Osječki matematički list 21 (2021), 57–74

Izgradnja funkcionalnog 16-bitnog računalnog sustava

Bartol Borozan*

Sažetak

Harvardska računalna arhitektura je jedna od najčešćih računalnih arhitektura korištenih u današnje vrijeme. Većina osobnih računala su građena prema njezinoj izmijenjenoj verziji. U ovome radu konstruiramo jednostavno 16-bitno računalo po harvardskoj arhitekturi sa svojim najvažnijim dijelovima: središnjom jedinicom za obradu, aritmetičko-logičkom jedinicom i memorijom. Posebnu pažnju ćemo posvetiti programabilnosti koji postižemo pažljivim dizajnom instrukcija.

Ključne riječi: *jezik za opis hardvera, harvardska računalna arhitektura, središnja jedinica za obradu, aritmetičko-logička jedinica, memorija*

Building a functional 16-bit computer system

Abstract

Harvard computer architecture is one of the most common computer architectures today and most personal computers are built on the principles of its modified version. In this paper, a simple 16-bit Harvard architecture computer system is constructed with its most important parts: central processing unit and memory, with the special attention given to the programmability achieved through a careful design of its set of machine instructions.

Keywords: *hardware description language, Harvard computer architecture, central pro- cessing unit, arithmetic-logic unit, memory*

*Sveučilište Josipa Jurja Strossmayera u Osijeku, Odjel za matematiku, email: bborozan@mathos.hr

1 Uvod

Funkcija najranijih računala je u potpunosti ovisila o njihovoj arhitekturi. Svaka promjena u radu takvih računala je često zahtijevala velike promjene u samoj arhitekturi. Takve su se promjene morale fizički provoditi rastavljanjem i ponovnim sastavljanjem raznih dijelova računala. Primjerice, reprogramiranje računala ENIAC, prvog reprogramabilnog računala, je trajalo nekoliko tjedana. Kako bi se olakšala reprogramabilnost, osmišljene su nove arhitekture u kojima je računalo u stanju izvršavati širok spektar generičkih operacija.

Jedna od takvih računalnih arhitektura koja se i u današnje vrijeme koristi u svom izmijenjenom obliku je von Neumannova arhitektura [4]. Sastoji se od središnje jedinice za obradu (procesor, CPU) i podatkovne memorije (RAM), u kojoj se nalazi i instrukcijska memorija. Instrukcijska memorija sadrži strojni kod, odnosno niz instrukcija pomoću kojih se upravlja procesorom. Procesor je povezan s memorijom s jednom sabirnicom. Sabirnica je vodič koji prenosi podatke između dijelova računala putem signala koje se interpretiraju kao nula ili jedan. Njezina širina određuje koliko bitova ona može paralelno prenijeti u isto vrijeme. Budući da je procesor povezan s memorijom preko samo jedne sabirnice, ograničeni protok podataka može loše utjecati na performanse. Ova pojava je u literaturi još poznata i pod nazivom von Neumannovo usko grlo [4]. S druge strane, harvardska arhitektura [1] razdvaja podatkovnu i instrukcijsku memoriju te na taj način značajno poboljšava performanse, unatoč povećanoj hardverskoj kompleksnosti. Većina današnjih računala su dizajnirana prema modificiranoj harvardskoj arhitekturi koja predstavlja instrukcije kao podatke i koristi nekoliko razina predmemorije (engl. *cache*) za pohranu podataka.

U ovome ćemo radu dizajnirati jednostavni 16-bitni model računala zasnovan na harvardskoj arhitekturi sa svojim osnovnim komponentama i specifikacijom strojnog jezika kako bi ilustrirali način rada te arhitekture. Rad prati predavanja iz računalne arhitekture Nand2Tetrisa [2] i kolegija Moderni računalni sustavi [3].

U prvome poglavlju predstavljamo jezik korišten za dizajniranje hardvera zvani HDL (engl. *hardware description language*) i navodimo nekoliko jednostavnih primjera. Koristeći HDL ćemo dizajnirati osnovne gradivne jedinice računala. Jedan od najvažnijih dijelova računala je aritmetičko-logička jedinica kojom ćemo se baviti u drugome poglavlju. U trećem poglavlju

ćemo pokazati kako implementirati registre i ostale memorijske čipove. Istaknut ćemo njihovu vremensku ovisnost i važnost hardverskog sata. U završnom dijelu ovog rada bavit ćemo se dizajnom CPU-a i njegovom ulogom u računalu baziranog na harvardskoj arhitekturi.

1.1 Jezik za opis hardvera

Da bi se dizajnirao računalni sustav, potrebno je postupno graditi sve složenije dijelove pomoću jednostavnih čipova kao što su AND (logička konjunkcija), ILI (logička disjunkcija), XOR (logička ekskluzivna disjunkcija) i NOT (logička negacija) koji se koriste za izvršavanje osnovnih bitovnih operacija. Prije nego što dublje zađemo u HDL, ukratko ćemo ga opisati putem nekolicine jednostavnih primjera. Svaki čip dizajniran u HDL-u se sastoji od tri glavna dijela: ulazne sabirnice, izlazne sabirnice i dijelova. Ulazne sabirnice su vodiči koji prenose ulazne podatke (bitove) u čip. Dijelovi čipa, koji su i čipovi samo po sebi, procesuiraju ulazne podatke te ih zatim vraćaju putem izlaznih sabirnica.

Svako definiranje čipa, koristeći HDL, započinje ključnom riječi čip nakon koje slijedi ime čipa. Zatim, nakon ključne riječi IN, slijede sve ulazne sabirnice. Nakon njih deklariramo sve izlazne sabirnice pomoću ključne riječi OUT. Širina sabirnice može biti veća od jednog bita, što je tada naznačeno upotrebom sintakse nalik na polje. Primjerice, sabirnicu foo širine četiri deklariramo kao $foo[4]$.

Započnimo s definiranjem jednostavnog čipa za zbrajanje zvanog poluzbrajalo koji sadrži dvije ulazne sabirnice *a* i *b*, dvije izlazne sabirnice *sum* i *carry* veličine jednog bita. Čip će izračunati sumu *a* i *b* te ju vratiti putem izlazne sabirnice *sum*. Ako rezultat zbrajanja premaši jedan, tada će vrijednost izlazne sabirnice *carry* biti postavljena na jedan. Definicija čipa poluzbrajalo slijedi. Primijetimo da će *carry* biti jedan ako i samo ako su obje ulazne vrijednosti *a* i *b* jednake jedan, dok će *sum* biti jedan ako i samo ako je točno jedna od ulaznih vrijednosti jedan.

```
CHIP HalfAdder
{
        // Ulazne sabirnice a i b, sirine 1-bit
        IN a, b;

        // Izlazne sabirnice sum i carry, sirine 1-bit
        // sum = a + b
```

```
        // carry = if (a == b == 1) then 1 else 0
        OUT sum, carry;

        PARTS:
        Xor(a = a, b = b, out = sum);
        And(a = a, b = b, out = carry);
}
```

Unutarnja je logika čipa definirana nizom dijelova (čipova izlistanim nakon ključne riječi PARTS) međusobno povezanih s proizvoljnim brojem unutarnjih sabirnica koji dodatno mogu biti i povezani s ulaznim i izlaznim sabirnicama. U prethodnom primjeru koristimo dva čipa, Xor i And, za izvršavanje operacije poluzbrajala. Ti su čipovi jedni od niza gotovih čipova koje pruža HDL. Lista svih gotovih čipova se nalazi u [5]. Svaki dio je definiran nizom ulaza i izlaza. Primjerice, u čip poluzbrajalo, Xor čip ima ulaze a i b (s lijeve strane) u koje su priključene sabirnice a i b (s desne strane). Taj Xor čip računa vrijednost ekskluzivne disjunkcije koristeći svoje ulaze kao operande te šalje rezultat na svoju izlaznu sabirnicu koja je usmjerena na *sum* izlaznu sabirnicu čipa poluzbrajalo. Primijetimo da je redoslijed čipova u dijelovima proizvoljan.

Još jedan zanimljiv primjer je multiplekser čip. On sadrži tri ulazne sabirnice, a, b i sel, širine jedan bit. Ako je sel postavljena na nula, čip prosljeđuje a na izlaznu sabirnicu out. U suprotnom se prosljeđuje n na izlaznu sabirnicu out. Ovaj se čip koristi kao dio većeg čipa te odlučuje o posredničkom rezultatu koji bi se trebao proslijediti. Donekle je sličan programskoj strukturi grananja (primjerice *if-then-else*) koja kontrolira tijek programa. Slijedi HDL implementacija multiplekser čipa.

```
CHIP Mux
{
        IN a, b, sel;
        OUT out;

        PARTS:
        Not(in = sel, out = nsel);
        And(a = b, b = sel, out = r1);
        And(a = a, b = nsel, out = r2);
        Or(a = r1, b = r2, out = out);
}
```

Multiplekser čip implementira sljedeću logičku formulu

$$Mux(a, b, sel) = (a \wedge \neg sel) \vee (b \wedge sel).$$

Ako je $a = 1$ dok je $sel = 0$, tada je rezultat u drugoj zagradi s desne strane jednakosti jedan. Kako je $sel = 0$, rezultat prve zagrade je nula te u tom slučaju formula vraća jedan, što je bila i vrijednost od a. Ako je $a = 0$, tada je konačni rezultat nula, što se opet podudara s vrijednošću od a. Stoga zaključujemo da je, u slučaju gdje je $sel = 0$, rezultat operacije opisan gornjom formulom uvijek jednak a. Slično se može pokazati da je za $sel = 1$ konačni rezultat vrijednost od b.

Zanimljiva značajka HDL-a, koja se može vidjeti u multiplekser primjeru, su sabirnice r_1, r_2 i $nsel$. One prenose posredne rezultate između dijelova multiplekser čipa te se mogu spojiti u ulaze drugih dijelova. Primjerice, r_1 i r_2 se koriste kao ulaz za Or čip. Dakle, unutarnje sabirnice čine glavninu komunikacije unutar čipa.

Sličan multiplekseru u svojoj upotrebi je demultiplekser čip [5] (DMux). On sadrži ulaze in i sel širine jedan bit i dva izlaze a i b širine jedan bit. Ako je sel postavljen na jedan, b se postavlja na vrijednost od in, a na nula. U suprotnom, ako je $sel = 0$, a se postavlja na vrijednost od in, a b na nula. Demultiplekser čip se uglavnom koristi za usmjeravanje protoka podataka u određeni čip u kolekciji čipova.

Budući da će procesor raditi sa 16-bitnim cijelim brojevima, sada ćemo opisati čip koji vrši operaciju zbrajanja nad 16-bitnim brojevima. U našem modelu računala, brojevi će biti reprezentirani kao nizovi od 16 uzastopnih bitova, gdje je krajnje lijevi bit indeksiran s 15, a krajnje desni s nula. Primjerice, niz 0000000000001001 reprezentira broj 9. Kada ručno zbrajamo binarne brojeve, u slučaju prelijevanja koristimo prijenos u zbrajanju iduće znamenke. Ovakvo ponašanje postižemo koristeći čip zbrajalo, čija definicija slijedi

```
CHIP FullAdder
{
        IN a, b, c;
        OUT sum, carry;

        PARTS:
        HalfAdder(a = a, b = b, sum = s, carry = c1);
        HalfAdder(a = s, b = c, sum = sum, carry = c2);
        Or(a = c1, b = c2, out = carry);
}
```

Čip zbrajalo zbraja tri bita, te vraća jednobitni rezultat zbrajanja i potencijalni prijenosni bit. Konačno, koristeći čipove poluzbrajalo i zbrajalo, mo-

žemo definirati čip za zbrajanje 16-bitnih cijelih brojeva. Primijetimo da je širina ulaznih i izlaznih sabirnica jednaka 16.

```
CHIP Add16
{
        IN a[16], b[16];
        OUT out[16];

        PARTS:
        HalfAdder(a=a[0], b=b[0], sum=out[0], carry=c0);
        FullAdder(a=a[1], b=b[1], c=c0, sum=out[1], carry=c1);
        FullAdder(a=a[2], b=b[2], c=c1, sum=out[2], carry=c2);
        // ...
        FullAdder(a=a[14], b=b[14], c=c13, sum=out[14], carry=
            c14);
        FullAdder(a=a[15], b=b[15], c=c14, sum=out[15], carry=
            c15);
}
```

Ulazne sabirnice a i b predstavljaju dva 16-bitna cijela broja koje će čip zbrojiti te rezultat zbrajanja poslati putem izlazne sabirnice out. Krajnji lijevi bitovi svakoga broja, $a[0]$ i $b[0]$, se prosljeđuju čipu poluzbrajalo. Preostale operacije obavlja čip zbrajalo, koji prima prikladne bitove iz ulaza a i b i *carry* bit iz prethodne operacije (iz unutarnjih sabirnica c_i). Primijetimo da se posljednji bit za prijenos, c_{15} ne koristi te zbog toga čip ne uzima u obzir prelijevanje (engl. *overflow*). Slično kao i prethodni čip za zbrajanje 16-bitnih brojeva, možemo također definirati i 16-bitni multiplekser, demultiplekser, čipove za logičke operacije itd. U ostatku rada, pri konstrukciji složenijih čipova, koristit ćemo gotove implementacije čipova koje pruža HDL [5].

1.2 Aritmetičko-logička jedinica

Aritmetičko-logička jedinica (ALU) se nalazi u svakom modernom procesoru. Njezin je zadatak pružiti procesoru osnovne aritmetičke (zbrajanje, oduzimanje itd.) i logičke (konjunkcija, disjunkcija, negacija itd.) operacije. U računalu kojeg gradimo, aritmetičko-logička jedinica će pružati procesoru 18 osnovnih operacija nad dva 16-bitna ulazna podatka. Odabir funkcije koju će ALU izvršavati se provodi postavljanjem vrijednosti na 6 kontrolnih bitova (zx, nx, zy, ny, f i no) kao što je prikazano u tablici 1.

Tablica 1. Tablica rezultirajućih operacija dobivenih kombinacijom kontrolinih bitova ALU-a. Neka su 16-bitni ALU ulazi označeni s x i y te njegov izlaz s out.

zx	nx	zy	ny	f	no	out
if zx then $x = 0$	if nx then $x = \neg x$	if zy then $y = 0$	if ny then $y =!y$	if f then $out = x + y$ else $out = x \wedge y$	if no then $out = \neg out$	
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	$\neg x$
1	1	0	0	0	1	$\neg y$
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	$x \wedge y$
0	1	0	1	0	1	$x \vee y$

Njezin izlaz je 16-bitna sabirnica *out* koja prosljeđuje rezultat operacije, i dva kontrolna bita *zr* (nula) i *ng* (negativ), koje redom ukazuju je li rezultat operacije nula ili negativan broj. Primijetimo da naš model računala reprezentira negativne brojeve koristeći dvojni komplement [4], što znači da je krajnje lijevi bit svim negativnim brojevima jednak jedan. Slijedi implementacija ALU.

```
CHIP ALU
{
  IN
    x[16], y[16], // 16-bitni ulaz
    zx, // postavi x na nula?
    nx, // negiraj ulaz x?
    zy, // zero the y input?
    ny, // postavi y na nula?
    f, // izracunaj out = x + y (ako je f 1) ili out = x & y (
        ako je f 0)
    no; // negiraj output?
```

```
    OUT
      out[16], // 16-bitni output
      zr, // 1 ako (out == 0), 0 inace
      ng; // 1 ako (out < 0), 0 inace

    PARTS:

          // Procesuiranje zx i zy
          Mux16(a=x, b=false, sel=zx, out=zerox);
          Mux16(a=y, b=false, sel=zy, out=zeroy);

          // Procesuiranje nx i ny
          Not16(in=zerox, out=notx);
          Not16(in=zeroy, out=noty);

          // Odabir ulaza nakon transformacije
          Mux16(a=zerox, b=notx, sel=nx, out=finalx);
          Mux16(a=zeroy, b=noty, sel=ny, out=finaly);

          // Racunanje x&y i x+y
          Add16(a=finalx, b=finaly, out=addxy);
          And16(a=finalx, b=finaly, out=andxy);

          // Procesuiranje f.
          Mux16(a=andxy, b=addxy, sel=f, out=op);

          // Procesuiranje no.
          Not16(in=op, out=notop);
          Mux16(a=op, b=notop, sel=no, out[0..7]=res1, out
              [8..14]=res2, out[15]=res3);

          // Jeli rezultat 0?
          Or8Way(in=res1, out=or1);
          Or8Way(in[0..6]=res2, in[7]=res3, out=or2);
          Or(a=or1, b=or2, out=or);

          // Postavljanje izlaza
          Or16(a[0..7]=res1, a[8..14]=res2, a[15]=res3, b=false,
              out=out);

          // Postavljanje zr
          Not(in=or, out=zr);

          // Postavljanje ng
          Or(a=res3, b=false, out=ng);
}
```

Prvo se izračunavaju vrijednosti kojima upravljaju *zx*, *nx*, *zy* i *ny* te se redom prosljeđuju unutarnjim sabirnicama *zerox*, *notx*, *zeroy*, *noty*. Primijetimo da smo u implementaciji koristili konstantu *false* koja predstavlja 16-bitnu nulu. Koristeći dva multipleksera, u ovisnosti o vrijednostima

na kontrolnim bitovima, odabiremo konačnu formu ulaznih vrijednosti. Ulazne vrijednosti se ili neće mijenjati, ili će biti postavljene na nulu, ili će biti logički negirane. Nakon toga računamo vrijednosti $x + y$ i $x \wedge y$ i odabiremo jednu od njih koristeći multiplekser čiji je *sel* postavljen na vrijednost od f. Za razliku od standardne prakse u grananju programa, na hardveru (i u HDL-u), prvo se izvode obje grane pa se zatim odabire jedna od njih. Primjerice, hardverska implementacija operacije

$$\text{ako } a == 0 \text{ onda } x + y \text{ inače } x - y,$$

se provodi izračunavanjem i $x + y$ i $x - y$ te se nakon toga, u ovisnosti o a, odabire jedan od ta dva rezultata. Zatim se rezultat, u našoj implementaciji ALU, negira u ovisnosti o bitu *no* te se zatim šalje putem sabirnica res_1, $res2$ i res_3. Ako je res_3, koji predstavlja krajnje desni bit rezultata, jedan, onda je po definiciji dvojnoga komplementa rezultat negativan broj te se vrijednost na izlazu *ng* prikladno mijenja. Ako su svi bitovi jednaki nuli, što provjeravamo koristeći čipove Or i Or8Way, vrijednost na *zr* postavljamo na jedan. Konačno, sabirnice res_1, $res2$ i res_3 se ponovno spajaju i šalju na izlaznu sabirnicu *out*.

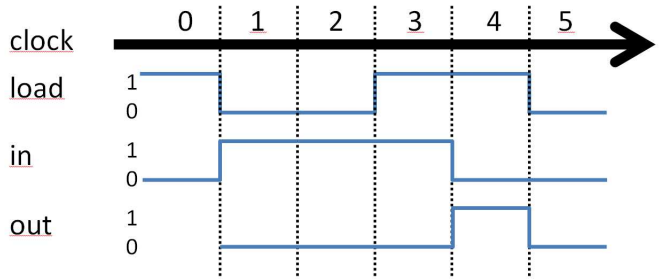
2 Memorija i registri

U harvardskoj računalnoj arhitekturi memorija se pojavljuje u tri različita oblika. Prvi je podatkovna memorija, ekvivalent RAM memoriji (engl. *random access memory*) u modernim računalima. Druga je instrukcijska memorija (engl. *read only memory*, ROM) u kojoj su zapisani programi. Zadnji tip memorije su registri koje procesor koristi prilikom izvršavanja instrukcija. Ona je fundamentalno drugačija od svih do sada implementiranih čipova (čipovi s kombinacijskom logikom) jer je ovisna o vremenu (čipovi sa sekvencijalnom logikom). Outputi ALU-a, koji je dobar primjer čipa s kombinacijskom logikom, ovise jedino o njegovim trenutnim inputima. Suprotno tome, vrijednosti kojima barataju memorijski čipovi ovise o stanju memorijskog čipa u prethodnom diskretnom vremenskom intervalu (hardverski sat). Memorijske ćemo čipove graditi koristeći se gotovim čipom zvanim bistabil (engl. *data flip-flop*, DFF). DFF čip ima jedan jedno-bitni input *in* i output *out*. On vraća input koji je primio tijekom prethodnog vremenskog intervala.

Započet ćemo konstruiranjem jedno-bitnog registra Bit čija je implementacija opisana u tekstu koji slijedi. Sastoji se od jednog DFF čipa sa svojim

inputom *in* i outputom *out* koji je poslan dvama sabirnicama *feedback* i *out* (izlaz iz čipa Bit). Napominjemo kako te dvije sabirnice prenose istu informaciju. Sabirnica *feedback* spojena je na input multipleksera zajedno s inputom *in* čipa Bit. Ako je ulazna sabirnica *load* čipa Bit postavljena na jedan, vrijednost sabirnice *in* šaljemo u DFF. U suprotnom, njegovo prethodno stanje (sadržano u sabirnici *feedback*) je zadržano. Dakle, dokle god je *load* postavljen na nulu, Bit zadržava svoju posljednje učitanu vrijednost. Ovo ponašanje, ilustrirano primjerom na slici 1, je sinkronizirano pomoću hardverskog sata tako da se samo jedna operacija učitavanja ili zadržavanja vrijednosti obavi u svakom od diskretnih vremenskih intervala. U primjeru su tijekom trećeg vremenskog intervala *load* i *in* postavljeni na jedan, što se reflektira na *out*-u tijekom sljedećeg intervala.

```
CHIP Bit
{
    IN in, load;
    OUT out;

    PARTS:
    Mux (a = feedback, b = in, sel = load, out = mout);
    DFF (in = mout, out = feedback, out = out);
}
```



Slika 1. Propagacija signala unutar jedno-bitnog registra tijekom nekoliko diskretnih vremenskih intervala, slika zasnovana na [2]

Nizajući jedno-bitne registre, moguće je implementirati 16-bitni registar. Slijedi HDL implementacija.

```
CHIP Register
{
    IN in[16], load;
    OUT out[16];

    PARTS:
    Bit (in = in[0], load = load, out = out[0]);
    Bit (in = in[1], load = load, out = out[1]);
    Bit (in = in[2], load = load, out = out[2]);
    //  ...
    Bit (in = in[15], load = load, out = out[15]);
}
```

Ovu ćemo sekciju zaključiti implementacijom RAM čipa koji sadrži 8 registara. Imat će tri ulazne sabirnice: 16-bitnu podatkovnu sabirnicu *in*, jedno-bitnu sabirnicu *load* te 3-bitnu sabirnicu *address*. Tri adresna bita odgovarat će adresama registara (000 je prvi registar, 001 drugi, ..., 111 zadnji). Pomoću bita *load* ćemo odlučivati hoćemo li u registar unutar RAM-a na adresi određenoj u sabirnici *address* upisivati vrijednosti (iz sabirnice *in*) ili ih iz njega čitati. Pristup registru na temelju njegove adrese ćemo postići pomoću gotove varijante demultiplekser čipa zvane DMux8Way. Kada smo u registar upisali vrijednost ili je iz njega pročitali, njegovo trenutno stanje je poslano na sabirnicu *out* pomoću gotovog multipleksera Mux8Way16. DMux8Way ima jedan podatkovni input i 3-bitni *select* input koji nam koristi kako bismo input poslali na jednu od njegovih 8 izlaznih sabirnica. Mux8Way16 sadrži 8 ulaznih 16-bitnih sabirnica, te njegov *sel* funkcionira na način ekvivalentan onom čipa DMux8Way. Slijedi implementacija RAM čipa.

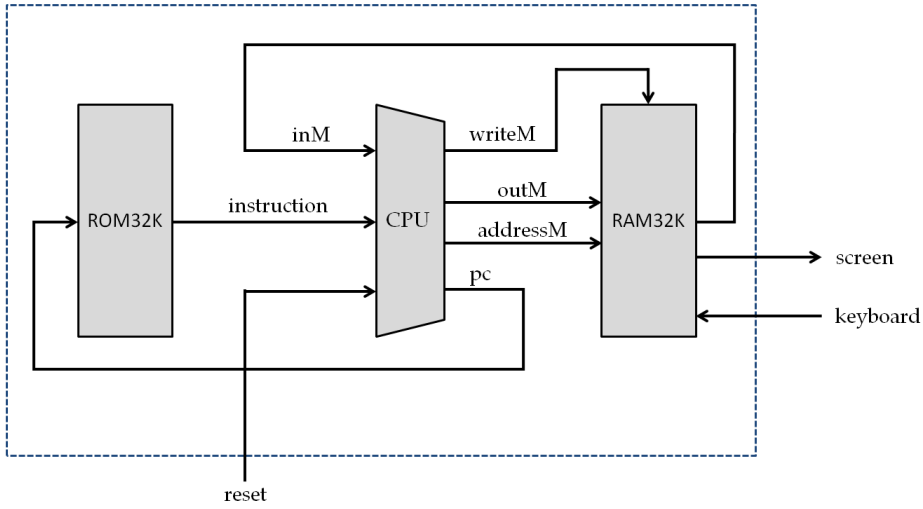
```
CHIP RAM8
{
    IN in[16], load, address[3];
    OUT out[16];

    PARTS:
    DMux8Way(in = load, sel = address,
             a = r0, b = r1, c = r2, d = r3,
             e = r4, f = r5, g = r6, h = r7);

    Register(in = in, load = r0, out = o0);
    Register(in = in, load = r1, out = o1);
    Register(in = in, load = r2, out = o2);
    Register(in = in, load = r3, out = o3);
    Register(in = in, load = r4, out = o4);
    Register(in = in, load = r5, out = o5);
    Register(in = in, load = r6, out = o6);
    Register(in = in, load = r7, out = o7);
```

```
    Mux8Way16(a = o0, b = o1, c = o2, d = o3,
              e = o4, f = o5, g = o6, h = o7,
              sel = address, out = out);
}
```

RAM memoriju većeg kapaciteta lagano je implementirati nizanjem manjih RAM čipova i proširivanjem adresne sabirnice.

3 Središnja jedinica za obradu

HACK računalo opisano u [2] je jednostavan stroj zasnovan na Harvard arhitekturi koji se sastoji od tri glavna dijela: procesor, RAM (podatkovna memorija) i ROM (instrukcijska memorija). Njegovu shemu je moguće vidjeti na slici 2.



Slika 2. Model HACK računala, slika zasnovana na [2]

Memorijski čipovi, RAM i ROM, su implementirani koristeći postupak opisan u prethodnoj sekciji i svaki sadrži 32 kilobajta memorije. Međusobno su povezani nizom unutarnjih sabirnica. Vanjski input i output je serijaliziran kroz sabirnice koje primaju input s tipkovnice i šalju output na zaslon. Programi napisani za HACK računalo upisani u ROM čipove, počinju sa

svojim izvršavanjem kada je poslan signal sabirnicom *reset* prema procesoru.
Uobičajeno je da se procesori dizajniraju imajući u vidu dobro definiran strojni jezik. Strojni jezik procesora, koji ćemo opisati, će imati dva tipa instrukcija: A-instrukcije i C-instrukcije. Ideja A-instrukcije je adresiranje specifičnog registra u podatkovnoj ili instrukcijskoj memoriji. Možemo ju predstaviti linijom koda

$$@value,$$

gdje je vrijednost *value* proizvoljna memorijska adresa. Fizički, procesor sadrži registar, zvan A-registar (adresni registar), u koji će vrijednost *value* biti upisana nakon izvršenja A-instrukcije. Kako bismo osigurali njegov ispravan rad, procesor će sadržavati još jedan registar, zvan D-registar (podatkovni registar). U njemu će procesor spremati vrijednosti potrebne za izvršavanje trenutne instrukcije. D-registar koristi većina C-instrukcija čija je općenita forma

$$dest = comp\ ;\ jump,$$

gdje je *dest* odredište u koje spremamo rezultat izračuna (A-registar, D-registar ili RAM), *comp* predstavlja jednu od već implementiranih ALU funkcija (popisanih u Tablici 1), a *jump* određuje sljedeću instrukciju koja će biti dohvaćena iz instrukcijske memorije. I A-instrukcije i C-instrukcije, koje su učitane iz ROM-a, predstavljene su pomoću nizova duljine 16-bita. Svaka A-instrukcija počinje s bitom nula na indeksu 15, dok svaka C-instrukcija počinje jedinicom.

Prvu nulu A-instrukcije slijedi 15-bitna RAM ili ROM adresa. Primijetimo kako je 15 bitova dovoljno kako bismo adresirali bilo koji registar unutar 32 kilobajta memorije. Za razliku od prilično jednostavne A-instrukcije, C-instrukcija je po svojoj definiciji ponešto kompliciranija. Njezina struktura je

$$1\ 1\ 1\ a\ zx\ nx\ zy\ ny\ f\ no\ d_1\ d_2\ d_3\ j_1\ j_2\ j_3.$$

Prva tri bita C-instrukcije su uvijek jedinice. Bitovi zx, nx, zy, ny, f i no odgovaraju kontrolnim bitovima na inputu u ALU, čiji je input x D-registar, a input y ili A-registar ili registar u RAM-u na adresi upisanoj u A-registru ($RAM[A]$). Izbor A ili $RAM[A]$ obavlja se na temelju a bita. Bit d određuje gdje će biti spremljen rezultat operacije koju je izvršio ALU: A-registar, D-registar i $RAM[A]$. Na temelju zr i ng outputa s ALU-a, bitovima j specificiramo hoće li sljedeća instrukcija biti $ROM[A]$ (skok). Bit j_1 inicira skok ako je rezultat operacije negativan, bit j_2 ako je rezultat operacije nula, te bit j_3 ako je on strogo pozitivan. Primijetimo da postavljanjem sva tri j bita

na nulu, skok neće biti iniciran, dok postavljanje istih bitova rezultira bezuvjetnim skokom.

Kako bismo bili u stanju dohvatiti instrukciju iz ROM-a, implementirat ćemo specijalizirani čip zvan programski brojač. On sadrži jedan 16-bitni input *in* te output *out* koji prosljeđuje instrukciju iz ROM-a do procesora. Procesor specificira koja mu je instrukcija potrebna koristeći se trima jednobitnim sabirnicama koje su spojene s ulazom u programski brojač pod nazivom *reset*, *load* i *inc*. Te smo sabirnice izlistali prema njihovom prioritetu. Ako je *reset* postavljen na jedan, iz početka se pokreće izvršavanje programa i dohvaća se prva instrukcija zapisana u ROM. Ako je *load* postavljen na jedan, instrukcija čija se adresa trenutno nalazi u A-registru je dohvaćena. U slučaju da je *inc* postavljen na jedan, programski brojač inicira dohvaćanje sljedeće po redu instrukcije iz ROM-a. Slijedi HDL implementacija programskog brojača.

```
CHIP PC
{
        IN in[16], load, inc, reset;
        OUT out[16];

        PARTS:
        Inc16(in = reg, out = outA);
        Mux16(a = reg, b = outA, sel = inc, out = outB);
        Mux16(a = outB, b = in, sel = load, out = outC);
        Mux16(a = outC, b = false, sel = reset, out = outD);
        Register (in = outD, load = true, out = reg, out = out)
            ;
}
```

Nakon završenog programskog brojača nastavljamo dizajn procesora. Njegova potpuna implementacija u jeziku HDL, dana na kraju sekcije, će biti postepeno objašnjena. Procesor ima dvije 16-bitne sabirnice na ulazu: *inM* i *instruction*. Koristit ćemo ih kako bismo učitali podatke iz RAM-a i instrukcije iz ROM-a. Osim već navedenih, procesor ima i ulaznu sabirnicu *reset* širine jednog bita. Output s procesora je 16-bitna sabirnica *outM* koja upisuje rezultat izvršavanja instrukcije u RAM-u ako je jedno-bitna sabirnica *writeM* postavljena na jedan. Sabirnice *addressM* i *pc* na izlazu sadrže adresu u RAM-u i ROM-u s kojih procesor čita vrijednosti ili, u slučaju RAM-a, na koju ih upisuje.

Kada primi instrukciju iz ROM-a, procesor treba odrediti kojeg je ona tipa. To je moguće uraditi pomoću dva Not čipa na sljedeći način.

```
Not(in = instruction[15], out = Ainstruction);
Not(in = Ainstruction, out = Cinstruction);
```

Sada unutrašnje sabirnice *Ainstruction* i *Cinstruction* sadrže informaciju o tipu instrukcije koje procesor treba izvršiti. Postavljamo A-registar.

```
And(a = Cinstruction, b = instruction[5], out = ALUtoA);
// C-instrukcija moze vrijednost spremiti u A-registar
Mux16(a = instruction, b = ALUout, sel = ALUtoA, out = Ain);
// Ucitavamo u A-registar?
Or(a = Ainstruction, b = ALUtoA, out = Aload);
ARegister(in = Ain, load = Aload, out = Aout);
// Saljemo A ili RAM[A] prema ALU-u
Mux16(a = Aout, b = inM, sel = instruction[12], out = AMout);
```

Svaka A-instrukcija i C-instrukcija za koju vrijedi $d_1 = 1$ učitava ili input iz *instruction* sabirnice ili output iz ALU-a u A-registar. Multiplekser odlučuje koja će informacija biti sačuvana. Ako se izvršava C-instrukcija te ovisno o a bitu, u ALU učitavamo ili vrijednost spremljenu u A ili vrijednost $RAM[A]$ putem $AMout$ sabirnice. Također, na temelju vrijednosti bita d_2, odlučujemo hoćemo li učitati vrijednost u D-registar.

```
And(a = Cinstruction, b = instruction[4], out = Dload);
DRegister(in = ALUout, load = Dload, out = Dout);
```

Nakon što je ALU završio izračun, njegov rezultat možemo poslati u RAM registar na adresu spremljenu u A-registru tako da output *writeM* postavimo na jedan ako je $d_3 = 1$.

```
Or16(a = false, b = Aout, out[0..14] = addressM);
Or16(a = false, b = ALUout, out = outM);
And(a = Cinstruction, b = instruction[3], out = writeM);
```

Na koncu, koristeći gotova logička vrata i elementarne logičke operacije, mogući skokovi su evaluirani i programski brojač prima adekvatnu instrukciju. Slijedi potpuna implementacija procesora.

```
CHIP CPU
{
    IN  inM[16],          // Input RAM[A]
        instruction[16], // Input ROM[A]
        reset;           // Ponovno pokrecemo program?
```

```
OUT outM[16],          // Output u RAM[A]
    writeM,            // Upisujemo vrijednost u RAM[A]?
    addressM[15],      // Adresa od RAM[A]
    pc[15];            // Adresa od ROM[A]

PARTS:
// A- ili C- instrukcija
Not(in = instruction[15], out = Ainstruction);
Not(in = Ainstruction, out = Cinstruction);

// A-registar
And(a = Cinstruction, b = instruction[5], out = ALUtoA);
Mux16(a = instruction, b = ALUout, sel = ALUtoA, out = Ain)
    ;
Or(a = Ainstruction, b = ALUtoA, out = Aload);
ARegister(in = Ain, load = Aload, out = Aout);
Mux16(a = Aout, b = inM, sel = instruction[12], out = AMout
    );

// D-registar
And(a = Cinstruction, b = instruction[4], out = Dload);
DRegister(in = ALUout, load = Dload, out = Dout);

ALU(x = Dout, y = AMout,
  zx = instruction[11], nx = instruction[10],
  zy = instruction[9], ny = instruction[8],
  f = instruction[7], no = instruction[6],
  out = ALUout, zr = ZRout, ng = NGout);

// Podatke saljemu u RAM[A], ako je potrebno
Or16(a = false, b = Aout, out[0..14] = addressM);
Or16(a = false, b = ALUout, out = outM);
And(a = Cinstruction, b = instruction[3], out = writeM);

// Programski brojac
// Output nula s ALU-a
And(a = ZRout, b = instruction[1], out = jeq);
// Negativan ALU output
And(a = NGout, b = instruction[2], out = jlt);
// Pozitivan ALU output
Or(a = NGout, b = ZRout, out = NGorZR);
Not(in = NGorZR, out = PSout);
And(a = PSout, b = instruction[0], out = jgt);
// Skok u slucaju C-instrukcije
Or(a = jlt, b = jeq, out = jle);
Or(a = jle, b = jgt, out = jmp);
And(a = Cinstruction, b = jmp, out = PCload);
// Ukoliko nema skoka, ucitaj iducu instrukciju
Not(in = PCload, out = PCinc);
PC(in = Aout, load = PCload, inc = PCinc, reset = reset,
    out[0..14] = pc);
```

```
}
```

Definicijom procesora smo uklonili posljednju prepreku na putu ka gotovoj implementaciji računala. Implementacijom A- i C-instrukcija, precizno smo definirali čitav strojni jezik našeg procesora. Programi pisani u strojnom jeziku HACK računala mogu se zapisati na ROM čipove koje je moguće umetnuti u računalo i, pritiskom na *reset*, početi s njihovim izvršavanjem. Autori [2] su dizajnirali pojednostavljenu varijantu asemblerskog jezika koji uvelike smanjuje potrebno vrijeme za pisanje programa za HACK računalo. Implementaciju parsera napisanu u programskom jeziku Python, koja kod napisan u HACK asembleru prevodi u naredbe strojnog jezika, moguće je pronaći u [3]. Simulacijski softver, razvijen od strane [2], pruža mogućnost simulacije HACK računala ili bilo kojeg drugog čipa napisanog u jeziku HDL na svakom računalu koji podržava JAVA platformu. Kasnije u svome radu, autori [2] koriste implementaciju virtualnog stroja koji radi na principu stogova kako bi razvili prevoditelj za objektno orijentirani programski jezik i operacijski sustav napisan za HACK računalo. U tome je radu demonstrirana snaga i svestranost jednostavne računalne arhitekture poput one korištene prilikom dizajna HACK računala.

4 Zaključak

U ovome smo radu izgradili model 16-bitnog računala te pokazali kako funkcionira harvardska arhitektura. Koristili smo se pri tom HDL-om, jezikom koji se koristi prilikom dizajna hardvera. Pomoću jednostavnih primjera smo pokazali način rada HDL-a. Potom smo izgradili čipove od kojih smo napravili računalo bazirano na harvardskoj arhitekturi. Među ostalom, dizajnirali smo nekolicinu jednostavnijih čipova od kojih su najvažniji aritmetičko-logička jedinica, memorijski registar, RAM i ROM memorija te procesor. Prilikom dizajna procesora smo vodili računa i o dizajnu strojnog jezika definiranog na temelju njegovih hardverskih značajki.

5 Literatura

[1] N. Kularatna, *Modern Component Families and Circuit Block Design*, Newnes, 2000, ISBN 9780750699921

[2] N. Nisan and S. Schocken, *The Elements of Computing Systems*, MIT press, 2008, ISBN 9780262640688

[3] L. Borozan, *Lecture notes from the course "Modern computer systems"*, Department of Mathematics, University of Osijek, 2020

[4] J. von Neumann, *First Draft of a Report on the EDVAC*, Moore School of Electrical Engineering, University of Pennsylvania, 1945

[5] M. Armbrust, *HDL Survival Guide*, `https://phoenix.goucher.edu/~kelliher/f2015/cs220/hdlSurvivalGuide.pdf`