Факултет техничких наука у Чачку

Универзитета у Крагујевцу

Имплементација Hack рачунара на FPGA платформи

дипломски рад

Врста студија: Основне академске

Назив студијског програма: Електротехничко и рачунарско инжењерство

Модул: Рачунарско инжењерство

Предмет: Организација рачунарских система

Име и презиме студента

Немања Богојевић 169/2018

Руководилац рада

др Урош Пешовић, доцент

Чачак, Септембар 2022.

Резиме

Аутор Немања Богојевић

Тема Имплементација Hack рачунара на FPGA платформи

Ментор Др Урош Пешовић

Представљено на Универзитет у Крагујевцу,

Факултет техничких наука у Чачку

Број страна 54

Овај дипломски рад има за циљ да упозна читаоце о томе како функционише један прост рачунарски систем. Да би смо довели рачунарски систем у реалан физички свет упознаћемо се са алатима који нам омогућавају да хардверске компоненте опишемо програмским путем помоћу **Verilog** језика и имплементирамо на **FPGA** систему.

Рад почиње са општим описом шта је рачунар и од којих свих компоненти се он састоји. Затим говоримо о апстракцији рачунара где се може сагледати сви слојеви једног рачунарског система.

Након тога упознаћемо се са **Hack** рачунаром. Овај рачунарски систем представља један теоретски опис рачунара где може се видети како од једног простог кола долазимо до функционалног рачунарског система. Сагледавају се све компоненте које чине један рачунар **CPU**, **ROM**, **RAM**. Затим се могу видети инструкције који тип инструкција и на који начин оне контролишу рад рачунара. Затим сагледавамо све програмске нивое од којих се рачунар састоји, како се код високог нивоа написан у JACK језику преводи у машинске инструкције.

У последњем поглављу описани је начин на који је Hack рачунар имплементиран на FPGA систем. Ту се упознајемо са проблемима које нисмо могли видети приликом практичног описа рачунара. Упознајемо се како се користи Quartus програмско окружење за FPGA систем. Такође читаоци могу се упознати са Verilog програмским језиком за опис хардвера и на који начин се реализују све компоненте Hack рачунара у овом језику. Сагледава се како функционише VGA стандард за приказивање слике и које компоненте је потребно реализовати да би омогућили рачунару да има приказ слике на монитору.

Кључне речи: FPGA, Hack рачунар, Verilog, JACK

Садржај

1. Увод

Рачунарски системи, односно рачунари, представљају електронске уређаје које обрађују улазне податке и генеришу неку излазну информацију за корисника. У данашње време, рачунари имају велику примену у свим областима науке и друштва. Рачунари су најзначајније откриће 20. века, јер су нам омогућили аутоматизацију и контролу разних процеса. Рачунари се састоје из два основна дела: хардвера и софтвера. Хардвер преставља физички део рачунара који је видљив голим оком и њега чине процесор, меморија и улазно/излазни уређаји. Сви ови делови су смештени у кућиште рачунара које садржи напајање. Софтвер представља имагинарни део рачунарског система који није видљив голим оком. Он управља свим хардверским компонентама и за неки улазне акције генерише одговарајући излаз. Да би смо управљали нашим рачунаром морамо му задамо сет инструкција које ће он извршити односно морамо да напишемо одговарајући програм за његову контролу. Из ове области рачунара произилази данас популаран појам Програмирање које нам је омогућило да реализујемо претходно дефинисане задатке

У овом раду бавићемо се развојем HACK рачунара ,помоћу програма за опис хардвера, је у могућности да извршава основне асемблерске инструкције. У данашње време могуће је хардвер рачунара описати програмским језиком и тај наш опис можемо применити на FPGA плочу. Уз помоћ ове плоче наш хардвер може из програмског описа постати реалан хардвер који је у могућности да буде тестиран и коришћен за реализацију рачунарског система.

1.1Апстракција рачунарског система

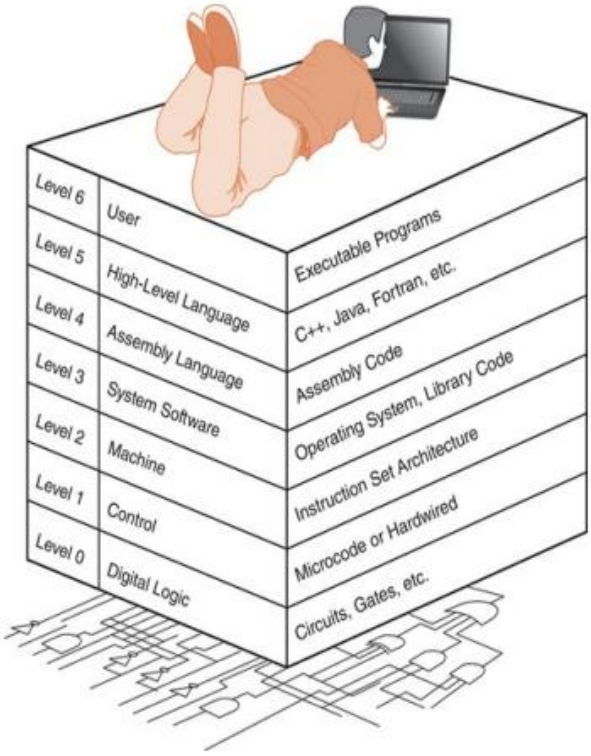
Као и сваки комплексан систем, рачунар се може посматрати на највишем нивоу са корисничког аспекта или на најнижем транзисторском нивоу. Сваки ниво посматрања представља одређену апстракцију рачунара. Ове апстракције су независне и дозвољавају различите нивое приступа. Тако да човек који уноси текст у едитору не мора ништа да зна о програмирању, слично томе програмер не мора ништа да знати о логичким структурама унутар рачунара. Унутар рачунарског система дефинисано је седам нивоа од корисничког до транзисторског ниво. Како се крећемо од врха према дну нивои постају све мање апстрактни и приказује се све више интерна структура рачунара.

1.1.1 Кориснички ниво

Кориснички ниво састоји се од апликација и представља ниво са којим смо сви упознати данас. На овом нивоу покрећемо програме као што су видео игре, програми за обраду текста, програми за приказ видео снимака и слично. Нижи нивои су скоро па невидљиви за корисника.

1.1.2 Програмски језици вишег нивоа

Овај ниво састоји се од програмских језика као што су C,C++, Java, Python, C# и слично. Ови језици морају бити преведени у језик који ће рачунар разумети. Језици који се компајлирају преводе се у асемблерски језик, а затим из асемблерског у машински језик. Корисник на овом нивоу види веома мало шта се дешава на нижим нивоима , иако програмер мора да зна о типовима података и инструкцијама које су доступне за тај тип података, али корисник не мора да зна како су ови типови имплементирани.



Слика 1. Апстракција рачунарског система.

1.1.3 Асемблерски језик

Овај ниво обухвата неку врсту асемблерског језика, који се директно преводи у машински језик. Као што смо малопре поменули, компајлерски језици вишег нивоа се прво преводе у асемблерски језик, који се затим директно преводи у машински језик. Ово превођење је у односу један према један, што значи да једна асемблерска инструкција се преводи у тачно у једну машинску инструкцију. Користећи одвојене нивое, смањујемо семантички јаз између програма високог нивоа и машинског језика.

1.1.4 Системски софтвер

Ниво системског софтвера одговоран је за управљањем системских инструкција. Овај ниво се такође бави мултипрограмирањем, заштитом меморије, синхронизацијом процеса и доста обаља доста других битних функционалности, Често инструкције које се преводе из асемблерског језика у машински пролазе кроз овај ниво.

1.1.5 Машински ниво

Архитектура сета инструкција (ISA) или машински ниво, састоји се од машинског језика који се препознаје од стране архитектуре рачунарског система. Програми који су написани у правом машинском језику од рачунара, могу да се извршавају директно од стране електричних кола без коришћења интерпретера, преводиоца или компајлера.

1.1.6 Контролни ниво

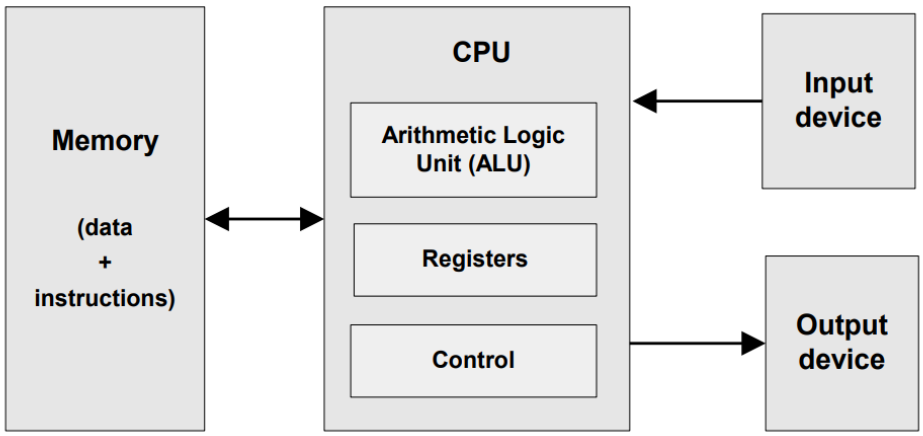
На овом нивоу контролна јединица има задатак да се инструкције декодирају и извршавају као што је то предвиђено и да се подаци премештају када и на које место је то потребно. Контролна јединица тумачи машинске инструкције које јој се прослеђују, један по једна, са виших нивоа, што узрокује да се одговарајуће акције спроведу. Контролне јединице могу да се реализују на два начина: Могу бити реализовани на самој плочи као хардвер или могу бити микроиспрограмиране. Ако су реализоване као хардвер они директно шаљу сигнале одговарајућим компонентама и у обичајно су знатно бржи, али недостатак је када се имплементирају тешко је извршити њихову модификацију. Микропрограм је програм који је написан у ником програмском језику који је имплементиран директно од стране хардвера. Машинске инструкције се шаљу у микропрограм који тумачи инструкције и активира одговарајуће хардверске компонента да изврше оригиналну инструкцију. Микропрограми су доста популарни јер могу лако да се модификују, али недостатак је што додавањем још једног слоја превођења доводи до споријег извршавања инструкција.

1.1.7 Ниво логичких кола

На нивоу логичких кола може се пронаћи физичке компоненте од рачунарског система (жице и логичка кола). Ове компоненте представљају основне градивне блокове и имплементирају Булову логику која је заједничка за све рачунарске системе.

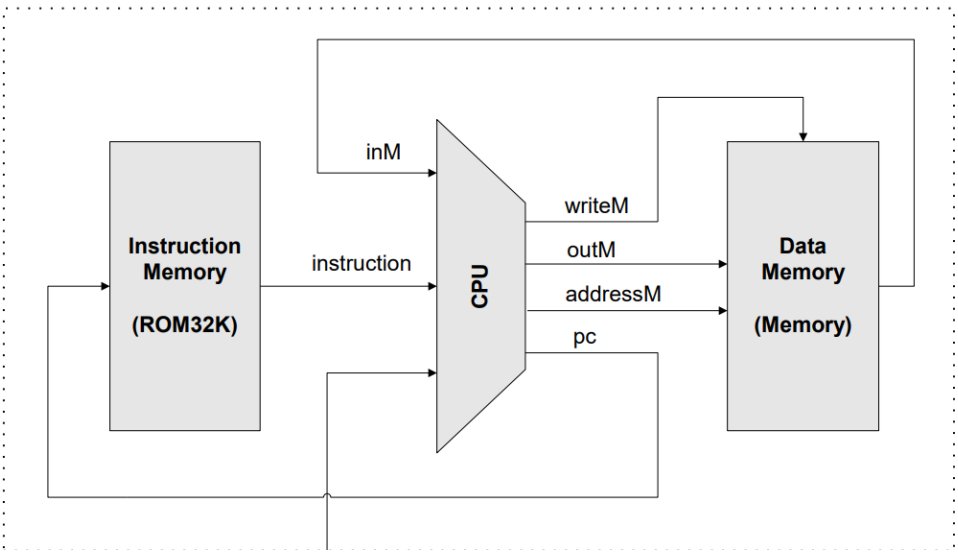
2. Теоретски опис Hack рачунарског система

Како би реализовали један прост рачунарски систем који ће да извршава основне асемблерске наредбе, потребно је приказати како изгледа архитектура рачунарског система. Најпростија архитектура рачунара је фон Нојманова архитектура која се састоји од меморијске јединице за чување инструкција и података која је одвојена од централне процесорске јединице са улазим и излазим јединицама из процесора.



Слика 2. Фон Нојманова архитектура рачунара

Осим ове структуре приказане на слици 2, постоји још једна модификована Харвард архитектура, код које је меморијски простор подељен на два дела. Један део у коме се смештају инструкције ROM меморија и други део меморије за складиштење података односно програмска меморија, као што је приказано на следећој слици

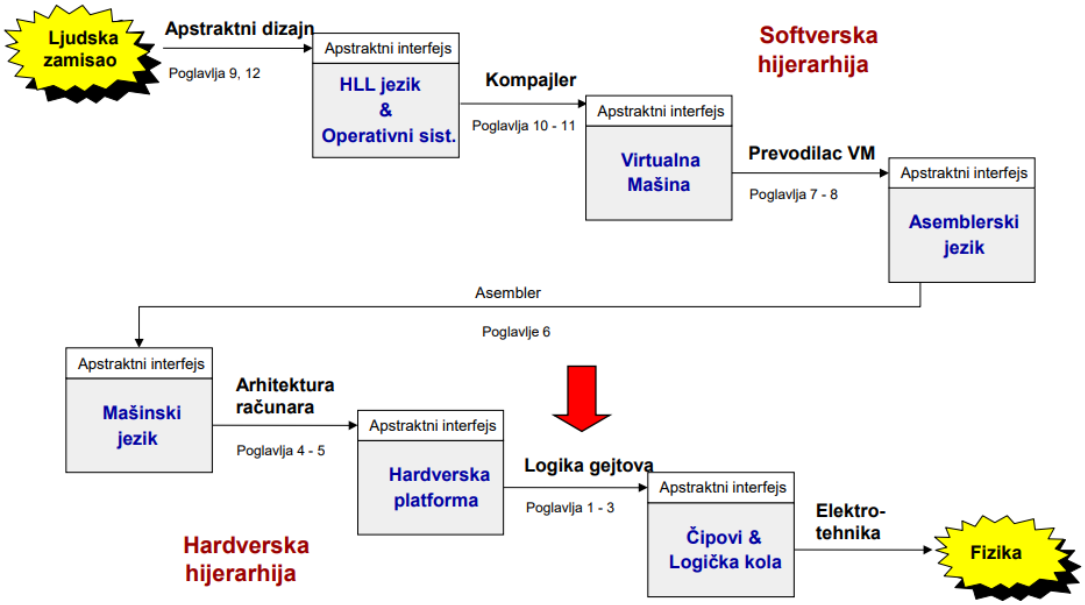


Слика 3. Харвард архитектура са подељеним меморијским простором

Рачунарски систем приказан у овом дипломском раду је дизајниран по принципу Харвард архитектуре и назива се Hack рачунар.

2.1Hack рачунар

Hack рачунарски систем представља један теоретски рачунарски систем који се обрађује на курсу *NandToTetris* и на предмету организација рачунарских система. Унутар овог курса сагледава се цео аспект једног рачунарског система. Полазећи од једног основног *nand* логичког кола и пролазе се сви аспекти хардвера и софтвера, као што је приказано на слици 4.



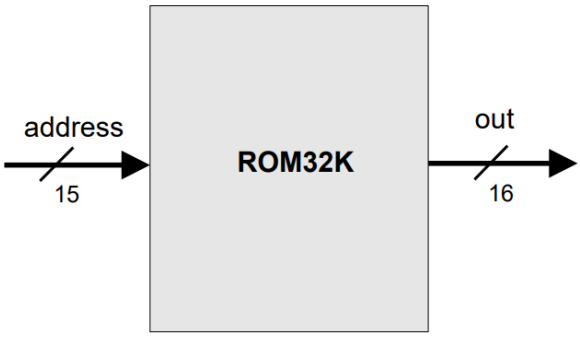
Слика 4. Опис курса

Hack рачунар има следеће карактеристике:

- Меморија за инструкције(ROM) и програме(RAM) је одвојена
- Излазна јединица Екран од 512 врста са 256 колона
- Улазна јединица Тастатура PS2
- Извршава програме који су написани у Hack машинском језику
- RAM меморија је подељена на три сегмента
  - Меморија за податке
  - Меморија за екран
  - Меморија за тастатуру
- Процесор за извршавање инструкција (CPU)

2.1.1 Програмска ROM меморија

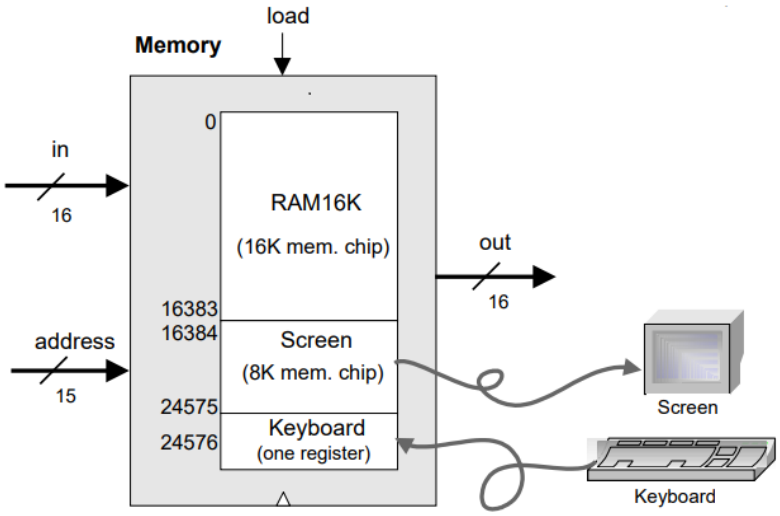
Програмска ROM (Read Only Memory) меморија је трајна меморија која чува свој садржај и приликом нестанка напајања. Унутар ове меморије обично се чувају системски подаци неопходни за рад рачунара. У архитектури Hack рачунара, ROM меморија се користи за складиштење програма написаног у Hack машинском језику. На излазу из ове меморије добијамо 16-бинтни број који представља једну машинску инструкцију, а на улазу у меморију се позива адреса на којој се налази машинска инструкција. ROM меморија се састоји од 32K адресабилних 16-битних локација.



Слика 5. Програмска меморија Hack рачунара

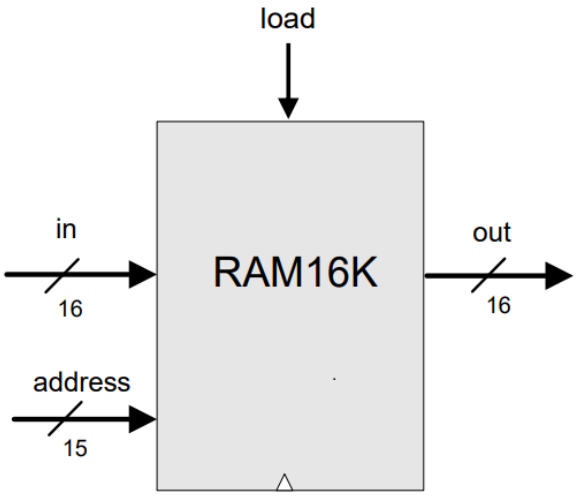
2.1.2 Меморија података

За разлику од ROM memorije, RAM (Random Access Memory) представља меморијски елемент који се користи за привремено чување података. Ова меморија после нестанка напајања губи свој садржај. Због оваквих особина ове меморије се обично користе за чување привремених променљивих направљених од стране програма који се извршава на рачунару. Унутар Hack рачунара меморија података је реализована као RAM меморија са 32K адресабилних 16-битних локација. Како би се омогућило повезивање додатних улазни/излазних уређаја, меморија података је подељена на три дела, односно целокупна меморија се састоји од три засебна меморијска сегмента.



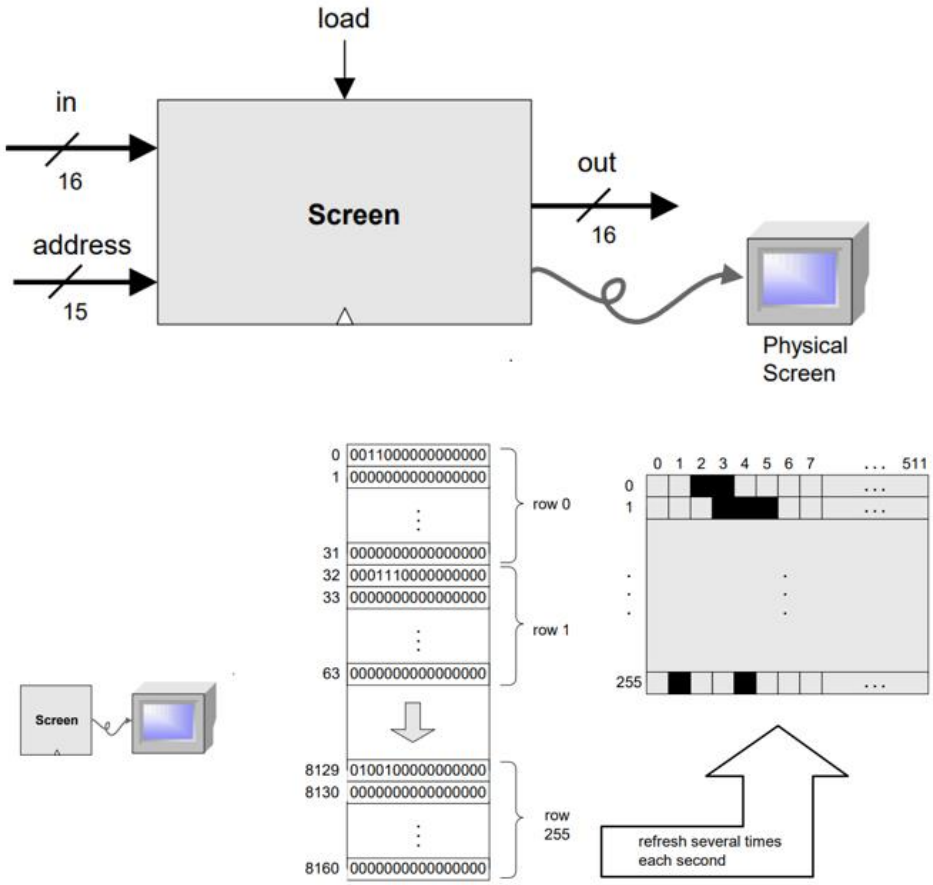
Слика 6. Расподела меморијског простора Hack рачунара

Први део меморије је класична RAM меморија која се користи за чување програмских података. Она се састоји од 16K адресабилних 16-битних адреса. Ова меморија сачињена је од in улаза која садржи податак који се уписује унутар меморије и address улаза који садржи меморијску локацију у којој се уписује/чита податак. Излаз out чита податак са одговарајуће меморијске локације.



Слика 7. RAM меморија Hack рачунара

Други део меморијског простора је резервисан за екран. Екран унутар Hack рачунара има резолуцију 512x216 пиксела. Сваки бит унутар меморијског простора представља један пиксел на екрану. Екран може само да прикаже две боје у једном тренутку, у овом раду користили смо црвену као позадину и белу за цртање по екрану, Ова RAM меморија се састоји од 8K адресабилних локације ширине 16-бита. Сваки бит унутар 16-битне локације одговара једном пикселу. Ако је бит на логичкој јединици, пиксел коме он одговара забиће осветљен белом бојом, док битови који су на логичкој нули приказују црвену боју.



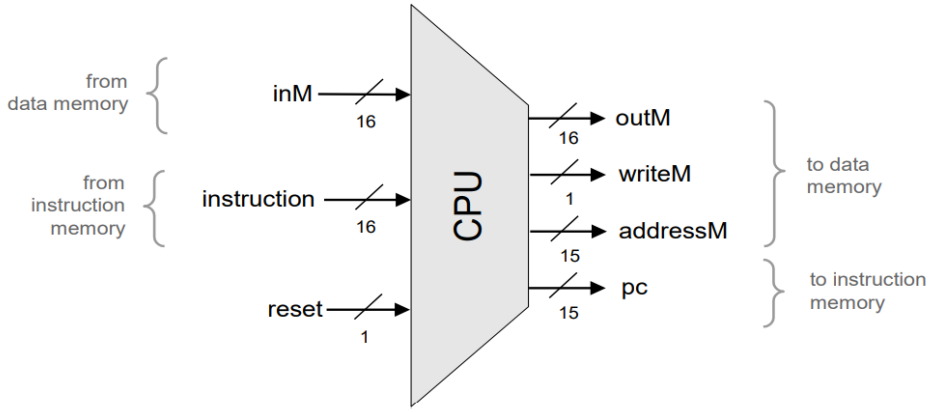
Слика 8. Структура меморијског сегмента екрана

Трећи део меморијског простора резервисан је за тастатуру. Меморијски чип за тастатуре се састоји само од једне 16-битне локације. Принцип рада је једноставан, унутар 16-битног регистра уписује се одговарајућа 16-битна вредност која одговара притиснутом тастеру. Када ниједан тастер није притиснут сви битови треба да буду на логичкој нули.

Слика 9. Структура меморијског сегмента тастатуре

2.1.3 Централни процесор

Централни процесор, скраћено CPU (Central Processing Unit) је основна јединица сваког рачунара. Процесор је интегрисано коло и у њему се реализују све рачунске и логичке операције и извршавају инструкције које су му задате од стране програма. У Hack рачунару CPU је реализован са три улаза и четири излаза. Први улаз процесора CPU је InM он има задатак да пренесе учитане податке из data меморије у процесор. Други улаз instruction преноси машинске инструкције из rom меморије у процесор. Последњи улаз reset користи се за ресетовање унутрашњих компоненти. Излаз OutM има функцију да 16-битни податак упише унутар data меморије, writeM излаз је контролни сигнал који одобрава или не одобрава упис података унутар data меморије, adressM прослеђује адресу на којој податак се уписује и PC излаз представља адресу на којој се налази следећа инструкција.



Слика 10. Централни процесор Hack рачунара

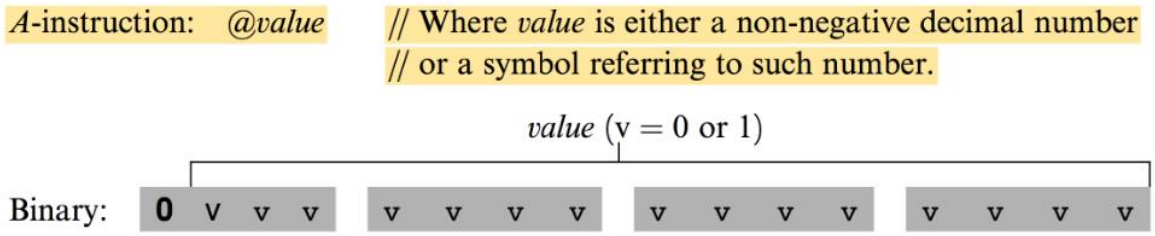
2.2Hack машинске инструкције

Као што смо у претходним поглављима нагласили, основни задатак рачунара је да извршава машинске инструкције које му се прослеђују. У Hack рачунару постоје две врсте инструкција А инструкције и С инструкције.

2.2.1 A тип инструкција

A инструкције се користе да вредност A регистра, који се налази унутар процесора, постави на неку 15-битну вредност. Бинарна вредност A инструкције састоји се из два дела. Први највећи бит који се налази са леве стране представља код операције, a ostalih 15 битова користе се за иницијализацију не негативног децималног броја. U Hack рачунару A инструкција има функцију:

a) Уноса константе вредности
b) Одабир локације унутар RAM меморије
c) Одабир локације унутар ROM меморије



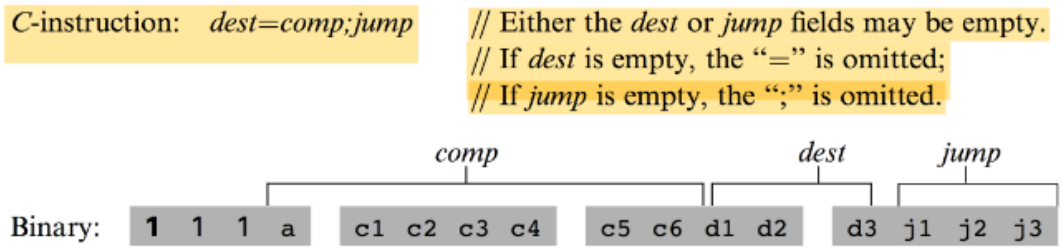
Слика 11. Структура A инструкције

2.2.2 C тип инструкција

C инструкција има знатно више функционалности које обавља за разлику од A инструкције. Ова инструкција решава три питања :

1) Шта је потребно да се израчуна(**comp**)
2) Где сместити израчунату вредност?(**dest**)
3) Шта је следеће потребно да се уради?(**jump**)

Први највећи бит са леве стране у овој инструкцији је такође код операције који се поставља на логичку јединицу. Друга два бита се не користе и по конвенцији постављају се на 1. Следећих седам бита представљају comp поље и следећа три представљају поље за одредиште dest и последња три бита дефинишу услове за скок jump поље.



Слика 12. Структура C инструкције

(when a=0) *comp*	c1	c2	c3	c4	c5	c6	(when a=1) *comp*
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

d1	d2	d3	*Mnemonic*	*Destination (where to store the computed value)*
0	0	0	null	The value is not stored anywhere
0	0	1	M	Memory[A] (memory register addressed by A)
0	1	0	D	D register
0	1	1	MD	Memory[A] and D register
1	0	0	A	A register
1	0	1	AM	A register and Memory[A]
1	1	0	AD	A register and D register
1	1	1	AMD	A register, Memory[A], and D register

j1 (*out* < 0)	j2 (*out* = 0)	j3 (*out* > 0)	Mnemonic	Effect
0	0	0	null	No jump
0	0	1	JGT	If *out* > 0 jump
0	1	0	JEQ	If *out* = 0 jump
0	1	1	JGE	If *out* ≥ 0 jump
1	0	0	JLT	If *out* < 0 jump
1	0	1	JNE	If *out* ≠ 0 jump
1	1	0	JLE	If *out* ≤ 0 jump
1	1	1	JMP	Jump

Слика 13. Табеле кодирања поља C инструкције

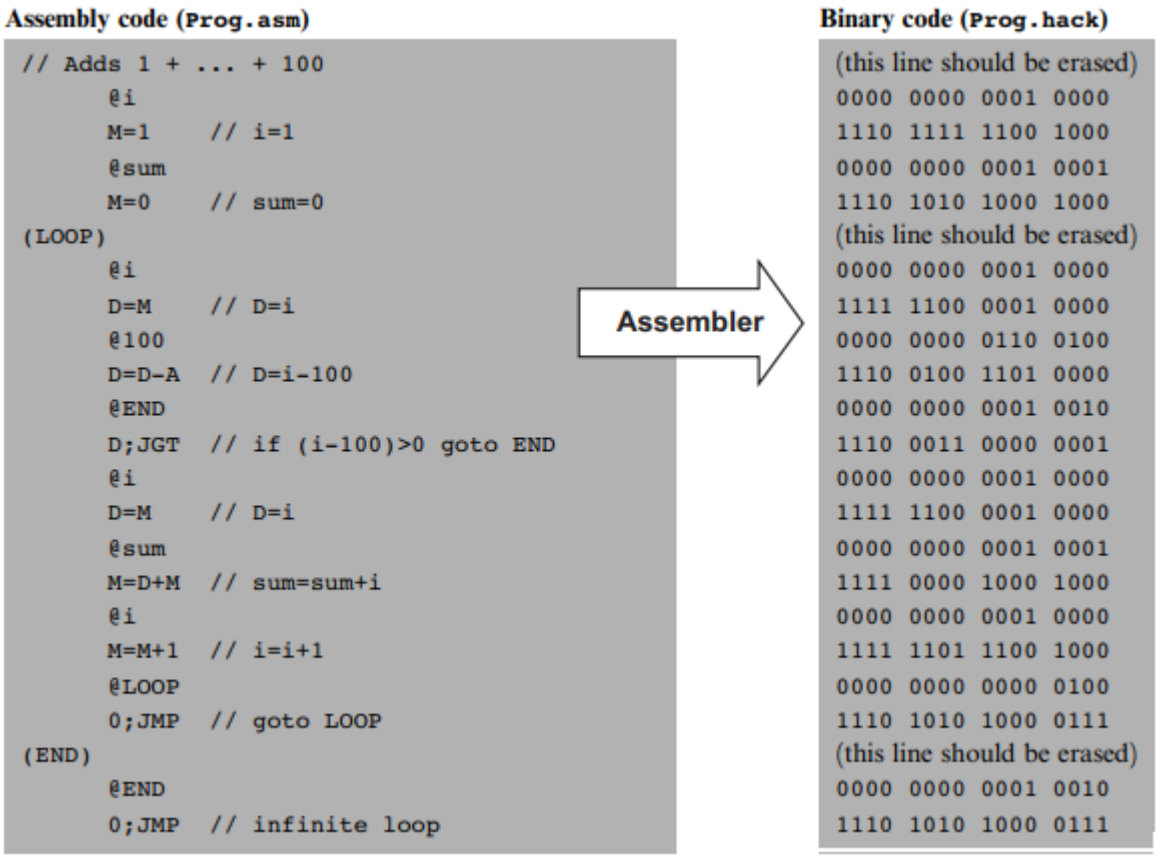
Поља comp, dest, jump унутар c-инструкције прослеђују процесору значајне информације и томе која операција треба да се изврши, где се вредност треба смести и да ли је потребно да се изврши скок на неки други део кода. У зависности од логичких вредности које се поставе у овим пољима свака вредност имаће своју одговарајућу акцију коју извршава, те акције можемо видети на следећој слици:

2.3Асемблер Hack рачунара

У прошлом поглављу, обрађивали смо машинске инструкције Hack рачунара и видели које врсте инструкција он подржава. Како би креирали програме који ће радити на рачунару, нећемо писати директно машинске инструкције, јер је доста компликовано написати програм у машинском коду, већ користимо програм ниског нивоа асемблер. Асемблер представља језик ниског нивоа који симболички текст преводи у одговарајуће машинске инструкције. Ако погледамо на слици 13. у првој и трећој колони можемо видети неке симболе који odgovaraju C инструкцијама. Сваки симбол има своју јединствену машинску инструкцију у коју се преводи. Једна линија асемблерског кода одговара једној линији машинског кода.

Поред симбола који представљају A и C инструкције постоје:

1) Лабеле које означавају дестинацију goto наредби:(LOOP),(END),…
2) Лабеле које означавају специјалне меморијске локације:@R0,@SCREEN,…
3) Променљиве:@i,@sum,@x,@y,...



Слика 14. Приказ програм у асемблерском и машинском језику

2.4Програмски језик високог нивоа Hack рачунара

До сада обрађивали смо ниске софтверске и хардверске нивое, што значи да људи не би требало директно раде са њима. У овом поглављу обрадићемо програм високог нивоа, под називом Jack, који је намењен за писање кода од стране корисника. Jack програмски језик је прост објектно оријентисан језик који дели сличне одлике као Java и C#,али са доста једноставнијом синтаксом и без могућности наслеђивања.

```
/** Hello World program. */
class Main {
    function void main () {
        // Štampa neki tekst koristeći standardnu biblioteku
        do Output.printString("Hello World");
        do Output.println();      // Nova linija
        return;
    }
}
```

Слика 15. Hello world програм у Jack-у

Овим програмским језиком могуће је направити различите програме, јер подржава:

1) Процедурално програмирање
2) Објектно оријентисано програмирање
3) Представљање апстрактних типова података
4) Представа структура података

White space and comments	Space characters, newline characters, and comments are ignored.
	The following comment formats are supported:
	`//  Comment to end of line`
	`/*  Comment until closing */`
	`/** API documentation comment */`

Symbols	`( )` Used for grouping arithmetic expressions and for enclosing parameter-lists and argument-lists `[ ]` Used for array indexing; `{ }` Used for grouping program units and statements; `,` Variable list separator; `;` Statement terminator; `=` Assignment and comparison operator; `.` Class membership; `+ - * / & | ~ < >` Operators.
Reserved words	`class, constructor, method, function` Program components `int, boolean, char, void` Primitive types `var, static, field` Variable declarations `let, do, if, else, while, return` Statements `true, false, null` Constant values `this` Object reference
Constants	*Integer* constants must be positive and in standard decimal notation, e.g., `1984`. Negative integers like `-13` are not constants but rather expressions consisting of a unary minus operator applied to an integer constant. *String* constants are enclosed within two quote (") characters and may contain any characters except *newline* or *double-quote*. (These characters are supplied by the functions `String.newLine()` and `String.doubleQuote()` from the standard library.) *Boolean* constants can be `true` or `false`. The constant `null` signifies a null reference.
Identifiers	Identifiers are composed from arbitrarily long sequences of letters (`A-Z, a-z`), digits (`0-9`), and "_". The first character must be a letter or "_". The language is case sensitive. Thus `x` and `X` are treated as different identifiers.

Слика 16. Синтакса Jack језика

2.4.1 Промењиве и типови података

Промењиве унутар Jack језика морају експлицитно бити декларисане пре употребе. Постоје четири врсте промењивих: filed, static, local и parameter, сваки тип поседује свој опсег и место у где се декларише.

Variable kind	Definition / Description	Declared in	Scope
Static variables	**static** *type name1, name2, ...* ; Only one copy of each static variable exists, and this copy is shared by all the object instances of the class (like *private static variables* in Java)	Class declaration.	The class in which they are declared.
Field variables	**field** *type name1, name2, ...* ; Every object instance of the class has a private copy of the field variables (like *private object variables* in Java)	Class declaration.	The class in which they are declared, except for functions.
Local variables	**var** *type name1, name2, ...* ; Local variables are allocated on the stack when the subroutine is called and freed when it returns (like *local variables* in Java)	Subroutine declaration.	The subroutine in which they are declared.
Parameter variables	*type name1, name2, ...* Used to specify inputs of subroutines, for example: `function void drive (Car c, int miles)`	Appear in parameter lists as part of subroutine declarations.	The subroutine in which they are declared.

Слика 17. Врсте промењивих у Jack језику

Унутар овог програмског језика постоје само три примитивна типа података (int, char, boolean), али промењива може да буде и типа објекат, која се декларише са именом класе. Класа које имплементира овај тип може да буде део стандарде библиотеке унутар Jack језика (Sting, Array, Keyboard, Math, Output,…), или може бити било која друга класа које је креирана од стране програмера.

```
/** Predstavlja račun u banci.
    Račun u banci ima vlasnika, id i stanje računa.
    Vrednosti id počinju od 0 i inkrementiraju se svaki
    put kada se novi račun kreira. */

class BankAccount {

    /** Kreira novi račun sa stanjem 0. */
    constructor BankAccount new(String vlasnik)

    /** Uplaćuje dati iznos na ovaj račun. */
    method void deposit(int iznos)

    /** Povlači dati iznos sa ovog računa. */
    method void withdraw(int iznos)

    /** Štampa podatke o ovom računu. */
    method void printInfo()

    /** Briše ovaj račun. */
    method void dispose()

}
```

Слика 16. Пример класе унутар Jack језика

Једна разлика која се може увидети код овог програмског језика, у односу на остале, је у томе што користи неколико кључних речи (do,let,function,method,…).

Кључна реч do користи се приликом позивања неке функције , let се користи када декларишемо променљиву и желимо да јој доделимо неку вредност, function користимо приликом креирања функције, method за функције унутар класе итд.

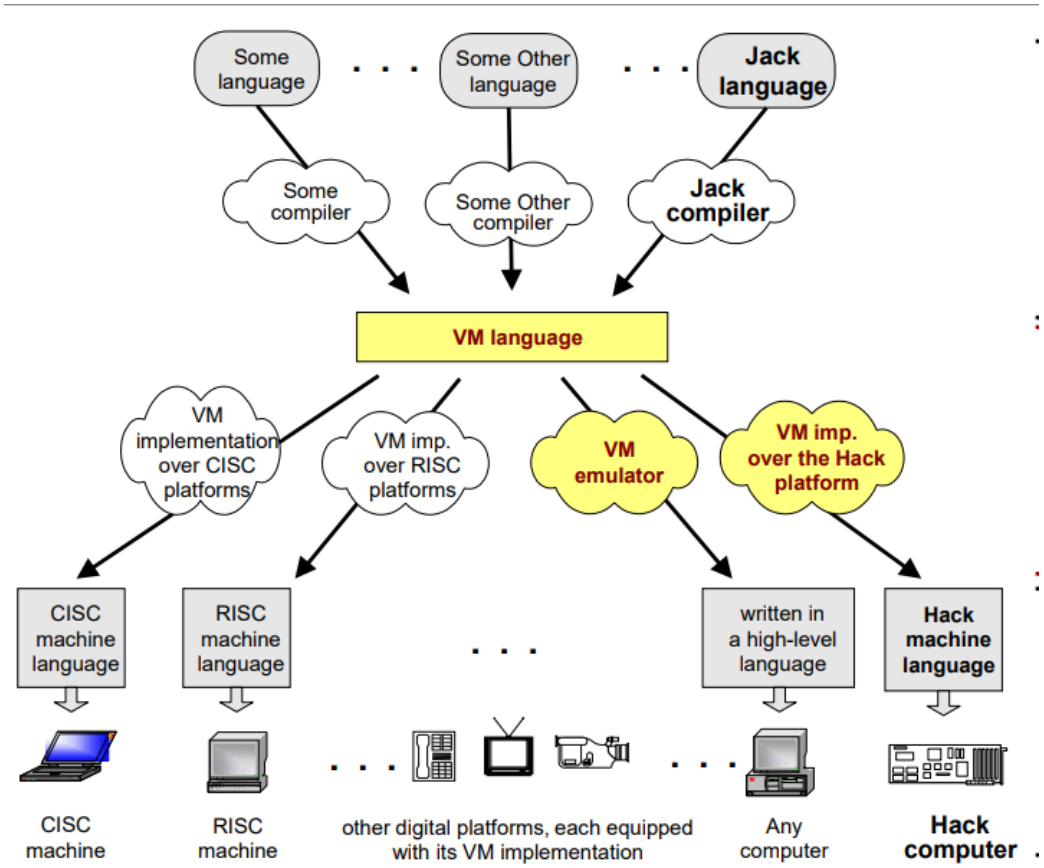
```
class Main {

  /** Sabira 1 + 2 + 3 + ... + n */
  function int sum (int n) {
    var int sum, i;
    let sum = 0;
    let i = 1;
    while (~(i > n)) {
      let sum = sum + i;
      let i = i + 1;
    }
    return sum;
  }

  function void main () {
    var int n;
    let n = Keyboard.readInt("Unesi n: ");
    do Output.printString("Rezultat je: ");
    do Output.printInt(sum(n));
    return;
  }
}
```

Слика 17. Пример програма у Jack језику

2.4.2 Виртуелна машина

Пре него што се програм ,написан у Jack програмском језику, покрене на неки рачунарски систем, он мора да се преведе у одговарајући машински код. Ово превођење је познато као compilation и представља један комплексан процес. У обичајно је да се праве посебни compiler-и за сваки пар програмског пакета високог нивоа и одговарајућег машинског језика. Како би омогућили да наш код ради на било ком уређају ми користимо VM (Virtaul Machine) језик између кода високог нивоа и асемблерског кода.

Виртуална машина представља језик који није ни вишег ни нижег нивоа. Овај међу језик се користи како би се могао покрене Jack код на било који уређај. Ако кренемо од кода вишег нивоа тај код се помоћу компајлера, као на слици 20, преводи у Jack језик. Након тога код високог нивоа се претвара у VM код који је мало ближи асемблерском коду. Када се креира тај код он може да се проследи и изврши на било коју хардверску платформу. Само је потребно да хардверска платформа поседује JVM implementation. Ove VM имплементације су у обичајно реализоване као client-side програми који преводе код из VM у машински код.

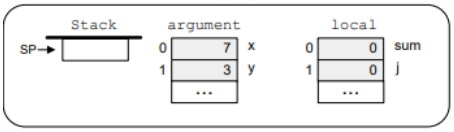


Слика 18. Процес двостепеног превођења програма

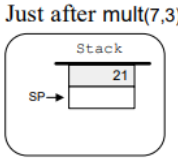
Kod visokog nivoa

```
function mult (x,y) {
  int result, j;
  result = 0;
  j = y;
  while ~(j = 0) {
    result = result + x;
    j = j - 1;
  }
  return result;
}
```

Just after mult(7,3) is entered:



Just after mult(7,3) returns:



VM kod (prva aproks.)

```
function mult(x,y)
   push 0
   pop result
   push y
   pop j
label loop
   push j
   push 0
   eq
   if-goto end
   push result
   push x
   add
   pop result
   push j
   push 1
   sub
   pop j
   goto loop
label end
   push result
   return
```

VM kod

```
function mult 2
  push   constant 0
  pop    local 0
  push   argument 1
  pop    local 1
label    loop
  push   local 1
  push   constant 0
  eq
  if-goto end
  push   local 0
  push   argument 0
  add
  pop    local 0
  push   local 1
  push   constant 1
  sub
  pop    local 1
  goto   loop
label    end
  push   local 0
  return
```

Слика 19. Пример VM кода

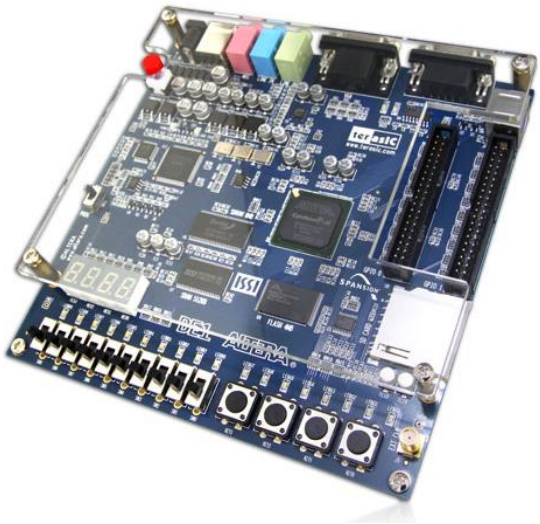
3. Практична реализација Hack рачунара

3.1Пројектни задатак

У досадашњем току рада теоретски смо видели како Hack рачунар функционише на хардверском и на софтверском нивоу. У следећем поглављу овај теоретски опис имплементираћемо на FPGA систему и покушати овај рачунар из теретског описа довести у реалност. Како доводимо овај рачунар у реални свет наилази се на нови сет проблема и изазова који нису били описани на теоретском моделу рачунара и имаћемо још већи увид о томе како данашњи рачунари функционишу.

3.2FPGA развојни систем

FPGA (Field-Programmable Gate Array) представља интегрисано коло чија унутрашња структура може да се конфигурише по жељи крајњег корисника. Да би смо могли да извршимо унутрашњу конфигурацију плоче морамо користити програмски језик за опис хардвера VHDL ili Verilog. Свака FPGA компонента се састоји од великог броја идентичних логичких блокова (ћелија), реконфигурабилних веза које омогућавају блоковима да буду међусобно повезани и улазно/излазног блока. Сваки логички блок се састоји од логичких ћелија. Логички блокови могу се конфигурисати тако да изводе сложене комбинаторне функције или једноставна логичка кола попут I кола и EXILI кола. Логички блокови такође могу укључивати и меморијске елементе, који могу бити једноставни флип-флопови или неки комплекснији меморијски елементи

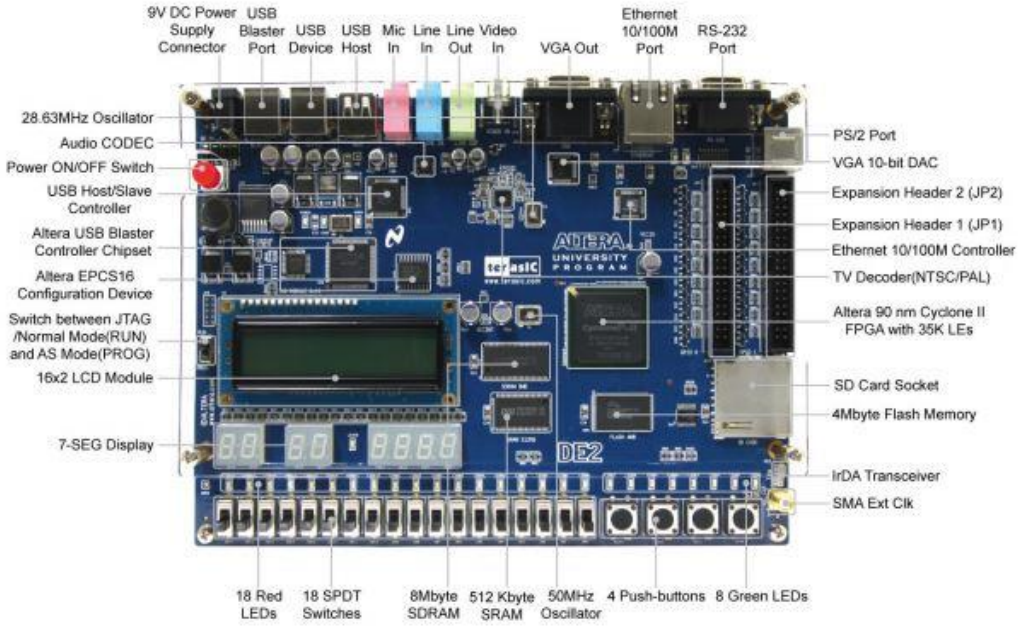


Слика 20. Primer FPGA ploce Altera Cyclone II

3.2.1 Altera DE2 ploča

Сврха Altera DE2 развојне плоче је да обезбеди идеалан медијум за напредне прототипе дизајна у мултимедији, складиштењу и умрежавању. Користи најмодернију технологију у хардверским и CAD алатима како би дизајнере изложио широком

распону тема. Плоча нуди богат скуп одлика које је чине погодном за употребу у лабораторијским вежбама за универзитетске и факултетске курсеве, за разне дизајнерске пројекте, као и за развој софистицираних дигиталних система. Алтера нуди комплет пратећи материјал за плочу DE2, укључујући туторијале, лабораторијске вежбе спремне за учење..



Слика 21. Altera DE2 развојно окружење

DE2 плоча има многе значајне карактеристике које омогућавају кориснику да имплементира широк распон дизајнираних кола, од једноставних кола до различитих мултимедијалних пројеката. Следеће компоненте се налазе на DE2 плочи:

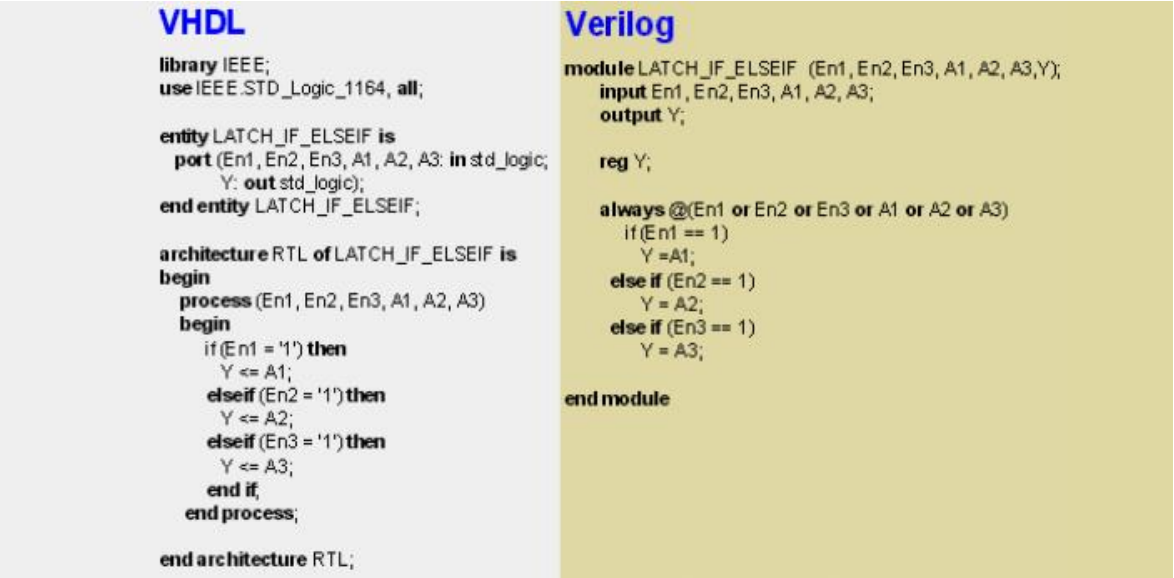
- Altera Cyclone II 2C35 FPGA device
- Altera Serial Configuration device – EPCS16
- USB Бластер (на плочи) за програмирање и корисничко контролисање API -а; и JTAG и Active Serial (AC) програмски модули су подржани
- 512-Kbyte SRAM
- 8-Mbyte SDRAM
- 4- Mbyte флеш меморије (1- Mbyte на неким плочама)
- SD Card Slot
- 4 притисна прекидача (типке)
- 18 прекидача
- 18 црвених LED диода
- 9 зелених LED диода

- 50- MHz осцилатор и 27- MHz осцилатор за такт
- 24-бит CD - quality audio CODEC са line-in, line-out i microphone-in прикључцима
- VGA DAC (10-бит high-speed triple DACs) са VGA - out конектором
- TV Децодер (NTSC/PAL) и TV -ин конектор
- 10/100 Ethernet контролер са конектором
- USB Host/Slave контролер са USB тип А и тип Б конекторима
- RS-232 примопредајник и 9-пински конектор
- PS/2 конектор за миш и тастатуру
- IrDA примопредајник
- Два 40-пинска заглавља за проширење са диодном заштитом

3.3Програмски језици за опис хардвера

Језик за опис хардвера(HDL) представљају специјализоване програмске језике који служе за опис структуре и понашања електронских и дигиталних кола. Ова група програмских језика има сличну структуру као C језик са својим изразима, променљивама и структуром. Главна разлика између оваквих и класичних програмских језика је у томе што код оваквих језика узима се у обзир појам времена. На основу тога можемо видети понашање сигнала унутар нашег хардверског система и кориговати према нашим потребама. Први HDL језици настали су почетком 1960, они су више изгледали као традиционални језици, а данас су најзатупљенији VHDL и Verilog.

VHDL је настао у Сједињеним америчким државама од стане Америчког департмана за безбедност и заснован је на Ада програмском језику. Verilog је био развијен око 1984 од стране Phil Moorby and Prabhu Goel и око 2001. год постао је IEEE стандард. За израду рачунарском система користићемо Verilog програмски језик и у наставку овог рада говорићемо о његовој синтакси.



VHDL

```
library IEEE;
use IEEE.STD_Logic_1164, all;

entity LATCH_IF_ELSEIF is
  port (En1, En2, En3, A1, A2, A3: in std_logic;
        Y: out std_logic);
end entity LATCH_IF_ELSEIF;

architecture RTL of LATCH_IF_ELSEIF is
begin
   process (En1, En2, En3, A1, A2, A3)
   begin
      if (En1 = '1') then
        Y <= A1;
      elseif (En2 = '1') then
        Y <= A2;
      elseif (En3 = '1') then
        Y <= A3;
      end if;
   end process;

end architecture RTL;
```

Verilog

```
module LATCH_IF_ELSEIF  (En1, En2, En3, A1, A2, A3,Y);
    input En1, En2, En3, A1, A2, A3;
    output Y;

    reg Y;

    always @(En1 or En2 or En3 or A1 or A2 or A3)
       if (En1 == 1)
          Y =A1;
     else if (En2 == 1)
          Y = A2;
     else if (En3 == 1)
          Y = A3;

end module
```

Слика 22. Verilog и VHDL програмски језици

3.4Структура Verilog програмског језика

Verilog програмски језик је доста сличан C језику и састоји се од неколико нивоа апстракције, а три главна су:

1. Ниво понашања
2. Ниво размене регистра
3. Ниво логичких кола

Ниво понашања описује систем у зависности од конкурентних алгоритама. Сваки алгоритам је секвенцијалан, што подразумева да се састоји од сета инструкција који се извршавају једна за другом. Функције задаци и блокови су главни елементи.

Ниво размене регистра дефинише карактеристику дигиталног кола користећи операције и размену података између регистра.

Код Ниво логичких кола карактеристике система описане су логичким везама и њиховим временским својствима. На овом нивоу сви сигнали су дискретни и могу само да имају логичке вредности (`0', `1', `X', `Z`).

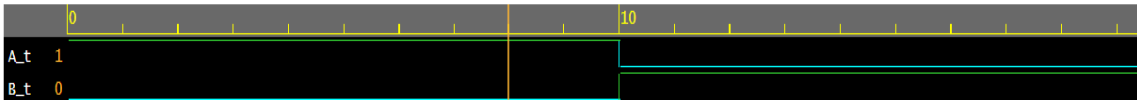
3.4.1 Модули

Модули представљају основи градивни блок Verilog програмског језика. Модул може бити основни елемент или скуп повезаних модула нижег нивоа који пружа одређену функционалност. Модули се декларишу помоћу кључне речи module и на крају дефинисања модула мора се наћи кључна реч endmodule. Модул преко својих портова који могу бити улазни и излазни обезбеђује дефинисану функционалност према модулима вишег нивоа. Сваки модул мора да се састоји од :

1. Име (једнозначно идентификује модул)
2. Интерфејс модула (описује улазне и излазне прикључке модула)
3. Садржај модула (садржи опис понашање модула који се са улазних прикључака преноси на излазне прикључке модула)

Пример модула који врши инвертовање улазног сигнала на излазу:

```
module Invertor(input A , output B);

 assign B=~A;

endmodule
```



Слика 23. Временски дијаграм улазног и излазног сигнала

Типови података

Унутар *Verilog* програмског језика постоје 4 главне вредности података које можемо да видимо на излазу:

1. 0 (логичка нула или нетачан исказ)
2. 1 (логичка јединица или тачан исказ)
3. x (непозната логичка вредност)
4. z (стање високе инпендансе)

Поред ових вредности постоје **nets** типови података који су постављене од стране излаза кола на које су повезане.

Net tip podataka	Funkcionalnost
wire, tri	Linija za povezivanje – nema posebnu funkciju
wor, trior	Ožičeno ILI – wired-OR (modeluje ECL kola)
wand, triand	Ožičeno I – wired-AND (modeluje open-collector kola)
tri0, tri1	pull-down ili pull-up kad nije pogonjen
supply0, supply1	Na liniji se nalazi konstantna logička nula ili jedinica (supply jačina)
trireg	Zadržava prethodnu vrednost kada je kolo u stanju visoke impedanse

Слика 24. Нетс типови података

Код излазних портова у неким случајевима је потребно да задрже своју вредност (флип флопови и лечеви) и они се тада морају декларисати као reg. Улазни портови се не могу декларисати као reg тип, јер они не требају да чувају вредност већ рефлектују промене спољашњих сигнала на које су повезане. Такође Verilog подржава основна логичка кола :

- and
  - nand
  - or

- nor
- xor
- xnor
- buf
- not
- bufifi1
- notifi1

3.4.2 Повезивање сигнала са портовима

Сигнали који се наводе при инстанцирању модула повезују се са портовима модула на два начина:

- Уређена листа
- По имену

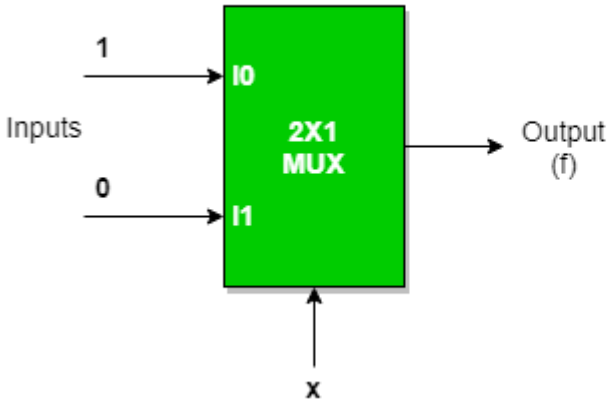
Код уређене листе, сигнали се при инстанцирању наводе у истом редоследу као портови у дефиницији модула. Везе се могу специфицирати у произвољном редоследу, ако се наведу имена портова.

```
module Top;
reg [3:0] A, B; // deklaracija promjenljivih za povezivanje
reg C_UL;
wire [3:0] ZBIR;
wire C_IZ;

// instanciranje -signali su redom povezani na portove (po poziciji)
fulladd4 fa_uredjena_lista(ZBIR, C_IZ, A, B, C_UL);

// instanciranje -signali su povezani preko imena
fulladd4 fa_uredjena_lista(.sum(ZBIR), .c_out(C_IZ), .a(A), .b(B),
.c_in(C_UL));
... <stimulus>...
endmodule

module fulladd4(sum, c_out, a, b, c_in);
output [3:0] sum;
output c_cout;
input [3:0] a, b;
input c_in;
...<unutrašnjost modula>...
endmodule
```

Слика 25. Пример Уређене листе и по именима

3.4.3 Пример реализације једног 16-битног мултиплексера

Да би реализовали један 16-битни мултиплексер потребно је прво да сагледамо структуру једног мултиплскксера 2/1. Мултиплскер 2/1 је логичко коло које се састоји од два улазна сигнала, једног излазног и једног селекционог сигнала који када има вредност логичке 0 селектује један улаз, а када има вредност логичке јединице селектује вредност са другог улаза и ту вредност представља на излазу.



Truth Table

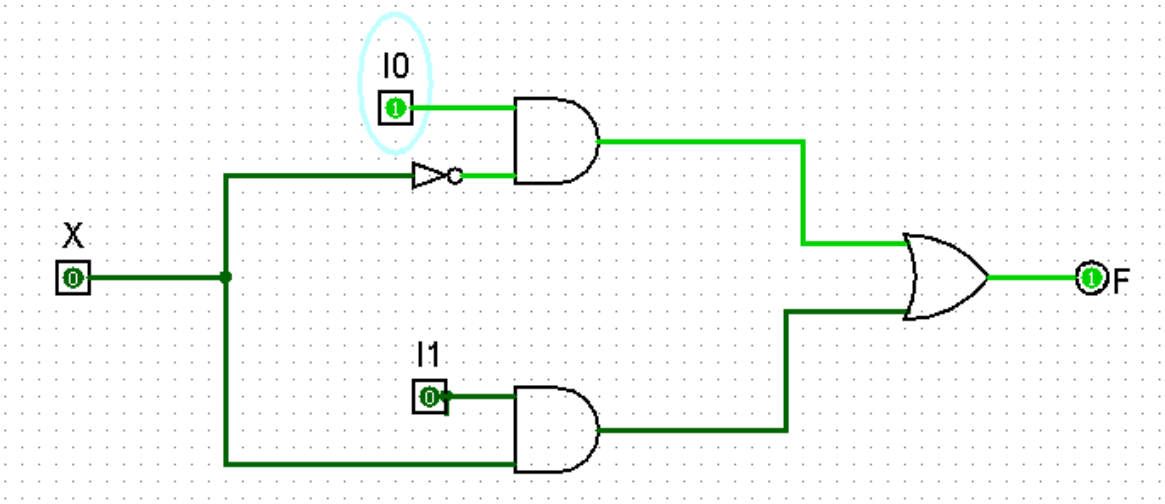
x	f
0	1
1	0

Слика 26. Мултиплексер 2/1

Рад мултиплексера можемо представити преко одговарајућих логичких кола. За реализацију мултиплексера преко логичких кола потребно је знати његову логичку једначину. Са слике 28. можемо видети да се мултиплексер састоји од улаза I0 и I1, селекционог бита X и излаза F. Тада ће логичка функција за овај мултиплексер гласити:

$$F = \bar{x}I0 + xI1 \tag{1}$$

На основу функције (1), можемо видети да се мултиплексер састоји од два I кола једног ili кола и једног инверторског кола.



Слика 27. Реализација мултиплексера са логичким колима

Као што смо малопре упознали за структуром verilog језика, креираћемо један модул и програмски описати наше логичко коло.

```
module mux_2_1 (output F , input I0,I1,X);

  wire nX,XI0,XB;

  not u1 (nX, X);
```

```
  and U2(XI0,I0,nX);

  and U3(XB,I1,X);

  or U4(F,XI0, XB);

endmodule
```

Слика 28. Мултиплексер описан у verilog програмског окружењу

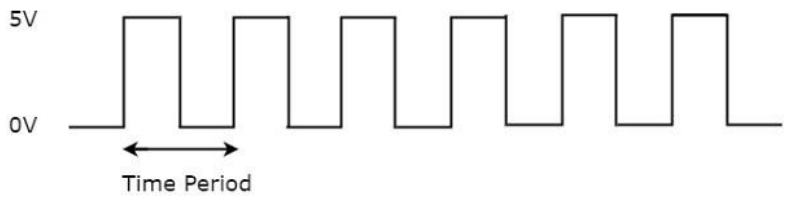
За реализацију 16-битног мултиплексера креираћемо нови модул, дефинисати да улази и излази буду 16-битни и позвати овај 1-битни мултиплексер за сваки бит нашег 16-битног.

```
module mux_2_1_16bit(output[15:0] F , input[15:0] I0,I1,input X);
  mux_2_1 U0 (F[0], I0[0],I1[0],X);
  mux_2_1 U1 (F[1], I0[1],I1[1],X);
  mux_2_1 U2 (F[2], I0[2],I1[2],X);
  mux_2_1 U3 (F[3], I0[3],I1[3],X);
  mux_2_1 U4 (F[4], I0[4],I1[4],X);
  mux_2_1 U5 (F[5], I0[5],I1[5],X);
  mux_2_1 U6 (F[6], I0[6],I1[6],X);
  mux_2_1 U7 (F[7], I0[7],I1[7],X);
  mux_2_1 U8 (F[8], I0[8],I1[8],X);
  mux_2_1 U9 (F[9], I0[9],I1[9],X);
  mux_2_1 U10 (F[10], I0[10],I1[10],X);
  mux_2_1 U11(F[11], I0[11],I1[11],X);
  mux_2_1 U12 (F[12], I0[12],I1[12],X);
  mux_2_1 U13 (F[13], I0[13],I1[13],X);
  mux_2_1 U14 (F[14], I0[14],I1[14],X);
  mux_2_1 U15 (F[15], I0[15],I1[15],X);

endmodule
```

Слика 29. Мултиплексер 16-битни описан у verilog програмском окружењу

3.5 Реализација Hack компоненти

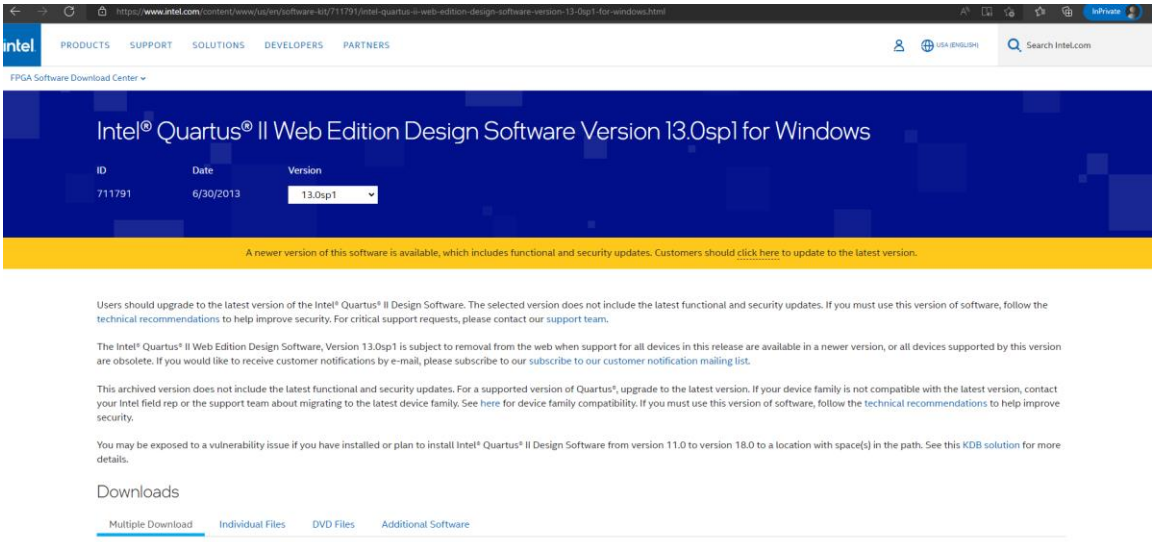
У прошлим поглављима говорили смо уопштено како функционише Hack рачунар. Приликом практичне реализације рачунара произилазимо на нови проблем који није био описан, како синхронизовати све компоненте да раде заједно? У теоретском опису рачунара није споменуто како се то постиже. Да би се делови рачунара синхронизовали морамо користити један континулани сигнал, који ће ићи на све компоненте, тај сигнал се назива Такт(CLK). Овај сигнал осцилује између високог и ниског стања и може се посматрати као неки метроном. Наше компоненте користе овај тактни сигнал да на позитивну или негативну ивицу такта изврше своје акције.



Слика 30. Сигнал Такта

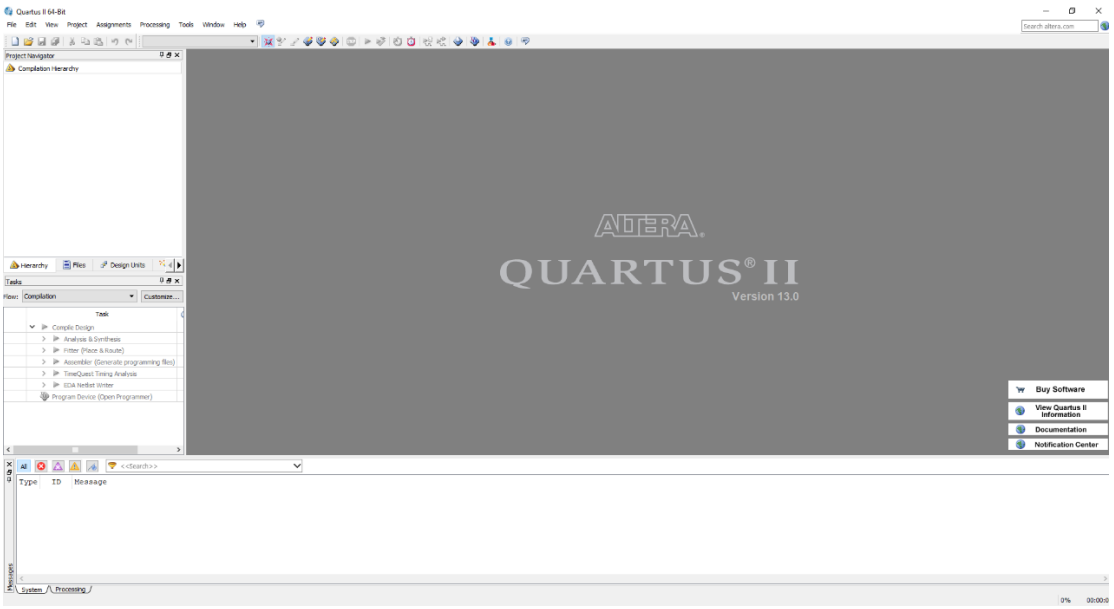
3.5.1 Quartus II окружење

Пре него што почнемо развијати наш рачунар у Verilog-у, прво морамо да се упознамо са програмским окружењем FPGA система који је коришћен у овом раду. У овом раду користићемо Altera Cyclone II систем, који може видети на слици 23, ова плоча је развијена од стране Alter-е коју поседује Intel. Да би смо били у могућности да програмирамо плочу, потребно је са intel-овог сајта преузети софтверски алат.



Слика 31. Сајт за преузимање Quartus *софртверског алата*

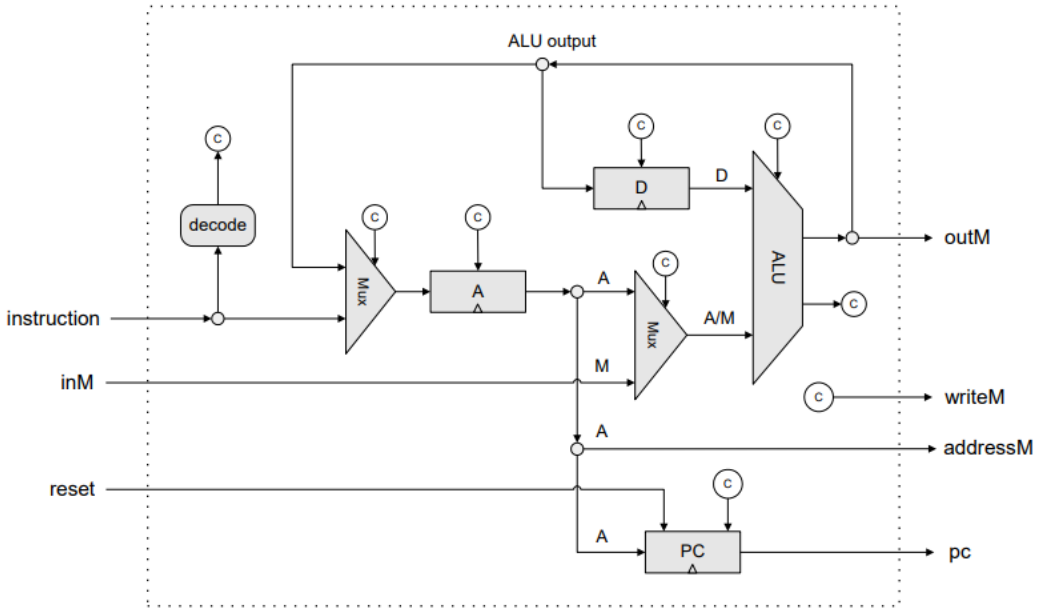
На слици 34 видимо окружење које представља један типичан софтверски алат. На врху имамо drop-down мениje са додатним опцијама, такође испод менија имамо скраћенице за неке брзе функционалности које су нам потребне. У централном левом делу имамо два проза Project navigation и Tasks. Први прозор служи да сагледамо све фајлове које смо креирали за пројекат(видећемо то ускоро) док други прозор даје информацију шта тренутно компајлер ради. Велики прозор са десне стране представља радну површ. Унутар овог прозора се учитава код, пише verilog код и слично. Доњи дугачки прозор има задатак да исписује грешке или упозорења која се јављају приликом компајлирања програма.



Слика 32. Софтверско окружење

3.6Реализација процесора

Како би реализовали процесор мора се сагледати његова унутрашња структура.



Слика 33. Унутрашња структура процесора

Као што можемо видети на слици 32, процесор се састоји од два регистра(A,D), програмског бројача(PC), два мултиплексера, Аритметичко логичке јединице(ALU) и декодера. Сваки елемент у садржи одговарајући контролни бит „C“ који се активира у зависности од C инструкције која је прослеђена. Уколико имамо A инструкцију она се шаље у A регистар и у зависности од следеће инструкције она се може проследити аритметичкој логичкој јединици, као адреса за меморијску локацију или адреса на коју се треба извршити следећа инструкција (прослеђује се PC). D регистар чува у себи излаз из аритметичко логичке јединице како би могли резултат неке операције опет

брзо да доведемо на улаз. Аритметичко логичка јединица извршава математичке и логичке операције, док PC има задатак да нам проследи адресу на којој се налази следећа инструкција.

3.6.1 Регистри

Реализација A i D регистра је поприлично једноставна, оба регистра су 16-битни који имају три улазна сигнала d,load.clk,rst и излазни q сигнал. Када је активан rst, регистар се сетује на вредност децималне нуле, а уколико је активан load сигнал учитава се вредност d која се доводи. У супротном регистар памти своју претходну вредност.

```
module reg16bit(output reg[15:0] q , input[15:0] d, input clk,load,rst
);


  always@(posedge(clk)) begin

    if(rst)
       q<=16'b0000000000000000;
    else if(load)
         q<=d;

      else
        q<=q;
    end

endmodule
```

Слика 34. 16-битни регистар

3.6.2 Програмски бројач

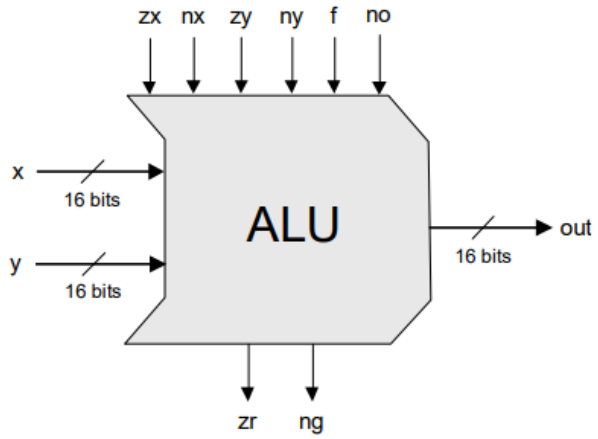
Програмски бројач има основну функцију да проследи адресу следеће инструкције тако што се бројач константно повећава за једну вредност више. Он може да крене од нуле или да му се зада адреса од које треба да крене да се инкрементира. Да би се он постепено повећавао унутар њега је реализован сабирач који сабира две бинарне вредности (FullAdder).

```
module PC(output[15:0] Q , input[15:0] A, input inc,load,reset,clk);

  wire[15:0]s1,s2,s3,s4;
  wire cout;
  supply1 v1;
  supply0[15:0] g1;


  FullAdder16 F1(cout,s1,Q,g1,v1);
  mux2_1_16bit M1(s2,Q,s1,inc);
  mux2_1_16bit M2(s3,s2,A,load);
  mux2_1_16bit M3(s4,s3,g1,reset);
  reg16bit R1 (Q,s4,clk,v1,reset);
endmodule
```

Слика 35.Програмски броја описан у Verilog језику

3.6.3 Аритметичко логичка јединица

Као што је наглашено, Аритметичко логичка јединица (ALU) обавља све аритметичке и логичке операције.



Слика 36. Аритметичко логичка јединица

Ако погледамо слику 38 можемо видети да се ALU састоји од два 16-битна улаза x и y, контролних битова zx ,nx, zy, ny, f, no, а излаз је 16-битни out сигнал и два контролна бита zr и ng. Како би аритметичко логичка јединица извршила неку математичку или логичку операцију над овим улазима, потребно је задати одговарајућу комбинацију над контролним битовима. Сваки контролни бит има своју акцију коју одрађује и заједно утичу на улазе, као на слици:

These bits instruct how to pre-set the x input		These bits instruct how to pre-set the y input		This bit selects between + / And	This bit inst. how to post-set out	Resulting ALU output
zx	nx	zy	ny	f	no	out=
if zx then x=0	if nx then x=!x	if zy then y=0	if ny then y=!y	if f then out=x+y else out=x And y	if no then out=!out	f(x,y)=
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

Слика 37. Излази из ALU

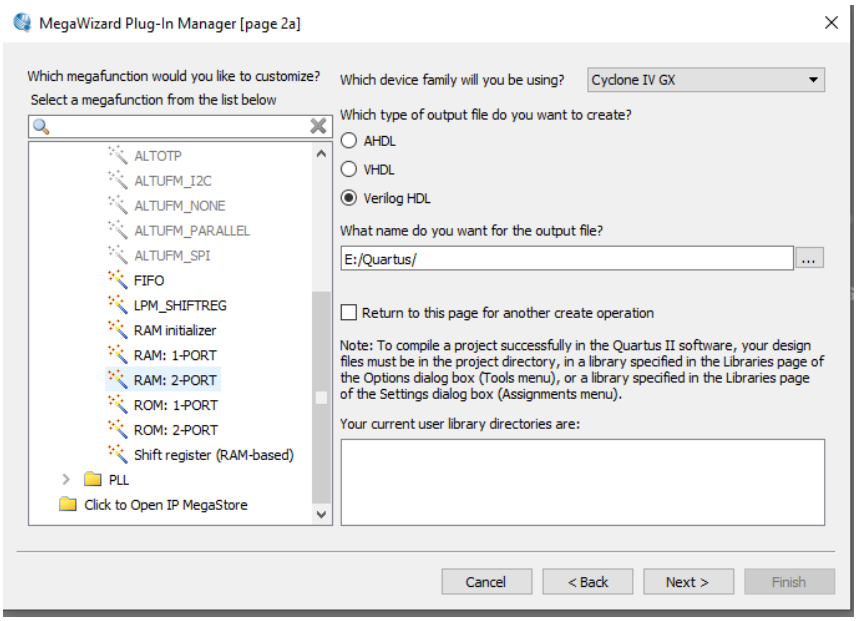
Излазни бит zr проверава да ли је број једнак нули, ако јесте заузима стање логичке јединице, а ng проверава да ли је број негативан и такође заузима стање логичке јединице ако је број негативан. За реализацију ALU користе се мултиплексери за прва четири сигнала, где сваки контролни бит представља један селекциони бит мултиплексера, у зависности да ли је он активан улаз се прослеђује или инвертује. За f контролни бит морамо реализовати операцију сабирања помоћу FullAdder који ће сабрати ова два улаза, а за операцију множења користимо једно I коло. И за последњи бит такође користимо један мултиплексер као за прва четири бита. Како би добили ng излаз само проследимо највиши бит out сигнала, а за реализацију zr излаза сваки бит излазног out сигнала морамо довести на логичко ILI коло и негирати излаз из тог кола.

```
module ALU(output[15:0] out,output zr,ng,input[15:0] x,y,input
zx,nx,zy,ny,f,no);

  wire[15:0] s1,s2,s3,k1,k2,k3,p1,z1,f1,if1;
  wire cout;
  supply0[15:0] g1,g2;
  supply0 g3;

  mux2_1_16bit m1(s1,x,g1,zx);
  INV I1(s2,s1);
  mux2_1_16bit m2(s3,s1,s2,nx);
  mux2_1_16bit m3(k1,y,g2,zy);
  INV I2(k2,k1);
  mux2_1_16bit m4(k3,k1,k2,ny);

  I_Gate And1(p1,s3,k3);
  FullAdder16 Add1(cout,z1,s3,k3,g3);
  mux2_1_16bit m5(f1,p1,z1,f);
  INV I3(if1,f1);
  mux2_1_16bit m6(out,f1,if1,no);

  assign ng=out[15];
  assign zr= ~(|out);
endmodule
```

Слика 38. Структура ALU описана Verilog језиком

3.7 Меморија података

Као што смо у прошлим поглављима видели, меморија података састоји се од три сегмента за податке, екран и тастатуру. За реализацију овог чипа унутар FPGA платформе користићемо алат које је уграђен унутар Quartus окружења који аутоматски генерисати Verilog модул и користи меморијске блокове који се налазе на FPGA плочи. Сва три чипа дизајнирају се на исти начин и после се пакују унутар једно модула.

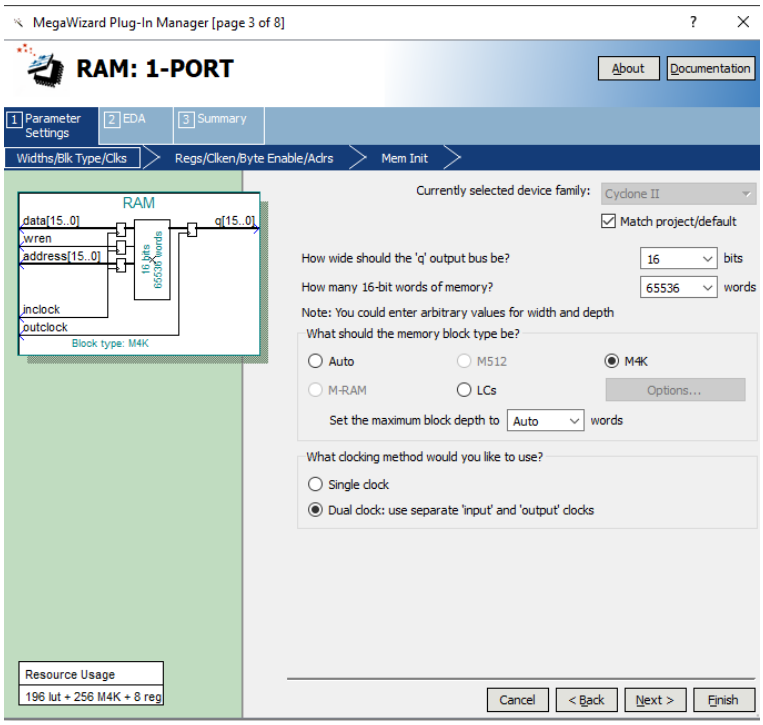


Слика 39. Алат за креирање меморијских компоненти

Ако погледамо слику 41 можемо видети све опције које су нам дате за креирање меморијског елемента. Реализова ћемо ram меморију и то RAM: 1-PORT, а меморију за екран и тастатуру као RAM: 2-PORT.

3.7.1 RAM меморија

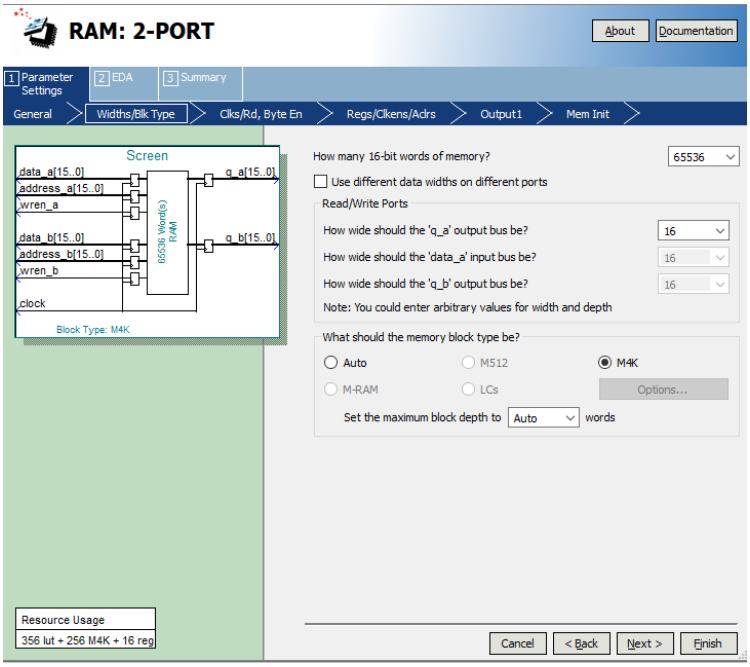
Ову меморију реализујемо као једнопортну меморију са два различита такта на улазу и на излазу. Зашто користимо два такта? Ако погледамо асемблерску инструкцију на пример M=M+D . Изгледа као сасвим нормална инструкција чита се податак из меморије сабира са вредношћу унутар D регистра и збир се враћа у меморију. Приликом практичне реализације овакве инструкције нам праве проблем. Од меморије се захтева да у једном такту прочита и упише податак унутар меморије, што је не могуће урадити. Зато приликом реализације меморије уводимо два тактна циклус и подешавамо да меморија рачунара ради дупло бржем такту од процесора. Овим приступом постижемо да са дупло бржим тактом меморија прочита и после упише податак позван овом инструкцијом.



Слика 40, RAM Verilog модул реализован помоћу алата унутар Quartus програма

3.7.2 Меморија за екран и тастатуру

За разлику од ram меморије за податке, RAM чипови за екран и тастатуру поред два такта морају да се реализују као двопортне меморије. Оне се реализују на овај начин из разлога што имамо два уређаја која комуницирају са меморијом процесор на једном порту, а екран или тастатура на другом порту.



Слика 41. Екран Verilog модул реализован помоћу алата унутар Quartus програма

Исти поступак се понавља се и за тастатуру.

3.7.3 Повезивање меморијских сегмената

Како би смо спојили све меморије у један модул, потребно је да сагледамо да ли је инструкција намењена за ram, екран или тастатуру. Ако погледамо слику 6 може се видети како је расподељен меморијски простор. Код практичне реализације како би проверили који меморијски простор се користи користимо addresM сигнал. На адреси од 0 до 16К резервисан је простор за ram меморију. Од 16К па до 24К резервисан је простор за екран и 24к+1 бит резервисан за тастатуру. Проверавајући 13 бит од addresM сигнала, да ли је на логичкој јединици или нули, можемо утврдити да ли се инструкција односи за ram или за екран и тастатуру, а проверавајући 14 бит можемо утврдити да ли се ради о екрану или тастатури. Да би имплементирали ову логику користе се демултиплексери на улазу и мултиплексери на излазу из меморије.

```
module RAM(output[15:0]Q,output[15:0]QVGA,
input[15:0]data,input[15:0]dataKBD,addresM, input [12:0] addressVGA,
input writeM,writeKBD,clk1,clk2,rst);

wire [15:0] RAMQ,SCRQ1,SCRQ2,KBDQ1,KBDQ2,muxOut1;
supply1 Vcc;
supply0 GND;
supply0[3:0] GND4;
supply1[3:0] VCC4;
supply0[15:0] GND15;
wire ramM,scrM,kbdM,k,ramW,scrW,kbdW;

DEMUX1bit D1(ramM,k,Vcc,addresM[14]);
DEMUX1bit D2(scrM,kbdM,k,addresM[13]);

assign ramW=ramM & writeM;
assign scrW= scrM & writeM;
assign kbdW= kbdM & writeM;



RAM16K R1(addresM[13:0],data,clk1,rst,clk2,ramW,RAMQ);
SCR
S1(addresM[12:0],addressVGA,clk1,clk2,data,GND15,scrW,GND,SCRQ1,QVGA);
KBD
K1(addresM[3:0],GND4,data,dataKBD,clk1,rst,clk2,kbdW,writeKBD,KBDQ1,KBD
Q2);

Mux16bit M1(muxOut1,SCRQ1,KBDQ1,addresM[13]);
Mux16bit M2(Q,RAMQ,muxOut1,addresM[14]);
endmodule
```

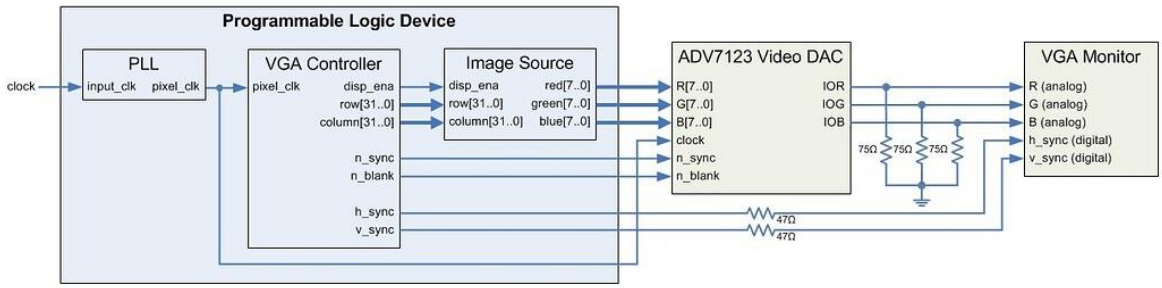
Слика 42. Data меморија представљена у Verilog језику

3.8Програмска меморија

Програмску меморију креирамо на исти начин као и остале меморија, али приликом практичне реализације ,због не довољних ресурса од стане FPGA система, морамо користити SRAM меморију FPGA система као ROM меморију. За попуњавање SRAM меморије користи се посебан програм о коме ћемо више споменути касније у раду.

3.9Графички контролер

Да би смо могли садржај нашег екрана представити на монитору морамо се упознати sa начином функционисања VGA монитора. За VGA комуникацију потребне су нам три компоненте контролер, генератор који ће да генерише слику и прескалер који ће да смањи фреквенцу тактног сигнала.



Слика 43. Шема имплементације

3.9.1 Комуникација преко VGA излаза

VGA је стандардни интерфејс за контролу аналогних монитора. Компјутерска страна интерфејса обезбеђује монитору хоризонталне и вертикалне синхронизационе сигнале, магнитуде боја и референце на земљи.

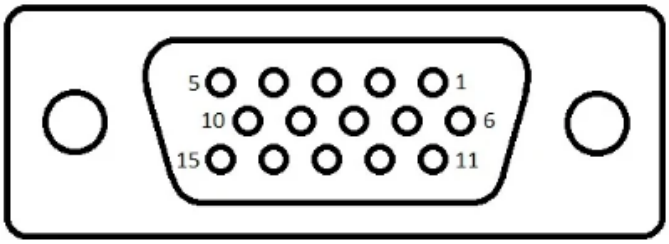
Хоризонтални и вертикални сигнали синхронизације су дигитални таласни облици 0V/5V који синхронизују тајминг сигнала са монитором. Будући да су дигитални, обезбеђују их директно из FPGA (3.3V задовољава минимални праг за логички висок ниво, па се може коистити 3.3V уместо 5B).

Магнитуде боја су аналогни сигнали 0V-0.7V који се шаљу преко R, G и B жица. Алтернативно, зелена жица може да користи сигнале 0.3V-1V који укључују и хоризонталне и вертикалне сигнале за синхрониацију, елиминишући потребу за тим линијама. То се назива sync-on-green. Три жице магнитуде боја су завршене отпорницима отпорности 75Ω. Ове линије су такође завршене отпорницима отпорности 75 Ω унутар монитора. За креирање ових аналогних сигнала, FPGA шаље 8-битну сабирницу за сваку боју у видео DAC. Овај видео DAC такође захтева да пикселни сат буде у тим вредностима.

VGA интерфејс такође одређује четири жице које се могу користити за комуникацију са ROM-ом на монитору. Овај ROM садржи EDID (проширене идентификационе податке), који се састоји од параметара монитора у стандардном формату. Постоји неколико комуникацијских стандарда за приступ овим подацима, али у најједноставнијем случају ове линије могу остати неповезане.

3.9.2 Повезивање

VGA konekcije koriste 15-pinski konektor koji se zove DB15.



Слика 44. VGA женски конектор (DB15 спремник)

1	R	Аналогна црвена 0-0.7V	DAC излаз
2	G	Аналогна зелена 0-0.7V или 0.3V-1V ако sync-on-green	DAC излаз
3	B	Аналогна плава 0-0.7V	DAC излаз
4	EDID Interface	Функције варирају у зависности од употребљених стандарда	Нема конеккције
5	GND	Опште	GND
6	GND	За R	GND
7	GND	За G	GND
8	GND	За B	GND
9	Без пина	Или опционо +5V	Нема конекције
10	GND	За h_sync и v_sync	GND
11	EDID интерфејс	Функције варирају у зависности од употребљених стандарда	Нема конекције
12	EDID интерфејс	Функције варирају у зависности од употребљених стандарда	Нема конекције
13	h_sync	Хоризонтална синхронизација, 0V/5V таласи	FPGA
14	v_sync	Вертикална синхронизација, 0V/5V таласи	FPGA
15	EDID интерфејс	Функције варирају у зависности од употребљених стандарда	Нема конекције

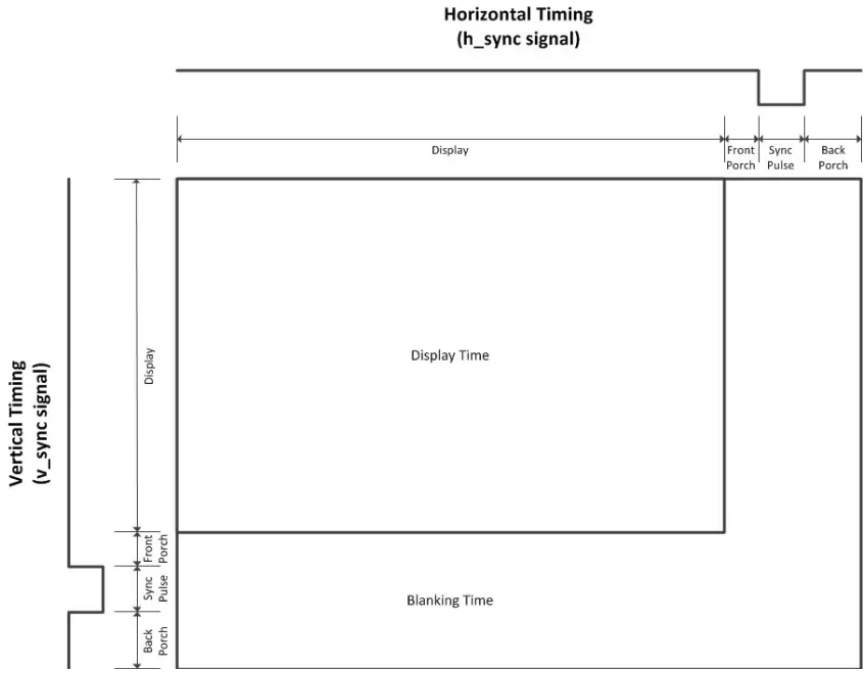
3.9.3 Закон функционисања

Контролер садржи два бројача. Један бројач се повећава на сатовима пиксела и контролише време h_sync (хоризонталне синхронизације) сигнала. Постављањем тако да време приказа почне на вредности бројача 0, вредност бројача је једнака координати колоне пиксела током времена приказа. Након хоризонталног приказа

следи време празног хода, које укључује хоризонтални предњи трем, сам хоризонтални синхронизацијски импулс и хоризонтални задњи трем, сваки одређеног трајања. На крају реда бројач се враћа да би покренуо следећи ред.

Други бројач се повећава како се сваки ред завршава, чиме се контролише време v_sync (вертикалне синхронизације) сигнала. Опет, ово је подешено тако да време приказа почиње од вредности бројача 0, тако да вредност бројача одговара координати реда пиксела за време приказа. Као и раније, вертикално време приказа прати време празног хода, са одговарајућим предњим темом, синхронизационим импулсом и задњим тремом. Када се заврши вертикално затамњење, бројач се враћа да би почео са следећим освежавањем екрана.

Омогућавање приказа дефинисано је логичким AND хоризонталних и вертикалних времена приказа. Помоћу ових бројача VGA контролер емитује хоризонталне синхронизације, вертикалне синхронизације, омогућавање приказа и сигнале координата пиксела. Синхронизовани импулси су специфицирани као позитивни или негативни поларитети за сваки VGA режим. GENERIC параметри h_sync (хоризонтални полартет) и v_sync (вертикални поларитет) подешавају поларитет h_sync и v_sync VGA контролера, респективно.



Слика 45. Сигнални временски дијаграм

3.9.4 Реализација VGA система

Како би смо смањили улазни такт који користимо, потребно је креирати прескалер који смањује улазни такт. На VGA Sync модул дововоди се сигнал такта, а на излазу добијају се сигнали за синхронизацију који се повезују са одговарајућим пиновима за VGA output, пикселима којима приступамо преносимо на ImageGenerator.

Помоћу ImageGenerator модула генеришемо слику на екрану. Овај модул за улазне параметре узима позицију пиксела и податке који су уписани унутар меморије.

Генератор чита те вредности и уколико је податак бинарна 0 треба да прикаже белу боју, а ако је бинарна 1 треба да покаже црвену боју.

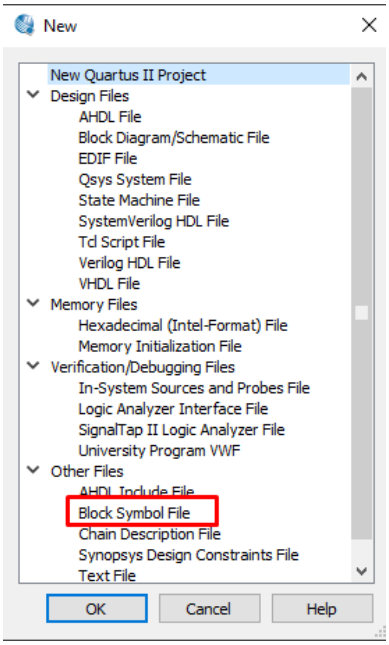
Са завршетком креирање под делова од којих се VGA конторлер састоји. Потребно је узети све ове делове и саставити у један модул као на слиици:

```
module VGA1(output [9:0] red,green,blue,output [12:0] address,output
horiz_sync_out, vert_sync_out,
output video_on, pixel_clock,input inclk0,input [15:0] data);

wire c0;
wire [9:0] pixel_row, pixel_column;

vga_pll V1(.inclk0(inclk0),.c0(c0));
VGA_SYNC
V2(.pixel_clock_int(c0),.horiz_sync_out(horiz_sync_out),.vert_sync_out(
vert_sync_out),.video_on(video_on),.pixel_clock(pixel_clock),.pixel_row
(pixel_row),.pixel_column(pixel_column));
ImageGenerator
V3(.red(red),.green(green),.blue(blue),.address(address),.row(pixel_row
),.columen(pixel_column),.data(data));


endmodule
```

Слика 46. VGA модул описан у verilog језику

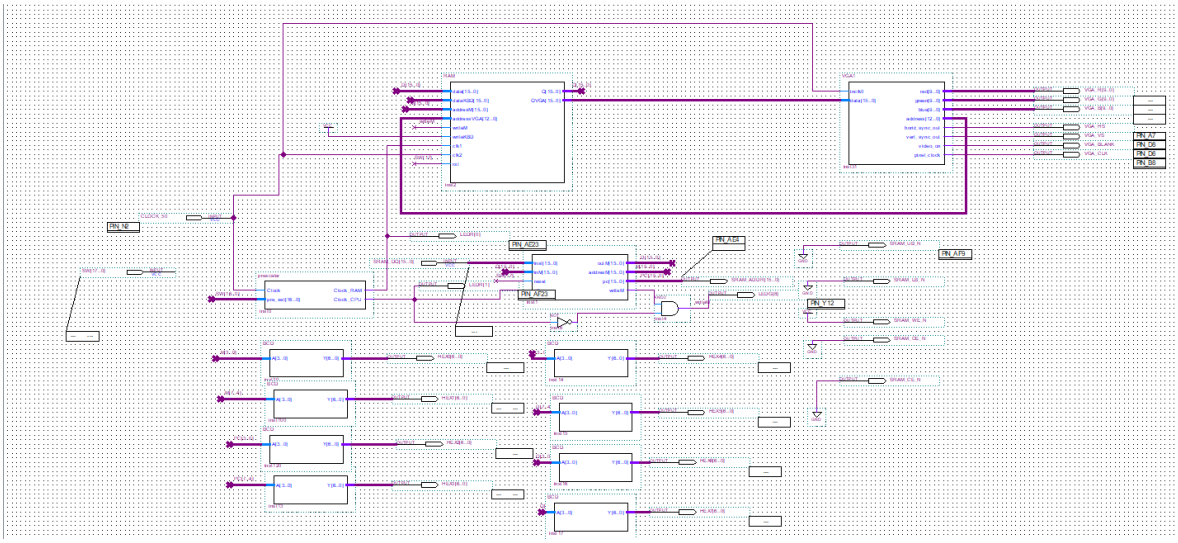
3.10 Повезивање компоненти Hack рачунара

Након креирања свих модула потребно је наше компоненте повезати да раде заједно. Унутар Quartus програма могуће је графички креирати verilog фајлове као симболе и те симболе повезати за улазним, излазним пиновима FPGA система, другим симболима и слично. Како би се ово постигло потребо је креирати нови фајл који има екстензију .bdf (Block Symbol File).



Слика 47. Фајл који се користи за креирање окружења

Када креирамо овај фајл отвориће нам се окружење у које можемо verilog фајлове убацивати као симобле и графичким путем спајати. Када убацимо све наше компоненте Hack рачунар ће изгледати као на слици:



Слика 48.Спајање свих компоненти Hack рачунара

3.11 Извршавање асемблерског програма

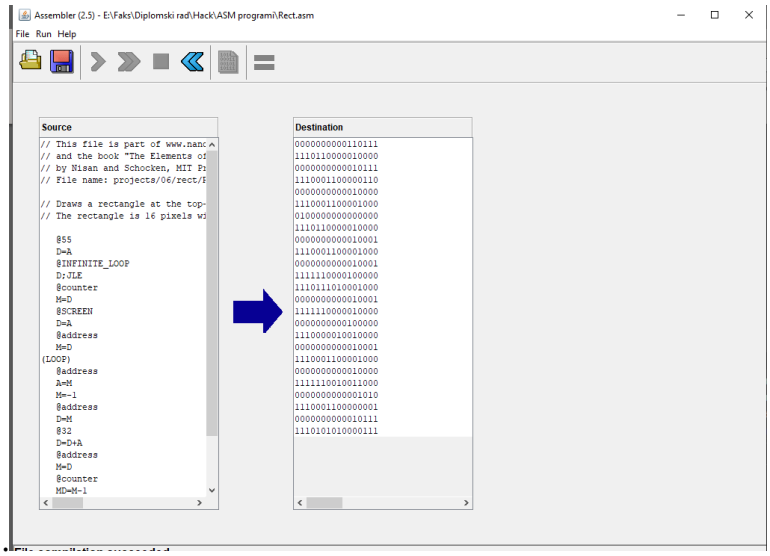
Сада када смо имплементирали Hack рачунар на FPGA систем, видећемо како се један прост асемблерски програм може покренути на рачунару. Као пример узећемо програм Rect.asm који креира квадрат одређене дужине почевши од леве ивице екрана код. Код можете наћи у прилогу

```
// This file is part of www.nand2tetris.org
// and the book "The Elements of Computing Systems"
// by Nisan and Schocken, MIT Press.
// File name: projects/06/rect/Rect.asm

// Draws a rectangle at the top-left corner of the screen.
// The rectangle is 16 pixels wide and 55 pixels high.

   @55
   D=A
   @INFINITE_LOOP
   D;JLE
   @counter
   M=D
   @SCREEN
   D=A
   @address
   M=D
(LOOP)
   @address
   A=M
   M=-1
   @address
   D=M
   @32
   D=D+A
   @address
   M=D
```

```
    @counter
    MD=M-1
    @LOOP
    D;JGT
(INFINITE_LOOP)
    @INFINITE_LOOP
    0;JMP
```

Слика 49. Rect.asm програм

Унутар курса nand2tetris дати су алати помоћу којих можемо асемблерски код претворити у машински



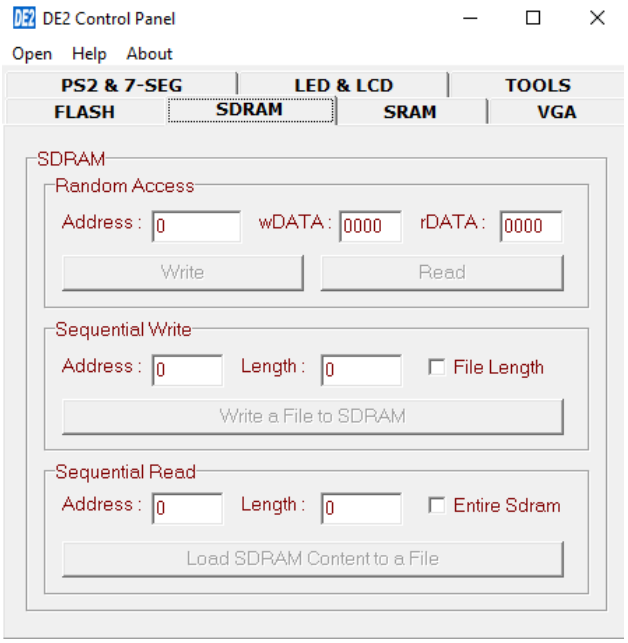
Слика 50. Алат за конвертовање асемблерски код у мапински

Након што је креиран машински код, потребо је машински код записати у бинарном фолдеру. Ово можемо постићи користећи прост Matlab програм са слике.

```
%Uciteti preko Import Data kao cell tipa text

fileID = fopen('Rect.bin','w');
for i=1:length(Code)
    X = uint16(bin2dec(Code(i)));

    Y = typecast(X, 'uint8');
    fwrite(fileID,Y);
end
fclose(fileID);
```

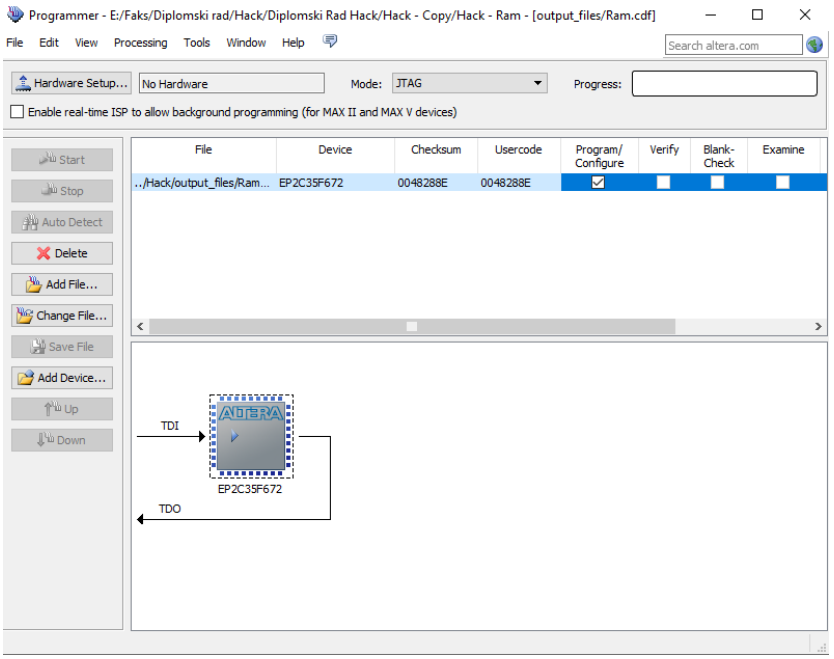
Слика 51. Matlab програм за конвертовање ASCII карактера у бинарни фајл

Када креирамо наш бинарни фајл потребно је садржај тог фајла уписати у SRAM меморију FPGA система. То се постиже користећи програм DE2 control panel.



Слика 52. DE2 contorl panel програм за попуњавање садржаја ROM меморије

Након што упишемо програм унутар SRAM(ROM) меморије потребно је покренути компајлирати програм унутар Qaurtus окружења и покренути програмер који ће да конфигурише плочу тако да се понаша као наш рачунар који смо програмски описали.



Слика 53. Програмер који кофигурише плочу

4. Закључак

У овом раду сагледали смо шта је све потребно да се реализује један једноставан рачунарски систем, а данас видимо колико су заправо наши рачунарски уређаји доста комплексне машине за које је било потребно године унапређивања и иновација да нам омогуће да радимо неке најелементарније ствари. Овај рачунар који у овом раду реализован има неке недостатке, није у могућности да креира програм унутар Jack језика и покрене због недостатка алата и лоших перформанси које располаже овај рачунар. Свакако је био један занимљив изазов покушати реализовати овај рачунарски систем о коме сам учио на предавањима из организације рачунарских система.

5. Литература

[1] Noam Nisan and Shimon Schocken: *The elements of computing systems Second edition* (2021) Massachusetts Institute of Technology

URL:https://www.nand2tetris.org/

[2] *Linda Null* Schocken: *Essentials of Computer Organization and Architecture Fifth edition (2003)*

[3] Предавања професора Др Уроша Пешовића

6. Прилог

Код можете пронаћи на GitHub линку: https://github.com/Nemanja5199/Hack-PC/tree/main/Hack