Arhitektura računala

Miličić, Ivan

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Ivan Miličić

Arhitektura računala

Završni rad

Osijek, 2016.

Sveučilište J. J. Strossmayera u Osijeku
Odjel za matematiku
Sveučilišni preddiplomski studij matematike

Ivan Miličić

Arhitektura računala

Završni rad

Voditelj: izv. prof. dr. sc. Domagoj Matijević

Osijek, 2016.

Sadržaj

Sažetak

U radu je opisana arhitektura 16-bitnog računala po imenu Hack računalo. Procesor, ključni dio svakog računala, služi za dohvaćanje i izvođenje naredbi. Osnovne naredbe podrazumijevaju spremanje nekog broja u računalo, iščitavanje ili izvođenje raznih operacija, aritmetičkih i logičkih, na danim podacima. Dio procesora koji izvodi operacije zove se aritmetičko-logička jedinica. Memorija je dio računala koji služi trajnom ili privremenom spremanju podataka. Uz pojam memorije vežu se ulazne i izlazne jedinice koje s računalom komuniciraju preko memorije. Naredbe se ovom računalu mogu pisati ili u strojnom jeziku, kao jedinice i nule, ili u asembleru, simboličkom jeziku, pri čemu ih je, ako su pisane simbolički, potrebno prevesti na strojni jezik prije samog izvođenja. Postoje dvije vrste naredbi: prva vrsta omogućava učitavanje i spremanje brojeva u računalo, a druga iščitavanje i izvođenje operacija. Sve spomenute jedinice mogu se povezati u jednu cijelinu – računalo, i u tom slučaju se rad računala može opisati vrlo jednostavno: procesor prima naredbu iz memorije, izvodi je, a potom eventualne podatke sprema u memoriju te ponovno šalje naredbu memoriji za učitavanje sljedeće naredbe.

Ključne riječi

Procesor, logički sklopovi, zbrajalo, aritmetičko-logička jedinica, upravljačka jedinica, ulazno/izlazne jedinice, bistabil, registar, brojač, memorija, strojni i asemblerski jezik, asembler.

Abstract

In this work I explain architecture of a 16-bit computer named Hack computer. Processor is main part of every computer and is used to fetch and execute instructions. Basic instructions are used for saving numbers in computer, reading or executing various arithmetic or logical operations. Arithmetic-logic unit is a part of a processor which executes operations with numbers. Memory unit is a part of a computer used for temporary or permanent saving of data. Input and output units communicate with computer via memory unit. There are two ways of writing writing programs for Hack computer: as zeros and ones (machine language) or simbolically (assembly language). If a program is written using assembler, then it must be translated to machine language in order to execute it. There are two types of instructions: first type is capable of loading and storing numbers in computer, and second type is used for reading and doing operations. All mentioned units can be connected together in order to make the Hack computer. This computer is working very simply: processor receives instruction from memory, executes it and, if there are outputs, saves them in memory and then sends another instruction to memory for loading next instruction.

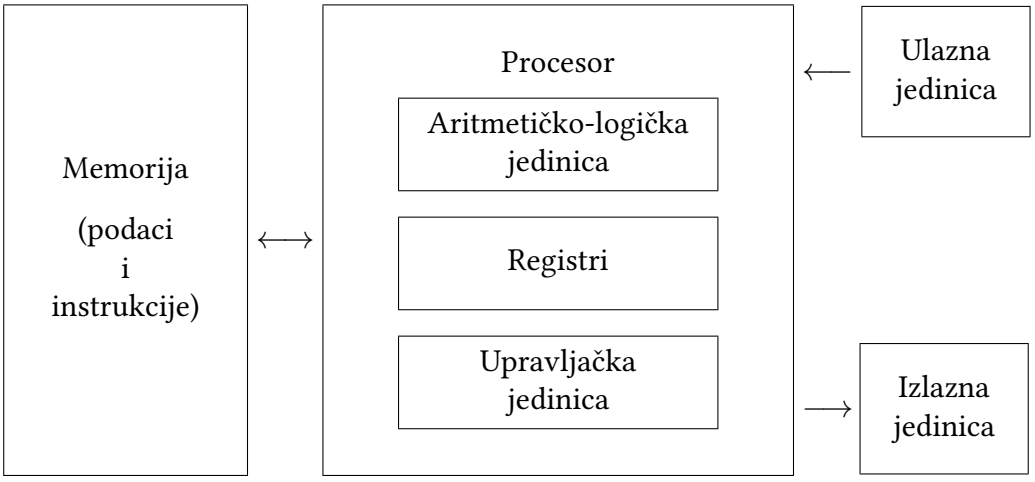
Key words

Processor, logic gates, adder, arithmetic-logic unit, control unit, input/output unit, flip-flops, register, program counter, memory, machine language and assembly, assembler.

Uvod

U ovom radu nastojat ću opisati rad jednog 16-bitnog računala po imenu Hack. Ono je napravljeno po von Neumannovoj arhitekturi[1] koja se sastoji od sljedećih jedinica:

- procesor (eng. *Central Processing Unit*; kratica: CPU) koji se sastoji od:
  - upravljačke jedinice,
  - aritmetičko-logičke jedinice,
  - registara,
- memorija,
- ulazne i izlazne jedinice.



Slika 1: Skica von Neumannove arhitekture računala.

Svaki od ovih dijelova računala bit će opisani u poglavljima. U prvom poglavlju ukratko su opisani logički sklopovi kao temeljni uređaji u izradi računala te aritmetičko-logička jedinica, bitna sastavnica svakog procesora. To poglavlje završava kratkim opisom rada upravljačke jedinice. U idućem, drugom poglavlju, ukratko je opisano kako se ostvaruje veza između ulazno/izlaznih jedinica i računala. Potom se detaljno objašnjava način na koji se može izraditi memorija: pomoću bistabila i registara, koji predstavljaju osnovne dijelove memorije. Nakon njihovog opisa slijedi kratak opis same memorije. U trećem poglavlju opisan je programski jezik Hack računala: strojni i asemblerski jezik paralelno. U tom poglavlju je dano i nekoliko jednostavnijih primjera uz popratne komentare. Završno poglavlje daje implementaciju procesora i kratki opis implementacije. Na samom kraju dana je implementacija Hack računala, tj. povezani osnovni dijelovi računala u jednu cijelinu.

U svrhu pisanja ovoga rada položio sam kurs Nand2tetris na Courseri (vidi [2]) te sam se većinom oslanjao na prvih šest poglavlja knjige *Elements of Computing Systems* (vidi [1]). U radu su korišteni i dijagrami koji su izrađeni pomoću programa Logisim (vidi [3]).

Sada ćemo krenuti s opisivanjem procesora i njegovih komponenti.

[1] John von Neumann, (1903.–1957.), američki matematičar i polimat mađarskog porijekla.

Poglavlje 1

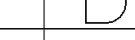
Procesor

Procesor je uređaj računala koji služi izvođenju određenog skupa unaprijed zadanih operacija. Te operacije dijelimo na aritmetičke i logičke, operacije upravljanja te operacije na ulaznim i izlaznim jedinicama. Od aritmetičkih operacija u Hack računalu bit će implementirano zbrajanje i oduzimanje, negacija (u aritmetičkom smislu) te određene konstante (0, 1 i -1), a od logičkih operacija konjukcija (I, u oznaci &), disjunkcija (ILI, u oznaci $|$) i negacija (NE, u oznaci $!$). Navedene operacije implementirane su direktno u hardveru Hack računala, a složenije (npr. množenje i složenije logičke operacije) se implementiraju na razini softvera.

Kako je svaki dio računala zapravo sastavljen od velikog broja logičkih sklopova međusobno povezanih sabirnicama na neki način, krenimo najprije od njih.

1.1 Logički sklopovi

Logički sklopovi su uređaji koji implementiraju logičke operatore (npr. I, ILI, NE,...), tj. izvode logičke operacije na jednom ili više logičkih ulaznih podataka, a vraćaju jedan ili više logičkih izlaznih podataka.[1] U tablici ispod dan je popis svih osnovnih logičkih operatora korištenih u izradi računala.

Logički sklop	Simbol	Logički sklop	Simbol
AND		NAND	
OR		NOR	
XOR		XNOR	
NOT			

Tablica 1.1: Popis osnovnih logičkih sklopova korištenih u izradi računala.

[1] Uz logičke operatore (a time i logičke sklopove) veže se jedan bitan teorem iz matematičke logike. Taj teorem kaže da se svi binarni operatori mogu na neki način dobiti kombiniranjem isključivo operatora NI ili NILI. Ta tvrdnja vrijedi samo za ta dva operatora pa kažemo da operator NI čini bazu za skup svih dvočlanih operatora (postoji $2^4 = 16$ takvih operatora). Također se može pokazati i da operatori I i NE te ILI i NE čine dvije baze za skup svih dvočlanih operatora.

A	B	A and B	A or B	not A	A nand B	A nor B	A xor B	A xnor B
0	0	0	0	1	1	1	0	1
0	1	0	1	1	1	0	1	0
1	0	0	1	0	1	0	1	0
1	1	1	1	0	0	0	0	1

Tablica 1.2: Tablica istinitosti logičkih operatora.

Kao što je rečeno u uvodu, svi dijelovi računala zapravo se sastoje od velikog broja logičkih sklopova međusobno povezanih na neki način sabirnicama. Sabirnice (eng. *bus*) su zapravo skup vodiča koji služe za prijenos podataka. Sabirnice se razlikuju ovisno o veličini podataka koje njima mogu „putovati" (npr. 16-bitna sabirnica). Označavaju se obično crnom crtom, a ponekada i crtom u boji. Naime, ukoliko dijagrami prikazuju neki primjer, onda će crnom crtom biti označena 16-bitna sabirnica (i općenito, više-bitne sabirnice crnom), a zelenim crtama 1 bit podatka pri čemu svjetlo zelena boja označava 1, a tamno zelena 0. Primjer takvog dijagrama je npr. slika 1.1.

U dijagramima će se ulazni podaci označavati kvadratićem unutar kojeg će stajati oznaka `x` zajedno s veličinom podatka u bitovima, a izlazni podaci kružićem (ili zaobljenim kvadratom). Primjerice, na prvoj slici ispod dan je 1-bitni ulazni podatak, zatim 16-bitni ulazni podatak, a potom pripadni izlazni podaci:

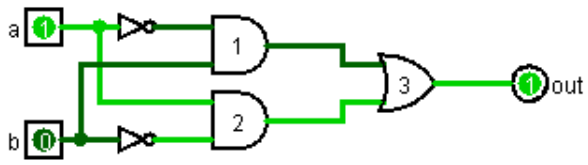
Često puta će uz ulazne i izlazne podatke stajati i njihovi nazivi (npr. `a`, `b`, `in`,...).

Osnovni logički sklopovi imaju svoje posebne oznake, a sklopovi izrađeni od njih najčešće se označavaju pomoću pravokutnika (ili kvadrata) unutar kojeg se nalazi oznaka što taj sklop predstavlja te, kod nekih sklopova, dodatne oznake uz ulaze. Primjerice,

Kod nekih sklopova se umjesto pravokutnika koristi trapez.

Na slici 1.1 je dana implementacije logičkog sklopa XOR korištenjem sklopova AND, OR i NOT. Uvjerimo se da dana implementacija zaista oponaša rad logičkog operatora XOR danog u tablici 1.2. Uočimo da sklop AND označen brojem 1 uvijek prima `!a` i `b`, drugi AND prima `a` i `!b`, a treći sklop OR prima rezultat prvog i drugog AND sklopa. Primjerice, za ulazne podatke `a=1`, `b=0`, prvi AND prima 0 i 0 i daje 0, drugi AND prima 1 i 1, a vraća 1, a OR prima 0 i 1 te kao konačan rezultat vraća 1. Na sličan način provjere se i ostale situacije.

Iz navedenih jednostavnih logičkih sklopova mogu se izraditi složeniji, kao funkcije od tri varijable ili više njih. Primjer logičkog sklopa koji implementira logičku funkciju s tri varijable je multipleksor.

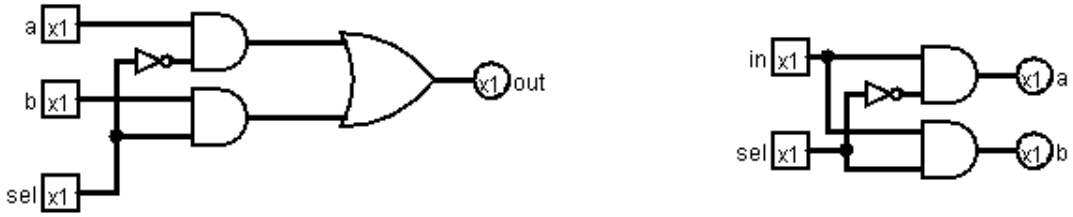


Slika 1.1: Dijagram implementacije logičkog sklopa XOR.

Multipleksori su logički sklopovi koji primaju tri ulazna podatka: `x`, `y` i `sel`. Multipleksor radi tako da pomoću podatka `sel` vraća jedan od preostalih dvaju podataka `x` i `y`. Ako je `sel=0`, onda vraća `x`, a ako je `sel=1`, vraća `y`.

Za razliku od multipleksora, demultipleksor radi suprotno: prima dva podatka `in`, `sel` i vraća dva podatka `a`, `b`. Ovisno o podatku `sel`, demultipleksor će vratiti dani podatak `in` kao `a` ili `b`. Naime, ako je `sel=0`, onda su vraćene vrijednosti `a=in` i `b=0`, a kada je `sel=1`, onda je `a=0` i `b=in`.

Na slikama 1.2 su dane njihove implementacije. Iz dijagrama multipleksora je vidljivo, kad je `sel=0`, da onda prvi AND sklop vraća `a` zato što prvi AND sklop prima `a` i `!sel`, tj. `a` i 1. U tom istom slučaju, drugi AND sklop prima `b` i `sel` te, kako je `sel=0`, vraća 0. Sada OR sklop prima `a` i 0 te je konačan rezultat `a`. Na sličan način se zaključi da u slučaju `sel=1` ovaj sklop vraća `b`. Provjerimo još na demultipleksoru slučaj kada je `sel=1`. Naime, tada prvi AND sklop prima `in` i 0, pa zbog toga vraća 0 te je `a=0`, a drugi AND sklop prima `in` i 1 te je `b=in`.



Slika 1.2: Dijagram implementacije multipleksora (lijevo) i demultipleksora (desno).

U dijagramima se multipleksori i demultipleksori označavaju trapezom:

Uz navedene logičke sklopove, koji rade isključivo s jednim bitom, postoje više-bitne verzije ovih sklopova, tj. logički sklopovi čiji ulazni i/ili izlazni podaci imaju više bitova (u našem slučaju 16). Označavamo ih s *And16*, *Or16*, *Not16* i *Mux16*. Nadalje, lako se generiraju i logički sklopovi koji primaju više podataka (eng. *multi-way logic gate*). Primjerice, OR u verziji s osam ulaznih podataka kojeg ćemo zvati *Or8Way* ili demultipleksori s 4 i 8 izlaznih podataka koje ćemo zvati *DMux4Way* i *DMux8Way*. Uočimo da ulazni podatak `sel` kod *DMux4Way* ima 2 bita. Slično, kod *DMux8Way*, ulazni podatak `sel` ima 3 bita.[2]

No, također postoje i logički sklopovi s obje opcije: više-bitni multipleksori koji primaju više podataka. Primjerice, 16-bitni multipleksor s 4 ulazna podatka *Mux4Way16* te 8 ulaznih podataka

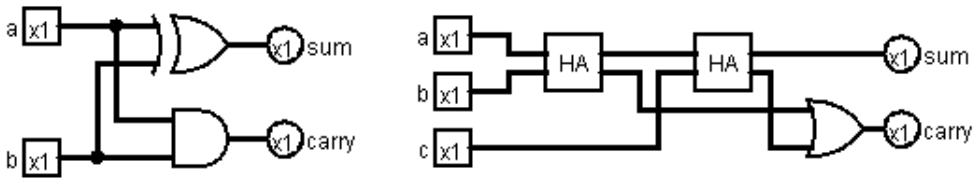
[2]Ulazni podatak `sel` uvijek će imati $\log_2 n$ bita, gdje n predstavlja broj izlaznih podataka!

Mux8Way16. Njihove implementacije mogu se dobiti korištenjem multipleksora s 2 ulazna podatka. Slično kao i kod demultipleksora, multipleksori s 4 ulazna podatka primaju 2-bitni `sel`, a multipleksori s 8 ulaza 3-bitni `sel`!

1.1.1 Zbrajala

Zbrajala su logički sklopovi koji izvode zbrajanje brojeva s dvostrukim komplementom. Mogu se podijeliti na poluzbrajala i potpuna zbrajala.

Poluzbrajalo (eng. *half adder*; kratica: HA) prima dvije binarne znamenke `a` i `b`, a vraća desnu znamenku njihovog zbroja kao `sum` i lijevu kao `carry` (znamenka prijenosa). Potpuno zbrajalo (eng. *full adder*; kratica: FA) prima tri binarne znamenke `a`, `b` i `c`, a vraća desnu znamenku njihovog zbroja kao `sum` i lijevu znamenku kao `carry`.



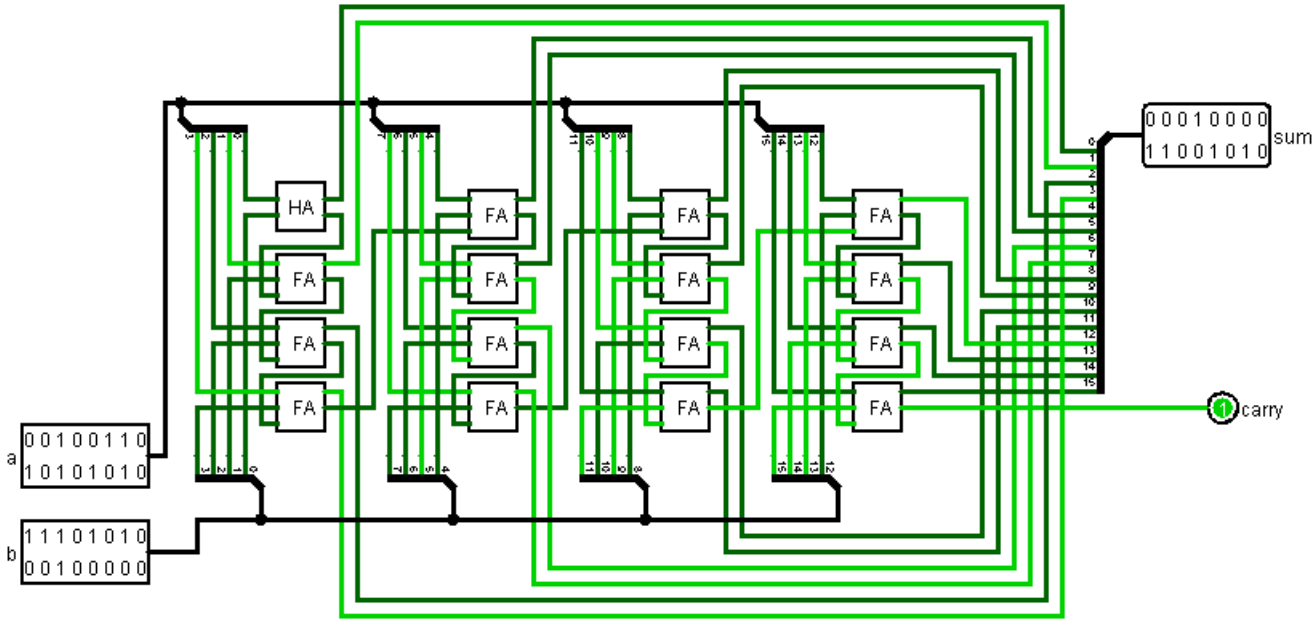
Slika 1.3: Dijagram implementacije poluzbrajala (lijevo) i potpunog zbrajala (desno).

Kao i do sada, lako se može provjeriti da dana implementacija poluzbrajala zaista radi na način koji je gore opisan. Drugi način provjere je da se ta implementacija izvede. Naime, želimo da `sum` u slučajevima `a=1`, `b=0` i `a=0`, `b=1` daje 1, a u slučajevima `a=1`, `b=1` i `a=0`, `b=0` daje 0. Iz tablice 1.2 zaključujemo da `sum` upravo odgovara rezultatu logičke operacije `a xor b`. Na isti način lako se zaključi da se `carry` bit dobije kao rezultat konjunkcije ulaznih podataka.

Potpuno zbrajalo može se implementirati koristeći poluzbrajalo. Naime, po istom zaključivanju kao i gore je `sum=(a xor b) xor c`, pa na taj dio primijenimo HA sklopove. Dakle, prvi HA prima `a` i `b`, a drugi HA prima `sum` bit od prvog HA i `c` te vraća konačni `sum`. Izlazni podatak `carry` možemo dobiti sljedećim zaključivanjem: kad god dođe do prijenosa znamenke unutar HA sklopa, odmah automatski dolazi i do prijenosa znamenke u cijelom zbrajanju. Bitno je uočiti da nije moguće da se u oba HA sklopa istodobno dogodi prijenos znamenke. To vrijedi zato što se zbrajanjem 3 binarne znamenke mogu dobiti najviše 2-bitni brojevi. Dakle, kako se može dogoditi najviše jedna znamenka preskoka, izlazne `carry` bitove HA sklopova treba povezati u OR sklop i time dobiti izlazni podatak `carry`.

Koristeći poluzbrajalo i potpuno zbrajalo moguće je napraviti zbrajalo koje će zbrajati više-bitne brojeve. Naime, ulazni podatak `c` potpunog zbrajala ima ulogu znamenke prijenosa dobivene u prethodnom zbrajalu. Upravo zbog toga se potpuno zbrajalo koristi u izradi više-bitnih zbrajala. Skica jednog 16-bitnog zbrajala dana je na slici 1.4. U dijagramima će se zbrajala, neovisno o veličini ulaznih podataka, označavati s kvadratom unutar kojeg će biti znak „+”.

Ponekada će u implementacijama, kada se koristi 16-bitni podaci, biti potrebno korištenje samo nekoliko bitova u određene svrhe. Drugim riječima, željet ćemo, iz više-bitnog podatka uzeti samo neke bitove i na njima obavljati neke operacije. To se postiže pomoću *splittera*. U dijagramima se primjerice prva četiri bita od nekog više-bitnog podatka razdjeljuju sljedećom oznakom:



Slika 1.4: Dijagram implementacije 16-bitnog zbrajala. Na slici je primjer zbrajanja brojeva 9898 i 59936 s tzv. *overflow*-om. Naime, stvarni rezultat je 69834, ali je prevelik da bi se mogao zapisati samo sa 16 binarnih znamenki pa je dobiveni rezultat 4298.

pri čemu se označe bitovi koji se koriste počevši od 0 i brojeći od najmanje značajnog bita (eng. *least significant bit*; kratica: LSB) prema najznačajnijem (eng. *most significant bit*; kratica: MSB)[3].

Objasnimo sada ukratko rad danog zbrajala sa slike 1.4. Uočimo da u prvom stupcu od ulaznih podataka uzimamo zadnja 4 bita (nulti bit je LSB, prvi bit je bit ispred LSB, itd.) i ubacujemo ih u pripadna zbrajala. Kako, kada tek krenemo zbrajati, imamo samo dvije znamenke, a ne tri, na početku koristimo HA sklop, a u svim ostalim slučajevima FA sklop jer zbrajamo dvije znamenke iz ulaznih podataka `a` i `b` te treću znamenku prijenosa dobivenu iz prethodnog zbrajanja. U svakom stupcu zbrajaju se po 4 znamenke ulaznih podataka, a njihov rezultat `sum` se šalje u veliki splitter koji formira konačni rezultat `sum`. Drugi izlazni podatak `carry` dobijemo iz zadnjeg FA sklopa u zadnjem stupcu zato što su svi ostali sklopovi međusobno povezani.

1.2 Aritmetičko-logička jedinica

Aritmetičko-logička jedinica (eng. *arithmetic-logic unit*; kratica: ALU) je logički sklop u kojemu su enkapsulirane sve osnovne aritmetičke i logičke operacije.

U Hack računalu ALU prima dva 16-bitna podatka `x` i `y` na kojima izvodi operacije (koje se određuju zadavanjem 6 kontrolnih bitova) i vraća 16-bitni rezultat. Šest bitova kojima se u ALU-u definira operacija označavaju se sa `zx`, `nx`, `zy`, `ny`, `f` i `no`. Uz 16-bitni rezultat, ALU još dodatno vraća dva bita (koje ćemo označiti sa `zr` i `ng`): `zr` daje informaciju o tome je li konačni rezultat jednak nuli, a `ng` informaciju o tome je li rezultat strogo negativan. Ta dva bita bit će nam potrebni

[3] LSB zapravo je zadnji bit binarnog zapisa nekog broja, a MSB prvi. Primjerice, $\underbrace{1}_{\text{MSB}}00111\underbrace{1}_{\text{LSB}}$.

kasnije kod strojnog jezika. Algoritam 1 daje kratki opis na koji način kontrolni bitovi djeluju na ulazne i izlazne podatke.

```
1: function ALU(x[16], y[16], zx, nx, zy, ny, f, no)
2:     if zx = 1 then
3:         x ← 0                                  ▷ 0 kao 16-bitna konstanta
4:     end if
5:     if nx = 1 then
6:         x ←!x                                  ▷ negacija po bitovima
7:     end if
8:     if zy = 1 then
9:         y ← 0
10:    end if
11:    if ny = 1 then
12:        y ←!y
13:    end if
14:    if f = 1 then
15:        out ← x + y                            ▷ zbrajanje s dvostrukim komplementom
16:    else                                       ▷ tj. kada je f = 0
17:        out ← x&y                              ▷ konjunkcija po bitovima
18:    end if
19:    if no = 1 then
20:        out ←!out
21:    end if
22:    if out = 0 then
23:        zr ← 1
24:    end if
25:    if out < 0 then
26:        ng ← 1
27:    end if
28:    return [out, zr, ng]                       ▷ Uočimo da je out 16-bitni podatak!
29: end function
```

Algoritam 1: Opis rada aritmetičko-logičke jedinice.

Postavljanjem kontrolnih bitova na određeni način moguće je, za ulazne podatke x i y, odrediti zbroj i razliku ili djelovanje logičkim operatorima I, ILI i NE.

U tablici 1.3 se nalazi popis operacija i kako ih dobiti preko kontrolnih bitova.[4] Nadalje, na slici 1.5 može se vidjeti implementacija aritmetičko-logičke jedinice, a na slikama 1.6 se nalaze dva primjera operacija na ALU-u.

zx	nx	zy	ny	f	no	operacija

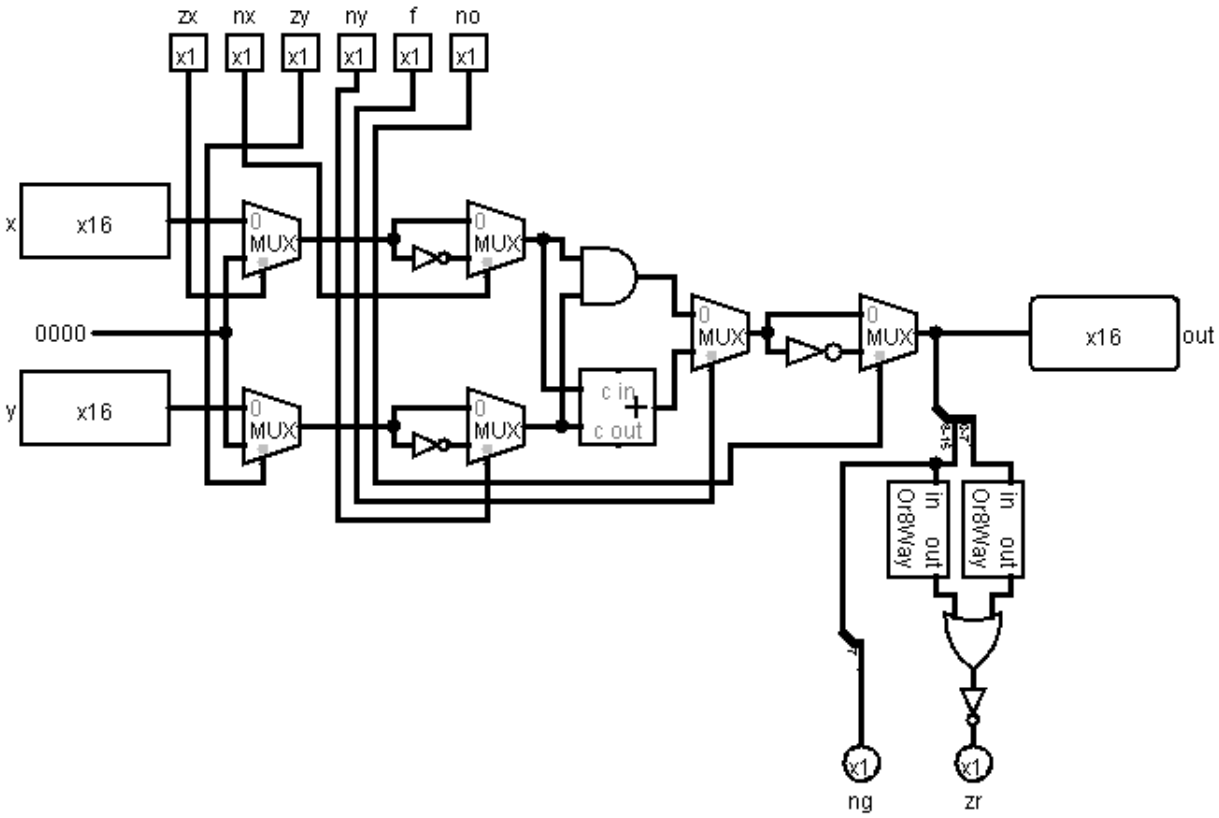
[4]Uočimo da postoji ukupno $2^6 = 64$ različita načina postavljanja kontrolnih bitova, pa i toliko različitih operacija. No, od svih njih, mi ćemo koristiti samo njih 18.

1	0	1	0	1	0	0
1	1	1	1	1	1	1
1	1	1	0	1	0	-1
0	0	1	1	0	0	x
1	1	0	0	0	0	y
0	0	1	1	0	1	!x
1	1	0	0	0	1	!y
0	0	1	1	1	1	-x
1	1	0	0	1	1	-y
0	1	1	1	1	1	x+1
1	1	0	1	1	1	y+1
0	0	1	1	1	0	x-1
1	1	0	0	1	0	y-1
0	0	0	0	1	0	x+y
0	1	0	0	1	1	x-y
0	0	0	1	1	1	y-x
0	0	0	0	0	0	x&y
0	1	0	1	0	1	x—y

Tablica 1.3: Prikaz operacija ALU-a s pripadnim kontrolnim bitovima.

Opišimo ukratko implementaciju ALU-a sa slike 1.5. Uočimo da prva dva MUX sklopa, postavljena u istom stupcu, zapravo izvode drugi i osmi redak iz algoritma 1. Dakle, prvi MUX vraća 16-bitni 0 ili x, ovisno o tome kakav je zx. Isto se događa sa MUX-om ispod njega. Druga dva MUX sklopa izvode 5. i 11. redak iz danog algoritma: vraćaju ili trenutni podatak ili ga negiraju. Potom slijedi konjugiranje i zbrajanje podataka te, pomoću MUX-a, odabiranje između x&y i x+y ovisno o tome kakav je f. Nakon toga preostaje još odabrati rezultat ili njegovu negaciju pomoću posljednjeg MUX-a koristeći no i dobiti konačan rezultat.

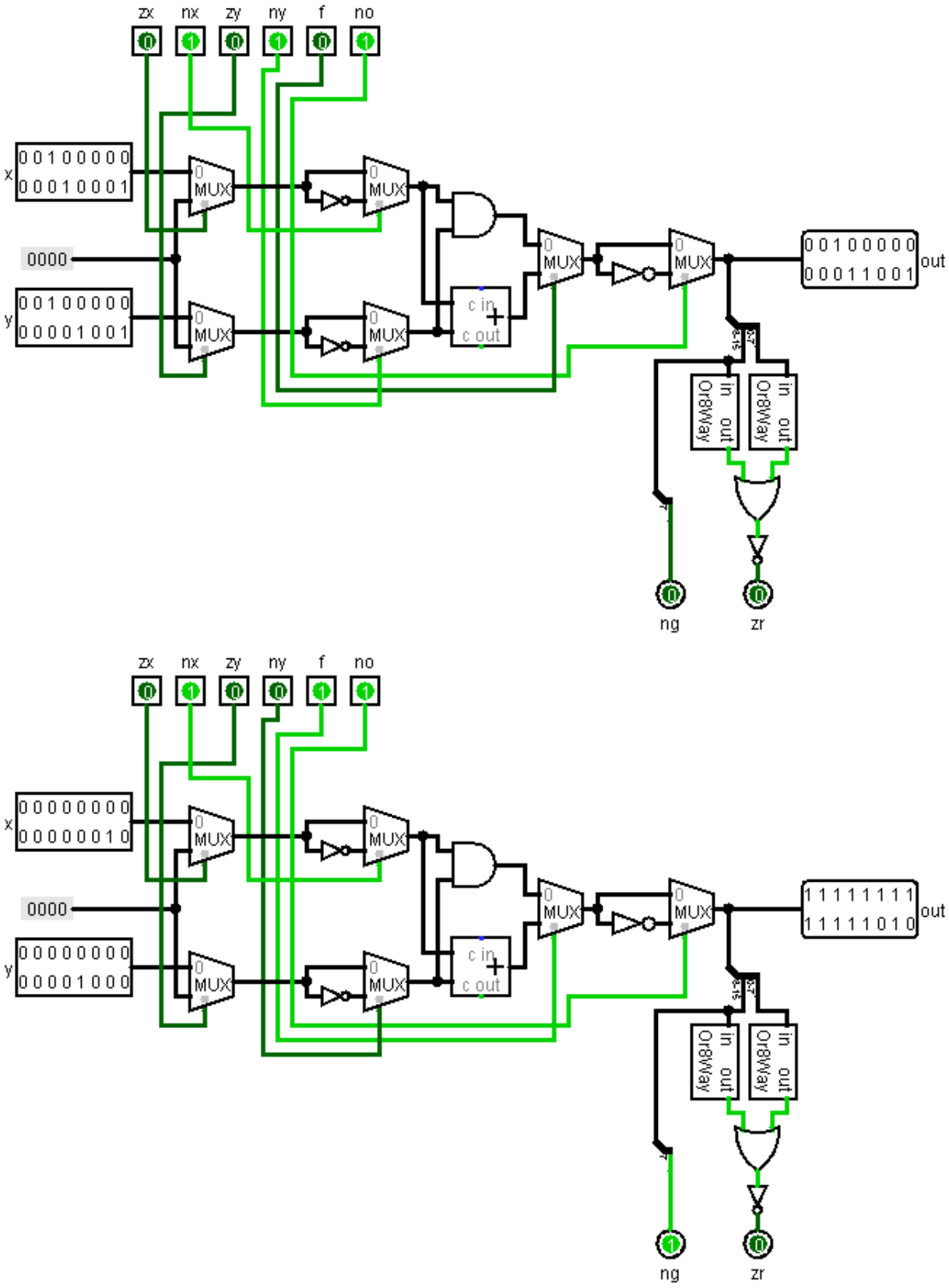
Nakon toga, na temelju konačnog rezultata određuju se izlazni podaci ng i zr. Uočimo da 15. bit (MSB, tj. 7. bit od dijela koji se sastoji od prvih 8 bitova) govori o tome je li broj negativan ili ne. Stoga, njega vratimo takav kakav je kao ng. Nadalje, kako bismo znali je li rezultat 0, uočimo da je dovoljno samo sve bitove konačnog rezultata međusobno ubaciti u OR sklop. To je napravljeno pomoću dva



Slika 1.5: Dijagram implementacije aritmetičko-logičke jedinice. Oznaka 0000 predstavlja 16-bitnu konstantu (*false*) prikazanu u heksadekatskom zapisu.

Or8Way sklopa koji bitove primljenog podatak međusobno disjungiraju. Njihovi se rezultati potom pošalju u jedan OR sklop i na kraju negira. Ako je primjerice konačan rezultat bio 16-bitna 0, tada su svi bitovi bili 0 pa je rezultat disjungiranja 0, a onda je `zr` bit jednak 1.

Prateći dani algoritam 1, moguće je zaključiti što se točno zbiva u dijagramima sa slike 1.6. Opišimo drugu sliku na kojoj je prikazano oduzimanje brojeva 2 i 8. Kako je `zx=zy=0`, rezultati prvih dvaju MUX-ova su `x` i `y`. Nadalje, budući da je `nx=0`, razultati druga dva MUX-a su `!x` i `y`. Kako je `f=1`, rezultat trećeg MUX-a je `(!x)+y` i konačno, jer je `no=1`, izlazni rezultat je `!((!x)+y)`. Logička negacija nekog podatka mijenja nule u jedinice i obratno. Primjerice, ako zamislimo da imamo 2-bitno računalo, tada negacija od 01 glasi 10. Uočimo da negacija zapravo odgovara oduzimanju najvećeg 2-bitnog broja s podatkom koji želimo negirati. U ovom slučaju, negaciju od 01 možemo dobiti kao $11 - 01 = 10$. Jednostavnije to možemo zapisati kao $(100 - 1) - 01 = (10^{10} - 1) - 01$ u binarnom zapisu, ili $(2^2 - 1) - 1$ u dekatskom zapisu. Jednostavnosti radi, neka nadalje najveći 16-bitni broj bude označen s b. Na sličan način, kako je `x` neki 16-bitni broj, njegova negacija je $b - x$. Dakle, tada izraz `!(!x+y)` postaje $b - ((b - x) + y) = b - (b - x + y) = x - y$ što je upravo ona operacija koju smo i željeli.



Slika 1.6: Primjeri operacija na ALU-u. Na prvoj slici se radi o logičkoj operaciji ILI, a na drugoj je prikazano oduzimanje brojeva 2 i 8 s rezultatom -6.

1.3 Upravljačka jedinica

Računalne naredbe se prikazuju binarnim kôdovima. No, prije nego što se ta naredba izvrši, potrebno ju je dekodirati, a informacije sadržane u njoj moraju se proslijediti različitim dijelovima računala (registrima, memoriji, ALU-u). To dekodiranje naredbi izvodi se pomoću upravljačke jedinice čija je uloga također i hvatanje te izvođenje sljedeće naredbe.

Stoga, kako bismo implementirali upravljačku jedinicu, potrebno je definirati naredbe, tj. strojni jezik Hack računala. Više o tome u četvrtom poglavlju.

Poglavlje 2

Ulazno/izlazne jedinice i memorija

2.1 Ulazno/izlazne jedinice

Hack računalo se može spojiti s dva vanjska uređaja: monitorom i tipkovnicom. Veza računala i tih ulazno/izlaznih jedinica (eng. *input/output unit*; kratica: I/O) ostvaruje se preko posebnog dijela memorije dodijeljenog monitoru i tipkovnici. To znači da se crtanje piksela na monitoru ostvaruje preko memorije dodijeljene monitoru, a unos znakova s tipkovnice preko memorije dodijeljene tipkovnici.

2.1.1 Monitor i tipkovnica

Monitor pridružen Hack računalu je crno-bijeli i sastoji se od 256×512 piksela. Kako je monitor s računalom povezan preko memorije, za pisanje ili čitanje piksela s monitora potrebno je pisati ili čitati pripadne bitove u memoriji dodijeljenoj monitoru, pri čemu 1 označava crni piksel, a 0 bijeli.

Tipkovnica je pridružena Hack računalu preko pripadne memorije. Kad god je pritisnuta neka tipka tipkovnice, njen ASCII kôd se prikaže u memoriji za tipkovnicu. Kada ni jedna tipka nije stisnuta, onda se na toj adresi nalazi 0.

2.2 Spremanje podataka

Spremanje podataka u računalu podrazumijeva pamćenje neke informacije obzirom na vrijeme. Stoga, potrebno je nešto čime ćemo u računalu mjeriti vrijeme, a to nešto se zove takt (eng. *clock*). Radi se o oscilatoru koji neprestano alternira u jednakim vremenskim razmacima između dvije faze: 0 (bez struje) i 1 (sa strujom). Vrijeme između početka 0 i kraja 1 se zove ciklus (eng. *cycle*) i uzima se kao jedinica vremena u računalu. Takt će se u dijagramima označavati sljedećim simbolom:

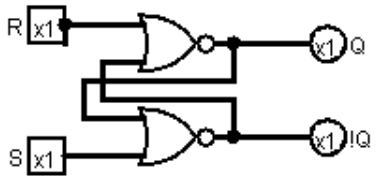
Logičkim sklopovima koji ovise o vremenu u računalu ćemo, unutar njihove oznake, stavljati trokut što će značiti da pripadaju tzv. sekvencijalnoj logici[1], a na mjestu trokuta će biti ulaz za takt, kao na slici:

[1] Sekvencijalna logika, za razliku od kombinacijske logike isključivo koju smo do sada koristili, je naziv za logiku strujnih krugova u kojoj izlazni podaci u trenutku t ovise o ulaznim podacima u trenutku $t-1$.

U implementaciji sekvencijalnih logičkih sklopova nećemo koristiti takt, nego samo uobičajeni ulaz umjesto takta s nazivom *clock*. Takt ćemo koristiti tek na kraju, kod implementacije Hack računala.

2.3 Bistabil

Budući da sada imamo jedinicu vremena u računalu, krenimo od logičkog sklopa koji će u trenutku t vraćati ono što je u prethodnom trenutku $t-1$ bio primio. Takav uređaj zvat ćemo bistabil (eng. *latch* ili *flip-flop*)[2]. Drugim riječima, bistabil je uređaj koji pamti jedan bit podatka. Postoje razne implementacije bistabila, a za početak ćemo objasniti rad tzv. SR-bistabila (eng. *set-reset latch*) implementiranog pomoću NOR sklopa na slici 2.1. Sastoji se od dva ulazna bita S i R te dva izlazna Q i njegove negacije !Q.



Slika 2.1: Dijagram implementacije SR-bistabila pomoću NILI sklopa.

U tablici 2.1 je dan popis istinosnih vrijednosti SR-bistabila. Uvjerimo se da on zaista radi u skladu s opisanom tablicom. Objasnimo najprije situaciju S=0, R=1. U toj situaciji prvi NOR sklop prima R=1 i !Q. Iz tablice 1.2, koja prikazuje istinosne vrijednosti logičkih operatora, vidimo da kad god je neki ulazni podatak jednak 1, da je rezultat jednak 0. Stoga, Q=0. Nadalje, drugi NOR sklop prima S=0 i Q=0 pa iz iste tablice vidimo da je tada !Q=1. Uočimo da je u ovoj situaciji rezultat u potpunosti stabilan i ne ovisi koji se NOR sklop prije izvrši. Kada bismo sada nakon ove situacije postavili S=0 i R=0, sličnim zaključivanjem uvjerili bismo se da su vrijednosti od Q i !Q ostale nepromijenjene.

Na sličan se način provjeri situacija S=1, R=0. Iz toga zaključujemo da kada su oba ulazna podatka postavljena na 0, da se onda radi o spremanju vrijednosti Q. Situacija kada je samo S postavljen na 1 uzrokuje postavljanje vrijednosti Q na 1 (eng. *set*), dok situacija kada je samo R postavljen na 1 uzrokuje Q=0 (eng. *reset*).

Provjerimo još situaciju S=1, R=1. Kako oba NOR sklopa primaju jedinicu, zaključujemo da oba sklopa vraćaju 0. Postavimo sada nakon ove situacije i S i R na 0. Naime, ako se prvi NOR sklop ostvari prije drugog, onda je njegov rezultat Q=1 jer prima R=0 i !Q=0, a drugi sklop, budući da prima S=0 i Q=1, vraća !Q=0. No, ako se drugi NOR sklop izvrši prije prvog, tada drugi NOR sklop vraća !Q=1, a prvi vraća Q=0. Zaključujemo da u ovoj situaciji ne možemo znati što će se dogoditi: nestabilna je i zbog toga se ne koristi. Kako bi se taj problem zaobišao, često se još dodaju logički sklopovi koji će situaciju S=R=1 pretvoriti ili u Q=1 ili u Q=0. Drugi način je da u slučaju S=R=1

[2]U engleskoj literaturi razlikuju se pojmovi latch i flip-flop. Naime, latch zapravo predstavlja jednostavniji oblik, samo s ulaznim podacima, dok se flip-flop sastoji od dva spojena latcha koji, uz ulazne podatke, ima i ulaz za takt.

izlazne vrijednosti zamijene mjesta. Takav uređaj se zove JK-bistabil. Više o bistabilima može se pročitati u [4].

S	R	Q_{next}
0	0	Q
0	1	0
1	0	1
1	1	X

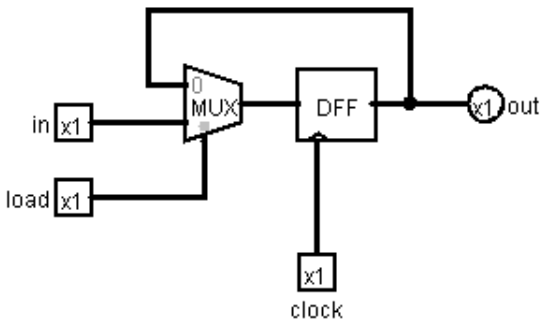
Tablica 2.1: Istinosna tablica za SR-bistabil.

U implementaciji Hack računala koristit ćemo D-bistabil (eng. *data flip-flop*; kratica: DFF). Taj tip bistabila i prima i vraća jedan bit podatka. Dodatno, ima i ulaz za takt koji se neprestano mijenja ovisno o glavnom taktu zbog čega se podatak sprema jedino kada je takt u stanju 1. Dakle, u trenutku t D-bistabil vraća ono što je primio u trenutku $t-1$. Implementacija D-bistabila može se naći u [4].

2.4 Registar

Pomoću danog D-bistabila sada je moguće napraviti registar. Najprije ćemo napraviti 1-bitni registar. To će zapravo biti samo proširenje D-bistabila: dodat ćemo još jedan ulazni podatak `load` koji će utjecati na spremanje ulaznog podatka `in`. Naime, samo kada je `load` postavljen na 1, podatak `in` se sprema u D-bistabil. Implementacija mu je dana na slici 2.2. Uočimo da kada je `load=1`, tada MUX sklop vraća `in` te se taj podatak sprema u DFF. Nadalje, ako je `load` postavljen na 0, onda MUX sklop odabire prethodni rezultat DFF-a pa se zapravo ništa neće promijeniti.

U dijagramima ćemo ovaj registar označavati kvadratom unutar kojeg će biti oznaka za ulaz `in` te skraćena oznaka `ld` za `load`.



Slika 2.2: Dijagram implementacije 1-bitnog registra.

Pomoću ovog registra lako se može dobiti i 16-bitni registar. Naime, može se mplementirati sličnim povezivanjem 1-bitnih registara kao i 16-bitno zbrajalo. U dijagramima ga označavamo pravokutnikom s istim oznakama kao i kod 1-bitnog registra. Sadržaj 16-bitnog registra u Hack računalu

zovemo riječ (eng. *word*), a duljinu riječi označavamo slovom w (eng. *word width*). U našem slučaju je $w = 16$.

Na skici von Neumannove arhitekture računala 1 unutar procesora su se nalazili i registri. Njihova uloga je povećanje brzine izvođenja operacija. Te registre možemo podijeliti na registre koji sadrže podatke (eng. *data register*; skraćeno: *D-registar*) i registre za adrese (eng. *addressing register*; skraćeno: *A-registar*). U Hack računalu, procesor ćemo implementirati s po jednim registrom od obje vrste.

Prije nego krenemo na objašnjenje memorije, opišimo ukratko brojanje u računalu koje se ostvaruje pomoću brojača.

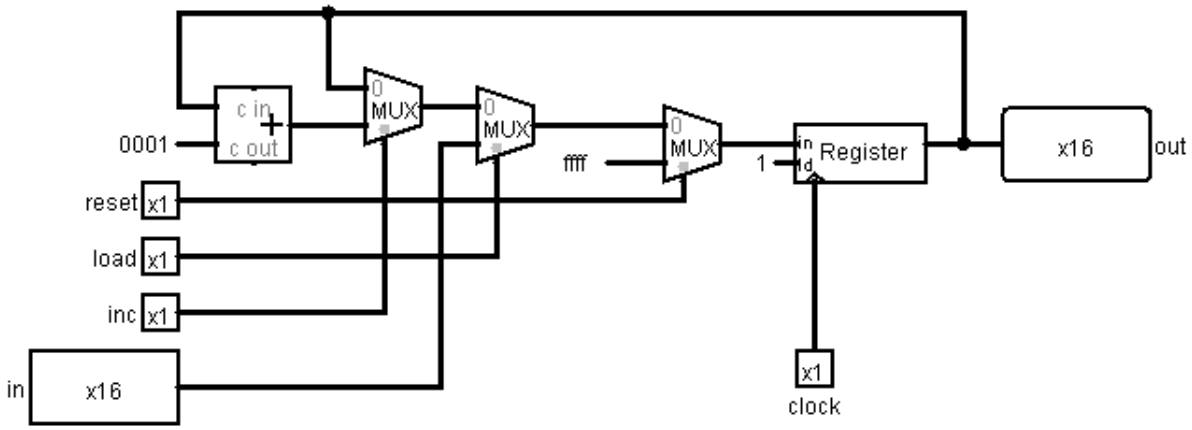
2.5 Brojač

Brojač (eng. *program counter*; kratica: PC) je logički sklop čija je vrijednost neki prirodni broj koji se povećava za 1 svake jedinice vremena u računalu.

U Hack računalu brojač počinje brojati od 0, a implementirat ćemo ga s tri ulaza: jedan za resetiranje brojača na -1 (`reset`), drugi za unos proizvoljnog broja u brojač (`load`) i treći za povećavanje vrijednosti brojača za 1 (`inc`). Najprije definirajmo slijed kojim se ove tri operacije izvode. Čim je `reset` postavljen na 1, bez obzira kakvi bili ostali ulazni podaci, odmah se vrijednost brojača resetira. Nakon resetiranja dolazi operacija unosa nekog broja u brojač, a na kraju povećanje vrijednosti brojača za 1. Primjerice, ako vrijednost brojača želimo povećati za 1, onda nužno moramo postaviti `reset` i `load` na 0, inače će se ostvariti neka od te dvije operacije. Implementacija je dana na slici 2.3.

Najprije uočimo da se u implementaciji brojača koristi jedan 16-bitni registar koji služi spremanju trenutne vrijednosti brojača. Ulazni podatak `load` tog registra postavljen je na konstantu 1 pa će u svakom trenutku registar spremati vrijednost `in` koju bude primio. Uočimo također da se željena operacija odabire pomoću MUX-ova: prvi MUX odabire između trenutne vrijednosti spremljene u registru i te vrijednosti uvećane za 1 ovisno o tome kakav je ulazni podatak `inc`, drugi MUX vraća 16-bitni ulazni podatak `load` ili ono što je vratio prethodni MUX, dok treći MUX odabire između resetiranja brojača na -1 i prethodno primljenog.

Kao i kod registra, brojač ćemo u dijagramima označavati pravokutnikom s oznakama `in`, `ld` i `inc` za ulaze.



Slika 2.3: Dijagram implementacije brojača. Uočimo da je kod zbrajala drugi 16-bitni ulazni podatak jedinica zapisana u heksadekatskom sustavu `0001`. Također uočimo da registar kao `load` prima konstantu 1 (tj. *true*), a treći MUX kao drugi ulazni podatak prima 16-bitni -1 zapisan u heksadekatskom sustavu kao `ffff`.

2.6 Memorija

Jedan način podjele memorije računala je na promjenjivu i nepromjenjivu. Promjenjiva memorija zahtjeva struju kako bi spremljene informacije ostale sačuvane pa, prilikom gašenja računala ili pri nestanku struje, svi se podaci odmah brišu. Suprotno od promjenjive je nepromjenjiva memorija. Dakle, nepromjenjiva memorija čuva podatke čak i nakon što je računalo isključeno. Više o ovakvoj podjeli memorije može se pogledati u [5].

2.6.1 Promjenjiva memorija

Primjer promjenjive memorije je memorija s nasumičnim pristupom (eng. *Random Access Memory*; kratica: RAM). Naime, naziv dolazi od zahtjeva da se svakoj riječi u RAM jedinici pristupi direktno jednakom brzinom bez obzira na fizičku lokaciju i broj registara. Taj je zahtjev ispunjen na sljedeći način: najprije se svakoj riječi u RAM jedinici dodijeli jedinstvena adresa, a potom se napravi logički sklop koji, za danu adresu i, označava registar s adresom i. Spomenuta adresa nije nikakav dodatni podatak, nego rezultat logičkih sklopova koji, ovisno o poretku registara, daju informaciju o njihovoj adresi.

Klasični RAM uređaj ima tri ulaza: ulaz za podatak `in`, ulaz za adresu `address` te ulaz `load` pomoću kojeg se izvršava spremanje (kada je postavljen na 1) ili iščitavanje (kada je postavljen na 0). RAM jedinica se vrlo lako implementira povezivanjem registara ovisno o tome koliku veličinu RAM-a želimo. Budući da se RAM sastoji od registara, pravilo je da u svakom trenutku bude označen samo jedan registar što se postiže ulaznim podatkom `address`. Veličina ulaznog podatka `address` ovisi o broju registara koje koristimo. Može se pokazati da je dovoljno $\log_2 n$ bitova za adresu, gdje je n broj korištenih registara.

U Hack računalu ćemo koristiti RAM koji se sastoji od 16K registara (veličina RAM-a je 32 KB).

2.6.2 Nepromjenjiva memorija

Primjer nepromjenjive memorije je memorija samo za čitanje (eng. *read-only memory*; kratica: ROM). Naime, ova vrsta memorije u nekim slučajevima dopušta promjenu podataka, ali ta promjena je spora, a u drugim slučajevima uopće ne dopušta pa se koristi za pohranu podataka i programa.

ROM Hack računala sastoji se od 32K registara (veličina ROM-a je 64 KB). Pretpostavit ćemo da je njegova implementacija dana, i to s 15-bitnim ulaznim podatkom `address` i 16-bitnim izlaznim podatkom `out` koji vraća podatak na danoj adresi.

Drugi način podjele memorije je na memoriju za podatke (eng. *data memory*) i memoriju za naredbe (eng. *instruction memory*). Često se te dvije memorije drže zajedno, no, zbog kompleksnosti koju takva implementacija uzrokuje, u Hack računalu se te dvije memorije implementirane odvojeno.

2.6.3 Memorija za podatke

Memorija za podatke je implementirana kao logički sklop s tri ulazna podatka: 16-bitni `in` koji kazuje što spremiti, `load` koji omogućuje spremanje podataka te 15-bitni `address` koji kazuje gdje će se ulazni podatak spremiti. Memorija za podatke vraća i podatak na danoj adresi. Ponekad ćemo element ove memorije simbolički označavati kao `memorija[adresa]`. Registre memorije za podatke ponekada zovemo i M-registre, kako bi se razlikovali od registara koji se nalaze u procesoru.

Ova memorija sastoji se od RAM-a, memoriju za ulazne jedinice (tipkovnicu) i memoriju za izlazne jedinice (monitor). [3]

Memorija za monitor mapirana je počevši od adrese 16384 i zauzima 8K registara. Svaki redak monitora u memoriji odgovara nizu od 32 riječi. Dakle, piksel koji se nalazi u retku r (gledajući od početka) i stupcu s (gledajući s lijeva) je mapiran bitom $s \bmod 16$ u riječi koja se nalazi na adresi $16384 + r \cdot 32 + c/16$. Memorija za monitor implementirana je preko logičkog sklopa koji ima tri ulazna podatka: 16-bitni `in`, 13-bitni `address`[4] i `load` te 16-bitni izlazni podatak `out`.

Memorija za tipkovnicu sastoji se od jednog registra na adresi 24576, a implementirana je kao logički sklop s isključivo izlaznim 16-bitnim podatkom `out`.

Činjenica da je adresa memorije za naredbe 15-bitna (a ne 16-bitna) bit će jasnija kasnije. No, uočimo da s 15-bitnom adresom možemo pristupiti 32K registara, a mi ih koristimo samo 24K+1.

2.6.4 Memorija za naredbe

U Hack računalu prethodno spomenuti ROM ima ulogu memorije za naredbe.

[3] Dijelovi memorije za podatke dijele isti adresni prostor!

[4] Memorija za monitor koristi samo 13 bitova za adresu jer s njima može pristupiti ukupno $2^{13} = 8192$ registara, što je upravo njena veličina.

Poglavlje 3

Programski jezik

Strojni jezik Hack računala sastoji se od isključivo nizova nula i jedinica. No, na strojni jezik Hack računala, iako najprimitivniji, gledat ćemo kao na bilo koji programski jezik. Programe pisane strojnim jezikom ćemo uređivati običnim *blokom za pisanje* i spremati ih na računalu s ekstenzijom *hack*. Nadalje, uz strojni jezik, bitno je spomenuti i asemblerski jezik (eng. *assembly*). Naime, radi se o nižem simboličkom jeziku, čovjeku čitljivijem nego li strojni jezik. Programi pisani u asemblerskom jeziku imaju ekstenziju *asm* i moraju se pretvoriti u strojni jezik, kako bi ga računalo moglo izvesti, a ta pretvorba radi se posebnim programom koji se zove asembler (eng. *assembler*). U daljnjem tekstu slijedi opis strojnog i asemblerskog jezika Hack računala.

U prethodnom poglavlju su bili opisani registri koje smo podijelili na A-registre i D-registre. U asemblerskom jeziku tim registrima pristupamo oznakama A i D. Dok se D-registar koristi isključivo za spremanje podataka, A-registar ima ulogu i podatka i adrese. Naime, ako želimo pristupiti nekom M-registru s nekom adresom, potrebno je prvo A-registar postaviti na tu adresu, a onda pristupiti tom M-registru koristeći oznaku M. Nadalje, A-registar ima ulogu i u memoriji za naredbe. Naime, postavljanjem A-registra na neki broj moguće je „skočiti" u samom programu na neki drugi redak.

Na sličan način, kao što smo podijelili registre unutar procesora, tako ćemo podijeliti i naredbe strojnog jezika Hack računala na A-naredbe (eng. *address instruction*) i C-naredbe (eng. *compute instruction*).

3.1 Naredbe

A-naredba se koristi za postavljanje A-registra na neku 15-bitnu vrijednost. Njena simbolička sintaksa je $@broj$ gdje je $broj$ neki pozitivni cijeli broj. Sintaksa ove naredbe u strojnom jeziku započinje s 0, a preostalih 15 bitova su proizvoljni. Dakle, prvi bit signalizira o kojoj je naredbi riječ pa se zbog toga u Hack računalu koristi 15-bitna adresa kod memorije za naredbe.

C-naredba se koristi za izračunavanje i spremanje. Sintaksa C-naredbe je $dest = comp; jump$ gdje $dest$ predstavlja gdje će se izračun spremiti, $comp$ predstavlja što treba izračunati, a $jump$ postavlja uvjet za „skok", tj. ako je uvjet, koji $jump$ predstavlja, ispunjen, onda će se dogoditi skok na liniju u programu određenu s vrijednosti spremljenoj u A-registru. U binarnom zapisu, sintaksa ove naredbe je $111accccccdddjjj$ pri čemu a i c bitovi određuju $comp$, d bitovi određuju $dest$, a j bitovi $jump$. Uočimo da prvi bit signalizira da se radi o C-naredbi, a druga dva početna bita su zapravo višak i dogovor je da ih se postavlja na 1. Sve moguće vrijednosti koje ove tri komponente mogu

imati su dane u tablicama 3.1, 3.2 i 3.3.

za $a = 0$	c_1	c_2	c_3	c_4	c_5	c_6	za $a = 1$
0	1	0	1	0	1	0	
1	1	1	1	1	1	1	
-1	1	1	1	0	1	0	
D	0	0	1	1	0	0	
A	1	1	0	0	0	0	M
!D	0	0	1	1	0	1	
!A	1	1	0	0	0	1	!M
-D	0	0	1	1	1	1	
-A	1	1	0	0	1	1	-M
D+1	0	1	1	1	1	1	
A+1	1	1	0	1	1	1	M+1
D-1	0	0	1	1	1	0	
A-1	1	1	0	0	1	0	M-1
D+A	0	0	0	0	1	0	D+M
D-A	0	1	0	0	1	1	D-M
A-D	0	0	0	1	1	1	M-D
D&A	0	0	0	0	0	0	D&M
D—A	0	1	0	1	0	1	D—M

Tablica 3.1: Popis operacija zajedno s pripadnim bitovima koji ih određuju. Uočimo da operacije odgovaraju danim operacijama ALU-a u tablici 1.3.

d_1	d_2	d_3	simbolički	destinacija
0	0	0	`null`	nigdje
0	0	1	M	memorija[A]
0	1	0	D	registar D
0	1	1	MD	memorija[A] i registar D
1	0	0	A	registar A
1	0	1	AM	registar A i memorija[A]
1	1	0	AD	registar A i D
1	1	1	AMD	registar A i D i memorija[A]

Tablica 3.2: Moguće destinacije.

j_1 ($out < 0$)	j_2 ($out = 0$)	j_3 ($out > 0$)	simbolički	efekt
0	0	0	`null`	nema skoka
0	0	1	JGT	ako je $out > 0$, skoči
0	1	0	JEQ	ako je $out = 0$, skoči
0	1	1	JGE	ako je $out \geq 0$, skoči
1	0	0	JLT	ako je $out < 0$, skoči
1	0	1	JNE	ako je $out \neq 0$, skoči
1	1	0	JLE	ako je $out \leq 0$, skoči
1	1	1	JMP	skoči

Tablica 3.3: Mogući uvjeti na skok.

3.2 Primjeri programa

Napišimo asemblerski program za zbrajanje brojeva 2 i 3.

```
0 @2
1 D=A
2 @3
3 D=D+A
4 @0
5 M=D
6 @6
7 0 ; JMP
```

Program 3.1: Primjer zbrajanja brojeva.

Uočimo da brojeve ne možemo direktno unijeti korištenjem operatora ili varijabli, nego preko A-naredbe. Kratak opis prethodnog programa po recima dan je ispod.

0. Učitavanje broja 2 u A-registar.
1. Spremanje vrijednosti iz A-registra u D-registar, tj. spremanje broja 2 u D-registar.
2. Učitavanje broja 3 u A-registar.
3. Zbrajanje vrijednosti koje se nalaze u D-registru i A-registru te spremanje dobivenog zbroja u D-registar.
4. Učitavanje broja 0 u A-registar.
5. Postavljanje M-registra na adresi 0 na vrijednost koja se nalazi u D-registru, tj. spremanje zbroja u `memorija[0]`.
6. Učitavanje broja 6 u A-registar.
7. Bezuvjetni preskok na liniju u programa, a broj linije određen je brojem spremljenim u A-registru; dakle, skok na 6. red programa

Redovi 6 i 7 zapravo označavaju kraj programa. Naime, to je potrebno zato što takt računala nikada ne miruje: kada ne bismo završili program na taj način, krenuli bi se izvoditi prazni redovi programskog kôda (jer programski je kôd spremljen u memorije za naredbe) kojih ima 32K.

Proširimo sada prethodni program na sljedeći način: neka se obavlja zbrajanje dvaju proizvoljnih brojeva koje korisnik zadaje u memoriji za podatke na adresama 0 i 1, a rezultat se sprema u memoriji za podatke na adresi 2.

```
0 @0
1 D=M
2 @1
3 D=D+M
4 @2
5 M=D
6 @6
7 0 ; JMP
```

Program 3.2: Proširenje prethodnog pograma.

Drugi program zapisan strojnim jezikom izgleda ovako:

```
0000000000000000
1111110000010000
0000000000000001
1111000010010000
0000000000000000
1110001100001000
0000000000000110
1110101010000111
```

Uočavamo da između prvog i drugog programa zapisanih simbolički i nema neke velike razlike. No, gledajući u prethodni strojni zapis lako je zaključiti da bi programiranjem na taj način bilo izrazito teško čitati, pisati i ispravljati programski kôd.

U asemblerskom jeziku Hack računala postoje i specijalne oznake koje olakšavaju programiranje:

- predefinirane oznake: jednom dijelu adresa RAM-a može se pristupiti koristeći sljedeće predefinirane oznake:
  - virtualni registri: oznake od `R0` do `R15` se koriste za adrese RAM-a od 0 do 15,
  - ulazno/izlazni pokazivači: oznake `SCREEN` i `KBD` koriste se za adrese 16384 i 24576, koje predstavljaju početak mapirane memorije monitora i tipkovnice,
- oznake za skok: definirane od strane korisnika, a služe za označavanje mjesta u kôdu na koji se može skočiti; označavaju se s `(...)` (s proizvoljnim nizom znakova unutar oblih zagrada),
- varijable: bilo koji simbol definiran od strane korisnika koji nije niti jedan od već predefiniranih simbola i nije prije upotrijebljen kao oznaka se koristi kao varijabla kojoj je pridružena jedinstvena memorijska adresa, s početkom od 16. adrese RAM-a.

Prilikom pisanja programa u asemblerskom jeziku dopušteno je ostavljati prazne redove ili uvlake, a komentari se pišu iza dvostrukih kosih crta `//`. Prilikom prevođenja, asembler je zadužen da sve oznake zamijeni pripadnim brojevima te ukloni prazne redove i uvlake, kao i komentare.

Koristeći prethodna pomoćna sredstva u pisanju programskog kôda, napišimo program koji će između dva zadana broja na memorijskim adresama 0 i 1 vratiti veći i spremiti ga na adresu 2.

```
0  @R0
1  D=M
2  @R1
3  D=D-M
4  @then
5  D;JGT  //if R0-R1>0 then *
6
7  @R1  //else R2=R1
8  D=M
9  @R2
10 M=D
11 @end
```

```
12 0;JMP
13
14 (then) // * R2=R0
15 @R0
16 D=M
17 @R2
18 M=D
19
20 (end)
21 @end
22 0;JMP
```

Program 3.3: Primjer određivanja većeg od dva broja.

U prvom dijelu programa (reci 0–5) računa se razlika brojeva na adresama 0 i 1 i sprema u D-registar (redak 3), a potom se provjeri je li dobiveni rezultat veći od 0 (redak 5). Ako je, onda će se dogoditi skok u programu na oznaku `then` (reci 14–18) i u tom slučaju će na poziciju 2 u memoriji spremiti prvi broj, a potom će program doći do retka 21 i završiti beskonačnom petljom. Ukoliko gornji uvjet nije bio ispunjen, onda će se nastaviti izvršavati program redak po redak pa će u recima 7–10 doći do spremanja drugog broja na memorijsku adresu 2, a potom bezuvjetni skok na oznaku `end` čime program završava beskonačnom petljom.

Nakon kratkog opisa programskih jezika Hack računala, može se konačno krenuti i na implementaciju samog računala.

Poglavlje 4

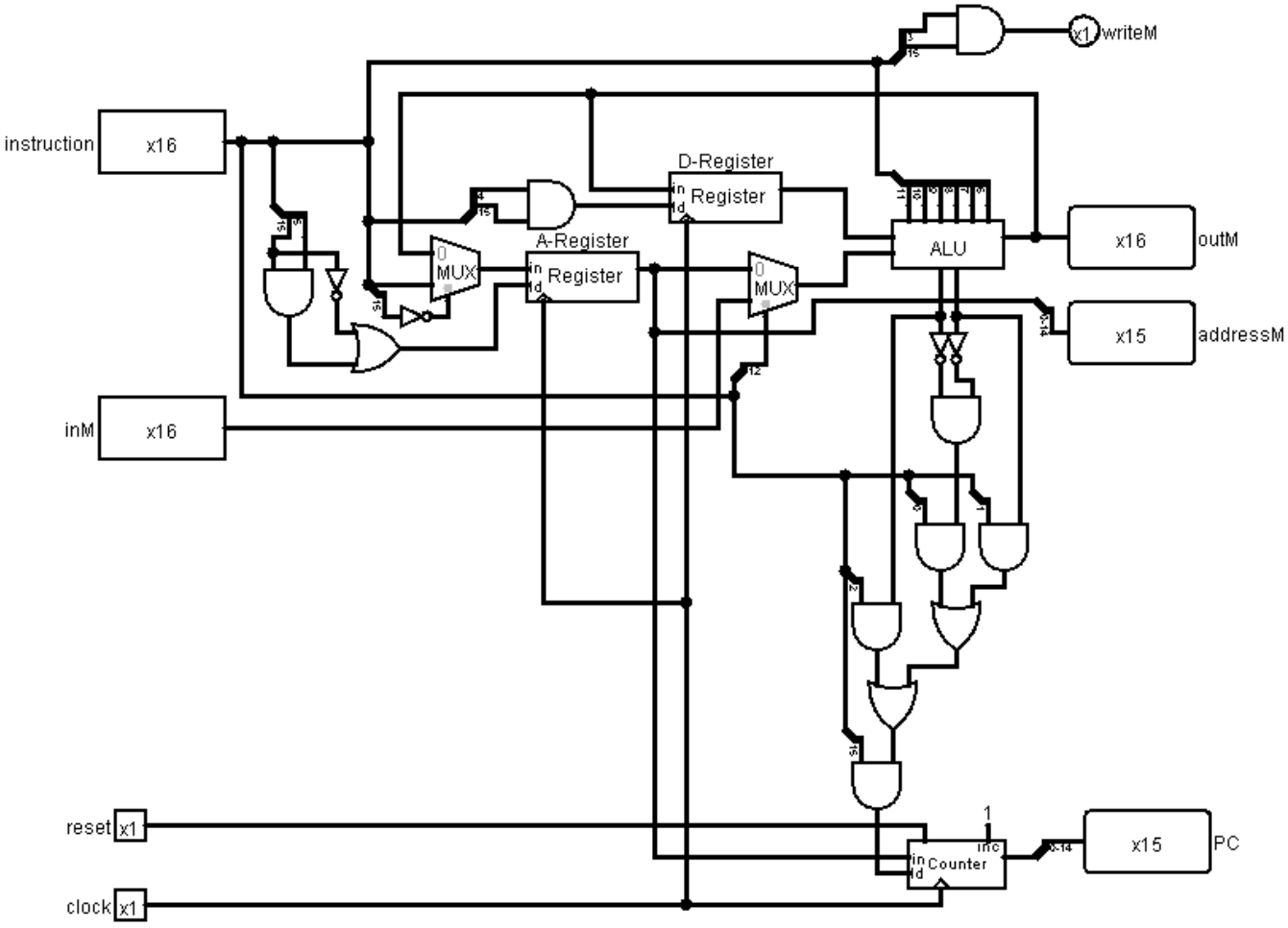
Hack računalo

Preostaje nam još završiti implementaciju cijelog računala. No, najprije, treba povezati do sada napravljene registre i aritmetičko-logičku jedinicu te brojač u procesor. To je napravljeno na slici 4.1. Na slici vidimo procesor kao logički sklop sa 16-bitnim ulaznim podacima `instruction` (dolazi od označenog registra iz memorije za naredbe) i `inM` (dolazi od označenog registra iz memorije za podatke) te jednim bitom `reset` koji zadaje korisnik i služi kao pokretanje procesora (tj. ponovno pokreće program iz memorije za podatke). Izlazni podaci implementiranog procesora su 16-bitni `outM`, 15-bitni `addressM`, jedan bit `writeM` te 15-bitni PC. Prva tri izlazna podatka, koja u imenu imaju oznaku `M`, spajaju se na memoriju za podatke, a posljednji se spaja na memoriju za naredbe. Na taj način vidimo da u svakom trenutku procesor daje memoriji za podatke i naredbe potrebne ulazne podatke pomoću kojih one vraćaju vrijednosti koje prima procesor (dohvaćanje podataka, eng. *fetch*), a potom procesor obrađuje te podatke (eng. *execute*) te vraća nove podatke koje ponovno primaju navedene memorije. U računalu se neprestano odvija ta povezanost između procesora i memorije.

U dijagramu sa slike 4.1 opisat ćemo samo ključne dijelove. Krenimo od ulaznog podatka `instruction`. On se dekodira tako da se određeni dijelovi te naredbe spoje na potrebna mjesta. Bitno je uočiti da je naredba spojena na MUX. Taj MUX sklop bira između naredbe i onoga što je ALU vratio, ovisno o tome kakva je naredba. Naime, ako se radi o A-naredbi, onda taj MUX izabire naredbu, a u suprotnom odabire rezultat ALU-a.

Pretpostavimo najprije da se radilo o A-naredbi. Procesor je tako implementiran da se u A-registar uvijek spremi A-naredba (dakle, ako se radilo o A-naredbi, ulazni podatak `load` A-registra će biti jednak 1). Nadalje, nakon što je A-naredba spremljena u A-registar, ona putuje prema sljedećem MUX-u, koji također prima i `inM`. Sada nam daljni tijek odvijanja nije toliko bitan. Drugim riječima, u slučaju A-naredbe, operacija koja se računa u ALU-u je proizvoljna s proizvoljnim rezultatom, no izlazni podatak `writeM` će uvijek biti jednak 0 pa zbog toga, koje god konačno rješenje bilo, ono se neće spremiti u memoriju. Jedini bitan podatak je `addressM` koji jednostavno vraća ono što je spremljeno u A-registru, tj. A-naredbu (jer A-naredba ima ulogu i adrese). Preostaje još vidjeti što se događa s brojačem. Naime, implementacija procesora je napravljena tako da se u slučaju A-naredbe u brojaču izvodi naredba uvećavanja za 1.

Sada ukratko proučimo što će se dogoditi ako je ulazna naredba bila C-naredba. Prvi MUX sklop prima rezultat ALU-a i `instruction`, no u slučaju C-naredbe uvijek se odabire rezultat ALU-a. Taj rezultat ALU-a potom ulazi u A-registar i sprema se u njemu jedino ako je $dest$ polje C-naredbe bilo postavljeno na A. Nadalje, sljedeći MUX, ovisno o tome je li C-naredba u polju $comp$



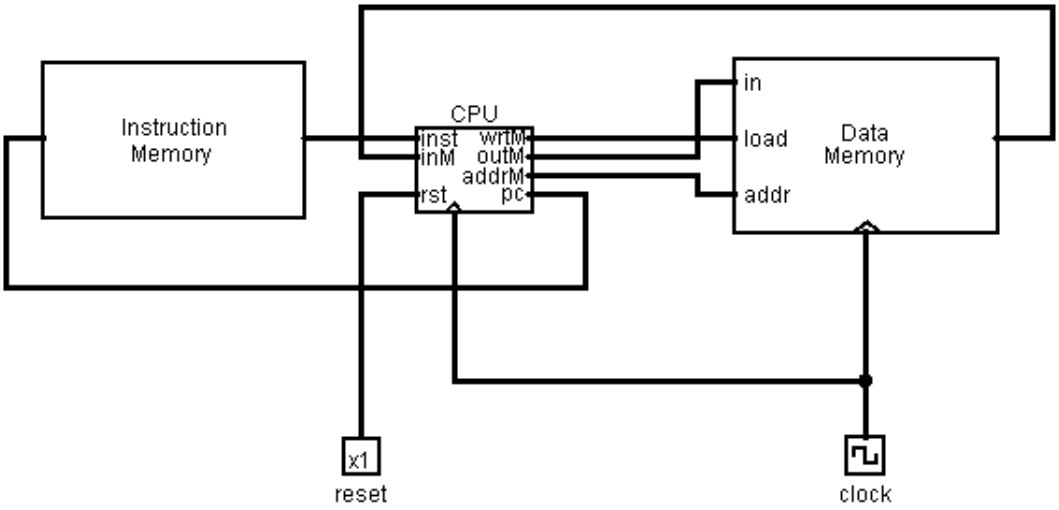
Slika 4.1: Dijagram implementacije procesora.

koristila podatak iz memorije ili ne, odabire između podatka iz A-registra i `inM`. U ALU-u se, ovisno o kontrolnim bitovima, računaju operacije na podacima. Dobiveni podatak će se potom spremiti u D-registar ukoliko se u *dest* polju C-naredbe koristila oznaka `D`. Nadalje, izlazni podaci iz ALU-a koji govore je li rješenje negativno ili jednako 0 utječu na brojač. Naime, veliki niz logičkih sklopova ispod ALU-a zapravo provjerava je li u C-naredbi korišten njezin *jump* dio. Ako je, onda će, ovisno o tome koji je tip preskoka korišten i ako su zadovoljeni uvjeti na rješenje, `load` bit kojeg prima brojač biti jednak 1 te će se vrijednost brojača promijeniti na broj spremljen u A-registru. Inače će se vrijednost brojača uvećati za 1. Još preostaje objasniti izlazni podatak `writeM`. Naime, ako je u C-naredbi u njenom *dest* polju korištena oznaka `M`, onda će taj izlazni podatak biti postavljen na 1 i krajnje rješenje će se spremiti u memoriji. Izlazni podatak `addressM` uvijek predstavlja sadržaj A-registra pa sve što je gore opisano za A-registar vrijedi i za ovaj izlazni podatak.

Sada se konačno može povezati procesor i memorije i time dobiti Hack računalo.

4.1 Implementacija Hack računala

Kao što je u prethodnim poglavljima bilo opisano, Hack računalo sastoji se od dvije zasebne memorije: memorije za naredbe i memorije za podatke. S procesorom su povezane na sljedeći način: memorija za naredbe od procesora prima broj koji kazuje redak u programskom kôdu koji se sljedeći treba učitati te vraća naredbu koja se nalazi u danom retku. Tu naredbu procesor prima kao `instruction`. Uz naredbu, procesor prima i sadržaj registra memorije za podatke kao `inM`, pri čemu je taj registar označen pomoću izlaznog podatka procesora `addressM`. Uz taj podatak, procesor još ima i izlazne podatke `writeM` i `outM` koji se spajaju na memoriju za podatke. Memorija za podatke i procesor povezani su taktom, a procesor još sadrži i dodatni ulazni podatak `reset` pomoću kojeg se resetira računalo. Implementacija cijelog računala dana je na slici 4.2.



Slika 4.2: Dijagram implementacije Hack računala.

Bibliografija

[1] Nisan, Noam; Schocken, Shimon. 2005. *The Elements of Computing Systems*. The MIT Press. London.

[2] Nisan, Noam; Schocken, Shimon. *Build a Modern Computer from First Principles: From Nand to Tetris*. Coursera: `https://www.coursera.org/learn/build-a-computer`

[3] Burch, Carl. 2005. *Logisim 2.7.1*: `http://www.cburch.com/logisim/`

[4] Flip-flop: `https://en.wikipedia.org/wiki/Flip-flop_(electronics)`

[5] Memorija: `https://en.wikipedia.org/wiki/Computer_memory`