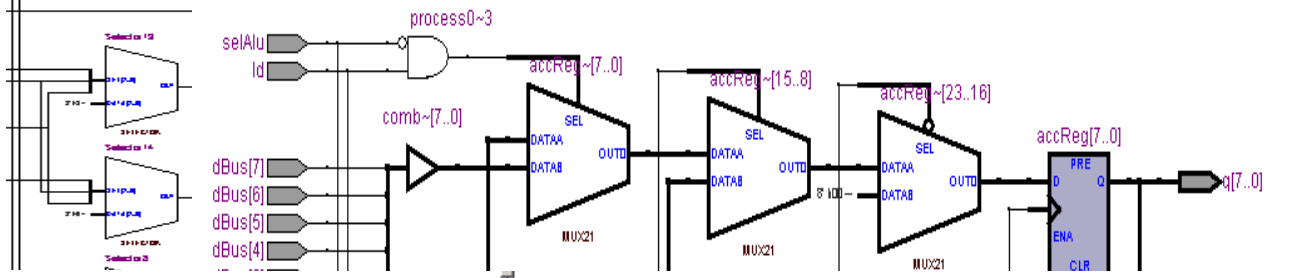
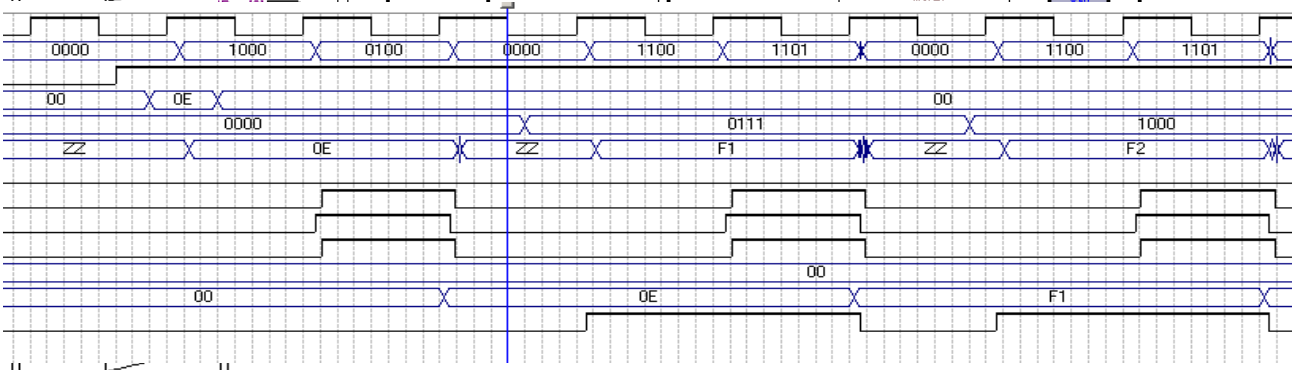
Георги К. Петров

Дизайн на цифрови електронни устройства с VHDL и Quartus II

1-во издание

част II - основи на VHDL в примери и задачи





УЧЕБНО ПОМАГАЛО

София 2010

e-Book първа версия 2010

Email: gwtt@abv.bg
http://www.instrumentation.hit.bg/

Авторът не поема отговорности относно повреди, причинени на хора, имущество и собственност по смисъла на безопасността и безотказността на продукти, или всякакъв вид методи, инструменти или идеи съдържащи се в изложения учебен материал.

Потребителите имат право да ползват свободно изложения код и примери изцяло на своя отговорност.

Текстообработката на настоящото учебно помагало е направена с OpenOffice. Ползвания за илюстрации софтуерен продукт Quartus II Web Edition може да се изтегли безплатно от сайта на компанията производител www.altera.com.

Тази страница умишлено е оставена празна.

СЪДЪРЖАНИЕ

Предговор

Настоящото учебно помагало е предназначено за използване от студенти и специалисти работещи в различни области на хардуерното и софтуерно производство. Основното предназначение е да се представи теоретичното съдържание на курсовете свързани с програмируемите логически устройства, които авторът води в департамент Телекомуникации на НБУ. Вниманието е фокусирано върху създаването на цифрови електронни устройства и модули чрез езика за поведенческо описание на цифрови интегрални схеми - VHDL (Very High Speed Integrated Circuit Hardware Description Language). За правилното използване и пълноценен дизайн на електронно оборудване, читателят би следвало допълнително да се запознае с други материали касаещи цифровата и аналогова схемотехника, а също така и да придобие допълнителни познания по програмиране на езици от високо ниво C/C++. Едновременно с това книгата може да бъде четена от съвсем начинаещи в областта, а при необходимост читателят да се обърне към цитираните литературни и Интернет източници. Разглежданите VHDL примери и приложените са оптимизирани за работа с интегрирана среда за разработка Quartus II Web Edition - производство на корпорация Altera. Самите VHDL кодове на могат да се ползват и в други среди за рабработка, като Web ISE производство на корпорация Xilinx.

Организацията е в 3 части:

- В първата част е направен кратък увод в булевата алгебра и цифровите интегрални схеми използвани за дизайн на електронно оборудване, този раздел е максимално олекотен и лесно разбираем дори за начинаещи;
- Втората част включва основните концепции и езикови конструкции на VHDL при реализацията на комбинационни и последователни логически блокове, както и основи за работа с Quartus II;
- Третата част разглежда приложение на VHDL в реализацията на цифрови модули за комуникация, сигнална обработка и програмирането на елементарно процесорно ядро.

Всички изходни кодове на разглежданите проекти могат да бъдат свалени свободно от сайта www.instrumentation.hit.bg. Ще Ви бъдем благодарни, ако изпращате своите препоръки и забележки на електронната поща на автора gpetrov@nbu.bg.

Увод

В част I на тази книга се запознахте с основните правила на булевата алгебра и придобихте представа за методите за синтез и дизайн на логически елементи, както и основите на оптимизацията на логически функции. Беше разгледан пример целящ реализацията на схемотехнически дизайн на комбинационна логичска функция чрез VHDL.

В част II ще се запознаете с основните концепции и понятия във VHDL. Абревиатурата VHDL буквално означава - език за описание на висикоскоростни цифрови интегрални схеми (Very High Speed Integrated Circuit Hardware Description Language). Възникването на този език, е част от проект на Щатският департамент по отбраната имащ за цел създаването на достатъчно гъвкав и прост инструмент за описание на действието на цифрови интегрални схеми ASIC (Application-specific integrated circuit). Основните версии на езика са дефинирани във следните стандарти IEEE 1076 VHDL–87, 1993 VHDL–93 и 1076 VHDL-08.

Необходимостта от специализиран език за дизайн на цифров хардуер е наложена от факта, че обикновенните схемотехнически описания – схеми и чертежи вече са прекалено обемисти и сложни за преглед и ръчен дизайн. Също така нуждата от непрекъснат ъпгрейд на хардуера налага ползването на инструменти за нанасяне на бързи изменения в схемата на устройствата. Като език, VHDL наследява много от свойствата на езикът ADA, който е стриктен императивен обектно ориентиран език за програмиране от високо ниво. Въпреки това много от концепциите във VHDL не може да бъдат открити в другите езици за програмиране. HDL езиците позволяват поведенческо и схемотехническо (структурно) описание предимно на цифрови интегрални схеми и устройства. Съществуват разновидности на езика /излизащи извън обхвата на част II на тази книга/ подходящи за дизайн и на аналогово цифрови интегрирани схеми, като **VHDL-AMS** (Analog and mixed-signal IEEE 1076.1).

След усвояване на материала от част II ще можете да създавате и управлявате проекти с Quartus II Web Edition. Ще умеете да избирате и адаптирате вашия дизайн към определена хардуерна платформа. В обхвата на примерите влизат: дизайн на комбинационна логическа схема със схемотехнически редактор, преминаване от схемотехническо описание във VHDL код и обратно, симулация и програмиране на устройството. В отделни примери са разгледани VHDL описанията на: тристабилен драйвер, пример с използване на конкурентни процеси, сигнали и променливи, тригери- RS, D, JK, мултиплексор и демултиплексор, паралелен запомнящ регистър, брояч, регистър с паралелен вход и сериен изход, цифров компаратор.

Основи на работата с Quartus II Web Edition

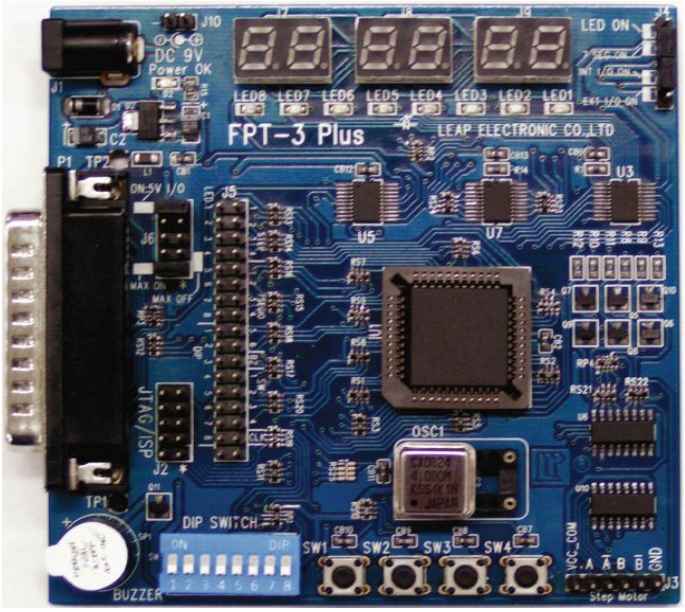
Моделирането и описанието на цифрови системи включват разработката и симулацията им от логически елементи на ниско ниво. Това означава, че всеки цифров блок с произволна сложност следва да може да бъде създаден с логически елементи И-НЕ или ИЛИ-НЕ. Напредъкът в интегралната електроника и интегрираните софтуерни среди за разработка на софтуер и хардуер позволяват да създаваме т.н. „системи върху чип“ и дори пълен дизайн на печатни платки. Този процес е сложен и налага ползването на специализирани софтуерни продукти за бързо и качествено изработване на работни прототипи и въвеждането им в серийно производство. За конкретното описание на цифровите блокове и схеми се ползват различни езици за описание на цифров фардуер, като най-популярните от тях са: VHDL, Verilog, AHDL, JHDL, SystemC и други. В рамките на тази книга ще разгледаме особеностите на дизайна на цифрови схеми и блокове с VHDL, като няма да се спираме на детайли на тяхната физическа реализация. Този език е избран, тъй като се поддържа от практически всички компилатори на по-големите производители на програмируеми логически устройства. Написаните кодове могат да се ползват върху програмируеми матрици с ниска интеграция PLA(D) (Programmable Logic Array-Device), комплексни програмируеми логически устройства CPLD (Complex Programmable Logic Device) и по-сложни устройства реализирани с FPGA (Field Programmable Gate Array).

Интегрирания софтуер за проектиране на цифров хардуер, с който ще се занимаваме и за който са адаптирани разгледаните в тази книга примери се нарича **Quartus II Web Eddition** и е производство на корпорация Altera. Софтуерът може да се изтегли безплатно от сайта на компанията – www.altera.com. С него може да се извършва пълноценен дизайн на системи базирани на CPLD и FPGA чипове без да се налага употребата на други програми. Предлагат се различни опции за лицензиране съвместими с Windows и Linux платформи, като платената версия на софтуера позволява дизайн на електронни схеми с всички произвеждани от Altera програмируеми логически устройства. Софтуерът интегрира инструменти за следните основни дейности свързани с проектирането на цифрови системи:

- структорно, поведенческо и схемотехнично описание на проектите
- логически синтез на проекта
- топологическо проектиране и оптимизация за конкретен чип
- симулация и времеви анализ
- анализ на консумацията
- програмиране

Инсталирането на Quartus II не се различава по нищо от инсталирането на

всяка друга програма под Windows. Размера на инсталационния файл за версия 9.1 е около 1400MBytes, следва да изтеглите и пакета ModelSim с обем 600MBytes, който се инсталира допълнително след основния софтуер. След инсталиране, софтуерът изисква да укажете мястото на лицензния файл, който може да получите безплатно след регистрация в сайта на Altera. Обръщам внимачие, че исканият от вас лценз следва да включва опцията за ползване на ModelSim. Основните принципи на работа с Quartus II ще разгледаме последователно в няколко поредни примера, като след пример три ще дефинираме само условията на задачата, която следва да изпълним и ще се концентрираме над усвояването на основните езикови конструкции с VHDL. Първият пример е даден с цел по-лесно запознаване с функциите и възможностите на програмата, като са адаптирани към ползване на методите за спемотехнически дизайн. Вторият пример запознава с основните концепции на VHDL. Третият пример разглежда методите за преминаване от един тип описание и дизайн в друг и съвместното ползване на компоненти.



Фиг. 1 Демонстрационна платка FPT-3 Plus

Повечето упражнения в част II на книгата позволяват да бъдат адаптирани към демонстрационната платка **FPT-3 Plus** (Фиг. 1) на цена до 70$, която може да бъде поръчана и доставена от www.farnell.com, като за България препоръчвам това да стане през фирма Комет Електроникс – www.comet.bg. За програмирането и се използва стандартен паралелен принтерен кабел - LPT, като това е по-евтината конфигурация. Кабелът не влиза в комплекта на FPT-3 Plus и при закупуването му следва да обърнете внимание да има налични всичките 25 проводника. В този случай ще ползвате интерфейсът за програмиране – **ByteBlaster II**, който е интегриран в самата платка. Тъй като повечето компютри днес не разполагат с паралелен принтерен порт, следва да купите USB-LPT конвертор, но няма гаранция че ще може да ползвате всеки

модел конвертори. Препоръчва се да се ползва или стандартна PCI към LPT платка, която не струва повече от 25-30$, или кабела за програмиране - **USB byte blaster (250$)**. Дефакто чрез този кабел се реализира стандартният индустриален интерфейс за програмиране на програмируемите логически устройства на Altera наричан **JTAG** (Joint Test Action Group). Първоначално създаден за проверка изправността на печатни платки, днес този интерфейс се ползва широко при програмирането и конфигурирането на множество микроконтролери и логически матрици.

Авторът препоръчва също така и следните други китове, имащи сравнително ниска цена и са масово прилагани за нуждите на обучението: **MODULE, FPT-1 WITH ALTERA CHIP**, както и по-скъпите FPGA китове: **KIT, FPGA, CYCLONE II DEVELOPMENT – DK-CYCII-2C20N.** Упражненията могат също така да бъдат имплементирани и на XILINX: **KIT, STARTER, COOLRUNNER-II, LO-PWR – SK-CRII-L-G**, като в настоящата книга не се дава описание за работа с развойната система. *Следва да се отбележи, че JTAG интерфейсите са предназначени за определен тип устройства на определена фирма производител, като не са съвместими. Ето защо когато избирате своята първа платка, независимо дали това ще бъде микроконтролер или матрица следва да проверите дали самата платка поддържа и има вграден интерфейс за програмиране, дадените по-горе примери за развойни системи имат вграден интерфейс за програмиране.*

Самата платка FPT-3 Plus представлява набор от интегрални схеми, бутони и индикация с които могат да се извършват упражнения за проектиране и програмиране на логически устройства и проверка на тяхната функционалност. Платката интегрира чип CPLD от фамилията MAX7000S (EPM7064 с 1250 логически клуча и общо 64 макроблока), към него се съединени драйвери за управление на вградените 7 сегментни светодиодни индикатори, 8 светодиода, приемане на команди от 8 ключета и 4 бутона, 4 проводен драйвер за управление на стъпков електродвигател и рейки с изведени всички входове и изходи на матрицата.

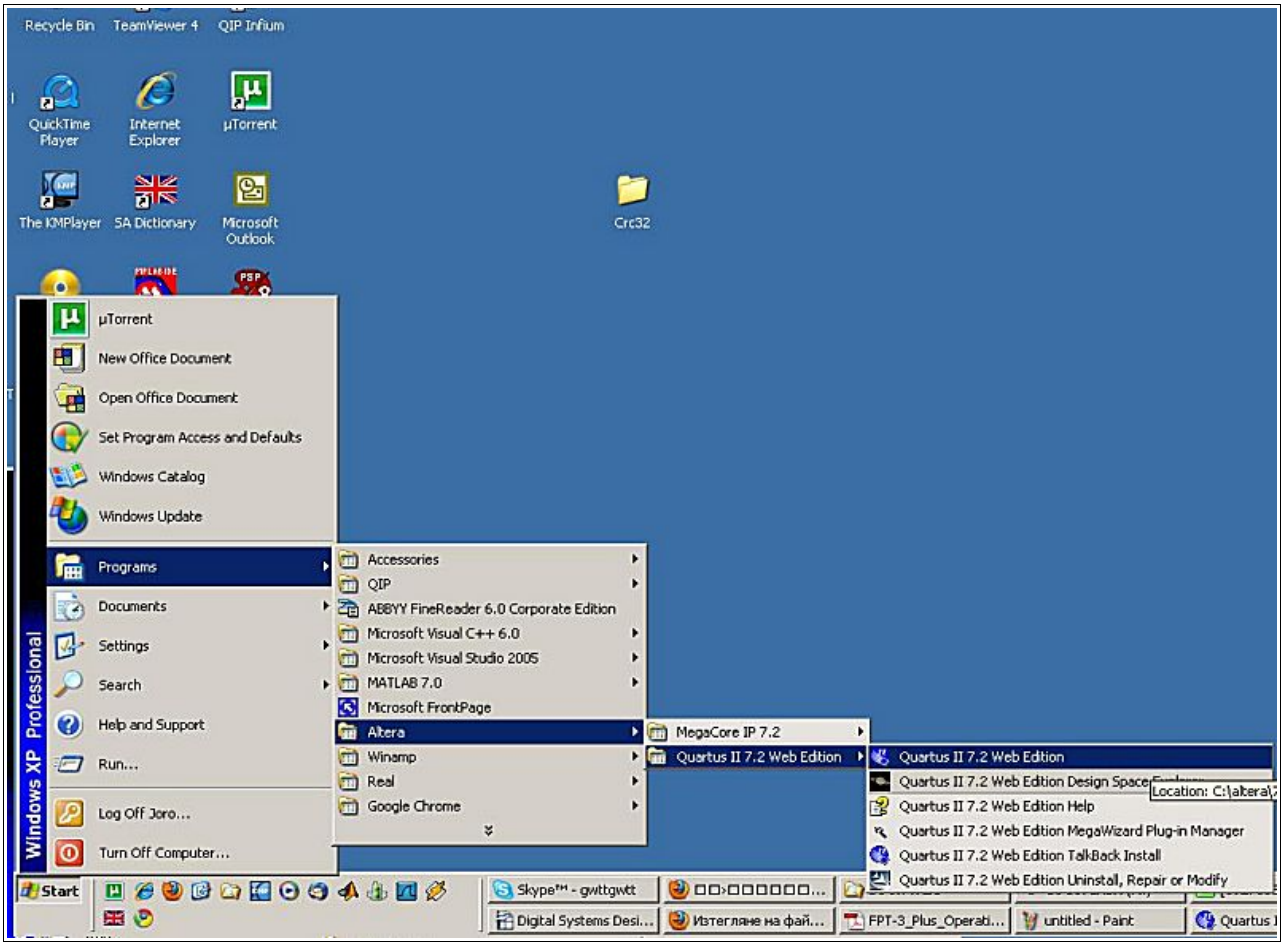
Схемотехнически дизайн на цифрово устройство

В този първи пример ще разгледаме как използвайки схемотехническия редактор на Quartus II може да създадем, симулираме и програмираме елементарна комбинационна логическа схема с 2 входа и 8 изхода. Примерът е адаптиран към използването на демонстрационен кит FPT-3 Plus с PLD MAX7000 имащ 64 логически блока. След приключване на упражнението ще можете да създавате и менажирате нов проект с Quartus II адаптиран към конкретно семейство PLD, да създавате нова принципна електронна схема използвайки готови елементи И, ИЛИ, НЕ и т.н., да създавате симулационна

последователност и да извършвате симулация, да разглеждате реалната схема на устройството, да следите за използването на русурсите в избраната хардуерна платформа и да програмирате реален CPLD чип монтиран върху демонстрационна платка FPT-3 Plus. Останалите упражнения в книгата обхващат само частта проектиране и симулация на схемите, като адаптирането към конкретна платка и чип следва да се извърши допълнително.

Създаване на нов проект

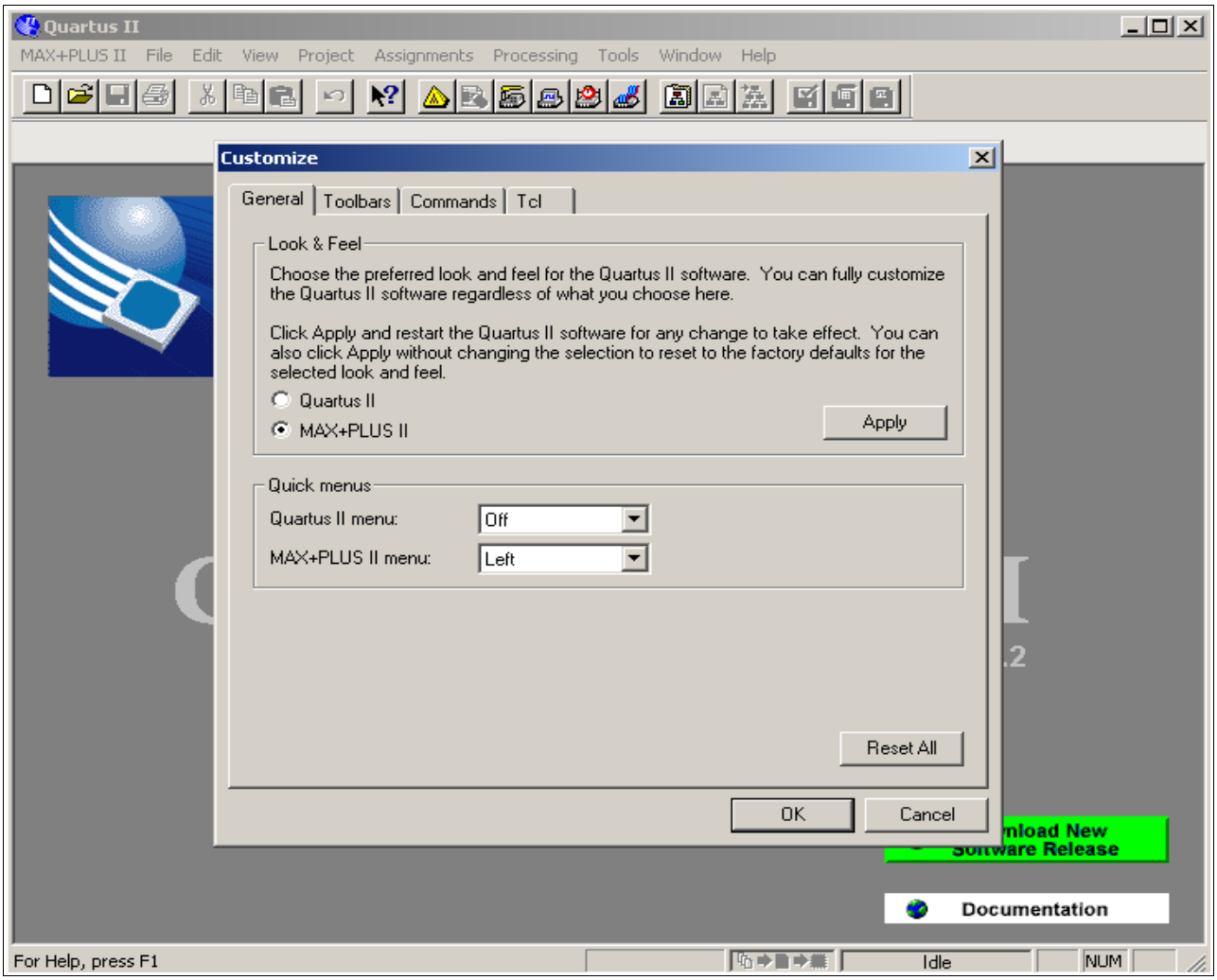
След инсталиране на Quartus II вие сте готови да започнете своят първи проект. Стартирането на програмата може да се извърши по стандартния начин през бутона Start или от иконата на работния плот на вашия компютър. При първото стартиране софтуерът ще изведе рекламен екран и ще предложи опция да продължите работата с 30 дневен лиценз, или да укажете мястото на 190 дневен лицензионен файл получен от Altera. Инструкции за неговата инсталация може да получите от сайта на компанията.



Фиг. 2 Стартиране на Quartus II

След успешно инсталиране на лицензионния файл следва да рестартирате програмата. Ако предходната операция е правилно извършена повторното стартиране ще приключи без съобщение за липса на лицензионен файл. Ако

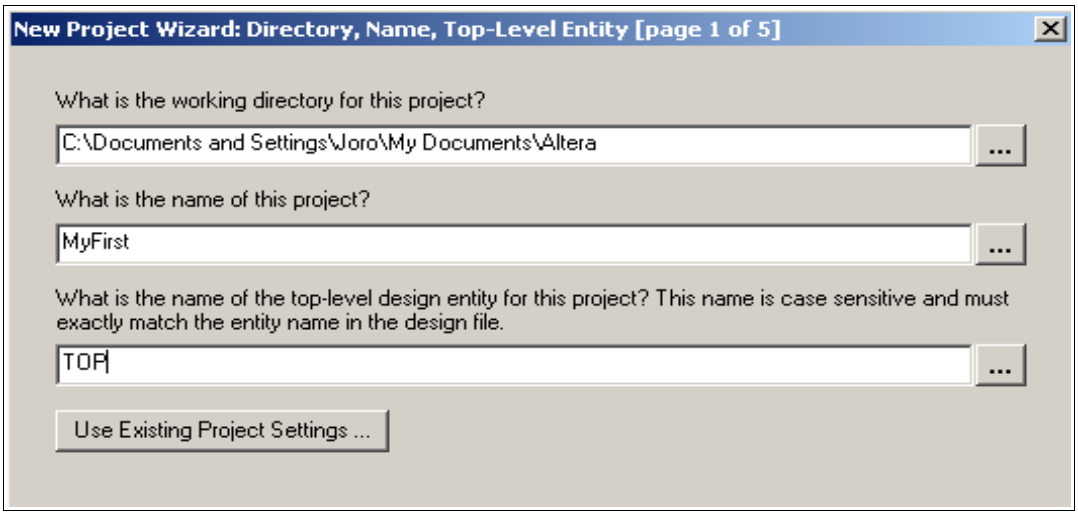
всичко е приключило успешно ще видите прозорец подобен на показания по-долу (Фиг. 3). Важно е да се отбележи, че Quartus II ви позволява да ползвате два стандартни изгледа на софтуера. Първият е нов и се нарича Quartus II, другият и по-удобен изглед на екрана позволява на програмата да изглежда като предната версия MAX+PLUS II. Упражненията в тази книга са правени с екранни снимки в режим MAX+PLUS II. Двата режима на работа ви предоставят равни права над използването на различни инструменти, въпрос на време е да свикнете да работите с двата екранни изгледа. Независимо от това дали в процеса на първо стартиране сте указали екранът да бъде в режим MAX+PLUS II или Quartus II, винаги може да направите промяна. Това става през менюто **Tools->Customize**, както е дадено на долната .



Фиг. 3 Първоначално стартиране на Quartus II

Създаването на нов проект започва с дефинирането на основните му параметри: директория и папка в която ще се съхраняват файловете и архивните копия на проекта, име на проекта и име на основното ентити на проекта. Понятието ентити ще бъде дискутирано по-късно, най-общата

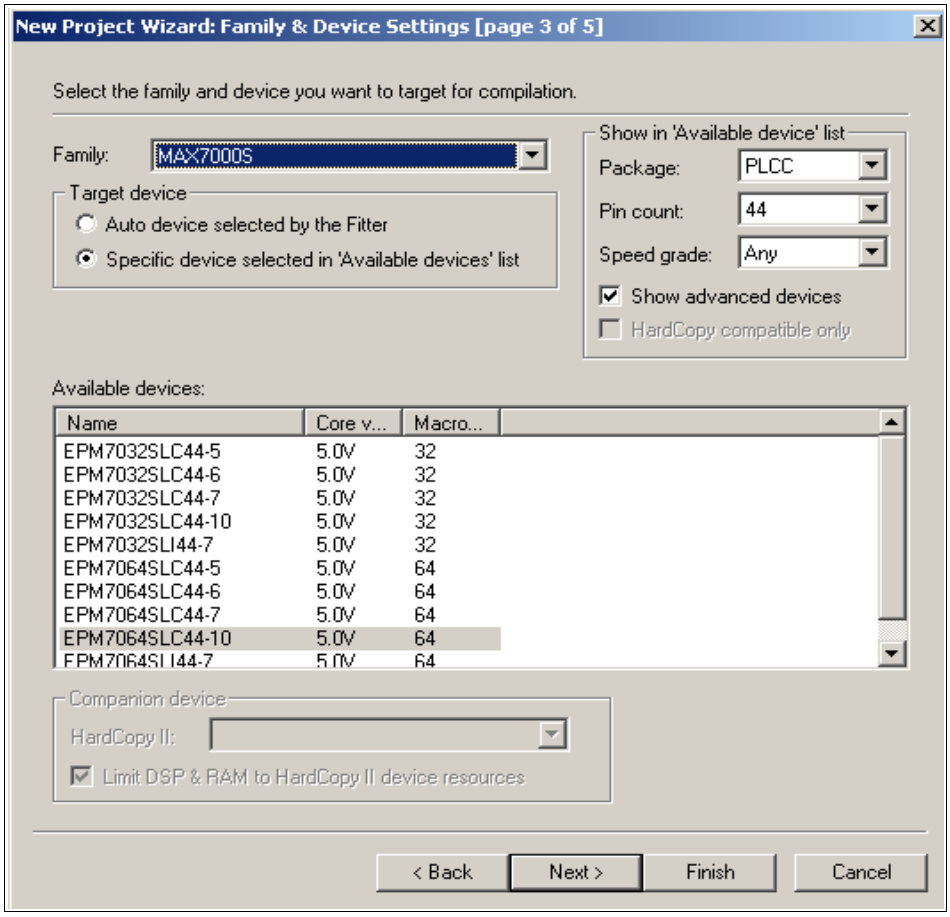
представа за него може да добиете ако приемете, че проектираното от нас устройство представлява черна кутия с входове и изходи. Тази черна кутия се нарича ентити. Главното ентити на проекта е най-външната обвивка на черната кутия, в него може да има влъжени по-малки черни кутии, а те от своя страна да са изградени от логически блокове, каито са направени от базови логически елементи. За да създадем първия проект ще ползваме помощникът за генериране на проекти (Wizard), който е достъпен през менюто **File->New Project Wizard** (Фиг. 4). Ще се отвори показания по-долу прозорец. Избираме бутонът **Next**, като на следващата стъпка ще трябва да изберем папка където ще съхраним проекта. За упражненията ще ползваме папката **My Documents**, в която създайте под папка **Altera** с под папка **Project1**. Изберете за име на проекта **Project1** и изберете за име на неговото основно ентити **TOP**.



Фиг. 4 Екран от Project Wizard

На следващата стъпка помощникът за създаване на проекти ще ви попита дали имате вече създадени файлове, които ще искате да ползвате в новия проект. Прескочете тази стъпка чрез натискане на бутона **Next**. Следващата стъпка изисква да изберете фамилията процесори върху които ще реализираме съответния дизайн. От падащото меню **Family** изберете **MAX7000S**, от **Package** изберете **PLCC** – това е корпуса на чипа монтиран върху демонстрационната платка, за **Pin Count** изберете **44.** Това е броят на изводите на чипа, в полето **Speed Grade** оставете параметър **Any**. След това от наличния списък с чипове изберете **EPM7064SLC44-10 5V** с **64** макро клетки (Фиг. 5). Натиснете бутона **Next** и попуснете поредните 2 стъпки избирайки **Next**. Тъй като на този етап дизайнът не се нуждае от повече настройки, за да продължим натиснете бутона **Finish**. Това действие приключва процеса на начално инициализиране на работната среда и параметрите на проекта. Създадохме работна папка и база данни с файлове съдържащи основните параметри на нашия дизайн. За да приключиите работата си на този етап, затворете прозореца на Quartus II. Ако

сега отидете в папката на проекта, която се намира в **My Documents/Altera/Project1** ще откриете файловете с описание на проекта. Това са: **Project 1.qpf** и **TOP.qsf** и под папка **db**. Тези файлове се ъпдейтват автоматично от средата за разработка и не е препоръчително ръчното им манипулиране. Ако все пак желаете да прочетете тяхното съдържание използвайте командата **Open With** и изберете програмата **Notepad**. В процесът на дизайн в тази папка ще бъдат създавани и съхранявани много видове файлове, не бива да променяте тяхното съдържание и местоположение с програми и инструменти не влизащи в пакета на Quartus II.

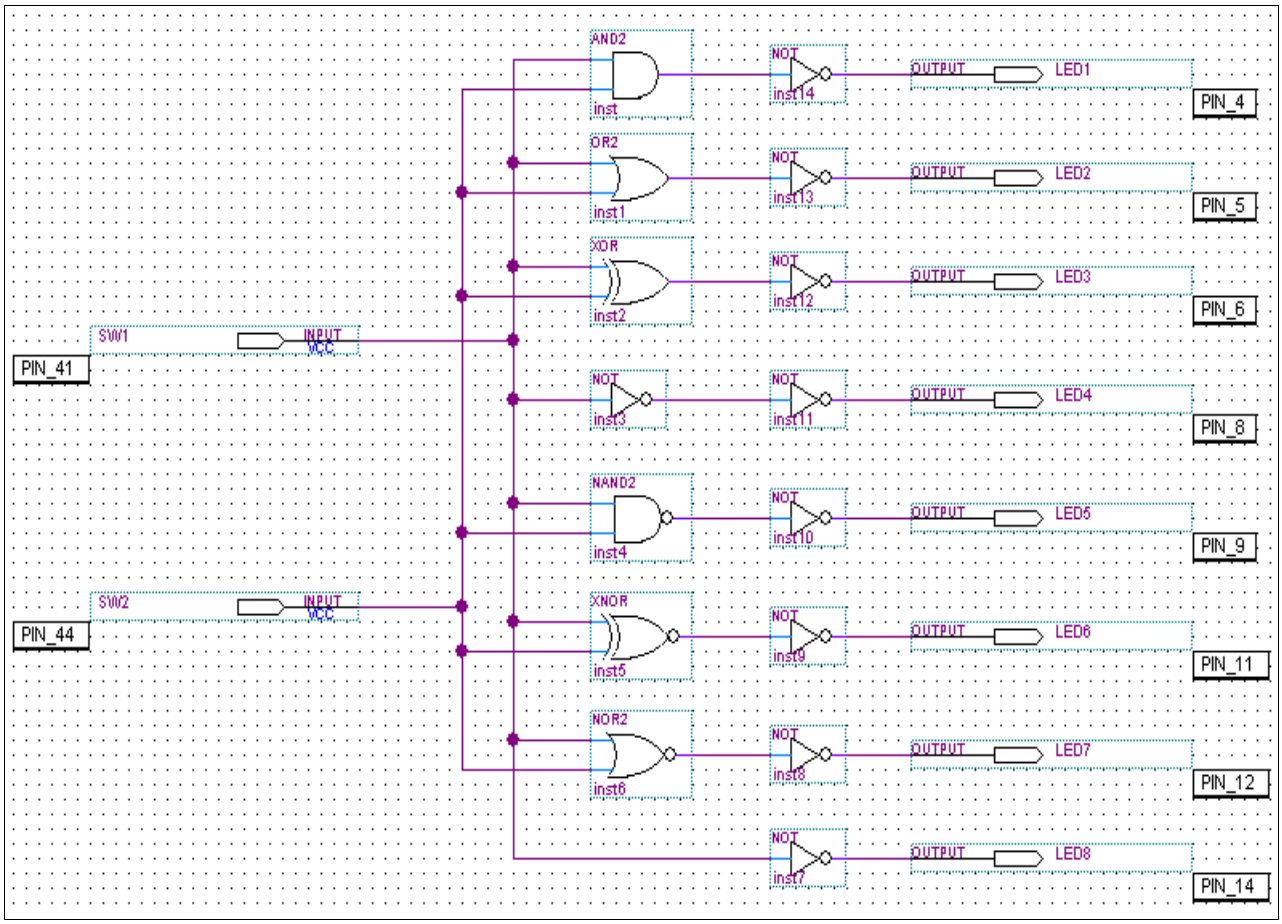


Фиг. 5 Избор на CPLD матрица за конкретния дизайн

Схемотехнически дизайн на комбинационна логическа схема

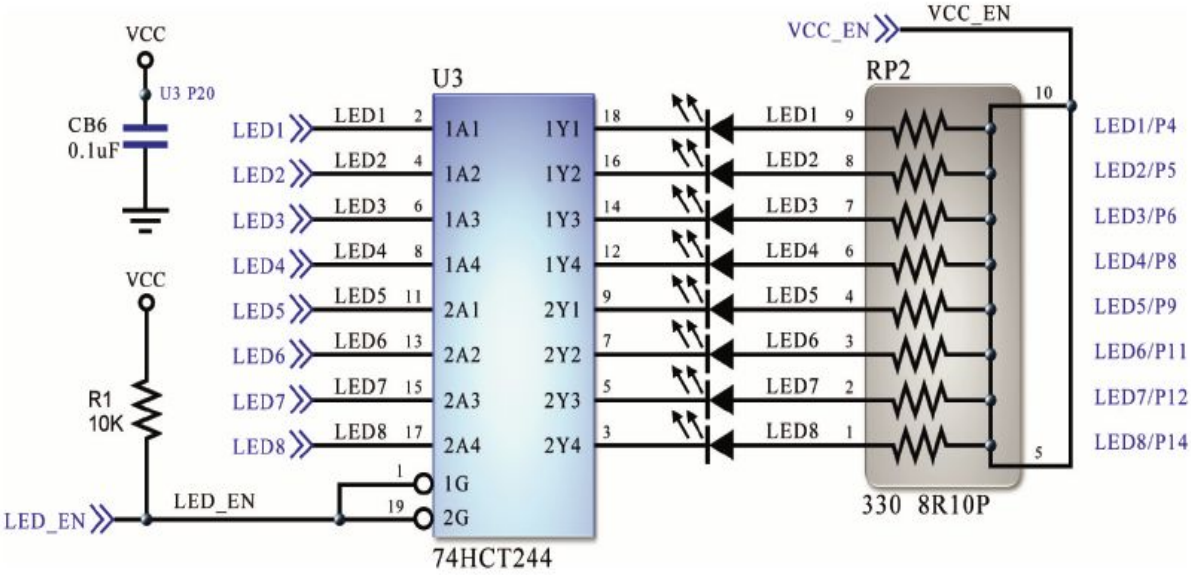
Макар и елементарна тази задача ще ви позволи да добиете представа за дизайна на електронни схеми с помоща на схемотехнически редактор. Този тип дизайн е удобен при създаването на реплики-копия на съществуващи интегрални схеми, чиито принципни схеми имаме, или да създаваме техни модификации. Освен това този режим е особено подходящ при дизайн на

сложни цифрови устройства, като микроконтролери и комуникационни интерфейси в блоков вид. Схемотехническия дизайн не е удобен при дизайн на прекалено сложни комбинационни и последователни логически схеми с много елементи и сложна структура, там за предпочитане е ползването на VHDL редактора. Следва да се знае, че по всяко време можете да преминете от символния файл или схемата във VHDL описание и обратното. Това позволява да се ползват различни методи за дизайн на устройствата, обезпечавайки по този начин максимално удобство за потребителите. Структурната схема, която следва да реализираме ще има 2 входа (SW1 SW1) и 8 изхода (LED1, LED2,... , LED8), а нейната принципна схема (Фиг. 6).



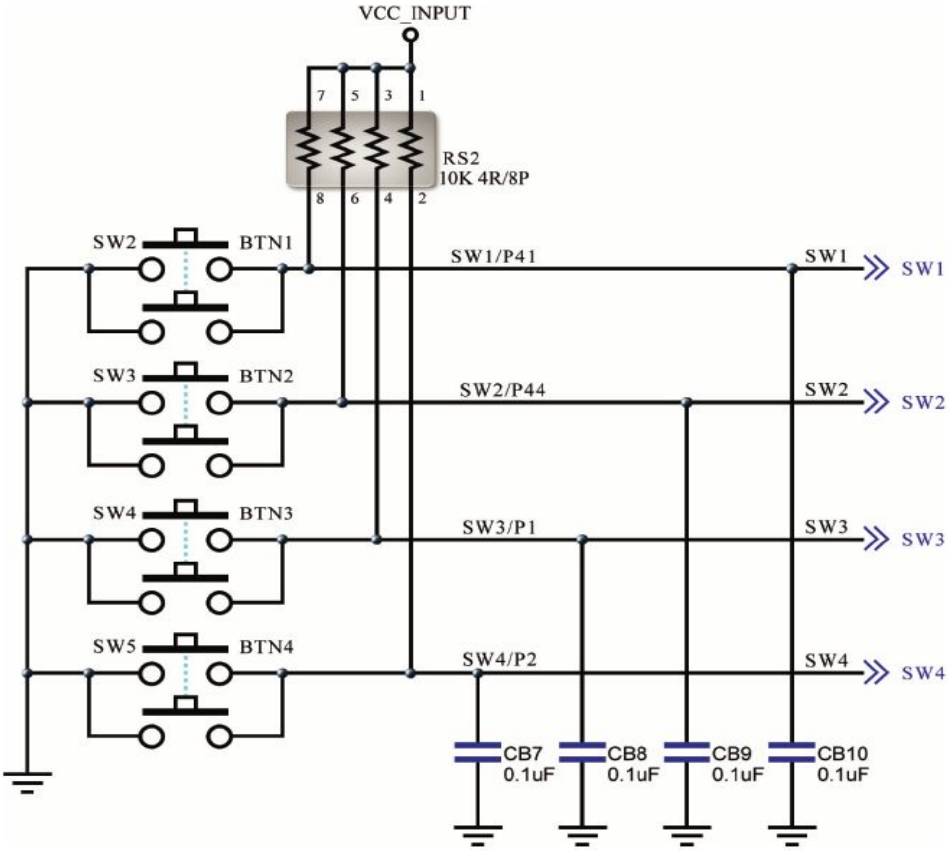
Фиг. 6 Схема на комбинационно логическо устройство

Обърнете внимание на включените в дизайна инвертори преди изходните пинове. Това се прави с цел правилната индикация на логическите функции върху демонстрационната платка. Изходните буфери реализирани с интегрална схема 74HCT244 (Фиг. 7) управляват светодиоди свързани към захранващото напрежение. Когато на техния вход се подаде логическа единициа ('1') през тях няма да протича ток и респективно те няма да светят.



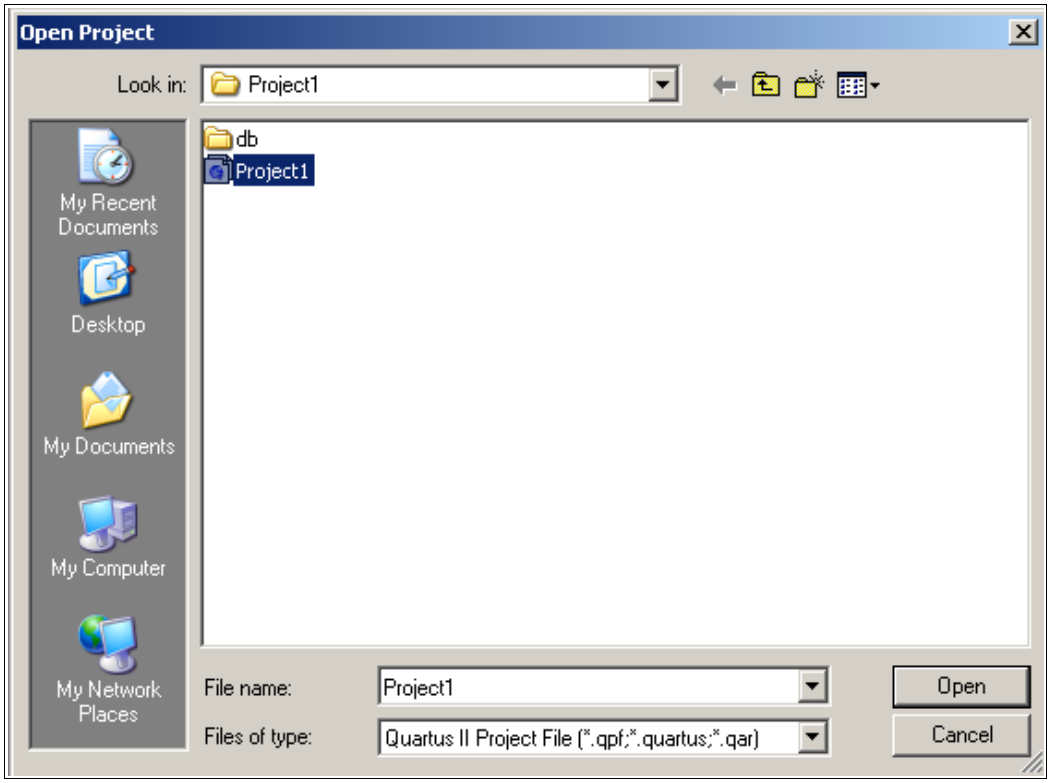
Фиг. 7 Управление на светодиодите върху платка FPT-3 Plus

Схемата на свързване на входните пинове **SW1** и **SW2** е дадена (фиг. 8), когато бутоните са в нормално състояние – не са натиснати, сигналът на входовете има стойност '1', а когато те са натиснати сигналите имат стойност '0'.



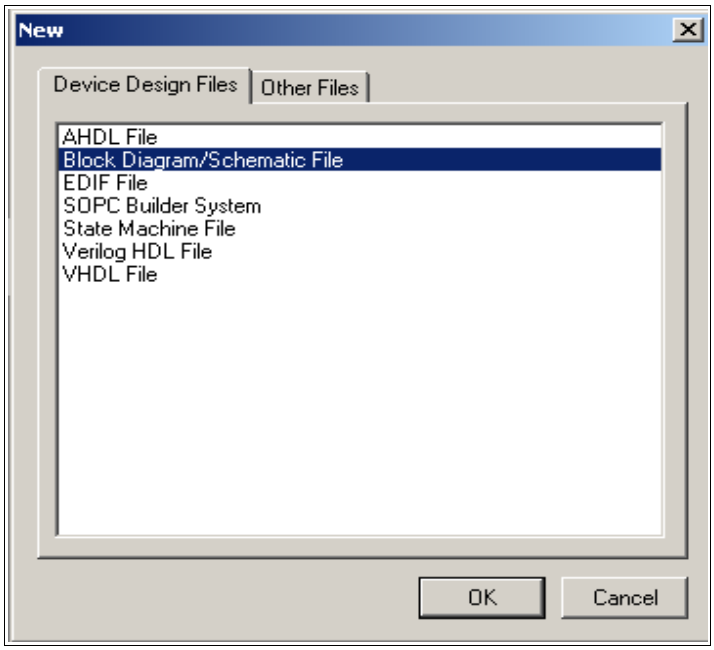
Фиг. 8 Схема на свързване на бутоните

Въпреки добавените кондензатори в схемата на бутоните, подобен вид входове не са подходящи за реализацията на бързо действащи схеми. При натискане на бутоните започва преходен процес на стабилизация на металната пластинка, която осъществява превключването, това довежда до поява на смущения. Тези смущения, ако бъдат подадени директно на входовете на цифровата схема и могат да доведат до непредвидими последици и грешни резултати. След като вече създадохме нов проектен файл можем да пристъпим към дизайна на нашата първа схема (Фиг. 6). За целта следва да стартирате Quartus II и да отворите вече създадения проект. Това може да направите от менюто **File->Open Project**, използвайте системата за навигация, за да отидете до **My Documents/Altera/Project 1**, където съхранихме заготовката на нашия проект. Софтуерът автоматично прилага филтър на показваните файлове, като единствения който може да отворите е файла на проекта – **Project1.qpf**.



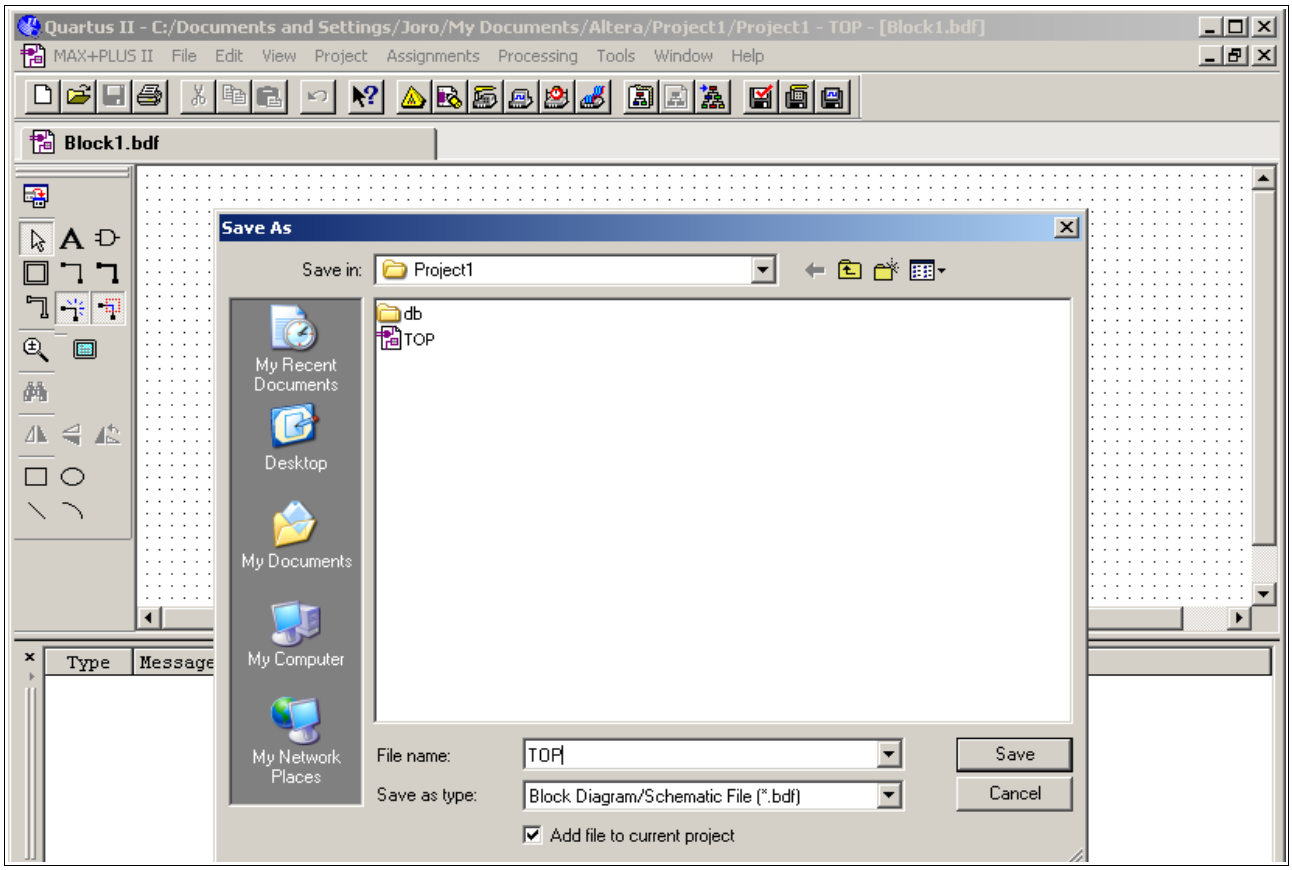
Фиг. 9 Отваряне на проектния файл

За да създадем първият схемотехнически файл използвайте менюто **File->New**, като от отворилия се прозорец изберете **Block Diagram/Schematic File** и натиснете **OK** (Фиг. 10). Това действие ще създаде нов файл за схемотехнически дизайн във вашия проект. Желателно е да го съхраните в момента на създаване, обърнете внимание, че тъй като нашият проект ще има само 1 файл съдържащ схемотехническо описание то той именно ще трябва да носи името на главното ентити на нашия проект - **TOP**.



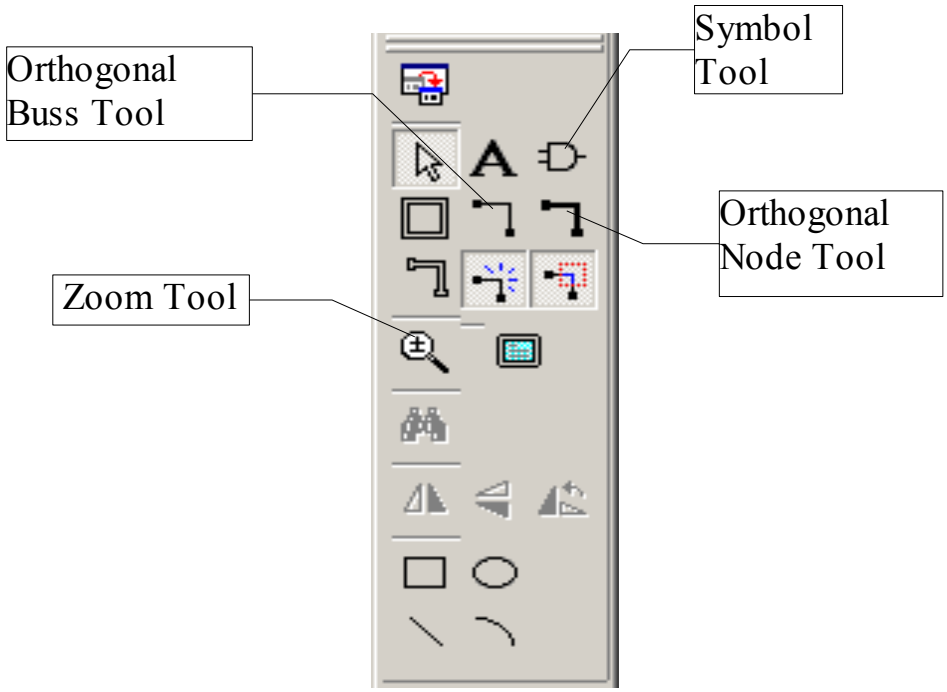
Фиг. 10 Добавяне на нов схемотехнически файл в проекта

За да съхраните новосъзданения файл използвайте менюто **File->Save As**, като в полето **File name** напишете **TOP.bdf** (Фиг. 11) и натиснете **Save**.



Фиг. 11 Съхраняване на новосъздадения схемотехнически файл

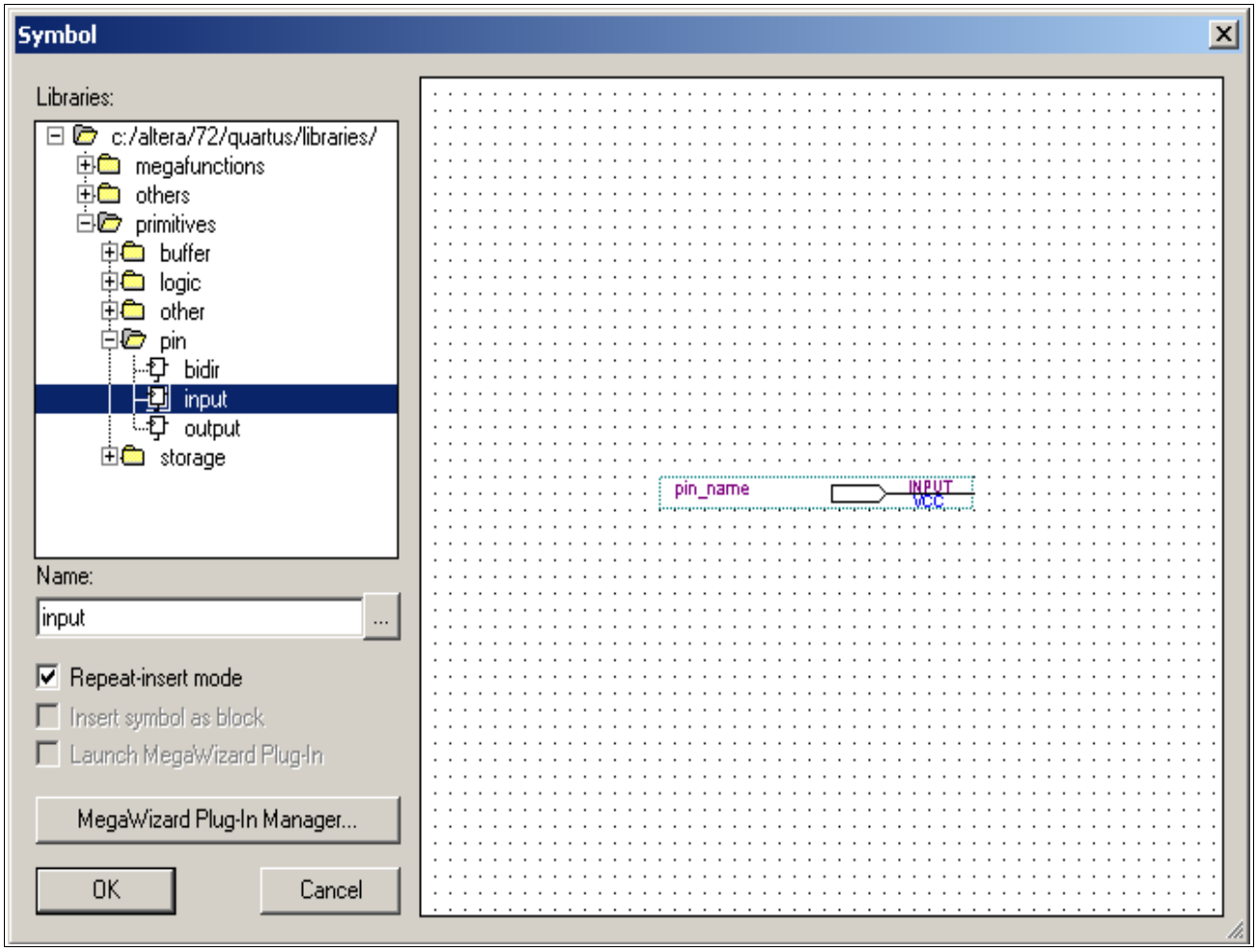
Ще използваме вертикалната лента с инструменти в схемотехническия редактор за да избираме нужните електронни компоненти и да ги свързваме в схема. Основните инструменти, които ще ползваме за това упражнение са: **Zoom Tool**, **Symbol Tool** и **Orthogonal Node Tool** (Фиг. 12). Първият инстлумент ви позволява да увеличавате или намалявате мащаба на схемата, правейки възможно виждането на отделни детайли или цялата схема в едър план. Чрез вторият инструмент се извиква библиотека с компоненти, а чрез третия ще свързваме отделните електронни компоненти. Освен това важен инструмент е **Orthogonal Buss Tool** чрез който се изчертават шини, това са множество свързващи проводници. Шините се използват при свързването на компоненти имащи многобитови входно изподни портове. За да прекратите употребата на всеки един избран инструмент следва да натиснете бутона **Esc** на клавиатурата. Може да тренирате самостоятелно употребата и на останалите инструменти, но те няма да са необходими за даденото упражнение.



Фиг. 12 Инструменти за работа в схемотехническия редактор

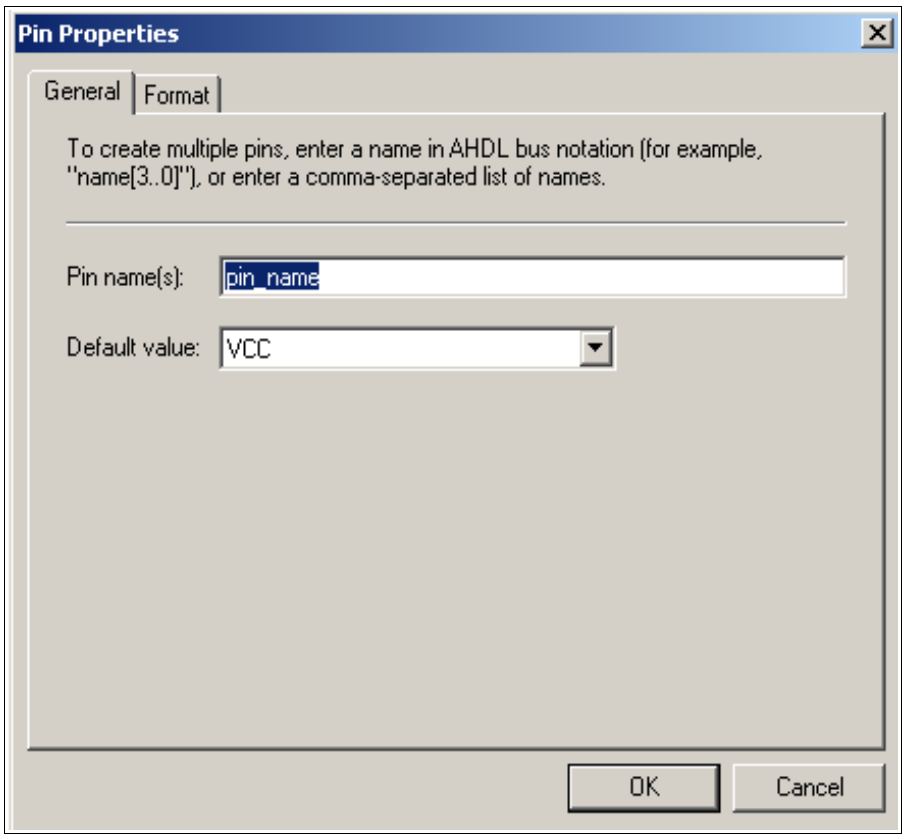
Ще използваме **Symbol Tool**, за да селектираме нужните ни логически елементи. Когато натиснете бутона **Symbol Tool** в лентата с инструменти ще се появи даденият по-долу прозорец (Фиг. 13). Може да разлистите списъка с компоненти, или да напишете името на компонента който ви трябва в тектовото поле **Name**. В случая ще трябва да изберем и поставим 2 входни пина и 8 изходни. След като изберете нужния компонент натиснете **OK**. Тази команда ще ви върне в прозореца на проекта където свободно ще може да поставите селектирания компонент. Поставете 2 входни порта, и след това 8 изходни, имената на които са output. За да прекратите изчертаването на даден компонент,

следва да натиснете бутона **Esc** на клавиатурата. Използвайте **Symbol Tool**, за да добавите и останалите логически компоненти, чиито имена са: **AND2, OR2, XOR, NOT, NAND2, XNOR, NOR2**.



Фиг. 13 Symbol – прозорец за избор на логически елемент

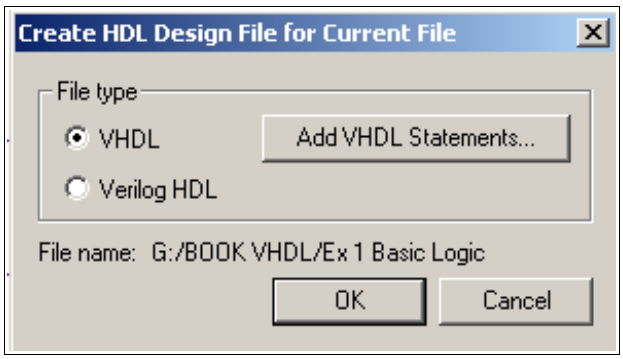
За да преименувате входните и изходните портове съгласно заданието, кликнете два пъти с левия бутон на мишката върху всеки един от тях, като при това ще се отвори по-долния прозорец (Фиг. 14). В полето **Pin name(s)** изберете **SW1**, за първия входен порт и **SW2** за втория входен порт, повторете тази операция и за изходните портове **LPT1**, **LPT2**,... , **LPT8**. Това преименуване се налага в следните случаи: първо то е задължително при именуване на входно изходни портове, второ когато ползвате елемент от стандартна библиотека чието име влиза в конфликт със запазените думи ползвани във VHDL за описание на определени действия. Добра идея е да имате предварително представа колко входни и изходни пина ще ползва вашия дизайн, така може да ги групирате и да изберете за име някакъв идентификатор, с който по-лесно ще ги различавате в последствие, например: RAM_ADR_0, RAM_ADR_1 … и т.н.



Фиг. 14 Преименуване на входните и изходни пинове

След като сте избрали нужните електронни компоненти и сте ги подредили по показания на начин следва да ги свържете както беше дадено на фигура 6. За свързване използвайте инструмента **Orthogonal Node Tool,** като последователно свържете всички логически елементи. Обърнете внимание на точността на изпълнение, там където два проводника следва да се свържат се получава по-голяма тъмна точка, там където два проводника се пресичат такава точка отсъства.

След като приключите със схемотехническия дизайн следва да проверите схемата за груби грешки и оставени несвързани електронни компоненти. Това действие се осъществява от менюто **Processing->Analyse Current File**, ако всичко е изчертано правилно този процес ще приключи със съобщение за успех. Натиснете **OK**, за да продължите. Тъй като схемата е сравнително елементарна не би следвало да имате проблеми на този етап. След като имаме готово схемотехническо описание на комбинационната логическа схема, лесно можем да преминем във VHDL описание. Това става чрез командата от менюто **File->Create Upgate->HDL File for current file** (Фиг. 15). Това ще предизвика отваряне на дадения по-долу прозорец, изберете типът на файла да бъде VHDL, той като примерите в тази книга са на VHDL. Натискането на бутона **OK** за да се генерира изходния VHDL код. Файлът ще носи името на схемотехническия файл, но с ***.vhd** разширение **TOP.vhd**.



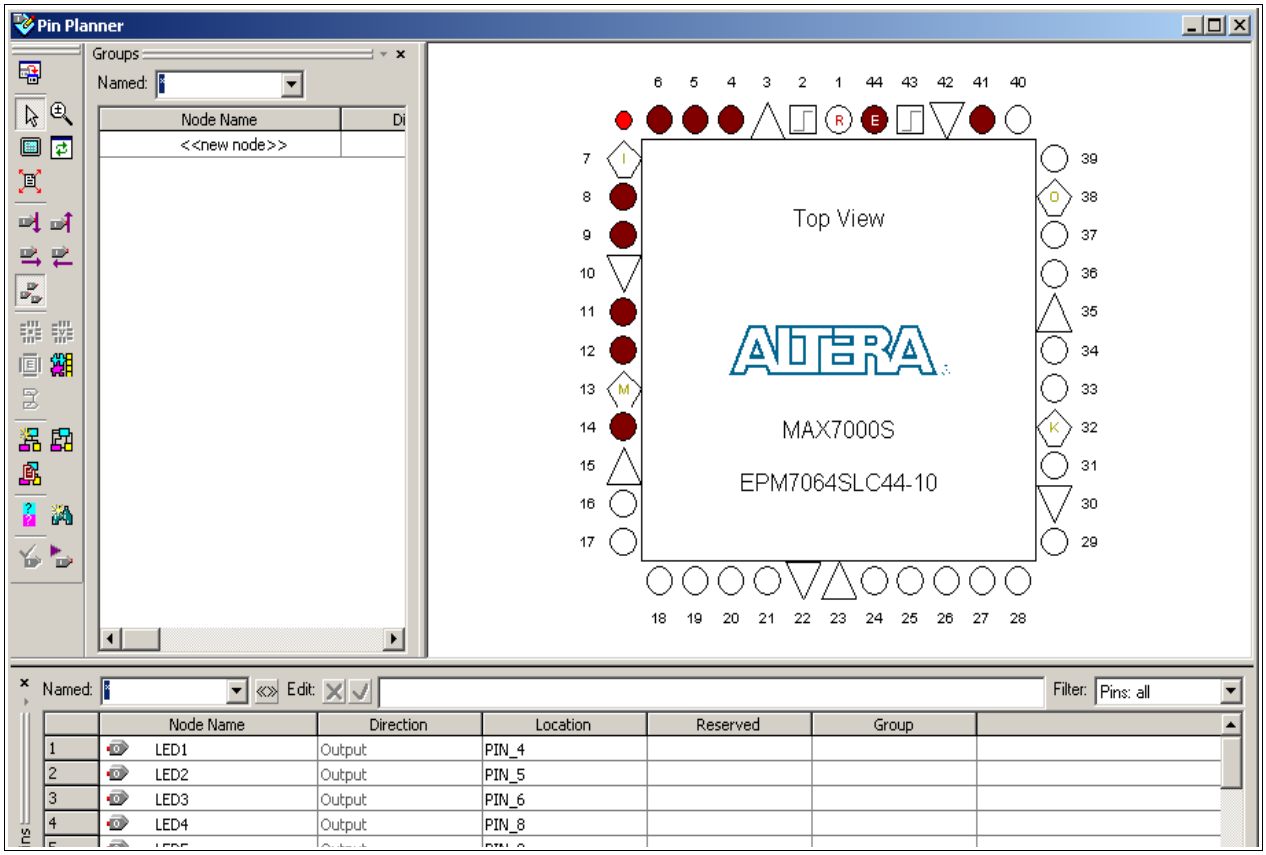
Фиг. 15 Генериране на VHDL код от схемотехническото описание на проекта

За да видите съдържанието на този файл използвайте командата от менюто **File->Open** и от отворилия се списък с файли изберете **TOP.vhd**. На екрана ще се отвори прозорец подобен на текстовия редактор **Notepad**. Уголемете прозореца и разгледайте съдържанието на файла. На този етап следва да обърнете внимание на това, че са ползвани 3 цвята за изписване на текста във файла. **Зеления цвят *представлява коментари към кода***, **синия цвят *показва така наречените служебни за VHDL думи***, а **черния цвят *е самото описание на кода и ползваните променливи, портове, сигнали, логически операции и други***.

Затворете прозореца **TOP.vhd** и от **Windows Explorer** отидете в папката **My Documents/Altera/Procet1,** като внимателно селектирате и изтриете файла **TOP.vhd**. Изтриваме тои файл, тай като на този етап ще работим само и единствено със схемотехническия редактор на Quartus II. Наличието на два файла съдържащи еднакви дефиниции на основното ентити на проекта – **TOP** ще доведе до грешки в процеса на компилация. За да се избегне това на този етап изтриването на **TOP.vhd** файла ще бъде най-лесният за вас начин за справяне с този проблем. След като вече имаме създаден проект и схемотехническо описание на комбинационно логическо устройство. Характерно за него е следното, входните данни се подават на два входа SW1 и SW2, като изходните данни ще бъдат извеждани на 8 светодиода. Всеки един светодиод се явява изход на една елементарна логическа функция. Тъй като всяка логическа функция се реализира от отделен логически елемент, всички те ще се изпълняват паралелно, тоест едновременно. Това е основно предимство при дизайна на цифров хардуер с VHDL и PLD матрици. Ако трябваше да напишем подобна програма за конвенционален микроконтролер, всяка една от логическите операции щеше да се реализира последователно, това би отнело много повече време, и като резултат повече консумация на електричество. Независимо от ниското ниво на сложност на реализираната задача, този пример показва възможността с PLD да се създават устройства съчетаващи много функции, които се изпълняват едновременно.

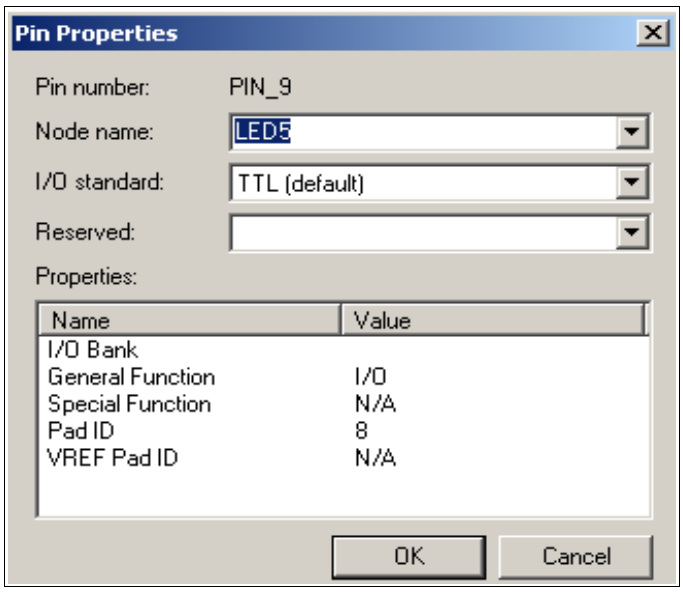
Планиране на входните и изходните портове

Нашия дизайн не може да се счита за завършен след като сме приключили компилацията на проекта. Не бива да забравяме, че всеки дизайн трябва да се свърже с точно определен хардуер, в случая с платка **FPT-3 Plus**. Това става с инструмента **Pin Planer**, който може да се стартира от менюто **Assignments->Pins.** Тази команда ще отвори дадения по-долу прозорец.



Фиг. 16 Използване на **Pin Planer** за избор на входно изходни пинове

Прозорецът **Pin Planner** се използва за назначаване на определени входно изходни пинове на PLD чипа към съответните входно изходни портове на схемотехническия или VHDL дизайн. Ползвания от нас чип е с малък брой пинове софтуерът не поддържа групово именуване на няколко съседни пина под формата на шина. Всеки входно изходен пин следва да се селектира и именува поотделно. Избирането на даден пин не може да става произволно, тъй като PLD чипът е запоен на печатна платка с определена топология и следва да съблюдаваме нейното описание. Самото именуване на пиновете става чрез двукратно кликване с ляв бутон на мишката върху избран пин (Фиг. 17). В полето **Node name** следва да се въведе името на входно изходния порт на нашия дизайн. В случая това е **LED5**. Същото действие следва да се извърши за всички останали портове ползвани в дизайна.



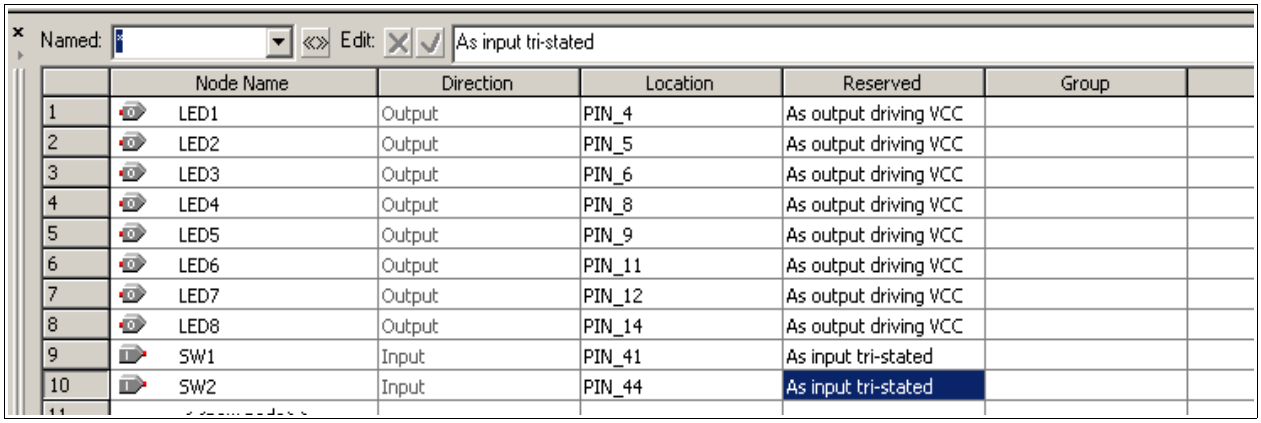
Фиг. 17 Прозорец за преименуване на пиновете

За правилното именуване на пиновете ползвайте дадената по-долу таблица 1. Веднъж направен файлът с описания на входно изходните портове за дадена платка и чип той може да се експортира и използва повторно.

PIN	Connect	PIN	Connect
3、15、23、35	VCC	10、22、30、42	GND
4	LED1	16	DIP1
5	LED2	17	DIP2
6	LED3	18	DIP3
8	LED4	19	DIP4
9	LED5	20	DIP5
11	LED6	21	DIP6
12	LED7	24	DIP7
14	LED8	25	DIP8
31	7SEGIO1	41	SW1
33	7SEGIO2	44	SW2
34	7SEGIO3	1	SW3
36	7SEGIO4	2	SW4
37	7SEGIO5	26	STEP1
39	7SEGIO6	27	STEP2
7	TDI	28	STEP3
38	TDO	29	STEP4
13	TMS	40	BZ
32	TCK	43	CLK

Табл. 1 Описание на входно изходните пинове на FPT-3 Plus

В колоната **PIN** е означен номерът на крачето на самия CPLD чип, а в колонката **Connect** е даден кой елемент от платката управлява или свързва. Така например **LED1** ни позволява да управляваме **светодиод №1**, а **SW1** да прочитаме данни от бутон **SW1**. Ако притежаваме списъка с портовете, можем да използваме табличният изглед на **Pin Planner** и да добавим нужните ни пинове, като в колона **Node Name** въвеждаме имената на портовете от нашия схемотехнически дизайн, в колона **Location** избираме показаните по-долу номера на пинове, а в колона **Reserved** избираме от падащото меню **As output driving VCC** – за изходите и **As input tri-stated** – за входовете (Фиг. 18). След като въведем всички портове извършете съхраняване на така създадения файл.

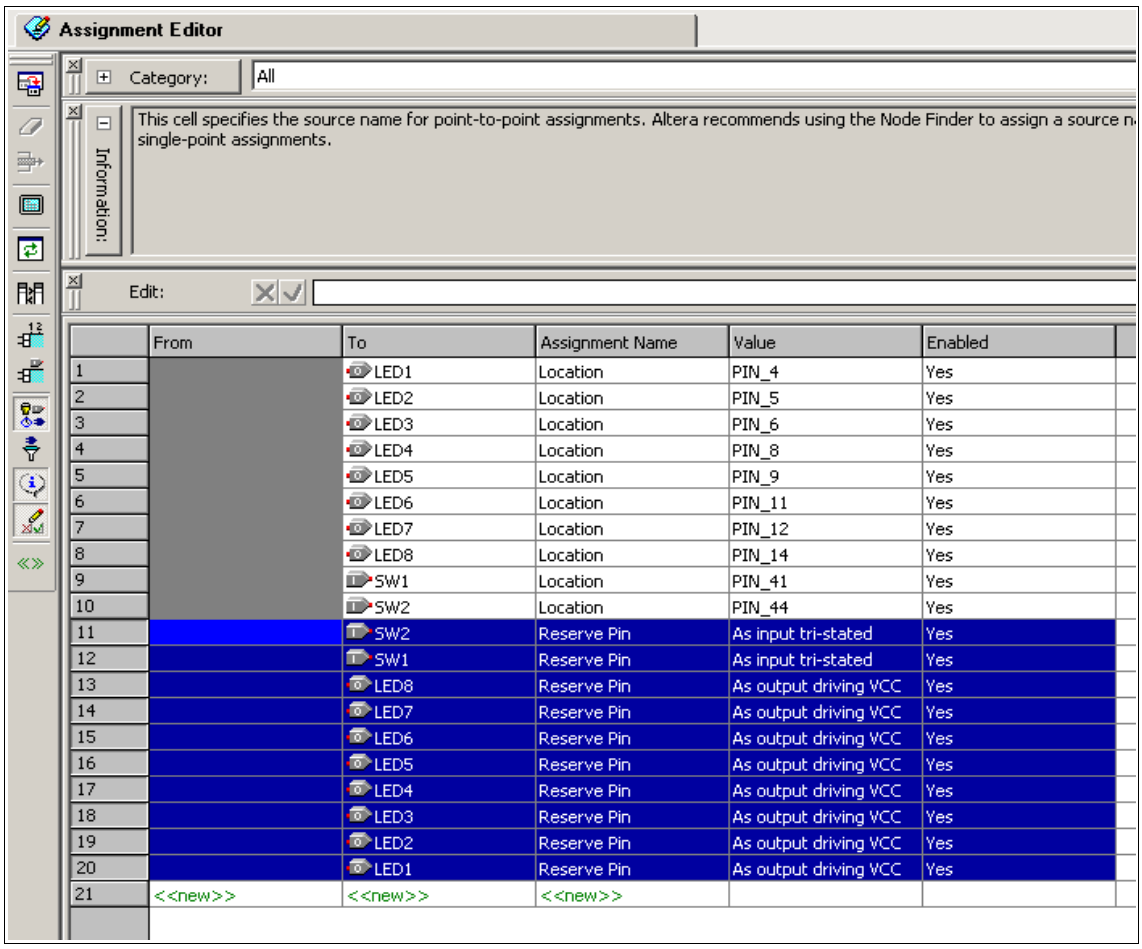


	Node Name	Direction	Location	Reserved	Group
1	LED1	Output	PIN_4	As output driving VCC	
2	LED2	Output	PIN_5	As output driving VCC	
3	LED3	Output	PIN_6	As output driving VCC	
4	LED4	Output	PIN_8	As output driving VCC	
5	LED5	Output	PIN_9	As output driving VCC	
6	LED6	Output	PIN_11	As output driving VCC	
7	LED7	Output	PIN_12	As output driving VCC	
8	LED8	Output	PIN_14	As output driving VCC	
9	SW1	Input	PIN_41	As input tri-stated	
10	SW2	Input	PIN_44	As input tri-stated	

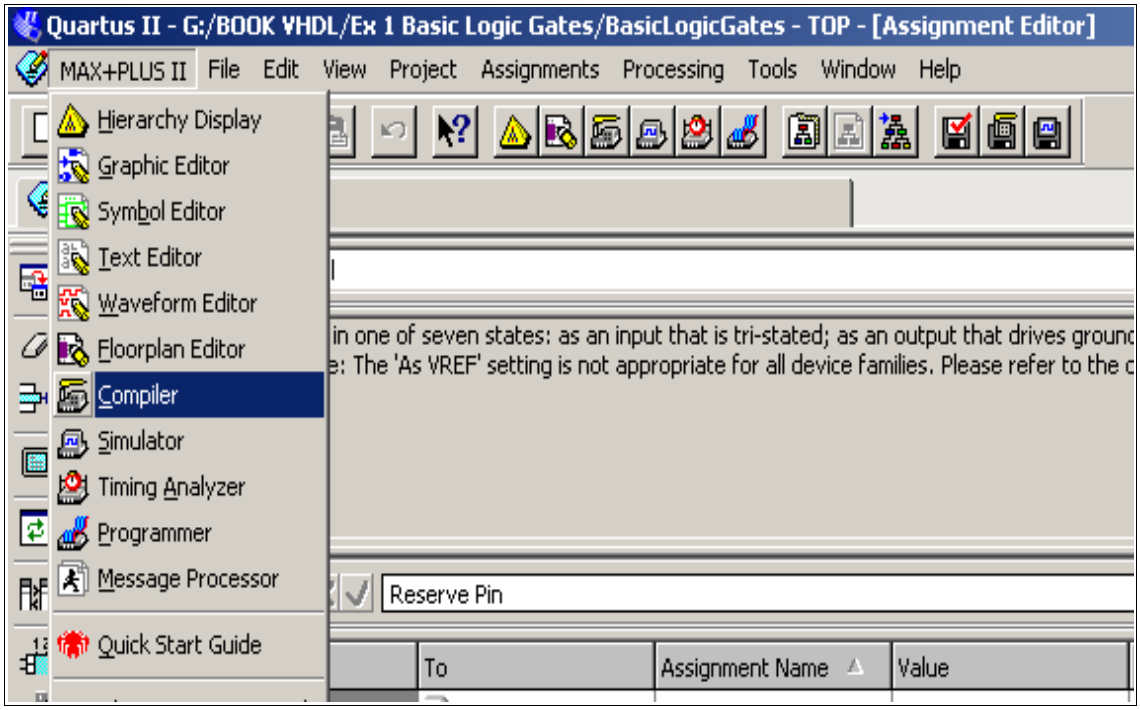
Фиг. 18 Табличен изглед на Pin Planner

Ако на този етап опитаме да компилираме проекта ще получим съобщения за грешки предизвикани от дублиране на имена за едни и същи входни пинове. За да отстраним този досаден проблем ще ползваме друг инструмент за планиране на пиновете, той е достъпен от менюто **Assignments->Assignment Editor**. Ще се отвори прозореца **Assignment Editor** показан по-долу. Кликнете веднъж в колона **Assignment Name**, за да подредите типовете пинове. Ще видите, че едни и същи входни и изходни портове от схемата са присъединени към различни видове изходи. Селектирайте онези изходи /селектирането става чрез натискане на левия клавиш на мишката в първата колона с номерата на портовете и задържане докато маркирате всички редове от 11 до 20/, както е показано по-долу. С клавиша **Delete** от клавиатурата изтрийте избраните пинове, след това можем да пристъпим към компилация на проекта (Фиг. 19).

Компилацията на проекта се извършва по няколко начина: от специално предназначеното меню **MAX PLUS II->Compiler** (Фиг. 20) или от менюто **Processing->Start Compilation**. Ако до този момент сте извършили всичко правилно, компилацията ще завърши успешно. При възникване на грешки, те ще се изведат в най-долния екран на Quartus II. При двукратно кликване върху тях с мишката ще бъдете позиционирани на мястото където грешката е открита. След редакция на грешките пристъпете отново към компилация на проекта.

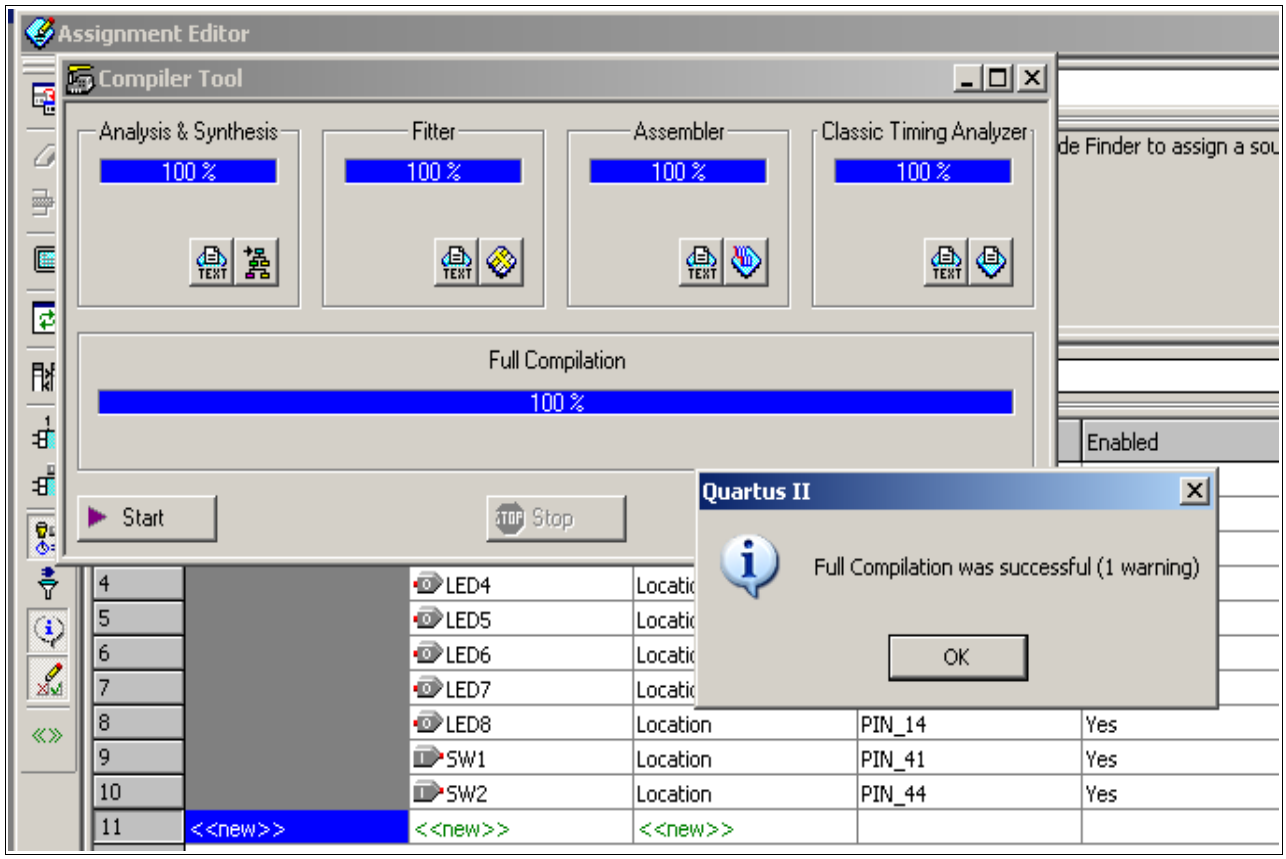


Фиг. 19 Изтриване на дублирани пинове



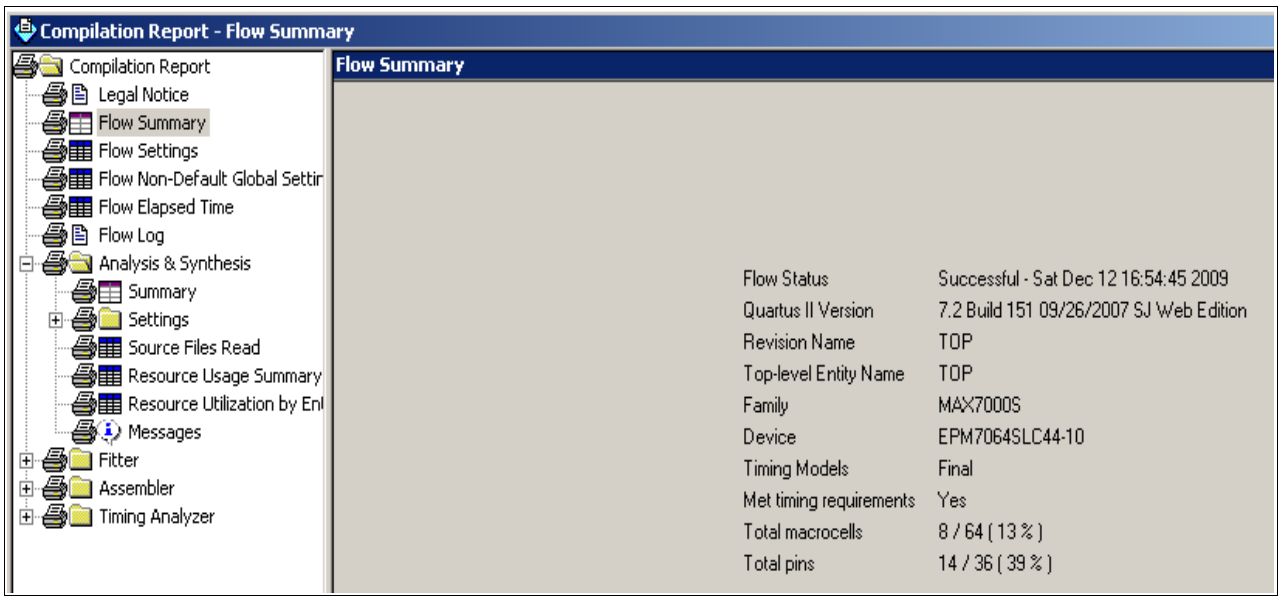
Фиг. 20 Компилиране на проекта MAX PLUS II->Compiler

Самият процес на компилация отнема 4 стъпки: анализ и синтез на схемата, филтър на схемата към даден чип и списък с портове, ассемблиране и класически времеви анализ на схемата. Ако компилацията спре на първата стъпка, най-вероятно има синтактична грешка във VHDL описанието или са забравени плаващи-несвързани входове и изходи на логически компоненти (Фиг. 21). Филтърът проверява дали избрания чип съответства на нуждите на текущия дизайт – брой портове, брой макроклетки. По време на дизайн и симулация на големи проекти се препоръчва да не се извършва асемблиране и времеви анализ на схемата, тъй като това отнема значително време. За да изпълните бърза компилация ползвайте менюто **Processing->Start->Start Analysis and Synthesis.**



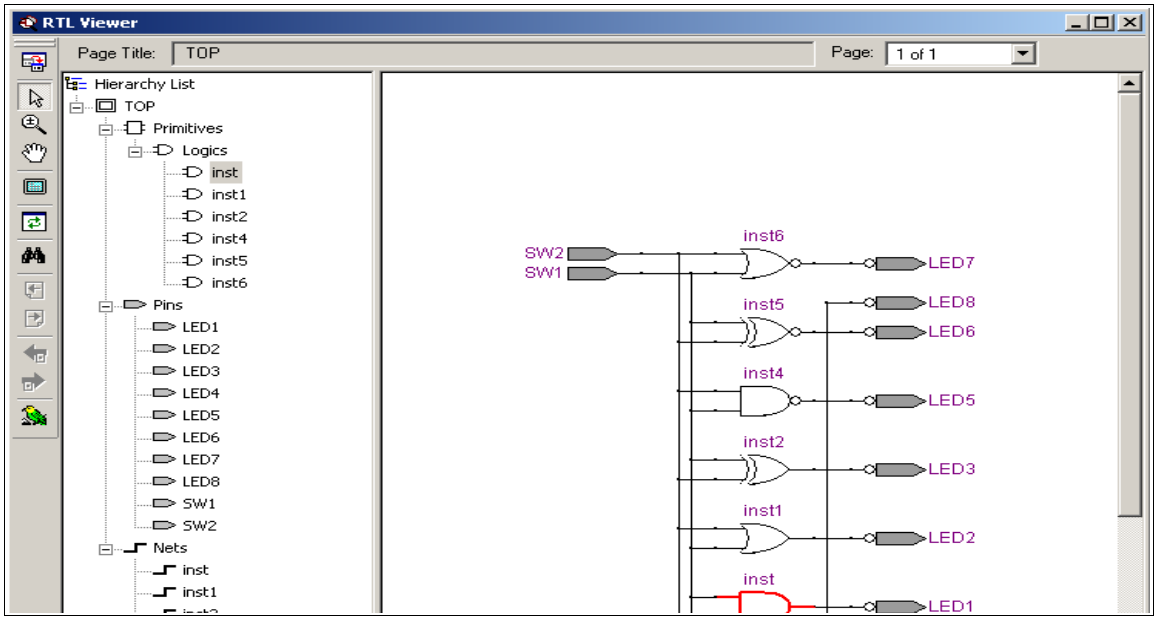
Фиг. 21 Резултат от успешна компилация, виждат се отделните прогресни индикатори на всеки един етап от компилацията . Analysis & Synthesis, Filter, Assembler, Classic Timing Analyzer

Ако всичко е преминало успешно можете да натиснете бутона **Report** (Фиг. 22), за да видите систематизиран отчет за използваните елементи и блокове на чипа, за който компилираме логическата схема. Ако процесът на филтрация и асемблиране не премине успешно ще трябва да изберем друг PLD чип с повече макро клетки или повече входно изходни пинове. На този етап нашият дизайн е компилиран и готов за симулация и програмиране.



Фиг. 22 Извеждъне на финален рапорт след успешна компилация

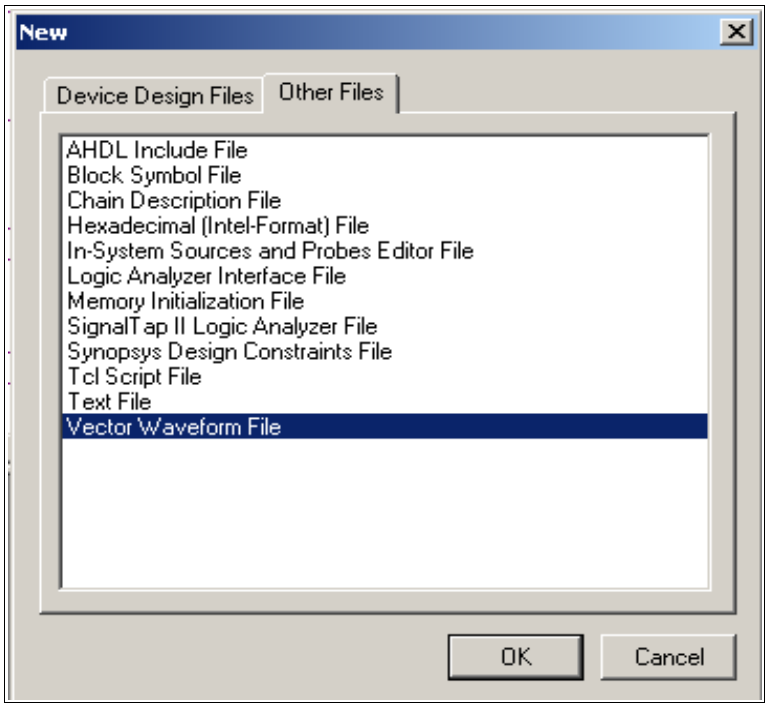
В много случаи е важно да сме сигурни каква е точната структурна схема на дизайна, която софтуерният компилатор генерира. Това може да се види през менюто **Tools->Netlist Viewers->RTL Viewer** (**RTL - register transfer level**), което ще изведе прозорец подобен на дадения на долната фигура. Това е еквивалент на принципната схема на дадено устройство, разбира се тази схема не е реално конфигурираната в устройството, но е достатъчно нагледна и от нея може да видите кои компоненти от дизайна реално ще бъдат ползвани и ще работят. Използването на тази опция ви позволява да сравнявате ефективността на един вариант на даден дизайн с друг негов вариант.



Фиг. 23 RTL Viewer

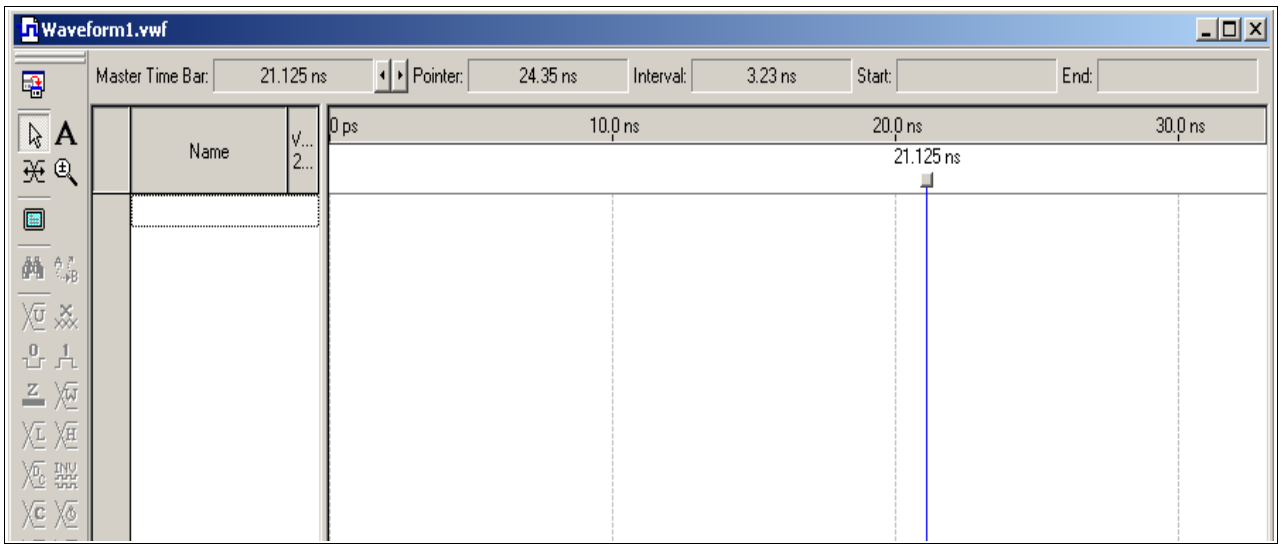
Симулация на проекта

След като приключите успешно схемотахническия дизайн и компилацията на проекта, мож да пристъпите към симулация на схемата. Преди това трябва да създадем файл с тестова последователност от импулси подавани в различни интервали от време на входните портове на схемата. Симулаторът ще извърши анализ на работата на устройството именно чрез тази тестова последователност. За да направите това следва да създадете нов симулационен файл, от менюто **File->New** отворете прозорецът за създаване на нови файлове. Прехвърлете се в **таб Other Files** и изберете **Vector Waveform File** и натиснете бутона **OK**.



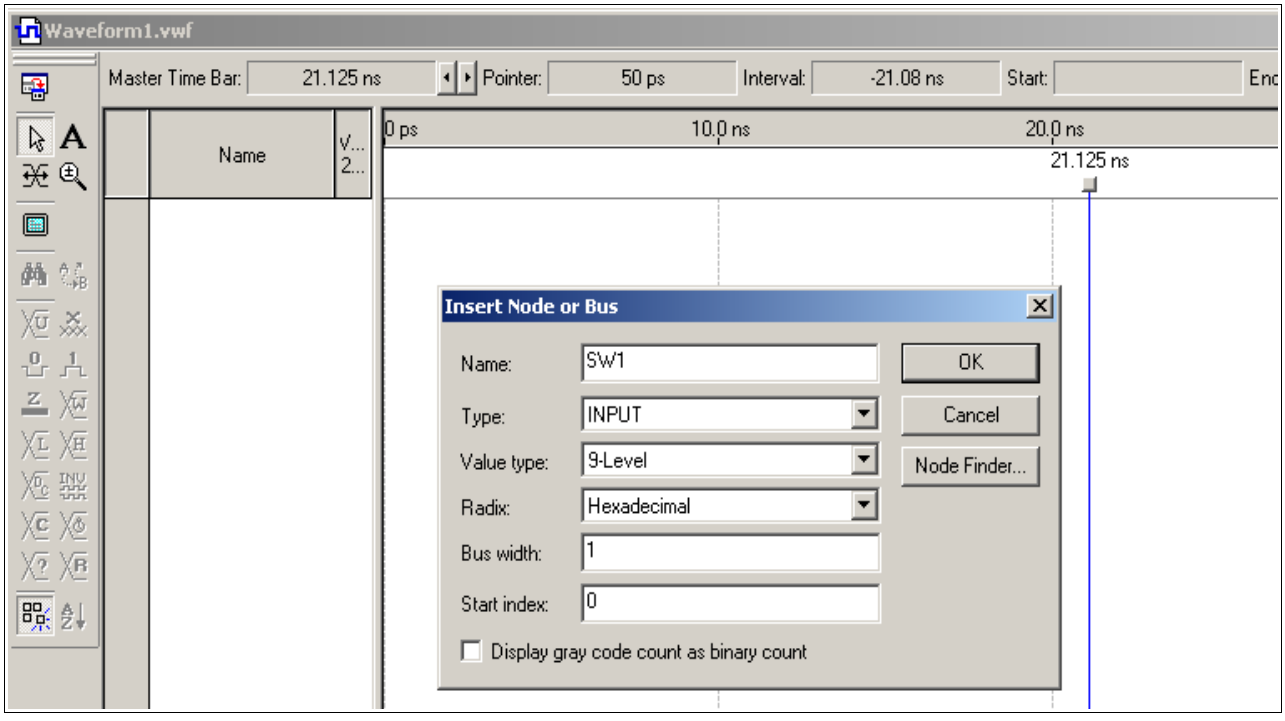
Фиг. 24 Създаване на нов Vector Waveform File

Тази команда ще отвори прозорец подобен на дадения на фигура 25. В него има няколко важни инструмента, първият от които е **лупата**. С нея можете да давате общи или детайлен вид на симулацията, която подготвяте. В първото вертикално бяло поле наречено **Name** се въвеждат последователно всички входни портове от дизайна, а именно **SW1** и **SW2**, това става като кликнете два пъти с ляв бутон на мишката в полето **Name**, ще се отвори прозорец показан на фигура 26. Препоръчително е при извършване на симулация да имате поглед на поведението на всички входно изходни портове участващи в конкретния блок или схема, която проектирате. Поради това препоръчваме изричното дефиниране на всички ползвани от вас портове в листа с портове подлежащи на симулация.



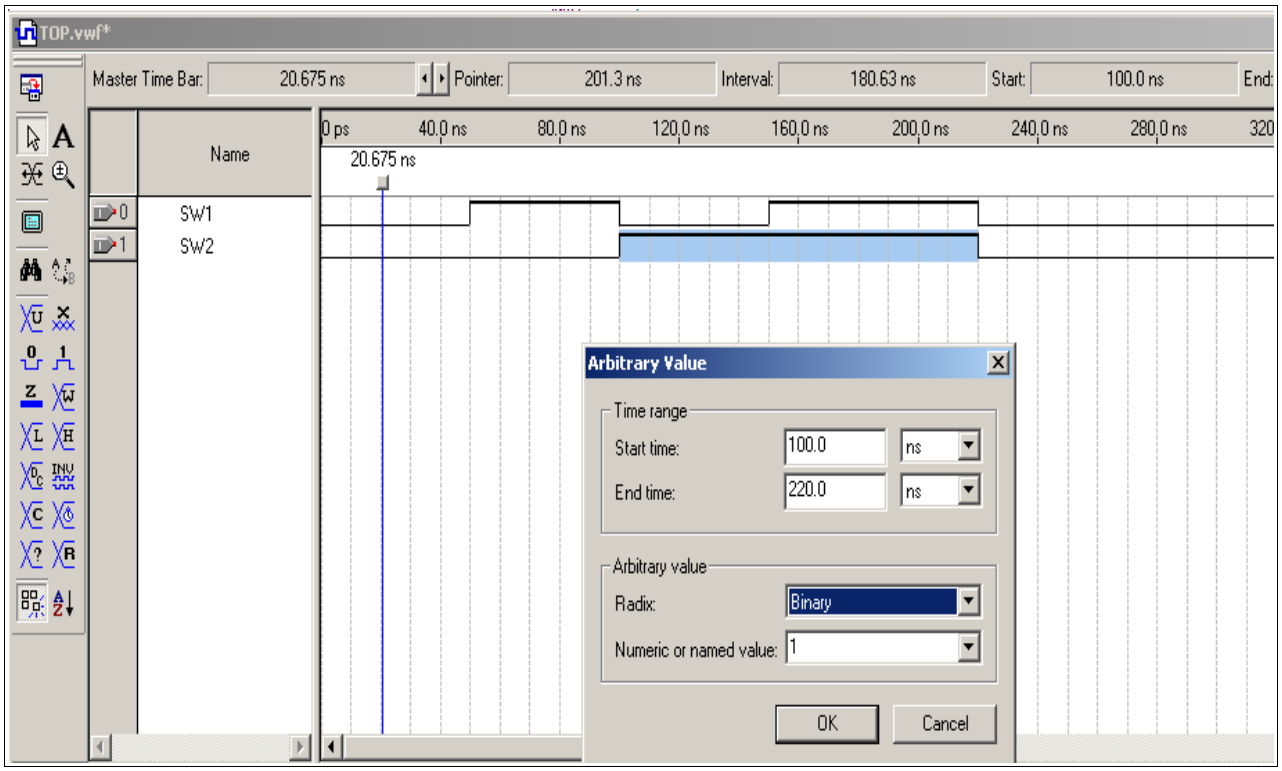
Фиг. 25 Прозорец за въвеждане на тестова последователност от импулси за текуща симулация

В прозореца **Insert Node of Bus** изберете **SW1** и натиснете бутона **OK**, повторете това действие и за бутон **SW2**. Принципно добрата практика на проектирането изисква да дефинирате абсолютно всички входно изходни портове на вашия дизайн. Може ръчно да въведете **LED1, LED2,... , LED8**. Тъй като всички тези изходни портове ще бъдат ползвани от нашия дизайн не е нужно да ги въвеждаме предварително, те ще се появят автоматично след като приключим процеса на симулацията.



Фиг. 26 въвеждане на входните портове, на които ще се подава тестова последователност от импулси за симулация

На този етап имаме добавени 2 входни порта **SW1** и **SW2**, но дефинираните нива на сигналите за цялата тестова последователност са **'0'** по подразбиране. Ще изолзваме лупата за да визуализираме по-дълъг интервал от време на екрана. В момента той е 30ns, тъй като времето за установяване на сигнала на изхода на всеки един логически елемент от момента на постъпване до стабилно извеждане е около 10ns, ще трябва да изберете такъв мащаб, че да може да виждате поне първите 300ns от симулацията. След това използвайки мишката като внимателно селектирате, задържайки левия бутон натиснат, време интервал от порядъка на 40-50ns (Фиг. 27). Селектираната област моментално се оцветява с бледо синьо. Кликнете два пъти в нея и ще се появи прозореца **Arbitrary Value**, в полето **Numeric or named value** въведете **'1'** и натиснете бутона **OK**.

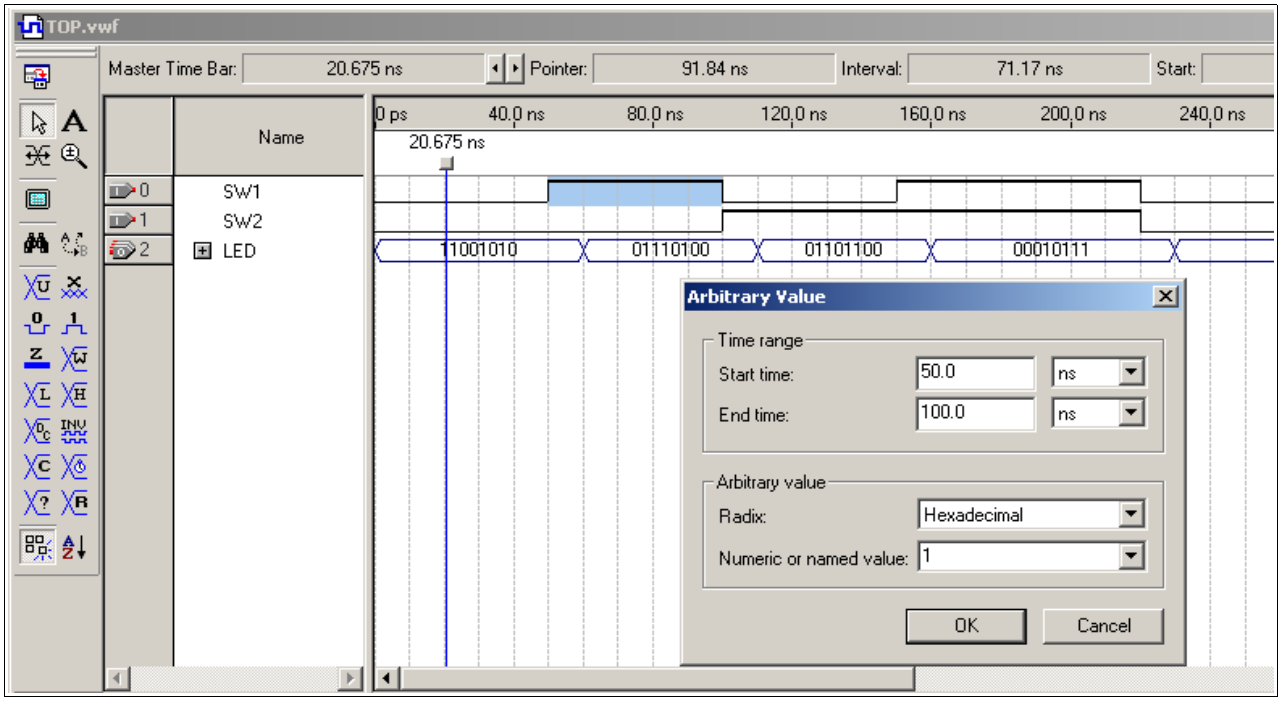


Фиг. 27 Задаване на стойност на сигнала за определен интервал от време

Тъй като симулационната последователност изисква да проверим значението на логическите функции за пълния набор от логически променливи, а в случая те са 4, следва да проверите действието за следните стойности на SW1 и SW2 съгласно:

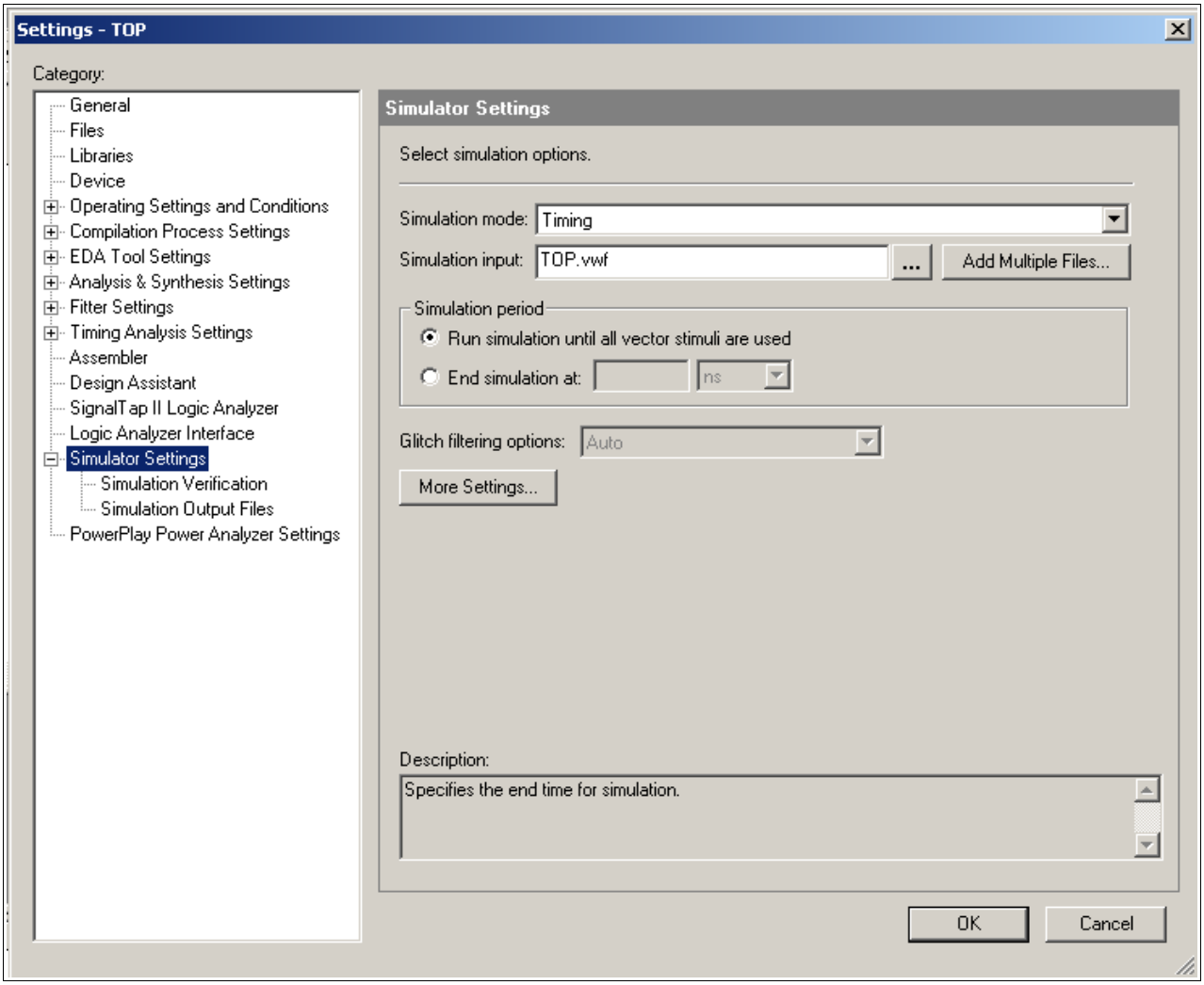
SW1	**SW2**
0	**0**
0	**1**
1	**0**
1	**1**

Само по този начин ще сте сигурни, че схемата функционира правилно за всички входни набори на логичуските променливи. В полето **Radix** – буквално преведено означава основа на бройната система, на прозореца **Arbitrary Value** може да избирате различен вид визуализация на променливите. Например двоичен-**binary**, шеснайсетичен-**hexadecimal** и други. Въведете следната тестова последователност (Фиг. 28). Преди да извършим симулация, съхранете файла с името **TOP.vwf**. Сега трябва да генерираме функционален списък на симулираните функции, което се извършва през менюто **Processing->Generate Functional Simulation Netlist**. След като тази операция приключи успешно следва да преминем към симулация.



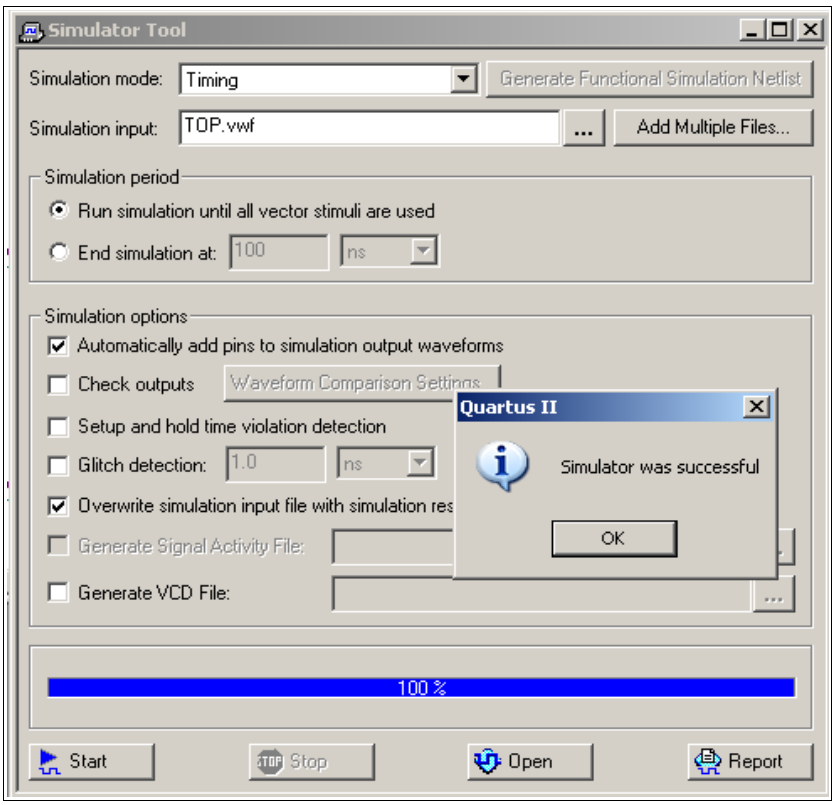
Фиг. 28 Изходна последователност за симулиране

Quartus II предоставя разнообразни възможности за симулация, от симулация на отделен файл до симулация на множество файли касаещи целия проект. За да определим кой файл ще симулираме ще ползваме менюто **Assignements->Setting,** отворете прозореца **Settings-TOP**, като от списъка **Category** изберете **Simmulator Settings** (Фиг. 29) Уверете се че в полето **Simulation mode** е избрана опцията **Timing**, а в полето **Simulation input** е избран файл **TOP.vwf**. След това натиснете бутона **OK**. Освен **Timing** съществува и опцията **Functional**, която ви позволява да симулирате сало действието на схемата без да се отчита времезакъснението на сигналите в отделните логически елементи. За момента този тип симулация не ни е нужна, тъй като искаме да видим какви времезакъснения ще имат изходните рузелтати спрямо подаването на входните сигнали.



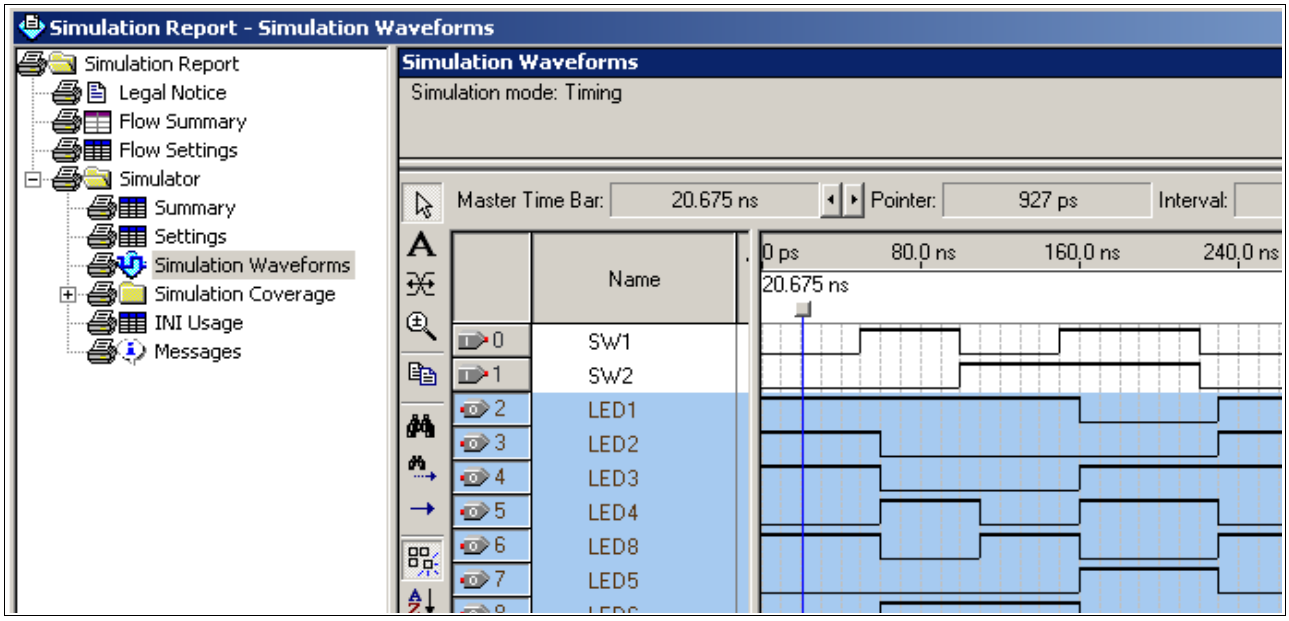
Фиг. 29 Settings-TOP за избор на вид симулация и файл с последователност от импулси, която ще бъде симулирана

Самата симулация се извършва от менюто **Processing->Start Simulation**. След като процесът приключи успешно може да изберете бутона **Report** (Фиг. 30). Когато изходния сигнал е '1' той се означава като високо ниво на даден порт – с по-плътна тъмна линия, когато изходния сигнал е '0' той се означала с ниско ниво и тънка черна линия. Когато на изхода е записано 'Z' това означава, че съответният порт е изведен във високоимпедансно състояние, а когато 'X' означава че не е инициализиран. Веднъж направена симулацията ще се съхранява до нейното ново стартиране. Това означава, че ако решите да промените архитектурата на схемата, ще трябва наново да преминете през процес на компилация и симулация.



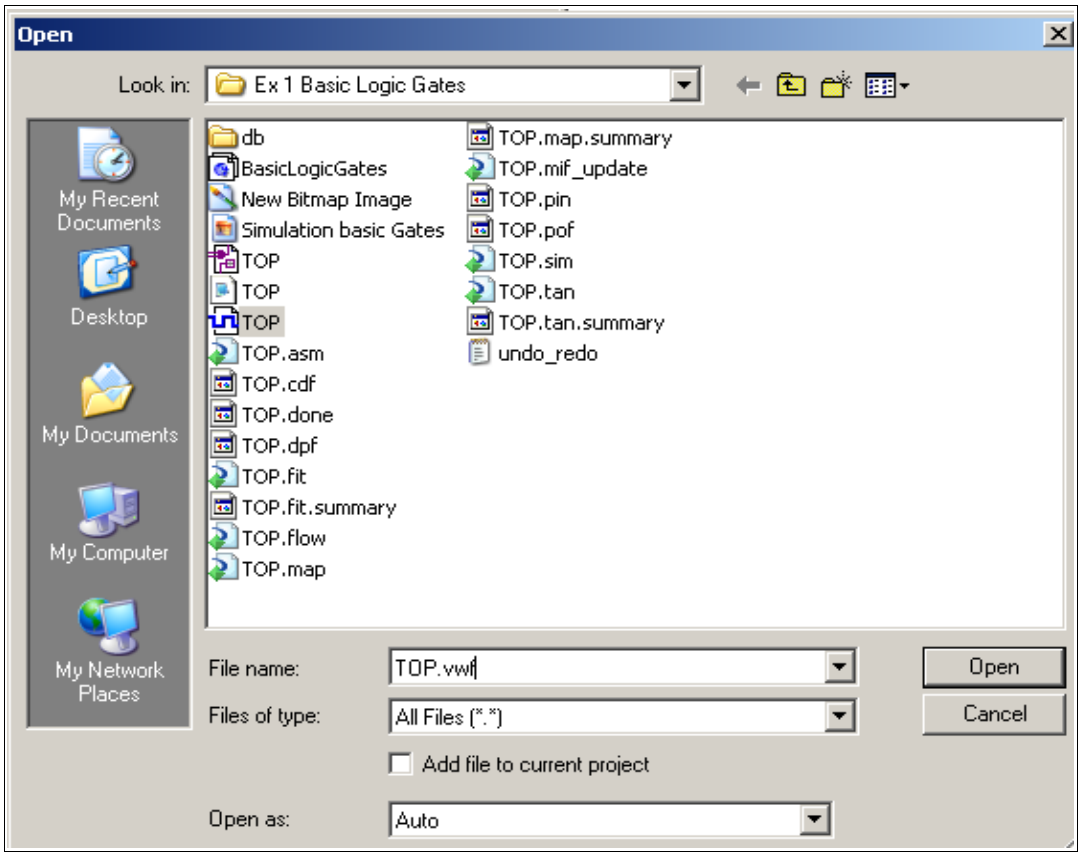
Фиг. 30 Успешен резултат след приключване на симулация

При успешна симулация ще се отвори прозореца **Simulation Report** даден на долната фигура. В него вече се виждат сигналите на изходните портове **LED1, LED2,... , LED8,** като обръшам внимание че е възможно номерацията да не съвпада с реалната ситуация.



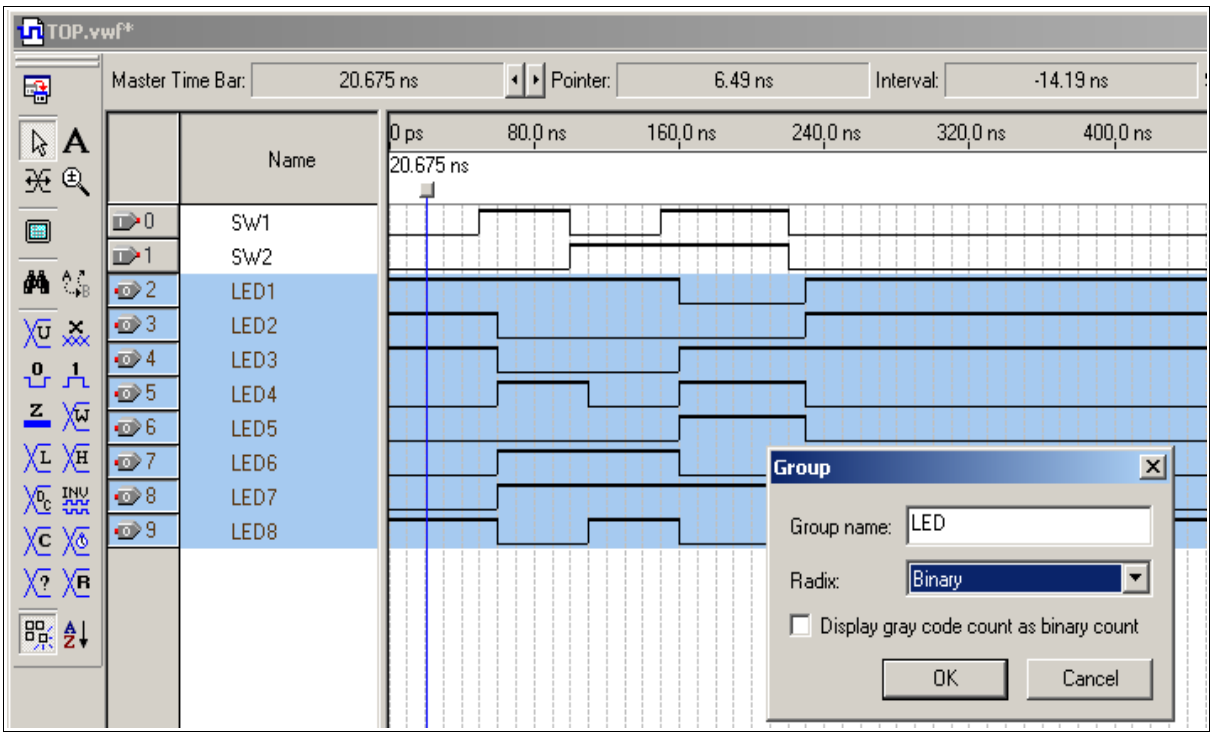
Фиг. 31 Резултат от приключилата симулация, LED1... LED7

На този етап можете да затворите прозореца **Simulation Report** и да отворите симулационния файл от менюто **File->Open**, като за да видите нужният ви файл **TOP.vwf** в полето **File types** изберете **All Files (*.*)** или в полето **File** name въведете **TOP.vwf**, след това натиснете **Open**.



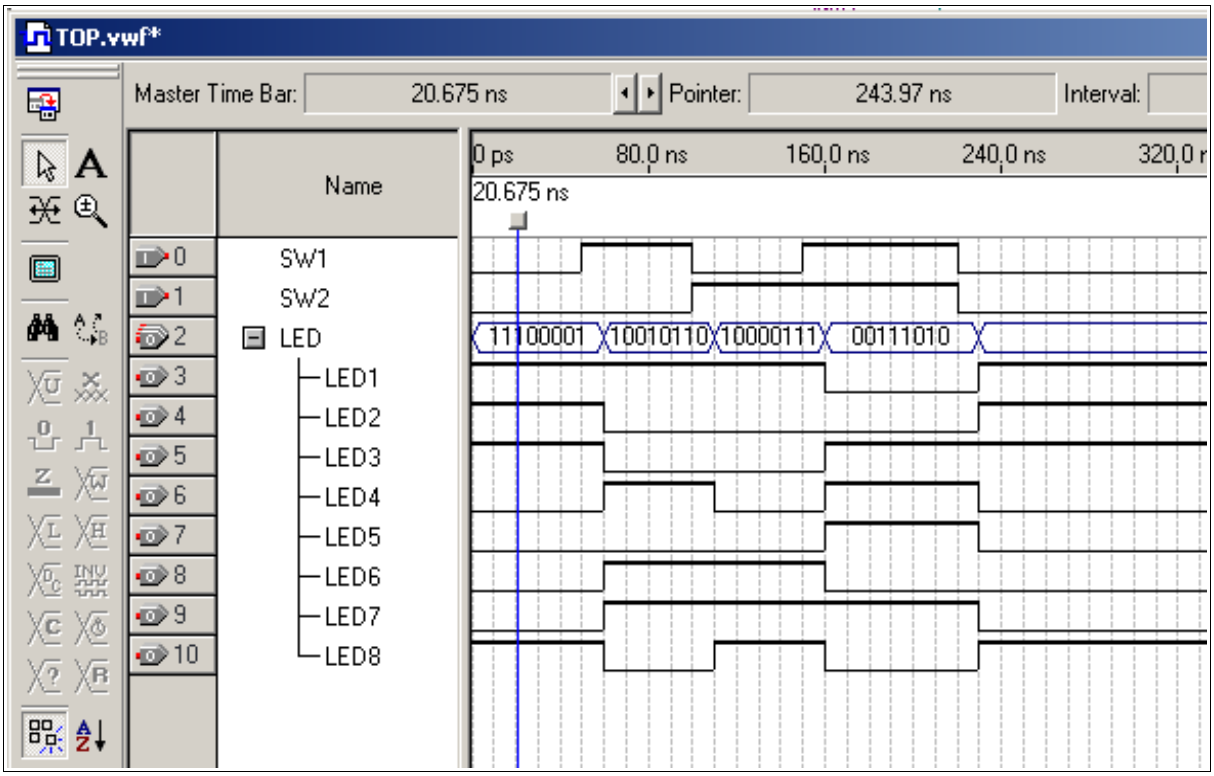
Фиг. 32 Отваряне на файл съдържащ вече направена симулация

В резултат на това действие, ще се отвори файлът който създадохте, но сега освен входните сигнали в него ще присъстват и изходните. Ако сигналите не са подредени по ред на номерата може да ги сортирате и с десен бутон на мишката, и да отворите опцията **Grouping**. Това ще отвори дадения по-долу прозорец **Group**. От падащия списък изберете данните да се извеждат в двоичен код - **Binary**, а за име въведете **LED** (Фиг. 33) Тази команда ще групира осемте сигнала в шина. Така по-лесно ще бъдат визуализирани изходните резултати. Избрахме да визуализираме изходните сигнали в двоичен код, тъй като дизайнът представлява реализацията на 8 елементарни логически функции с отделни изходи. По този начин тяхното състояние най-лесно ще бъде отчетено. Операцията по групиране може да извършите и за набор входни портове. Този тип позволява да имате един компактен изглед на симулираната последователност, като при нужда може да отворите всяка една група и да наблюдавате сигналите поотделно.



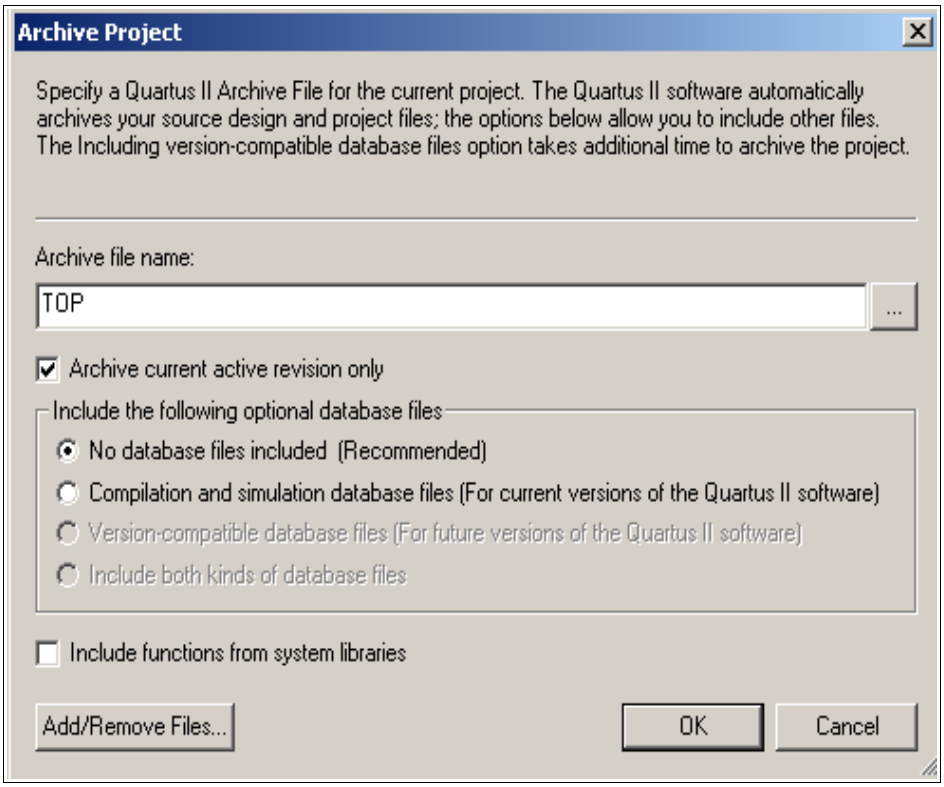
Фиг. 33 Групиране на набор изходни портове

След като групираме изходите в обща шина **LED**, може да разгледате резултатите, които следва да са подобни с дадените на фигура 34.



Фиг. 34 Разглеждане на единичните сигнали от група LED

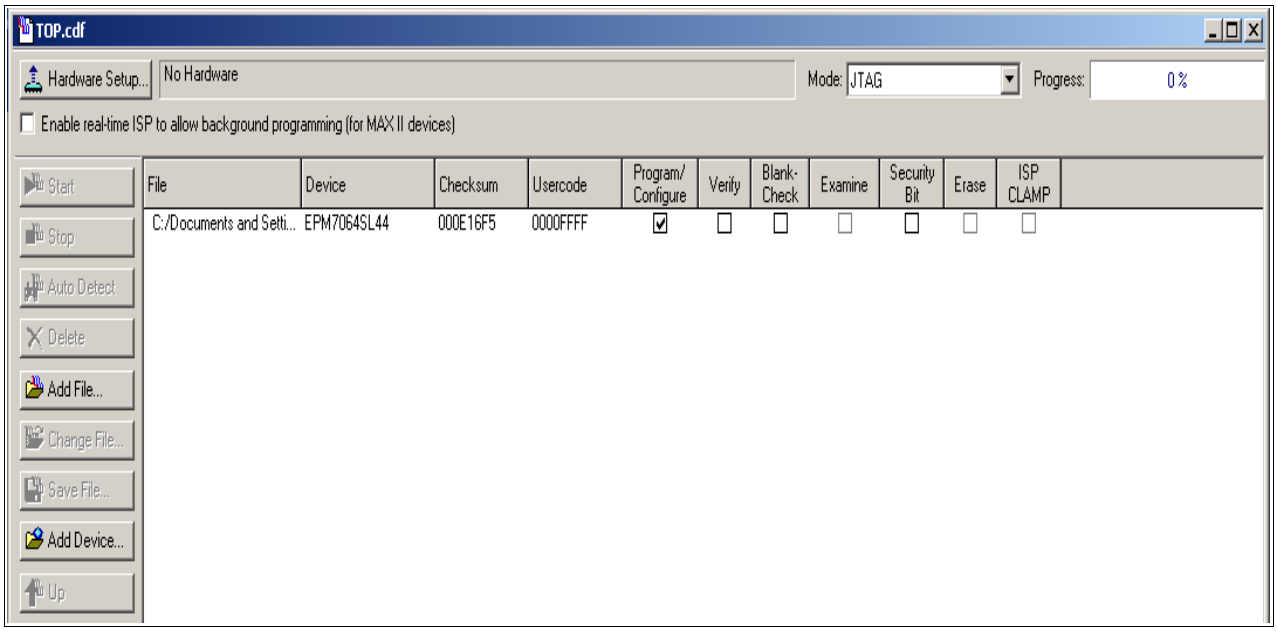
На този етап приключихме дизайнът и симулацията на комбинационно логическо устройство. Следва да архивираме нашият проект. Архивирането на проекта в края на работния ден или работен етап е важно и следва да се прави по два начина. Или чрез архивиране на папката на проекта със стандартен архиватор Zip или RAR, или чрез използване на вградените в Quartus II инструменти за архивация на проектите. Това може да стане от менюто **Project->Archive Project** (Фиг. 35). Възможностите да архивирате и базата данни със симулациите към проекта е приемлива само в случай че ще ползвате архивирания проект към същата версия на Quartus II. Може да изберете мястото където архивният файл ще бъде съхранен. Оставете настройките по подразбиране и извършете архивирането. Сега архивният файл на проекта се съдържа в основната папка на проекта. При нужда можете да възстановите вашият проект от менюто **Project->Restore Archived Project**.



Фиг. 35 Архивиране на проекта с Archived Project

След като създадохме и симулирахме проекта следва да програмирате реалната платка и да проверите наистина ли схемата работи правилно. За целта ще ползваме интерфейса за програмиране **Byte Blaster** с паралелен кабел през **LPT (Local Printer Port)**. Самото програмиране изисква да изберете подходящият интерфейс за програмиране и да укажете името и мястото на двойчния конфигурационен файл. За да програмирате файла изберете менюто **MAX+PLUS II->Programmer.** Ще се отвори прозорецът даден по-долу. От бутона **Hardware Setup** следва да изберете наличният и инсталиран в системата

хардуер за програмиране. Много JTAG програматори изискват предварително инсталиране на специфичен драйвер, това важи основно за програматори свързвани през USB интерфейс към персоналния компютър.

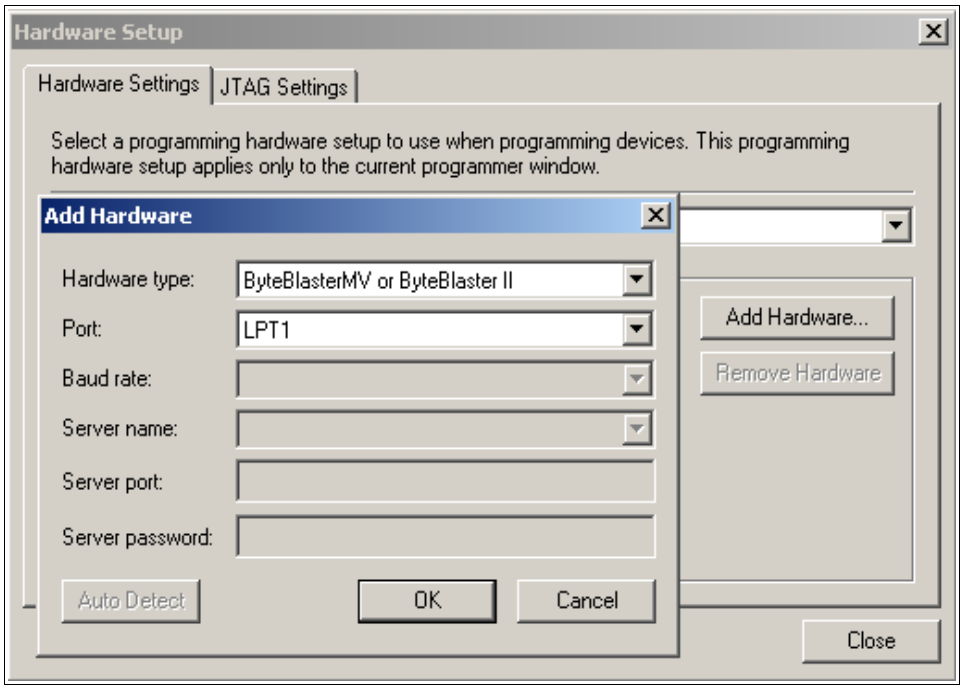


Фиг. 36 Команден прозорец за избор на конфигурационен файл, програматор и програмиране на устройството

От отворилият се прозорец **Hardware Setup** изберете **таб Hardware Settings** и натиснете бутон **Add hardware**, което ще отвори под прозореца **Add Hardware**. Ако преди това сте включили захранването на платката и сте свързали компютъра с нея софтуерът автоматично ще дететектира платката и интерфейсът **Byte Blaster** през **LPT**. Обърнете внимание, че за успешното програмиране при ползване на **USB Byte Blaster**, наличен на други модели демонстрационни платки, следва да сте инсталирали първоначално драйверите (фиг. 37). Ако не сте направили това и свържете платката съм компютъра ще се наложи след това ръчно да отстраните драйверите и да ги качите наново. Драйверите не се инсталират автоматично с инсталирането на Quartus II, но са достъпни в папка: **...\altera\72\quartus\drivers\usb-blaster**, като изберете под папка **x32** или **x64** в зависимост от операционната система която ползвате. За да инсталирате драйвера намерете файл **usbblst** и кликвайки с десен бутон на него изберете опцията **install**. Възможно е след това системата да поиска да бъде рестартирана. Чак след това може да включите USB програматора към вашия компютър и успешно да го използване.

След като изберем правилния интерфейсен кабел за програмиране натиснете **OK** и след това **Close**. Ще се върнете в основния прозорец от където да изберете файла с който да програмирате устройството. На този етап може да

проверите и опцията **Auto Detect**, която ще разпознае автоматично свързаният към компютъра хардуер за програмиране. След това трябва да изберете файл който да заредите в него. При правилната компилация на проекта този файл би следвало да се намира в основната му папка и да има името **TOP.qpf**. Изборът на файл става чрез бутона **Add File**. Отметнете полето **Program/Configure** и натиснете **Start**. След успешното програмиране на конфигурационниа файл платката може да бъде тествана. За целта задавайте избрани от вас комбинации с бутон **SW1** и **SW2** и наблюдавайте изобразяването на данните върху **LED1** до **LED8**.



Фиг. 37 Избор на програматор Byte Blaster

След успешното приключване на това първо упражнение може да пристъпите към подготовка на по-сложни проекти. Запознахте се с ползването на схемотехническия редактор за дизайн на електронни схеми, симулатора и програмирането на PLD чипове. Можете да адаптирате конкретен дизайн към определена печатна платка с реално действащ хардуер. В следващия раздел ще се запознаем с основите на VHDL и начините на дизайн на хардуер с него. Схемотехническия редактор и VHDL текстовия редактор са основните инструменти за дизайн на хардуерни блокове и модули. Умението за ползване ви гарантира успеваемост при реализацията на сложни проекти от идеята до конкретната реализация.

Дизайн на цифрови устройства с VHDL

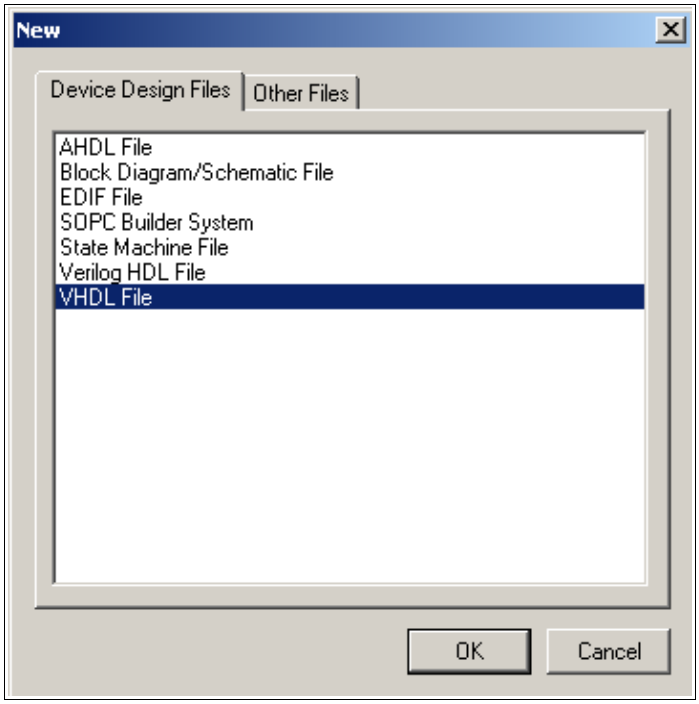
До момента се запознахте как с помоща на Quartus II можете да извършите класически схемотехнически дизайн на цифрово електронно устройство, да компилирате съответния проект и да извършите конфигуриране на PLD чип. Въпреки, че класическият схемотехнически подход е удобен в редица случаи той не е достатъчно подходящ за създаване на сложни блокове, като броячи, дешифратори и други, тъй като в процеса на работа се допускат множество технически грешки, чието откриване изисква симулации и множество реални проверки. Това е в сила за всички устройства в които се ползват сходни блокове с малки различия. Съществува по-удобен начин за описание на подобно устройство, а именно с VHDL. Характерно за този език е че можете да извършвате структурен и поведенчески модел на дадено устройство или блок. Този модел лесно може да бъде адаптиран за 8, 16, 32 и повече битов дизайн. Създадените описания са лесно четими, докато една сложна схема не винаги е разбираема за всички. Логиката на описание е сравнително проста и се ползва достъпен и лесен за усвояване набор от езикови конструкции. (Изисква се известна практика, особено за програмисти на C/C++, Java, Fortran и др. езици.) Дефакто може да твърдим, че човек който може да чете и разбира английски език и е технически грамотен би могъл да се ориентира в един VHDL код без много усилия и с малко първоначална подготовка. Освен това този код е преносим, тоест едно устройство може да бъде компилирано за различни типове хардуерни платформи производство на Altera, Xilinx и други фирми, с различен тип градивни логически блокове, което не винаги е валидно за схемотехническия дизайн.

Дизайнът с VHDL е забавен и не изисква от дизайнера да познава в детайлно платформата която ще конфигурира. Едни и същи файлове могат да бъдат ползвани в други дизайни без да се налага никаква промяна в тях. За да добиете по-добра представа за този процес следва да възприемете опростеното схващане, че всяко едно цифрово устройство може да се представи като черна кутия с входни и изходни портове и някаква електронна схема, която седи затворена в нея. Този подход опростява значително процеса на дизайн. Черната кутия се нарича ентити - **entity**, а електронната схема **architecture** – архитектура. За тези две понятия следва да знаете следните неща: в едно ентити може да стоят различни архитектури, а също така една и съща архитектура може да се постави в различни ентитита. Класически пример за това са електронните схеми произвеждани в различни типове корпуси DIP и SMD. Едни и същи електронни устройства се поставят в корпуси с различен брой изводи и форми, същото важи и за самите корпуси. Едни и същи корпуси се ползват от производителите на електронни компоненти, за да поставят в тях най-разнообразни електронни схеми, процесори, операционни усилватели и други устройства.

В предишния пример видяхте как можем да генерираме VHDL описание за определена схема, същият процес е валиден за създаване на схема от VHDL файл. В повечето случаи отделните компоненти на едно по-сложно устройство се описват в отделни VHDL файлове, а цялото устройство се създава, като отделните символни файлове – блокове се свързват в схемотехническия редактор. Този процес е напълно осъществим и при ползване само на VHDL, но тогава главният VHDL файл изисква прекалено много усилия за описание на взаимо връзките мужду отделните модули. Именно за това ефективен дизайн може да се постигне при комбиниране на VHDL със схемотехническия редактор.

Дизайн на 8 битов тристабилен драйвер

Това упражнение ще ви научи как да създавате и симулирате хардуерно устройство чрез използване на VHDL описание. Целта е да се реализира тристабилен 8 битов драйвер. Ползването на тристабилни изходни и входни портове позволява свързването на множество електронни компоненти през една шина (група проводници или писти). Този процес се нарича мултиплексиране, но за да не си пречат две и повече работещи устройства тяхното действие се разпределя във времето. Когато едно устройство ползва шината останалите трябва да бъдат изведени в състояние на висок импеданс. Повече за типови чипове реализиращи тази функционалност ще намерите ако прочетете описанието на схема **74LS240**. Към разглеждания пример студентите се запознават с **процесите,** техните функции, събития които ги активират и оператор за условно изпълнение **if-else**. Като начало създайте нов проект с Quartus II в нова папка намираща се в **My Documents/Altera/project2**. Кръстете името на проекта **Project2** и главното ентити **TOP**, оставете останалите настройки и избора на PLD чип, като при първия пример. След като създадете тялото на проекта следва да добавите нов VHDL файл, това може да стане през менюто **File->New** (Фиг. 38).



Фиг. 38 Създаване на нов празен VHDL файл

След като създадете нов файл следва да го съхраните. Използвайте името **TOP**, тъй като това е името на основното ентити на вашия проект. Първото нещо което следва да включите са определени библиотеки, в тях има предварително дефинирани логически и аритметични функции, както и

различни типове променливи. За момента следва да помните, че във всички проекти трябва да включвате дадената библиотека **ieee и std_logic_1164.all.** Обърнете внимаче, че текстовият редактор ще оцветява различно някои думи от вашия код, това са запазени команди, методи или служебни думи, които не бива да ползвате като имена на променливи, процеси и други елементи на вашия код. Всеки ред от VHDL кода завършва с точка и запетайка, както в езика „C“. Когато дадена секция код завършва със скоба, точката и запетайката на последния изброен елемент се поставя зад скобата. Описанието на библиотеките, ентитито и архитектурата следва да седят в един и същи файл.

```
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity TOP is
port(
        d_in: in std_logic_vector(7 downto 0);
        en:    in std_logic;
        d_out:        out std_logic_vector(7 downto 0)
);
end TOP;

architecture TRISTATE of TOP is
begin
  process(d_in, en)
  begin
        if en='1' then
          d_out <= d_in;
        else
          d_out <= "ZZZZZZZZ";
        end if;
  end process;
end TRISTATE;
-----------------------------------------
```

Описание на ентитито

Както казахме ентитито ви позволява да описвате едно цифрово устройство, като черна кутия с набор входни и изходни портове. Ентитито започва със служебната дума „**entity“**, като след нея следва неговото име, в случая ТОП, като след това следва декларацията му след думата **„is”.** За да укажете на компилатора за край на ентитито следва да ползвате декларацията “**end TOP;“**.

```
entity TOP is
        ...
end TOP;
```

В ентитито следва да се опишат входно изходните портове, което става с директивата „**port“**, в скобите се извършва последователно описание на листа с портовете.

```
port( … );
```

В нашия случай това са: 8 битов входен порт, 8 битов изходен порт и 1 битов входен порт, които се описват, както следва:

```
port(
        d_in   : in std_logic_vector(7 downto 0);
        en      : in std_logic;
        d_out : out std_logic_vector(7 downto 0)
);
```

Имената на портовете са съответно: **d_in, en, d_out**, за да укажем дали са входни или изходни се ползват дефинициите „**in” и “out”**. Разредността на порта може да бъде единична, тогава той се декларира, като „**stg_logic”**, или многоразрядна, когато се дефинира, като „**std_logic_vector”.** В скоби се указва колко бита е самия порт „**(7 downto 0)“ -** дефинира се 8 битов порт. Следва описанието на архитектурата. Преди да коментираме всеки ред от кода следва да дадем следното пояснение, относно функционирането на схемата. Драйверът, който реализираме има възможност да извежда подадените на входа стойности през изходните пинове. Това може да се случи, само когато е подаден разрешаващ сигнал „**en“**. В останалите случаи, когато разрешаващия сигнал не е подаден, изходът на драйвера остава във високо импедансно състояние. Следоватлно устройството, което реализираме следва да може да проверява състоянието на разрешаващия вход и да взима решения съобразно него, при **'1'** да изведе данните подадени на входа, а при **'0'** да установи изхода във високоимпеданско състояние означено с главно **„Z”**.

Както казахме по-рано, конкретната функционалност на едно цирово устройство се описва в неговата архитектура – схема. Архитектурата във VHDL може да бъде зададена строго – структурно или не императивно - поведенчески. И двата варианта имат своите преимущества и недостатъци. Структурното описание ви позволява точно да дефинирате пътят на сигналите в самата схема. При поведенческото описание задавате описателно, какво вашето устройство следва да прави, без да се интересувате от конкретната схемна реализация. Тъй като една архитектура може да се пъстави в много различни ентитита, следва да

имаме начин с който да укажем на компилатора към кое ентити ще се прикрепи нашата архитектура, това става по следния начин:

```
architecture TRISTATE of TOP is
begin
  ...
end TRISTATE;
```

Началото на архитектурата започва със служебната дума „**architecture ... of … is**” и завършва с „**end ...** ;“. Където, „**TRISTATE**” е името на архитектурата, а „**TOP**” указва към кое ентити тя следва да се привърже. Буквално това може да се прочете така: архитектурата наричана „**TRISTATE**” отнасяща се към ентити „**TOP**“ е дефинирана след секцията „**begin**”, краят на архитектурата се указва с „**end TRISTATE;**“. След секцията **begin** може да седи нашето същинско описание. Ако реализираният компонент беше елементарен буфер, то всичко което трябва да напишем, като описание е:

```
architecture TRISTATE of TOP is
begin
        d_out <= d_in;
end TRISTATE;
```

Тук символът „<=” указва посоката на прехвърляне на данните, в класическите езици за програмиране тази операция се означава с „=” и се нарича присвояване. Действието, което се описва е следното: всички промени подадени, като сигнали на 8 битовата шина „**d_in**” ще бъдат незабавно изведени на 8 битовата шина „**d_out**”. Това е важно свойство на архитектурите, те могат да обработват и да реагират на измененията на входните портове описани в ентитито, и съответно да извеждат данни на изходните портове. По този начин в архитектурата може да се дефинират различни логически, аритметични и други операции ползващи, като променливи данните подавани на входните портове. Резултатът от операциите се извежда на изходните портове на устройството. Предимството на PLD пред конвенционалните чипове и микроконтролери, е че те могат да се ползват за едновременното изпълнение на множество комбинационни и последователни функции, вместо това да става разделно във времето, както това се осъществява в микропроцесорните системи. При микропроцесорите програмните инструкции се изпълняват една след друга, и действието в реално време се постига за сметка на много бързото изпълнение на инструкциите. При PLD ние конфигурираме даден чип да действа, като набор от отделни логически схеми. Всяка една хардуерно дефинирана схема може да работи независимо от другите смехи и блокове дефинирани в дизайна. Това позволява да повишим бързодействието на отделните операции, като те реално се изпълняват в паралел. В основата на

абстракцията на този модел лежи понятието процес – **process**. Процесът е независимо дефинирана функция, но не по смисъла на функция който се влага в програмните езици, която се изпълнява като отделен обособен логически блок в конкретната PLD. Процесът може да се синхронизира с другите процеси или с определен набор разрешаващи и тактови сигнали. Тези сигнали предизвикват изпълнението на процеса. За да дефинираме процес използваме дадения по-долу формат:

```
име : process (събития предизвикващи процеса)
 begin
        ...
end process;
```

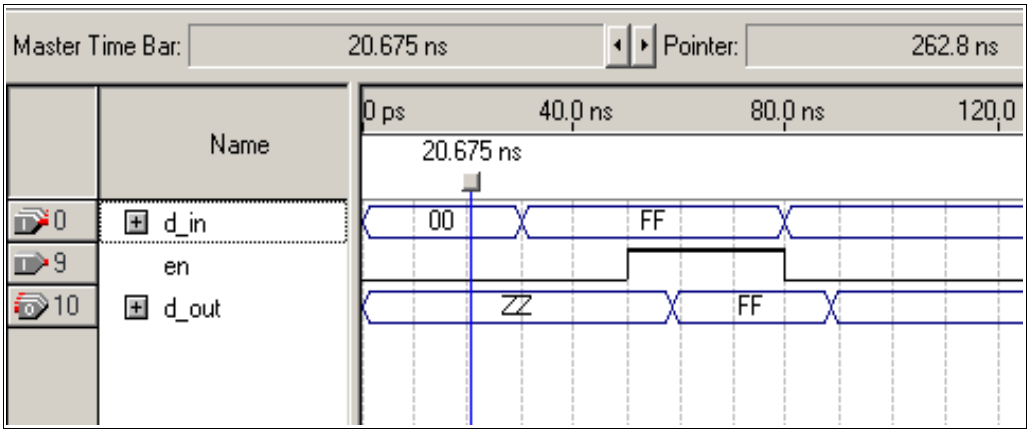
Процесът може да се дефинира с име, например „**data_flow : process (d_in, en)**“, кодето името на процеса е „**data_flow“**, или да се дефинира без да указваме конкретно име. В скобите се описват сигналите, които ще активират процеса. Това ни посволява да активираме процеса само при настъпването на определена комбинация от събития. В нашия случай процесът следва да се активира в 2 конкретни случая: когато на разрешаващия порт се подаде сигнал 1/0 и когато на входния 8 битов порт се смени подаваната комбинация от битове. Тялото на процеса се описва след директивата „**begin”**.

```
process(d_in, en)
begin
       if en='1' then
         d_out <= d_in;
       else
         d_out <= "ZZZZZZZZ";
       end if;
end process;
```

Процесът ще притежава функционалността да извежда в изходния порт подадените данни на входния порт или да установява във високоимпедансно състояние изходния порт. Това става с оператора за условно изпълнение „**if”**. Ако четем буквално написаното то ще има смисъла на: ако състоянието на сигнала подаван на вход „**en”** е равно на '1', тогава - “**then“** ще изведем подадените на вход „**d_in”** данни в изход „**d_out”**, в противен случай изходния порт „**d_out”** ще се изведе във високоимпедансно състояние „***ZZZZZZZZ***“. За да укажем на компилатора къде свършват исканите от нас проверки за условия ползваме израза „**end if”**. Този на пръв поглед елементарен пример имаше две основни задачи, да ви даде представа, за понятията: **ентити, архитектура и процес**, и да ви покаже, че умението да четете буквално английски език ще ви позволи да четете и разбирате различни VHDL описания.

Симулация

За да проверим правилността на действие на тристабилния драйвер следва да извършим симулация. За целта създайте нов симулационен файл и добавете нужните входно изходни портове: **d_in, en, d_out**. Създайте симулационна последователност (Фиг.). Достатъчно е да проверим двете крайни комбинации подавани на входния порт d_in, това са 0x00 и 0xFF. Съответно, ако VHDL описанието ви е правилно, на изходния порт d_out ще се извеждат данни само ако на вход en сме подали логическа 1. Във свички останали случаи изходния порт d_out ще се извежда във високоимпедансно състояние „ZZ“.



Фиг. 39 Симулационна последователност на тристабилен драйвер

Задача 1: Създайте тристабилен драйвер за 16 и 32 битова шина, тествайте компилирания проект. Може да използвате следният подход за лесна промяна на разрядността:

```
entity TOP is
        -- добавете този код за да дефинирате стойност
        -- използвана по време на компилацията
        generic ( n : integer := 16); - -  32 за 32 битова шина
port(
        d_in: in std_logic_vector(n-1 downto 0);
        ...
```

Задача 2: Обяснете защо в декларацията на портовете за старши бит в дефинираните вектори се ползва **n-1**, вместо **n**.

Задача 3: От създадения VHDL файл генерирайте символен файл, създайте нов схемотехнически файл за дизайн и използвайки новосъздадения символ създайте нова схема с него.

Процеси, сигнали и променливи

Използвайки придобитите до момента умения създайте нов проект, кръстете го **Process,** а главното ентити **TOP**, прибавете нов VHDL файл към проекта и наберете дадения по-долу код. Приложението което правим има 3 еднобитови входни порта и 2 еднобитови изходни порта. В тялото на архитектурата му са дефинирани два еднакви по функционалност процеса реализиращи логическото уравнение: **y = (A & B) xor C** по различен начин.

```
-------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity TOP is
port( d1, d2, d3:  in std_logic;
      res1, res2:     out std_logic);
end TOP;

architecture behv of TOP is
-- дефиниране на сигнал виден за процесите в цялата архитектура
      signal sig_s2: std_logic;
      begin
      proces1: process (d1, d2, d3)
      -- дефиниране на променлива видима само от процеса
      variable var_s1: std_logic;
      begin
            var_s1 := d1 and d2;
            res1 <= var_s1 xor d3;
      end process;

      proces2: process (d1, d2, d3)
      begin
            sig_s2 <= d1 and d2;
            res2 <= sig_s2 xor d3;
      end process;
end behv;
-------------------------------------------------------
```

Описание на архитектурата

Двата процеса се дефинират в тялото на архитектурата след директивата **begin**. Техните имена са съответно „**proces1**” и „**proces2**”. Обърнете внимание на още две особености, веднага след дефиницията на архитектурата „**behv”** е деклариран нов сигнал кръстен „**sig_s2**”.

```
signal sig_s2: std_logic;
```

Този сигнал е от типа **std_logic** и в рамките на архитектурата се държи като входно изходен порт, сигналът може да се свързва към входове и изходи на елементи вътре в архитектурата, и също така той може да се инициаризира със стойности подавани на входните портове или с междинни резултати получени в рамките на изчисления в архитектурата. Забележете също така че за сигнала не указваме посока на предаване на данните. Най общо казано може да гледате на сигнала, като на свързващ проводник между отделните елементи в архитектурата, но такъв който изпълнява функциите и на D тригер, в някои случаи.

```
proces1: process (d1, d2, d3)
-- дефиниране на променлива видима само в тялото на процеса
variable var_s1: std_logic;
begin
        var_s1 := d1 and d2;
        res1 <= var_s1 xor d3;
end process;
```

Важно особеност на тази архитектура е дефинирането на два процеса, proces1 и proces2. Сами по себе си процесите ви позволяват да описвате отделни независимо работещи хардуерни блокове във архитектурата. Процесът притежава списък от вектори, които го активират. В случая това е изменение на нивата на входните портове - **d1, d2, d3**. Важно особеност на разглеждания процес е наличието на локална променлива, декларирана преди директивата begin. Тази променлива ще бъде ползвана от процеса за междинно съхраняване и обработваните данни.

```
variable var_s1: std_logic;
```

За разлика от порта и сигнала, една променлива се инициализира по следния начин:

```
var_s1 := d1 and d2;
```

Като за да прочетем данните от нея и да ги изведем д съответен изходен порт, или да извършим някаква логическа операция, не правим нищо различно от начина за инициализиране на стойности записвани в портовете.

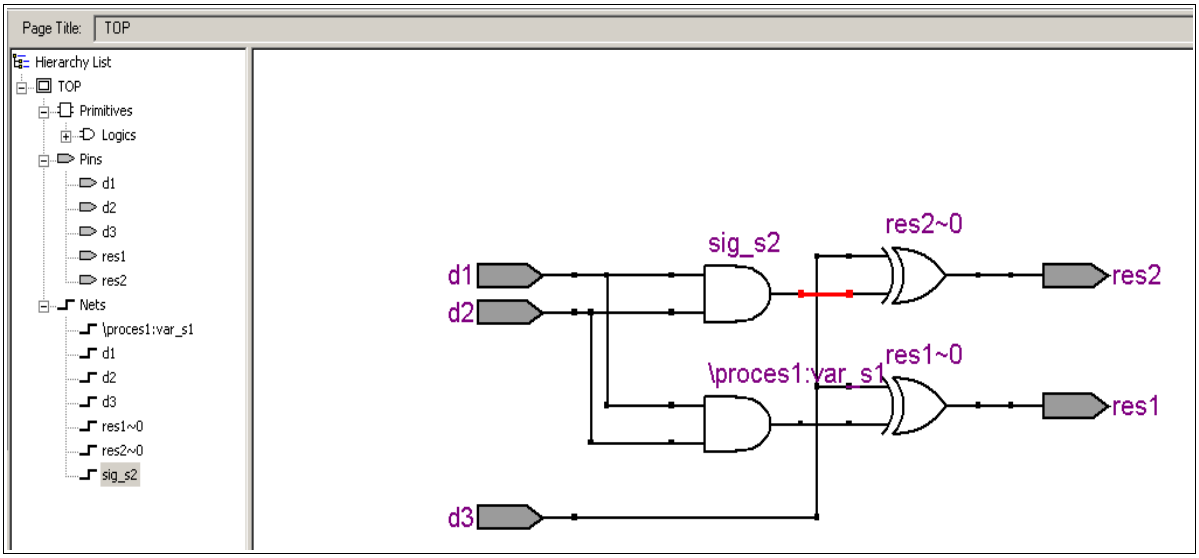
```
res1 <= var_s1 xor d3;
```

Вторият процес използва сигнала „**sig_s2”** за реализацията на същата функционалност:

```
proces2: process (d1, d2, d3)
        begin
                sig_s2 <= d1 and d2;
                res2 <= sig_s2 xor d3;
end process;
```

Какъв хардуер се генерира за реализацията на двата процеса?

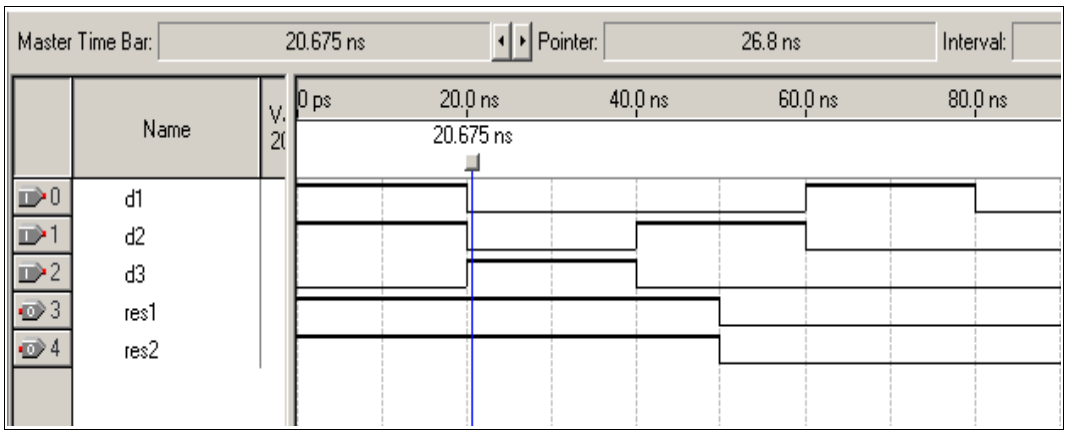
За да разберете действителната схема на генерирания хардуер, след като компилирате успешно проекта използвайки менюто **Tools->Netlist Viewers->RTL Viewers** проверете RTL схемата на вашия дизайн (Фиг. 40). Дефинирани са 2 идентични схеми използващи по 1 елемент „И“ и един елемент „ИЗКЛЮЧВАЩО ИЛИ“. За да откриете къде точно седят дефинираните от вас сигнал и променлива разлистете списъка **Herarchy List**, и от **Nets** изберете **sig_s2** и **proces1:var_s1,** тази команда ще оцвети в червено съответният избран от вас елемент.



Фиг. 40 RTL Viewer

Симулация

След като се убедихме в истинноста, че двата процеса генерират еднакъв хардуер остана една особеност, да извършим симулация на тяхното поведение и да се убедим и в еднаквостта на получените резултати. За целта добавете нов симулационен файл и създайте показаната по-долу тестова последователност. Резултатите трябва да са идентични с показаните изходни нива на изходни портове **res1** и **res2** (Фиг. 41).



Фиг. 41 Тестова последователност за проверка действието на двата процеса

Задача 1: Дефинирайте нов еднобитов изходен порт в ентитито и трети процес в рамките на същата архитектура, който да реализира функцията:

y = d1 & d2 & d3

Модифицирайте симулационния файл и извършете симулация с направените промени.

Задача 2: Обяснете как би могло да стане предаването на данни между два независими процеса, как бихме могли да синхронизираме изпълнанието на два процеса, които трябва да се стартират един след друг.

Дефиниране на константи, променливи и сигнали

По-горе се запознахме как могат да бъдат дефинирани и използвани променливи, сигнали и процеси във VHDL моделите. Като всеки език за програмиране VHDL разполага с широк набор типове променливи и оператори, които дизайнерите могат да ползват наготово. Често ползвани при симулиране на различни хардуерни модели са константите. Те се употребяват, когато дадена стойност на променлива трябва да остане неизменна по време на симулацията. Константи могат да бъдат дефинирани за всички типове данни по следния начин:

```
constant SIGN: std_logic := '1';
constant I : integer := 1;
```

Дефинирането на променливи и сигнали беше описано по-горе, тук само за пример ще посочим следната дефиниция с начална инициализация:

```
signal S : std_logic := '1';
variable V : std_logic := '0';
```

Типове оператори във VHDL

Като всеки език за програмиране и VHDL разполага с широк набор предварително дефинирани логически и аритметични оператори, като основните от тях са:

Булеви: AND, OR, NOT, NAND, NOR, XOR, XNOR;

Аритметични:

+ - * / събиране, изваждане, умножение и деление
ABS изчислява абсолютна стойност
MOD дефинира се като: **A mod B = A – B * N**
REM дефинира се като: **A rem B = A – (A/B) * B**
** изчислява експонента на цяло число, като резултата може да бъде целочислен или с плаваща точка
примери:
A <= “1000”;
B <= “0001”;
C <= A & B; - - в този случай C ще е равно на “10000001“
7 mod 4 резултат **3**
-7 mod 4 резултат **–3**
7 mod (-4) резултат **–1**
-7 mod (-4) резултат **–3**

Оператори за преместване:

sll логическо отместване вляво
srl логичуско отместване вдясно
sla аритметично отместване вляво
sra аритметично отместване вдясно
rol ротиране наляво
ror ротиране надясно

примери:
“1100” sll 1 резултат **“1000”**
“1100” srl 2 резултат **“0011”**
“1100” sla 1 резултат **“1000”**
“1100” sra 2 резултат **“1111”**
“1100” rol 1 резултат **“1001”**
“1100” ror 2 резултат **“0011”**
“1100” ror –1 същото като **“1100” rol 1** резултат **“1001”**

Оператори за сравнение:

= проверка за равенство
/= проверка за неравенство
< по-малко от
<= по-малко или равно на
> по-голямо от
>= по-голямо или равно на

Оператори за условно разклоняване:

Това са оператори за проверка на условие: **IF-THEN ELSE** и **CASE**. Действието на оператор **IF** бе разгледано при реализацията на 8 битов тристабилен драйвер. Този оператор може да се ползва за еднократна проверка на условие:

```
if (условие) then
        изпълни някакво действие;
end if;
```

и за проверка ня няколко условия:

```
if (условие 1) then
        изпълни някакво действие;
elsif (условие 2) then
        изпълни някакво действие;
else;
        изпълни някакво действие;
end if;
```

Оператор **CASE** обикновенно се ползва за проверка на няколко еднотипни варианта, този тип проверка на условие е особено полезно при дефинирането на състоянията на дадено устройство при описание на крайни автомати или за многократна проверка на условие имащо еднаква дължина:

```
signal TEST : std_logic_vector (2 downto 0);
signal A : std_logic_vector (1 downto 0);

case TEST is
        when “001” =>
                A <= “01”;
        when “010” =>
                A <= “10”;
        when “011” =>
                A <= “11”;
        when others =>
                A <= “00”;
end case;
```

По-подробни примери за използване на **CASE** ще бъдат разгледани по-долу, тук следва да се обърне внимание на „**when others”,** което ви позволява да дефинирате действие на устройството за неизвестен в момента на дизайн входен набор. Това ви позволява да дефинирате така наречения безопасен изход от функия, за която нямате предварителни данни какви биха били всички варианти на входните величини. Друго преимущество, е че така следващите версии могат да поддържат повече опции, които в предходните не са били реализирани, но са планирани като възможност.

Цикли

Съществуват два типа цикли **FOR** и **WHILE** (вторите не са изпълними при VHDL синтеза, тъй като обикновенно дават грешка в процеса на компилация). Цикълът **FOR** позволява да изпълним дадено действие определен брой пъти:

```
signal C : std_logic_vector (7 downto 0);
for i in 0 to 7 loop
        C(i) <= '1'; -- инициализираме последователно
                        -- всеки елемент от C с единица
end loop;
```

ТРИГЕРИ – RS, D, JK

По типа на своето действие тригерите биват асинхронни и синхронни, има три типа синхронни тригери:

- **с управление по ниво на сигнала** - ползва се отделен разрешаващ сигна (EN – Enable);
- **с управление по фронт на сигнала** - превключват се само при преход на синхро сигнала (от 0 към 1 или от 1 към 0)
- **двустъпални** – при тях при подаден синхронизиращ сигнал първото стъпало приема входните данни, при преминаване на синхронизиращия сигнал в '0' сигнала се прехвърля във второто стъпало.

За нуждите на това упражнение ще реализираме проект с VHDL описание на три тригера: RS, D и JK. Трите тригера ще бъдат описани, като различни архитектури в рамките на един и същи VHDL файл. За целта създайте нов проект, използвайте чип MAX7000S с 64 макроклетки. Кръстете проекта **Trigger**, а името на главното ентити кръстете **TOP**. Създайте нов VHDL файл и въведете даденото по-долу описание.

Ентити

Входовете на трите тригера са описани в едно ентити, разбира се могат да бъдат описани в отделни ентитита с различни архитектури, като при това се ползват като модули или пакети в по-сложни дизайни.

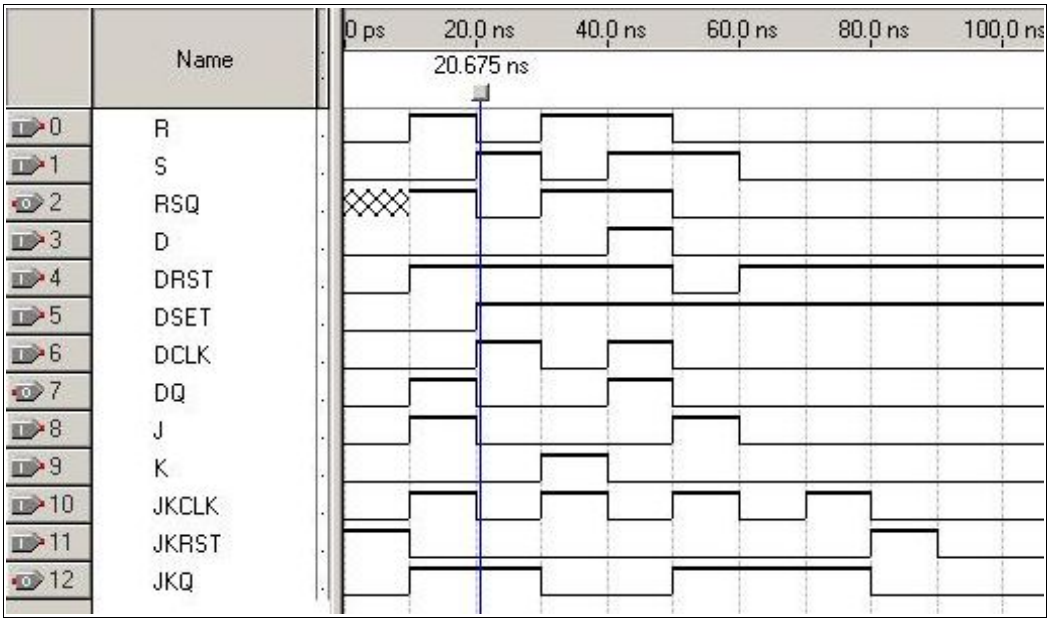
```
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity TOP is
generic ( n : integer := 8);

port(
        R: in std_logic; S: in std_logic; RSQ: out std_logic;
        D: in std_logic; DRST: in std_logic; DSET: in std_logic;
        DCLK: in std_logic; DQ: out std_logic;
        J: in std_logic; K: in std_logic; JKCLK: in std_logic;
        JKRST: in std_logic; JKQ: out std_logic
);
end TOP;
----------------------------------------
```

Архитектура

```
-----------------------------------------
architecture TRIGGER of TOP is
begin
	RS : process (R, S)
	begin
		if (R='0' and S='1') then
			RSQ <= '0';
		elsif (R='1' and S='0') then
			RSQ <= '1';
		end if;
	end process;
	JK : process (J, K, JKCLK, JKRST)
		variable state : std_logic;
	begin
		if (JKRST='1') then
			state := '0';
		elsif (JKCLK='1' and JKCLK'event) then
			if(J='1' and K='1') then
				state := not state;
			elsif (J='1' and K='0') then
				state := '1';
			elsif (J='0' and K='1') then
				state := '0';
			elsif (J='0' and K='0') then
				state := state;
			end if;
		end if;
		JKQ <= state;
	end process;
	DD: process (D, DRST, DSET, DCLK)
	begin
		if (DRST = '0') then
			DQ <= '0';
		elsif (DSET = '0') then
			DQ <= '1';
		elsif (DCLK = '1' and DCLK'event) then
			DQ <= D;
		end if;
	end process;
end TRIGGER;
-----------------------------------------
```

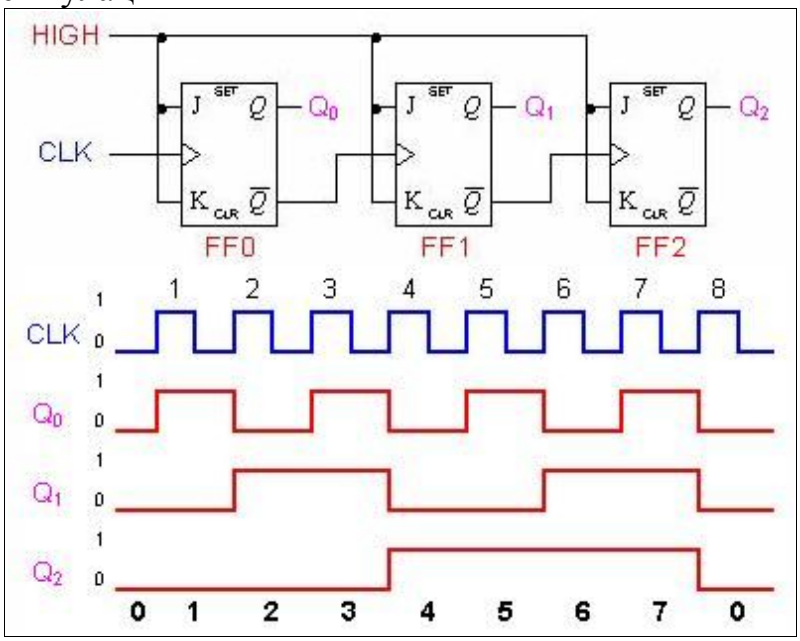
Симулация

За да извършите симулация на тригерите създайте файл със следната последователност от входни сигнали. Изходните сигнали са дадени съответно в редове: **RSQ, DQ** и **JKQ**.



Фиг. 42 Тестова последователност за проверка действието на трите тригера

Задача 1: Реализирайте дизайн съдържащ една архитектура с JK тригер, генерирайте от него символен файл, който да ползвате в схемотехниески дизайн даден на фигура 43. След компилацията на проекта извършете неговата симулация



Фиг. 43 Реализация на брояч с JK тригери

Многократно използване на програмен код

Много често в програмирането, независимо от ползвания език се налага многократното ползване на един и същи код. За да не се налага системното преписване на тези сегменти от програмата, повечето езици предлагат разнообразни варианти за решаване на този проблем. При асемблерните програми това се решава с макроси, при програмите писани на бейсик това са процедури, а при С програмите това са функции. VHDL също предлага подобна функционалност. Това става или чрез **функции, пакети, компоненти и процедури**. Следва да се отбележи, че за оптималното синтезиране на описвания хардуер следва да се обърне особено внимание на това кой тип инкапсулация ще бъде ползван, това не е обект на настоящия учебник.

За нуждите на това упражнение ще разгледаме създаването и ползването на елементарен пакет, реализиращ една единствена функция за умножение на две числа. Програмния код на пакета може да се ползва в основното описание, само ако бъде дефиниран съответният пакет в списъка с декларации в началото на програмата. Създайте нов проект и го кръстете **Package** с име на главното ентити **TOP**, оставете настройките по подразбиране, като в примера разгледан по-горе. Създайте нов VHDL файл и въведете даденото по-долу описание. След като въведете кода съхранете файла с име **Package.vhd**.

```
----------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use ieee.std_logic_arith.ALL;

package PACKAGE1 is
        constant n : integer := 8;
function MULTIPLY (A, B : std_logic_vector) return std_logic_vector;
end package;

package body PACKAGE1 is

function MULTIPLY  (A, B : std_logic_vector) return std_logic_vector is
                variable C : std_logic_vector (n*2-1 downto 0);
                begin
                                C := A * B;
                return C;
        end function MULTIPLY;
end package body;
----------------------------------------
```

Описанието на конструкиите тук е подобно, като при декларацията на ентити и архитектура, но вместо ентити имате декларация на пакет и тяло на пакет. В декларацията на пакета се описват включените в пакета функции, а в тялото му всяка една от включените функции. В пакетите и функциите може да използвате само променливи, ползването на сигнали ще доведе до грешки в процеса на компилиране. В случая имаме само една функция в пакета. За да можем да ползваме току що въведената функция и да симулираме нейното поведение трябва да я използваме в архитектура. Следва да създадете нов VHDL файл и да въведете даденото ентити и архитектура.

Ентити

Обърнете внимание, че ползваната константа **n** е дефинирана в тялото на пакета, а не в ентитито. Изходния порт е с два пъти по-голяма разрядност от входните, *за справка вижде в част I - умножение на двоични числа*.

```
----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use work.PACKAGE1.all;

entity TOP is
port(
        IN1 : in std_logic_vector (n-1 downto 0);
        IN2 : in std_logic_vector (n-1 downto 0);
        OUT1 : out std_logic_vector (2*n-1 downto 0)
);
end TOP;
----------------------------------------------
```

Архитектура

В декларираната архитектура ползваме готова функция **MULTIPLY** декларирана в пакета **PACKAGE1**, нейни параметри са данните постъпващи от портове **IN1** и **IN2**, а изхода се извежда през порт **OUT1**.

```
----------------------------------------------
architecture MODULE of TOP is
begin
        OUT1 <= MULTIPLY(IN1, IN2);
end MODULE;
----------------------------------------------
```

Задача 1: Извършете симулация на проекта, обяснете резултата и корегирайте параметрите на симулацията, ако се налага.

Задача 2: Добавете нова функция в пакета, която да реализира операцията събиране C := A + B. Използвайте нововъведената функция в архитектурата, добавете съответен изходен порт за визуализация на резултата, компилирайте проекта. Извършете симулация на новосъздадената функционалност. *Заб. Следва да предвидите подравняване на A и B към разрядността на C.*

```
variable C : std_logic_vector (n downto 0);
C := ('0' & A)+('0' & B);
```

Мултиплексор 8 към 1

Този тип устройства ви позволяват да предаваме данни от множество източници по една шина. Те спадат към комбинационните логически устройства, но функционирането им е последователно във времето, тъй като използването на изходната шина може да става последователно във времето. Например може да комутирате няколко входни потока от битове към няколко изходни потока. Мултиплексирането е процес, който се ползва в системите за пренос на информация. Съществуват няколко вида мултиплексиране: по време, по честота и по код. Разглежданият тук мултиплексор може да комутира 8 входа (8 бита всеки) към един 8 битов изход. Изборът на вход ще се осъществяма с подаването на 4 битова кодова дума, която след дешифриране ще позволява комутацията на определен вход към изхода на устройството.

Създайте нов проект с име „**MUX**” и име на главното ентити „**TOP**”, тъй като нашия дизайн ще използва като минимум 75 пина следва да изберем матрица с по-високи възможности **MAX7000B** (**100** извода). Създайте нов VHDL текстов файл и въведете дадения по-долу код. В него са дефинирани всички портове на мултиплексора: 1 разрешаващ вход **EN**, 8 входа за данни, 1 вход за подаване на сигнал за избор на вход и 1 изходен порт. Входните и изходни портове за данни са 8 битови и са от типа **std_logic_vector.**

Ентити

Възможно е да дефинирате ентити на мултиплексор с повече входове, например за реализация на 64 входа, каквато би била нужда при реализация на

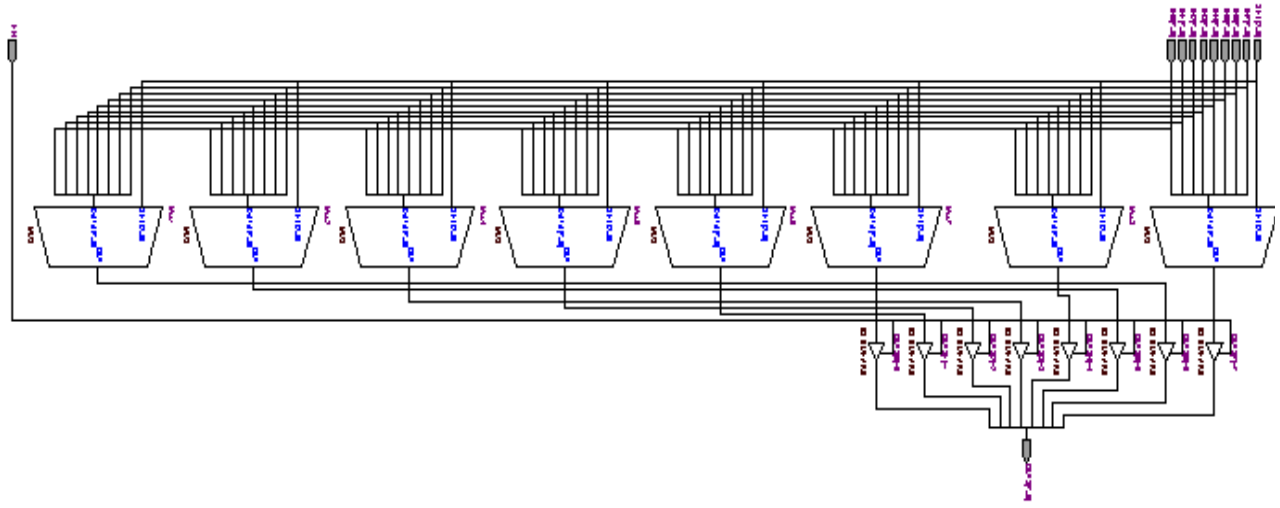
```
-------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity TOP is
        port(
                EN:   in std_logic;
                IN7:  in std_logic_vector(7 downto 0);
                IN6:  in std_logic_vector(7 downto 0);
                IN5:  in std_logic_vector(7 downto 0);
                IN4:  in std_logic_vector(7 downto 0);
                IN3:  in std_logic_vector(7 downto 0);
                IN2:  in std_logic_vector(7 downto 0);
                IN1:  in std_logic_vector(7 downto 0);
                IN0:  in std_logic_vector(7 downto 0);
                SEL: in std_logic_vector(2 downto 0);
                OUT8: out std_logic_vector(7 downto 0)
);
end TOP;
-------------------------------------------
```

В архитектурата на мултиплексора трябва да се осигури следната функционалност. При подаване на разрешаващ сигнал EN=1 мултиплексора трябва да бъде активиран, в противен случай изхода му OUT8 следва да се установява във високо импедансно състояние „Z”. Когато мултиплексорът е в активно състояние всяка смяна на входните сигнали на останалите портове следва да предизвиква смяна на изходния сигнал. Функционалността на мултиплексора следва да се имплементира в отделен процес, който да се активира при смяна на всеки един от входните сигнали (**IN7, IN6, IN5, IN4, IN3, IN2, IN1, IN0, SEL**). Изборът на входен 8 битов порт, който следва да бъде комутиран към 8 битовия изход става с подаване на 3 битова кодова комбинация на вход **SEL**. За опростяване на дизайна няма да има синхронизация на мултиплексора. При реализация на сложни схеми, при които от особено значение е правилната работа на устройството по време следва да се предвиди метод за синхронизация с тактов сигнал.

Архитектура

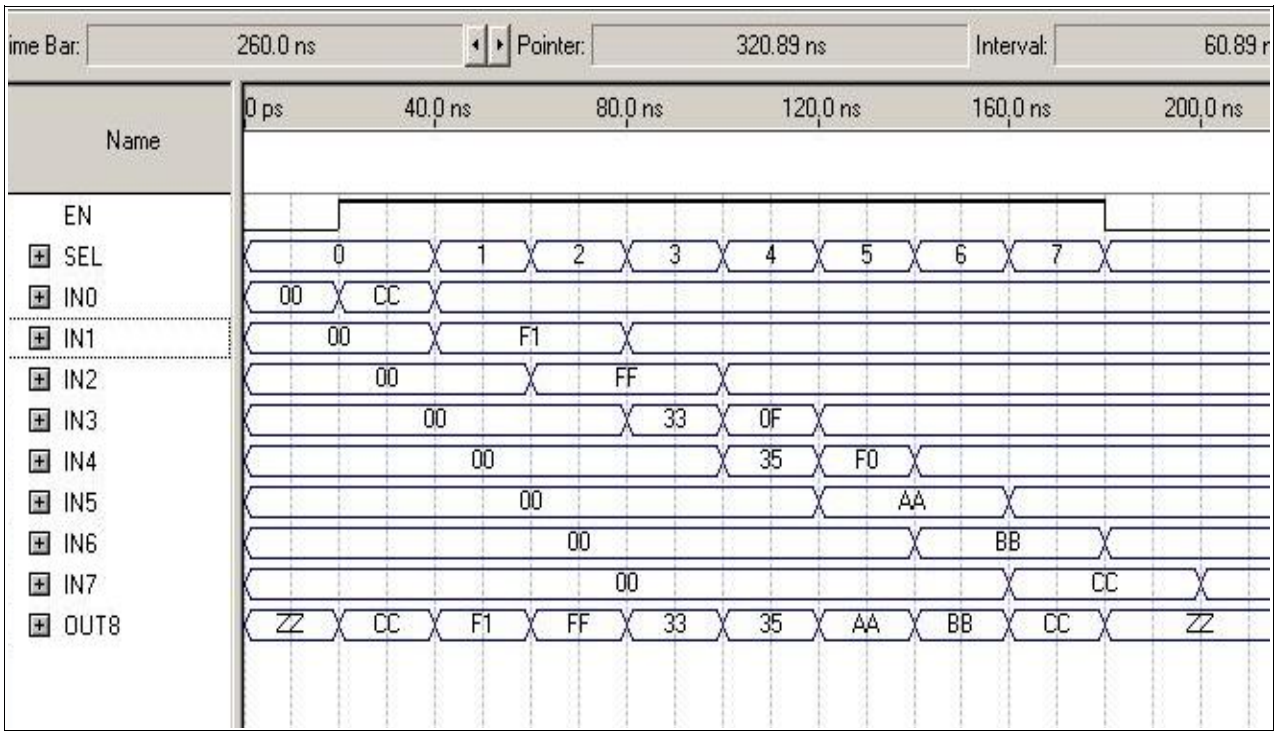
```
-------------------------------------------------
architecture MUX_8 of TOP is
begin
   multiplexing:  process(IN7, IN6, IN5, IN4, IN3, IN2, IN1, IN0, SEL)
   begin
       if(EN = '1') then
              case SEL is
                     when "000" =>    OUT8 <= IN0;
                     when "001" =>    OUT8 <= IN1;
                     when "010" =>    OUT8 <= IN2;
                     when "011" =>    OUT8 <= IN3;
                     when "100" =>    OUT8 <= IN4;
                     when "101" =>    OUT8 <= IN5;
                     when "110" =>    OUT8 <= IN6;
                     when "111" =>    OUT8 <= IN7;
                     when others =>   OUT8 <= "ZZZZZZZZ";
              end case;
       else
              OUT8 <= "ZZZZZZZZ";
       end if;
   end process;
end MUX_8;
-------------------------------------------------
```

След като приключите описанието и компилацията на схемата погледнете генерирания хардуер през **Tools->Netlist Viewer->RTL Viewer**, който следва да изглежда по начина даден по-долу.



Фиг. 44 Синтезираната схема на мултиплексора

За да приключим работата по този проект следва да извършим симулация, доказваща правилността на работата на мултиплексора (Фиг. 45). Генерирайте нов симулационен файл, обърнете внимание, че за по-добра прегледност изходния формат на 8 битовите шини следва да бъде в шестайсетичен код – **hexadecimal**. Променяйки стойностите на входните сигнали проверете за коректността при работа на синтезираното устройство.



Фиг. 45 Симулация работата на мултиплексора

Задача 1: Добавете още един 8 битов вход за данни, като обезпечите функционалност за неговата комутация към общия изход. Какви изменения в ентитито на проекта следва да предвидите, какви изменения в архитектурата са нужни? Компилирайте и симулирайте действието на модифицираното устройство.

Задача 2: Колко входни порта би могло да обслужи модифицираното устройство имащо 4 битова шина за селекция на входен порт?

Демултиплексор от 1 към 8

Демултиплексорът (или декодер) е комбинационно логическо устройство по функции обратно на мултиплексора. При него данните постъпващи през един вход се комутират към множество изходи в зависимост от управляващия сигнал. Демултиплексорът който ще разгледаме има един 8 битов вход, един 3 битов вход за подаване на код за селекция на изхода, един разрешаващ вход за цялата схема и 8x8 битови изхода. Създайте нов проект, като единствената разлика с предишния следва да бъде избора на логическа матрица, за избягване на грешки към настоящия момент изберете чип **EPM7512BFC256-5**, който има достатъчно количество изходни портове. След това създайте ном VHDL файл и въведете даденото по-долу описание на ентити и архитектура. Съществената разлика с предния пример, е че тук се ползват 8x8 бита изхода и 1x8 бита вход с 1x3 бита селектиращ вход.

Ентити

```
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity TOP is
port(
        EN:   in std_logic;
        OUT7:       out std_logic_vector(7 downto 0);
        OUT6:       out std_logic_vector(7 downto 0);
        OUT5:       out std_logic_vector(7 downto 0);
        OUT4:       out std_logic_vector(7 downto 0);
        OUT3:       out std_logic_vector(7 downto 0);
        OUT2:       out std_logic_vector(7 downto 0);
        OUT1:       out std_logic_vector(7 downto 0);
        OUT0:       out std_logic_vector(7 downto 0);

        SEL: in std_logic_vector(2 downto 0);
        IN8:  in std_logic_vector(7 downto 0)
);
end TOP;
-----------------------------------------------
```

Функционалността, която следва да бъде реализирана в архитектурата е следната: когато имаме разрешаващ сигнал EN=1 следва да прочетем данните подавани на входния порт за селекция на изхода. В зависимост от подадената кодова комбинация прочетените 8 бита ще бъдат изведени на съответния изходен порт. Изходният порт ще помни последно въведените в него данни до

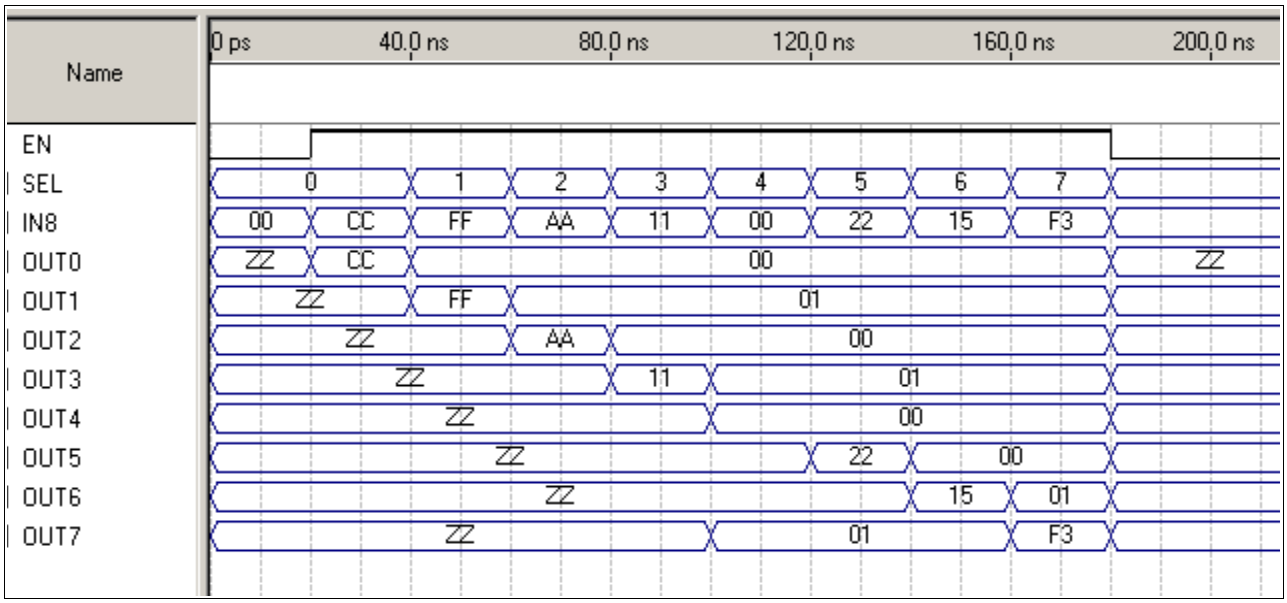
момента в който не бъдат записани нови. В зависимост от конфигурацията и нужната функционалност в практиката може да се ползват и допълнителни сигнали за външна сонхронизация, които не са реализирани в този модел. Когато EN=0 всички 8 изходни порта ще бъдат установявани във високо импедансно състояние Z. След компилация на схемата погледнете RTL схемата и я съпоставете със схемата на мултиплексора. Архитектурата разгледана по-долу може да бъде изпълнена и с **CASE** и с **IF** условни оператори.

Архитектура

```
-----------------------------------------------
architecture DE_MUX_8 of TOP is
begin
   multiplexing: process(IN8, SEL)
   begin
       if(EN = '1') then
              case SEL is
                     when "000" =>   OUT0 <= IN8;
                     when "001" =>   OUT1 <= IN8;
                     when "010" =>   OUT2 <= IN8;
                     when "011" =>   OUT3 <= IN8;
                     when "100" =>   OUT4 <= IN8;
                     when "101" =>   OUT5 <= IN8;
                     when "110" =>   OUT6 <= IN8;
                     when "111" =>   OUT7 <= IN8;
              end case;
       else
              OUT0 <= "ZZZZZZZZ";
              OUT1 <= "ZZZZZZZZ";
              OUT2 <= "ZZZZZZZZ";
              OUT3 <= "ZZZZZZZZ";
              OUT4 <= "ZZZZZZZZ";
              OUT5 <= "ZZZZZZZZ";
              OUT6 <= "ZZZZZZZZ";
              OUT7 <= "ZZZZZZZZ";
       end if;
   end process;
end DE_MUX_8;
-----------------------------------------------
```

Симулация

Създайте нов симулационен файл и въведете дадената по-долу тестова последователност (Фиг. 46). Обърнете внимание на начина на извеждане на данните, входният и изходни 8 битови портове следва да показват данните в шеснадесетичен формат.



Фиг. 46 Симулация на демултиплексора, обърнете внимание, че след запис на данни в изходен порт те се изменят неконтролируемо

Задача 1: Предложете изменения в кода, така че след запис на стойност в изходен порт тя да остава непроменена до следващия запис.

Задача 2: Реализирайте архитектурата на демултиплексора с **IF** оператор за проверка, ползвайте подобна конструкция:

```
if(SEL = "000") then
        OUT0 <= IN8;
elsif (SEL = "001")  then
        OUT1 <= IN8;
...
else
...
end if;
```

Задача 3: Сравнете RTL схемата генерирана при описание с **CASE** и **IF** оператори за проверка на условие.

8 битов паралелен регистър

В предходните упражнения реализирахме схеми ползващи многобитови портове за паралелно въвеждане и извеждане на данни. Неявно изходните регистри, които реализирахме при мултиплексора и демултиплексора представляват **D** тригени групи. Характерно за тях, е че веднъж записани данните, състоянията на изходите им остават непроменени до следващ запис на нови данни в тях. Тук ще реализираме 8 битов паралелен регистър, който ще има възможнаст за установяване на изходния порт във високоимпедансно състояние - **EN**, синхронен запис на данните при подаване на тактов сигнал - **CLK**, сигнал за нулиране на регистъра – **CLR**, и съответно 8 битов порт **DIN_8,** и 8 битов изходен порт **DOUT_8**.

Създайте нов проект, тъй като сложността на устройството е относително ниска, може да ползвате чип **MAX7000S** с **64** макроклетки. Кръстете проекта **FlipFlop**, а името на главното ентити кръстете **TOP**. Създайте нов VHDL файл и въведете даденото по-долу описание.

Ентити

```
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity TOP is
        generic(n: natural :=8);
port(
        IN_8:in std_logic_vector(n-1 downto 0);
        CLK:in std_logic;
        EN:      in std_logic;
        CLR:in std_logic;
        OUT_8:      out std_logic_vector(n-1 downto 0)
);
end TOP;
-------------------------------------------------
```

Архитектура

При реализацията на архитектурата ще разгледаме две нейни версии. Първата ще интегрира цялата функционалност в един единствен работен процес, а втората ще използва два отделни процеса. Целта на това упражнение е да се сравнят по сложност схемите генерирани от компилатора и да се убедите, че за оптимално решаване на паралелни изчислителни задачи, проблемът е по-

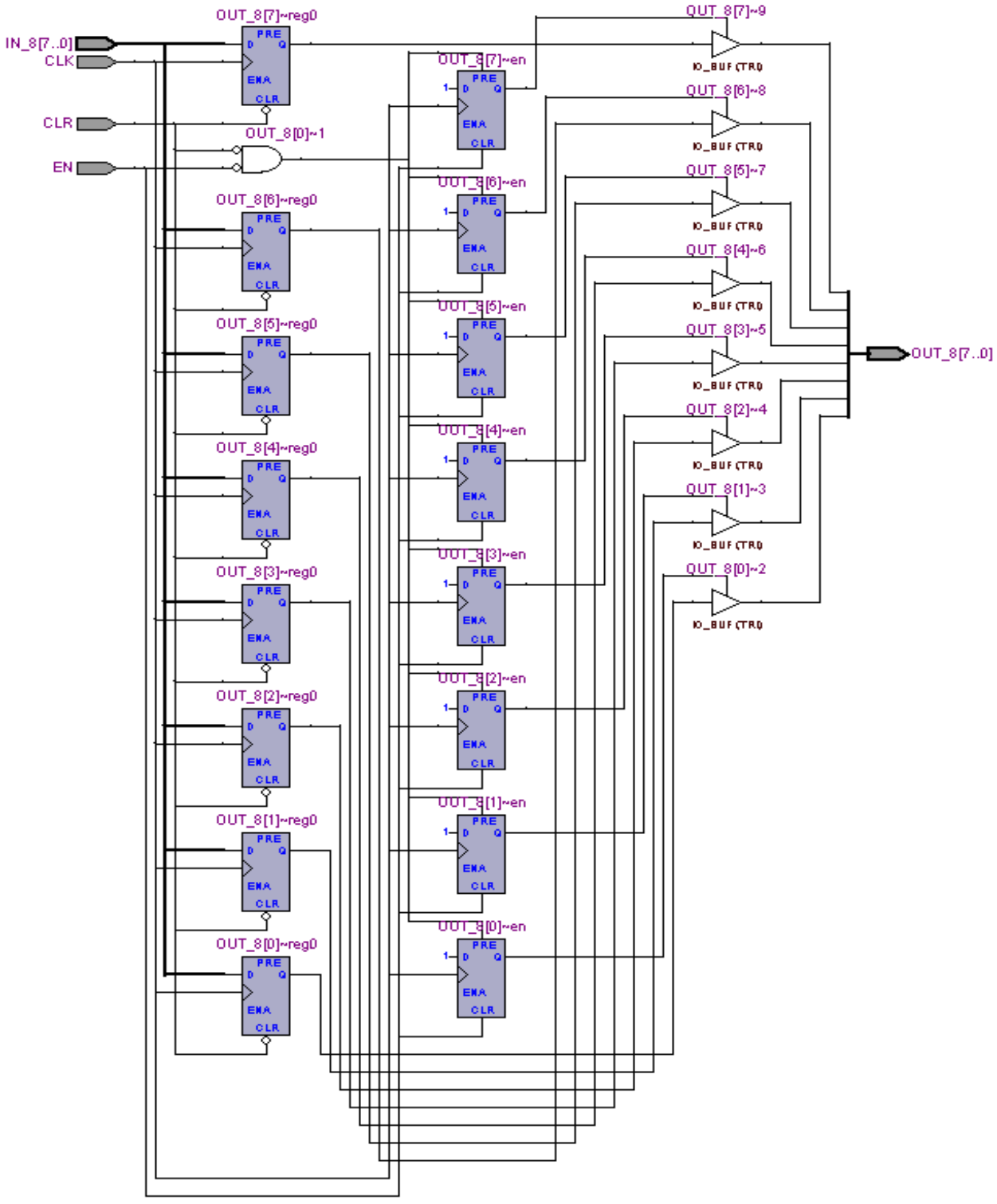
добре да бъде разбит на отделни взаимно свързани процеси. Този подход ще позволи по-лесен синтез на схемата на устройството и е за предпочитане пред интегрирането на цялата функционалност в един единствен процес.

Архитектура 1

Тази версия на архитектурата има следната функционалност, изпълнявана от един единствен процес: проверява се дали чипът е селектиран **EN=0**, след това се прави проверка за това дали имаме нарастващ фронт на CLK и дали **CLK** е установен в '1', ако това е така данните от входния порт се подават на изходния, ако обаче имаме подаден сигнал **CLR** в регистъра се записват нули, при подаден сигнал **EN=1** изходния порт се извежда във високоимпедансно състояние '**Z**'.

```
-------------------------------------------------
architecture D_FlipFlop_8 of TOP is
begin
   process(EN, CLR, CLK, IN_8)
   begin
              if( EN = '0')then
                    if (CLR = '0') then
                          OUT_8 <= "00000000";
                    elsif (CLK ='1' and CLK'event) then
                          OUT_8 <= IN_8;
                    end if;
              else
                    OUT_8 <= "ZZZZZZZZ";
              end if;
   end process;
end D_FlipFlop_8;
-------------------------------------------------
```

След успешно компилиране на тази архитектура разглейдайте RTL схемата (Фиг. 47). Опитайте да обясните защо са използвани 2 набора по 8-D тригера. Какво би могло да се направи, за да се намали броят ползвани логически елементи и тригери в нея, без да се променя функционалността и. Отговорът е да опитаме да разбием изпълняваните проверки и действия в отделни процеси.

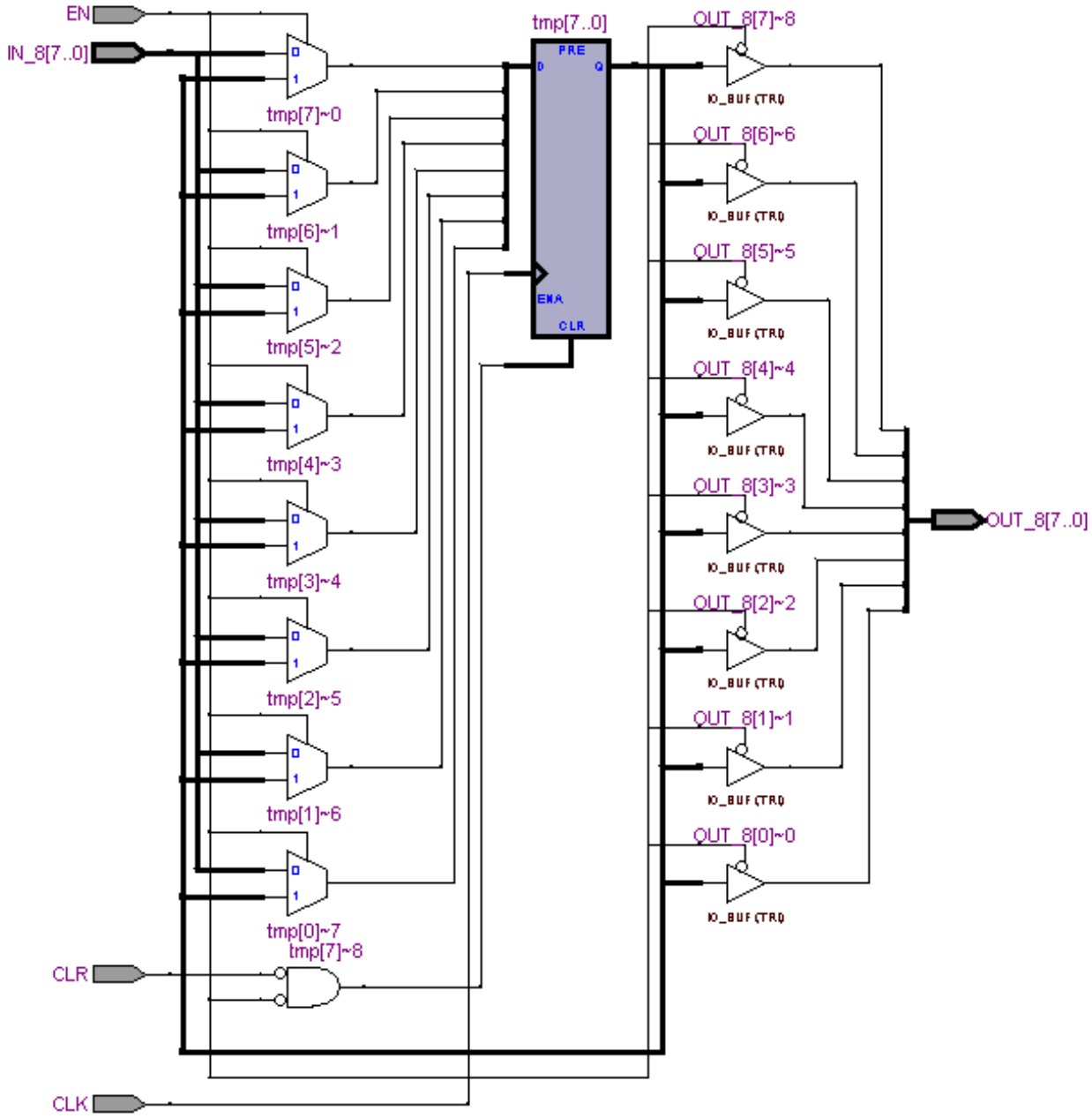


Фиг. 47 RTL схема синтезирана от първото описание на архитектурата, отнема 9 макроклетки от PLD чипа.

Архитектура 2

За да ускорим действието на схемата и да облегчим изпълнявания процес, ще опитаме да реализираме същата функционалност с 2 независими процеса и една променлива. В този случай първият процес ще следи за промяна на входните сигнали и ще записва данните в междинна променлива. При изменение на тази променлива, вторият процес в зависимост от сигнала подаден на вход **EN** ще изведе данните от променливата в изходния порт или ще го установи във високо импедансно състояние **'Z'**. Този подход добре онагледява, как разбиването на един проблем на отделни паралелно изпълняващи се части може значително да опрости реализирания хардуер, което е видно и от RTL схемата (Фиг. 48).

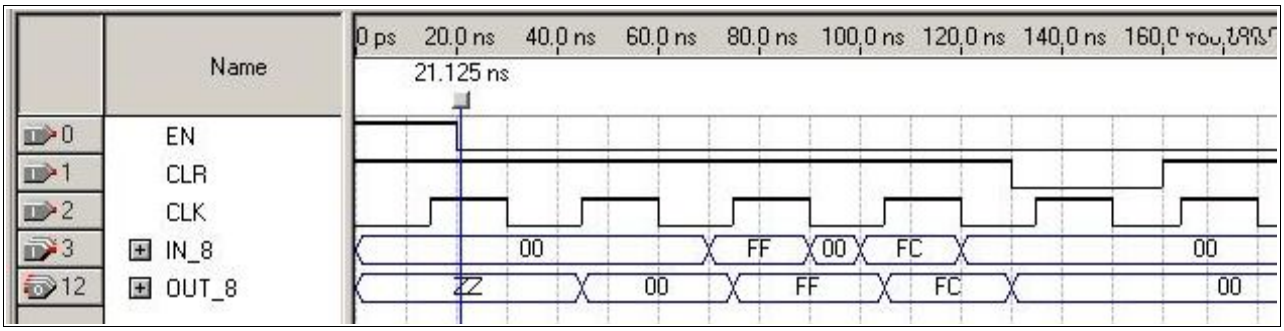
```
-----------------------------------------------------------------
architecture D_FlipFlop_8_1 of TOP is
        signal tmp: std_logic_vector(n-1 downto 0);
begin
  -- процес 1
   process (EN, CLR, CLK, IN_8)
   begin
                if( EN = '0')then

                        if (CLR = '0') then
                                tmp <= "00000000";
                        elsif (CLK ='1' and CLK'event) then
                                tmp <= IN_8;
                        end if;
                end if;
   end process;
   -- процес 2
   process (EN, tmp)
        begin
                if (EN = '1') then
                        OUT_8 <= "ZZZZZZZZ";
                else
                        OUT_8 <= tmp;
                end if;
        end process;
end D_FlipFlop_8_1;
-----------------------------------------------------------------
```



Фиг. 48 RTL схема синтезирана от второто описание на архитектурата, отнема 8 макроклетки от PLD чипа.

Симулация

За да симулирате действието на синтезираната схема създайте нов файл за симулация и въведете дадените по-долу стойности за входните сигнали (Фиг. 49). Получената от вас симулация би следвало да не се отличава от дадената тук. Извършете симулация за двата модела на архитектурата, които разгледахме по-горе.



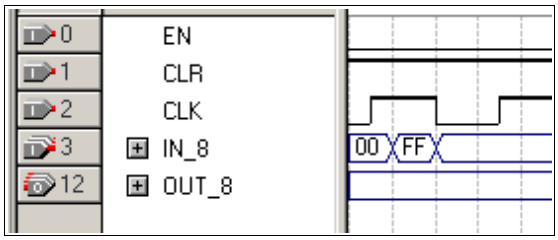
Фиг. 49 Времедиаграма от симулацията на действието на паралелния регистър

Задача 1: Обяснете защо при падането на сигнал **EN=0** и при подаден тактов сигнал **CLK=1** в изходния порт не се извеждат подадените на вход **IN_8=00**, а изходния порт остава във високоимпедансно състояние?

Задача 2: Защо във времеинтервала 90ns – 100ns при подадени данни **IN_8=00** изхода остава непроменен?

Задача 3: Защо е нужно данните на вход **IN_8** да са подадени преди установяването на тактовия сигнал **CLK** в **1**?

Задача 4: Променете така времедиаграмата на симулацията (виж Фиг. 50), че данните които подавате на входния порт **IN_8** да се установят след като сигнал **CLK** е установен в **1**, анализирайте резултата.



Фиг. 50

4 битов брояч

Броячите намират приложение при реализацията на широка гама логически устройства: за делене на честота, за отчитане на броя постъпили импулси, за реализация на крайни автомати, за реализация на преместващи регистри и др. Конкретната реализяция е на 3 битов брояч. Реализираният модул има сигнал за нулиране **CLR**, сигнал за отчитане на входните импулси **CLK**, сигнал за арзрешаване и спиране процеса на борене **CE** и 3 битов паралелен изходен порт **OUT_3**. *Реализацията на тази схема има отношение към релизацията на следващото упражнение.*

Ентити

```
-------------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;
use IEEE.numeric_std.all;

entity TOP is
        generic(n: natural :=4);
port(
        CLK: in std_logic;
        CLR: in std_logic;
        CE   : in std_logic;
        OUT_4:  out std_logic_vector(n-1 downto 0)
);
end TOP;
-------------------------------------------------
```

Архитектура

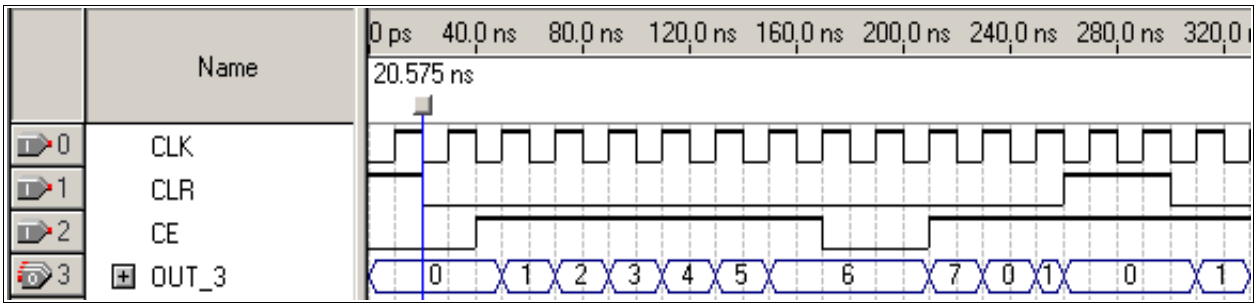
Архитектурата на брояча ще реализира следната функционалност: при подаване на **CLR=0** ще се нулира изходна, при регистрация на нарастващ фронт на тактовия сигнал **CLK** ще се инкрементира стойността на брояча ако сигнал **EN=1**.

```
-------------------------------------------------
architecture Counter of TOP is
        signal tmp: std_logic_vector(n-1 downto 0);
begin
        COUNT: process(CLK, CE, CLR)
         begin
                if CLR = '1' then
                        tmp <= "0000";
                elsif (CLK='1' and CLK'event) then
```

```
                    if (CE = '1') then
                        tmp <= tmp + 1;
                    end if;
                end if;
        end process;
OUT_3 <= tmp;
end Counter;
-------------------------------------------------
```

Симулация

Създайте нов файл за симулация и въведете по показания по-долу начин съответните входни и изходни сигнали. Изберете времеинтервал на тактовия импулс 20ns със запълване 50%.



Фиг. 51 Симулация действието на брояча

Задача 1: Променете кода на брояча, така че да брои до 255. *Заб. Изменете разрядността на изходния паралелен порт и променливата ползвана за съхраняване на междинния резултат.*

Задача 2: Предложете модификация, така че когато брояча достигне зададена стойност процеса на броене да прекъсне до следващо нулиране. Добавете един еднобитов сигнал в архитектурата и го кръстете S, добавете още един процес в архитектурата. След успешната компилация извършете симулация, коментирайте наблюдаваните процеси.

```
signal S: std_logic;
STOP: process(tmp, CLR)
begin
        if(tmp = "1000") then
                S <= '0';
        end if;
        if (CLR='1') then
                S <='1';
        end if;
end process;
```

8 битов преместващ регистър с паралелен вход и сериен изход

Упражнението е за самостоятелна работа.

При реализацията на комуникационни интерфейси често се използват преместващи регистри. Повечето от тях имат интегриран паралелен вход/изход, и сериен вход/изход. Този тип регистри се наричат универсални, като пълната функционалност не винаги е нужна. В този пример ще разгледаме 8 битов паралелен регистър със синхронен сериен изход. За повече информация вижте потърсете в google описанието на интегрална схема **74HC165**. Веднъж записани данните в регистъра, ще бъдат извеждани от него при нисък фронт на тактовия импулс **CLK**. Този начин за извеждане на данни се ползва при реализацията на **SPI** серийни интерфейси налични в широка гама интегрални схеми и готови електронни модули.

Този тип устройства най-добре се описват с крайни автомати, но поради тяхната специфика те ще бъдат детайлно разгледани в част III на този учебник. За нуждите на това упражнение ще използваме придобитите до тук умения за направа на броячен регистър с нулиране и създаването на мултиплексор. За целта създайте нов проект с име **PinCount**, оставете **TOP** за име на главното ентити, изберете чип от серията **MAX7000S** с **64** макроклетки. Функционалността която устройството реализира е следната: при началното установяване на регистъра серийния изход се установява във високо импедансно състояние **SOUT='Z',** при подаден сигнал **PL=0** в регистъра ще може да се запишат данните подадени на паралелния вход **IN_8**. При подаване на **CLR=1** броячният регистър ще се нулира, това действие е последвано от подаване на **CS=1**, което ще инициира процесът на последователно извеждане на осемте бита. Всеки пореден бит ще бъде изведен при нисък фронт на сигнал **CLK**. След като бъдат изведени 8 бита, броячът ще спре инкрементацията, а серийният изход ще бъде изведен отново във високо импедансно състояние **SOUT='Z'**. Създайте нов VHDL файл и въведете дадения по-долу код, след което компилирайте проекта.

Ентити

```
----------------------------------------------
library ieee ;
use ieee.std_logic_1164.all;
use ieee.std_logic_unsigned.all;

entity TOP is
        generic(n: natural :=8);
port(
        CLK:        in std_logic;--clok signal
        PL:         in std_logic;--paralel load
```

```
        CLR:        in std_logic;--clear counter
        CS:         in std_logic;--chip select
        IN_8:       in std_logic_vector(n-1 downto 0);--parallel data input
        SOUT:       out std_logic --serial data output
);
end TOP;
-------------------------------------------------
```

Архитектура

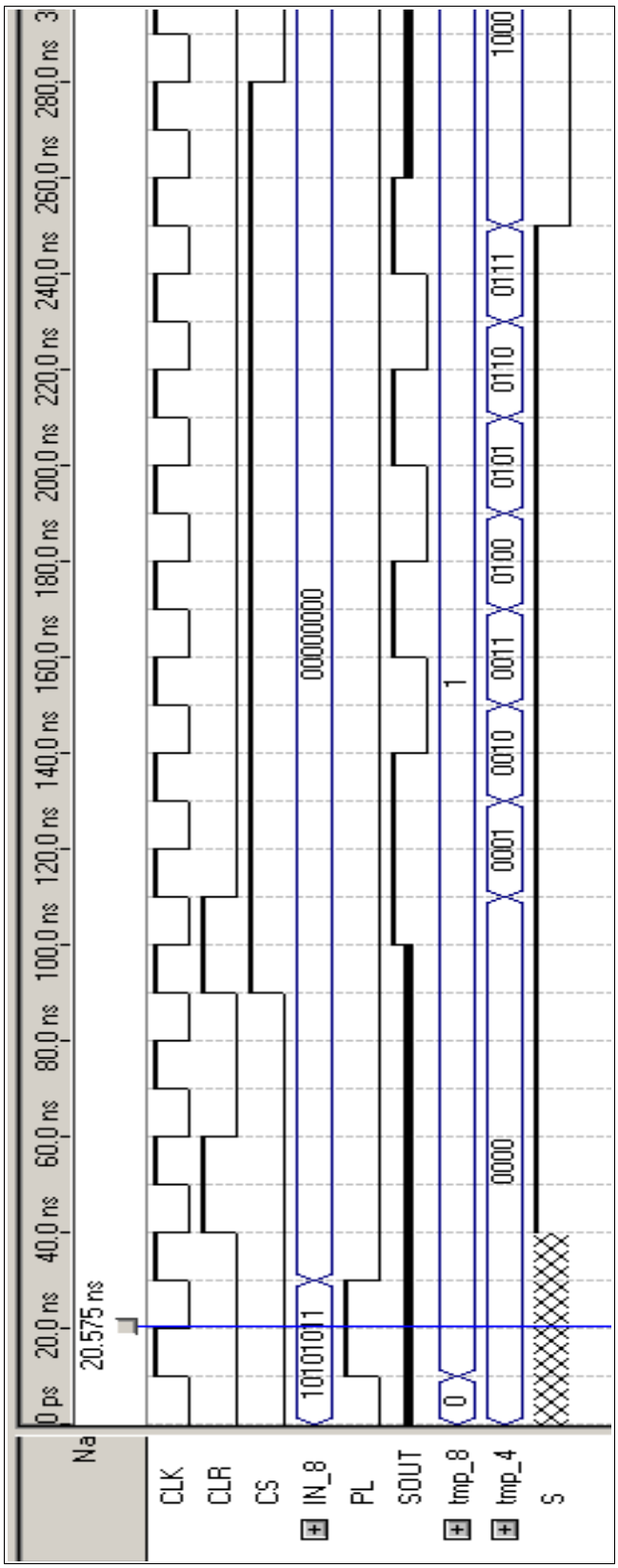
Архитектурата използва 5 отделни процеса: **COUNT:, ChipSel:, STOP:, ParalelLoad:** и **Shift.** Първият процес реализира броячен регистър, като поредно отчетената стойност на брояча се съхранява в сигнала **tmp_4**. Процесът **STOP** осигурява автоматично спиране броенето при отброяване на 9 поредно постъпили импулса. Процесът **Shift** анализира стойността на **tmp_4** и според нея извежда един пореден бит от сигнал **tmp_8**, който съдържа 8 бита записани през параления порт **IN_8**.

```
-------------------------------------------------
architecture Counter of TOP is
        signal tmp_4: std_logic_vector(3 downto 0);
        signal tmp_8: std_logic_vector(n-1 downto 0);
        signal S: std_logic;        signal CE: std_logic;
begin
        COUNT: process(CLK, CE, CLR)
        begin
                if CLR = '1' then
                        tmp_4 <= "0000";
                elsif (CLK='1' and CLK'event and CE='1') then
                        if (S = '1') then --if (CE = '1')then--
                                tmp_4 <= tmp_4 + 1;
                        end if;         end if;
        end process;
        ChipSel: process(CLK, CS)
        begin
                if (CLK='0' and CS='1') then
                        CE <= '1';
                elsif (CS='0')then
                        CE <= '0';   end if;
        end process;
        STOP: process(tmp_4, CLR)
        begin
                if(tmp_4 >= "1000") then
```

```
                S <= '0';      end if;
        if (CLR='1') then
                S <='1';       end if;
end process;

ParalelLoad: process (PL, IN_8)
begin
        if(rising_edge(PL))then
                tmp_8 <= IN_8;   end if;
end process;
Shift: process(CLK, CE)
begin
        if (CLK='0' and CLK'event and CE='1') then
                case tmp_4 is
                        when "0000" => SOUT <= tmp_8(0);
                        when "0001" => SOUT <= tmp_8(1);
                        when "0010" => SOUT <= tmp_8(2);
                        when "0011" => SOUT <= tmp_8(3);
                        when "0100" => SOUT <= tmp_8(4);
                        when "0101" => SOUT <= tmp_8(5);
                        when "0110" => SOUT <= tmp_8(6);
                        when "0111" => SOUT <= tmp_8(7);
                        when others => SOUT <= 'Z';
end case; end if; end process; end Counter;
-------------------------------------------------
```

Симулация

Създайте нов файл за симулация и въведете дадените по-долу сигнали (Фиг. 52).



Фиг. 52 Симулация на регистъра с паралелен вход и сериен изход

Задача 1: Задайте друга комбинация от битове подавана на паралелния вход **IN_8=11001010** и анализирайте резултата получаван на серийния изход **SOUT**.

Компаратор

При реализяцията на цифрови устройства често се налага да бъде извършвано сравнение на две стойности. Тази функционалност се реализира чрез използването на компаратори. Ако трябва да се сравнят две числа A и B, компараторът има три възможни изходни състояние: A=B, A>B, B>A. За да реализирате компаратор сравняващ целочислени положителни 8 битови стойности създайте нов проект с име **Comparator** и главно ентити с име **TOP**, след това създайте нов VHDL файл, въведете и компилирайте дадения по-долу код.

Ентити

```
-----------------------------------------------
entity TOP is
generic(n: natural :=8);
port(
        A:      in std_logic_vector(n-1 downto 0);
        B:      in std_logic_vector(n-1 downto 0);
        A_less_B:           out std_logic;
        A_greater_B:        out std_logic;
        A_equal_B:          out std_logic
);
end TOP;
-----------------------------------------------
```

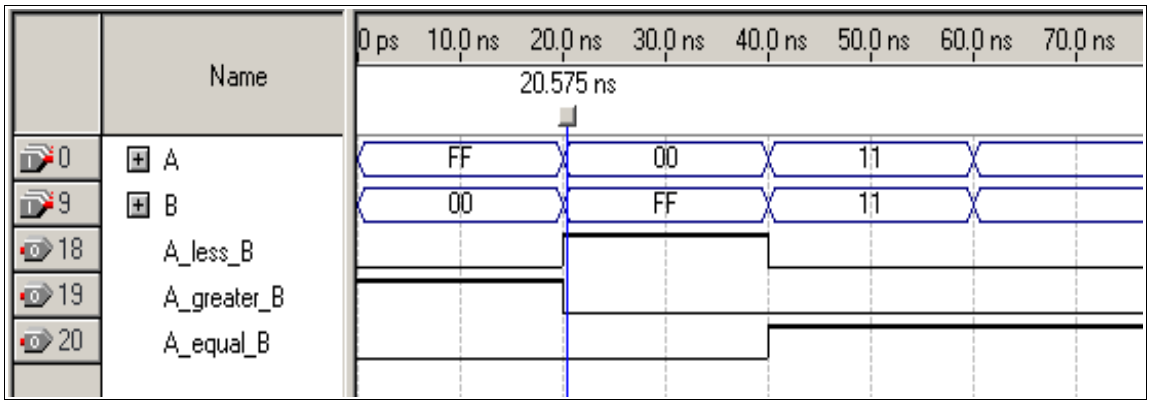
Архитектура

Архитектурата използва един процес **COMPARE** за реализация на пълната функционалност на схемата. Така създадения модел може да сравнява само челочислени положителни стойности.

```
-----------------------------------------------
architecture Comparator of TOP is
begin
   COMPARE: process(A,B)
   begin
      if (A<B) then
          A_less_B    <= '1';
          A_equal_B   <= '0';
          A_greater_B <= '0';
      elsif (A=B) then
          A_less_B    <= '0';
          A_equal_B   <= '1';
          A_greater_B <= '0';
      else
          A_less_B    <= '0';
```

```
            A_equal_B   <= '0';
            A_greater_B <= '1';
        end if;
    end process;
end Comparator;
-------------------------------------------------
```

Симулация



Фиг. 52 Симулация действието на компаратора

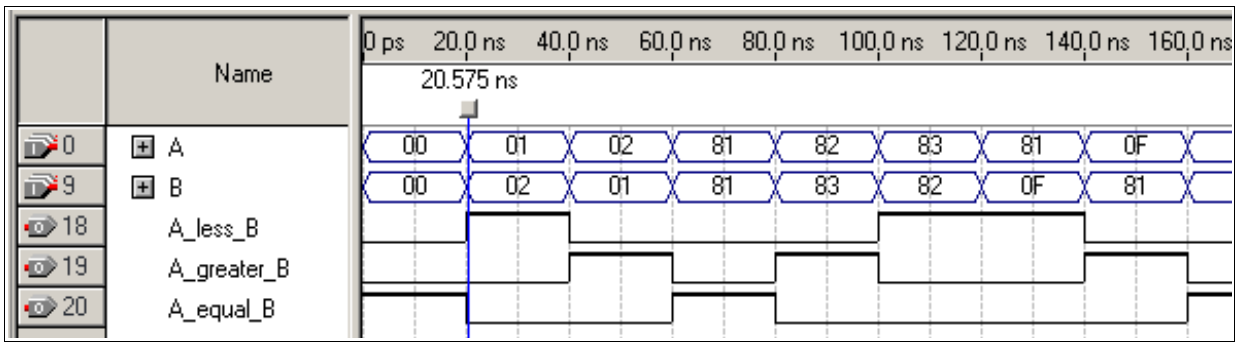
Задача 1: Предложете изменения в архитектурата, така че да може да се сравняват положителни и отрицателни стойности. При тази реализация имайте предвид, че знакът на числото се задава с установяване в '1' или '0' на най-старшия бит от числото A или B, когато битът е '1' то числото е отрицателно. За по-голямо удобство ползвайте отделни сигнали за представянето на знака на A и B, като инициализирате тяхната стойност със бит 7 на двете числа по следния начин, отделете този процес.

SignA <= A(7);
SignB <= B(7);

Знак на А	**Знак на Б**	**Необходима проверка**
0	**0**	Определете по-голямото по абсолютна стойност if(|A|>|B|) then A_greater_B <= 1
0	**1**	A>B
1	**0**	B>A
1	**1**	Определете по-малкото по абсолютна стойност if(|A|<|B|) then A_greater_B <= 1

```
architecture Comparator of TOP is
        signal SignA: std_logic; signal SignB: std_logic;
begin
   COMPARE_SIGN: process (SignA,SignB)
   begin
   if (SignA='0' and SignB='1') then
          A_less_B  <= '0';          A_equal_B <= '0';      A_greater_B <= '1';
       elsif (SignA='1' and SignB='0') then
          A_less_B  <= '1';          A_equal_B <= '0';      A_greater_B <= '0';
       elsif (SignA='0' and SignB='0') then
              if (A<B) then
                    A_less_B <= '1'; A_equal_B  <= '0'; A_greater_B <= '0';
              elsif (A=B) then
                    A_less_B <= '0'; A_equal_B  <= '1'; A_greater_B <= '0';
              else
                    A_less_B <= '0'; A_equal_B  <= '0'; A_greater_B <= '1';
              end if;
       elsif (SignA='1' and SignB='1') then
              if (A<B) then
                    A_less_B  <= '0'; A_equal_B  <= '0'; A_greater_B <= '1';
              elsif (A=B) then
                    A_less_B  <= '0'; A_equal_B  <= '1'; A_greater_B <= '0';
              else
                    A_less_B  <= '1'; A_equal_B  <= '0'; A_greater_B <= '0';
              end if;        end if;
   end process;
       SignA <= A(7); SignB <= B(7);
end Comparator;
```

Задача 2: Симулирайте работата на схемата, използвайте подобна последователност, обяснете кои стойности на A и B са положителни и кои отрицателни (Фиг. 53).



Фиг. 53 Симулация на компаратор сравняващ цели числа със знак

ЛИТЕРАТУРА

[1] Quartus II Handbook, Volume 2 Design Implementation & Optimization, http://www.altera.com

[2] Jari Nurmi, Tampere University of Technology, "Processor Design System-on-Chip Computing for ASICs and FPGAs"

[3] Р.И.Грушвицкии, А.Х.Мурасев, Е.П.Угрюмов, "Проектирование систем на микросхемах с програмируемой структорой – bhv", 2006

[4] Кр. Н. Филипова, В. Христов, М. Христов, И. Панайотов, „Използване на (v)HDL за синтез на електронен хардуер", 2004

[5] Dr Peter R. Wilson, "Design Recipes for FPGAs"

[6] http://www.altera.com/education/training/curriculum/cpld/trn-cpld.html "Introduction to VHDL (IHDL110)"

[7] http://www.altera.com/education/training/curriculum/fpga-asic/trn-fpga-asic.html?GSA_pos=3&WT.oss_r=1&WT.oss=FPGA "ASIC-to-FPGA Designer Curriculum"

Георги Костадинов Петров има магистърска степен по телекомуникации, получава степен Доктор на Нов български университет през 2008г. Преподавателската му дейност е от 2006г. до сега и е свързана основно с курсове в областта на мултимедийни и широколентови комуникации, телекомуникационен софтуер и конфигурируеми интегрални схеми. Работл е по редица национални и международни проекти в областта на образованието, телекомуникациите и медицински системи. Професионалните му интереси са в области: медицински системи за събиране и анализ на сигнали, системи за телеметрия и мултимедийни системи за обработка и сегментация на цифрово видео.

Дизайн на цифрови електронни устройства с VHDL и Quartus II

част II основи на VHDL в примери и задачи

1-во издание

Това учебно помагало е предназначено за студенти изучаващи следните курсове в програма Телекомуникации на НБУ: „Увод в програмируемите логически устройства“, „Съвременни конфигурируеми интегрални схеми“ и „Проектиране с програмируеми интегрални схеми“.

Обобщените примери и коментари, както и последователността на изложението го правят подходящо за всички начинаещи дизайнери на хардуер с VHDL. Книгата е подходяща и за програмисти на C/C++, които желаят да разширят своите познания в областта на хардуера и дизайнът на процесори и електронни блокове. Настоящото учебно пособие е подходящо и изчерпателно начално четиво в областта на разработката на проекти с програмиреуми логически устройства и използването на интегрирани среди за разработка на хардуер.