Универзитет у Београду
Електротехнички факултет

Хардверска дигитална мрежа за сортирање

Мастер рад

Ментор:
доц. др Живојин Шуштран

Студент:
Алекса Величковић 2020/3358

Београд, септембар 2023.

Садржај

1 Увод

У овом раду ће бити описана рекурзивна имплементација хардверске дигиталне мреже за сортирање битоник сортер (енг. *bitonic sorter*). Имплементација је урађена коришћењем језика за опис хардвера Чизел (енг. *Chisel*). Чизел је базиран на програмском језику Скала (енг. *Scala*) и омогућава једноставније дефинисање хардвера од других језика за моделовање хардвера (енг. *hardware description language*) и прегледнији програмски код.

Истраживачки рад обухвата разматрање предности и мана програмског језика Скала, функционалног програмирања и програмског језика Чизел. Представљена су постојећа решења за сортирање у хардверу и објашњена је софтверска имплементација битоник сортера у програмском језику Пајтон (енг. *Python*).

Циљ овог мастер рада је имплементација битоник сортера у програмског језику Чизел, која се базира на постојећој софтверској имплементацији у програмском језику Пајтон. Због ограничења које постоје у хардверу, имплементација у хардверу захтева одређене измене у односу на софтверску имплементацију.

Добијена имплементација је поређена са постојећим решењима. Поређење се вршило на основу простора које различите имплементације заузимају на чипу, као и на основу брзине самог сортирања. Тестирана је и скалабилност решења провером како се те особине решења мењају у зависности од величине улазних података.

На крају функционалност добијене имплементације је тестирана и верификована помоћу симулатора. Уз помоћ алата за синтезу, код је преведен и проверен да ли задовољава све услове да се добијено решење користи на чипу.

У другом поглављу овог рада су представљени коришћени алати: програмски језик Скала, функције вишег реда, програмски језик Чизел, као и алат за синтезу Quartus Prime Lite.

Мреже за сортирање су дефинисане и представљене у трећем поглављу рада. Детаљно је описан битоник сортер, који се показао као најпогоднији за имплементацију у хардверу.

У четвртом поглављу рада представљена су постојећа решења у програмским језицима Чизел, Пајтон и Верилог (енг. *Verilog*). Представљена су итеративна и рекурзивна решења, као и њихове карактеристике.

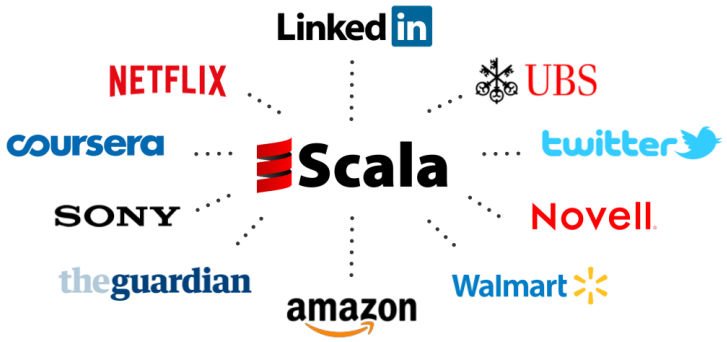
Опис архитектуре и имплементације решења дат је у петом поглављу рада. Извршено је поређење са постојећим решењима и детаљно је објашњен процес тестирања.

Последње поглавље преставља закључак у коме је наведено шта је урађено, као и који су предлози за даљи рад и надоградњу добијеног решења.

2 Коришћени алати

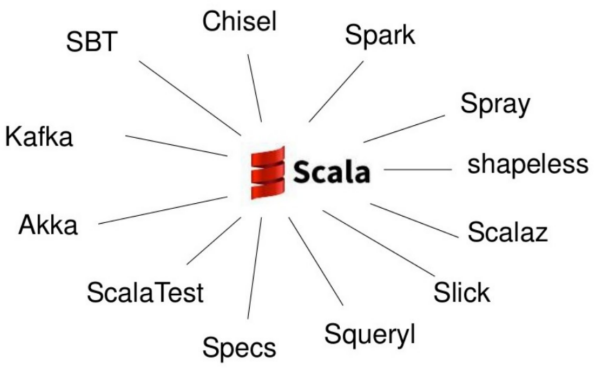
2.1 Скала

Скала је програмски језик који подржава и објектно-оријентисано и функционално програмирање. Дизајниран је да буде концизан и да надомести недостатке које има програмски језик Јава (енг. *Java*) [1]. Име Скала потиче од „скалабилан језик". Језик је тако назван зато што је дизајниран да расте са потребама корисника. Скала ради на стандардној Јава виртуелној машини и може користити све Јава библиотеке. Примена конструкта функционалног програмирања омогућава брзо стварање програма коришћењем једноставних делова, док конструкти објектно-оријентисаног програмирања омогућавају изградњу већих система и прилагођавање новим потребама [2]. Скала је популаран програмски језик и користе га популарне компаније као Goldman Sachs, Twitter, IBM, Verizon, Spotify, Netflix итд. Слика 2.1.1 приказује неке најпознатије компаније које користе Скалу.



Слика 2.1.1: Компаније које користе Скалу

Постоје многи софтверски алати који су написани у Скали или су надограђени на овом програмском језику као нпр. Apache Spark који се користи за обраду и анализу великог обима података у дистрибуираном окружењу. Такође, на слици 2.1.2 се може приметити и Чизел (Chisel), језик за пројектовање хардверских уређаја, који ће касније бити детаљније описан и који је коришћен у овом раду.



Слика 2.1.2: Окружења која користе Скалу

2.1.1 Функције вишег реда

Функције вишег реда (ФВР) су општи концепт који се јавља како у функционалном програмирању тако и математици. Функционални језици третирају функције као „грађане првог реда", а то значи да:

- функција може бити прослеђена као параметар другој функцији
- функција може бити резултат друге функције

Пример употребе ФВР за сумирање [3]:

```
def sum(f: Int => Int, a: Int, b: Int): Int =
  if a > b then 0
  else f(a) + sum(f, a + 1, b)

def id(x: Int): Int = x
def cube(x: Int): Int = x * x * x
def fact(x: Int): Int = if x == 0 then 1 else x * fact(x - 1)

def sumInts(a: Int, b: Int) = sum(id, a, b)
def sumCubes(a: Int, b: Int) = sum(cube, a, b)
def sumFactorials(a: Int, b: Int) = sum(fact, a, b)
```

Овде се може приметити да функција `sum()` користи функцију `f()` као параметар, такође има и доњу границу `a` и горњу границу `b`. Посебно су дефинисане функције `sumInts()` (сума бројева), `sumCubes()` (сума кубних вредности) и `sumCubes()` (сума факторијела бројева) на опсегу од `a` до `b`. Може се приметити да су једине разлике њихових дефиниција функције која се прослеђује као аргументи `id()`, `cube()` и `fact()`. Такође, функције `sumInts()` (сума бројева) и `sumCubes()` је могуће дефинисати и употребом **Анонимних функција**:

```
def sumInts(a: Int, b: Int) = sum(x => x, a, b)
def sumCubes(a: Int, b: Int) = sum(x => x * x * x, a, b)
```

Овако се може избећи писање великог броја кратких функција. Функцију `sum()` је могуће дефинисати и тако да није неопходно прослеђивати `a` и `b` при дефинисању функција за различита сумирања.

```
def sum(f: Int => Int): (Int, Int) => Int =
  def sumF(a: Int, b: Int): Int =
    if a > b then 0
    else f(a) + sumF(a + 1, b)
  sumF
// или скраћено
def sum(f: Int => Int)(a: Int, b: Int): Int =
  if a > b then 0 else f(a) + sum(f)(a + 1, b)
// где би сумирање кубова било овако дефинисано
def sumCubes = sum(x => x * x * x)
```

```
// или чак ни то није неопходно..
sum (cube) (1, 10)
```

Овакав стил дефинисања и примене функција, где се функција са више параметара може представити у виду секвенце функција једног параметра, назива се **керификација** (currying).

2.2 Чизел

Чизел (енг. *Chisel - Constructing Hardware in a Scala Embedded Language*) је језик за опис хардвера заснован на Скали. Чизел наслеђује аспекте објектно-оријентисаног и функционалног програмирања Скале за опис дигиталног хардвера. Коришћење Скале као основе омогућава опис генератора кола. Чизел код којим су описана кола може бити претворен у Верилог за синтезу и симулацију [4].

Омогућава корисницима да конструишу хардвер управљајући и повезујући Скала објекте да би се направио граф жељеног хардвера. Чизел бекенд потом анализира резултујући граф да би се добио брз, циклусно-прецизан C++ симулатор, који одмах потом може да комуницира са Скала тестним окружењем или Верилог синтезом која ће бити процесирана на FPGA (*Field Programmable Arrays*) интегрисаном колу. Чизел дозвољава генератору хардвера да изврши различита израчунавања у Скали над улазним параметрима током израде графа хардвера. Додатно, програмери могу да искористе одлике функционалног програмирања у Скали за опис апстракција хардвера (нпр. имплементација ФИР филтера коришћењем `map` и `reduce` функција) и Јава стандардних библиотека (нпр. претварање бројева са покретним зарезом у или из репрезентације битова која би се користила у тестном окружењу) као начине за поједностављивање свеукупног развоја хардвера и верификације.

Отклањање грешака током дизајна хардвера у Чизелу може бити изазовно. Дизајнери који користе Чизел често пишу сажет програмски код науштрб коректности. Тако, корисник може да погреши у куцању при дефинисању хардвера, а да не добије никакве грешке. Такође, доста грешака које Чизел враћа су веома нејасне, где само пише "assertion failed". На пример, декларисање литерала (непроменљиве вредности) уместо порта унутар улазно-излазних декларација је грешка коју би Чизел ухватио. Међутим, порука о грешци која се добија је донекле бескорисна [5].

2.2.1 Генератори кола и условни изрази

Сви језици за опис хардвера подржавају писање појединачних инстанци. Међутим, писање инстанци је заморно. Верилог омогућава ограничену параметризацију, као број битова и `generate` израз, али само толико. Пошто није могуће написати генератор у Верилогу, то би значило да треба да напишемо нову инстанцу, што би дуплирало величину програмског кода. Генератори би требало (у идеалном случају) да их је могуће направити, да су моћни и да омогућавају прецизну контролу над генерисаним дизајном. Провера грешака је неопходна да бисмо били сигурни ли је композиција дозвољена; без тога, отклањање грешака је тешко. Ово тражи да језик генератора разуме семантику дизајна тј. да зна шта јесте, а шта није дозвољено. Такође, генератор не би требало да буде преопширан!

Желимо да програм генератора сажето изрази више различитих дизајна, без потребе да се посебно повезује у `if` условима за сваку инстанцу [6].

У наставку је приказан једноставан пример `Passthrough`, који има 4-битни улаз `in` и 4-битни излаз `out`. Модул повезује `in` и `out`, па `in` управља `out`-ом.

```
class Passthrough extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(4.W))
    val out = Output(UInt(4.W))
  })
  io.out := io.in
}
```

`Modul` је уграђена Chisel класа коју сви хардвер модули морају да наследе. Сви улазни и излазни портови се декларишу у посебној `io val`. Мора да се зове `io`, и мора да буде `IO` објекат или инстанца у форми `IO(_instantiated_bundle_)`.

```
new Bundle {
    val in = Input(...)
    val out = Output(...)
}
```

Декларише се нова хардверска структура типа (`Bundle`) која садржи сигнале `in` и `out` са одговарајућим смеровима `Input` и `Output`, редом. `UInt(4.W)` представља тип сигнала, у овом случају неозначени (unsigned) цео број ширине 4. Улазни порт се повезује са излазним изразом `io.out := io.in`, и тако `io.in` управља `io.out`.

Након декларисања Чизел модула, користимо Скалу да позовемо Чизел компајлер да преведе Чизел `Passthrough` у Верилог `Passthrough`. Овај процес се назива обрађивање (elaboration).

```
println(getVerilog(new Passthrough))
```

Као било која Скала класа и Чизел модул би могао да прими неке параметре. У наставку је приказан пример генератор модула, где се дефинише нова класа `PassthroughGenerator` која прима целобројну вредност `width` која одређује ширине улазног и излазног порта:

```
class PassthroughGenerator(width: Int) extends Module {
  val io = IO(new Bundle {
    val in = Input(UInt(width.W))
    val out = Output(UInt(width.W))
  })
  io.out := io.in
}

println(getVerilog(new PassthroughGenerator(10)))
println(getVerilog(new PassthroughGenerator(20)))
```

Може се приметити да Верилог користи различит број битова за улаз/излаз зависно од параметра `width`.

```
//Elaborating design...
//Done elaborating.
module PassthroughGenerator(
  input        clock,
  input        reset,
  input  [9:0] io_in,
  output [9:0] io_out
);
  assign io_out = io_in; // @[cmd4.sc 6:10]
endmodule

//Elaborating design...
//Done elaborating.
module PassthroughGenerator(
  input         clock,
  input         reset,
  input  [19:0] io_in,
  output [19:0] io_out
);
  assign io_out = io_in; // @[cmd4.sc 6:10]
endmodule
```

Пошто су Чизел модули нормалне Скала класе, можемо да искористимо предност конструктора Скала класа да параметризујемо обрађивање дизајна. Још једном, ову параметризацију омогућава Скала, не Чизел. Пошто `PassthroughGenerator` не описује само један модул, већ описује породицу модула са параметром `width`, овај модул представља један **генератор**.

Основни условни изрази у Чизелу су `when`, `elsewhen` и `otherwise`. За разлику од `if` услова у Скали, овде се не враћају вредности на крају делова кода везаних за `when`. На пример, израз `val result = when(squareIt) { x * x }.otherwise { x }` неће радити. Начин да се заобиђе ово ограничење је `Wire` конструкт. `Wire` се може налазити на десној или на левој страни `:=` оператора. Следећи пример зависно од вредности улазног селектора рачуна један од три полинома $x^2 - 2x + 1$, $2x^2 + 6x + 3$ или $4x^2 - 10x - 5$:

```
class Polynomial extends Module {
  val io = IO(new Bundle {
    val select = Input(UInt(2.W))
    val x = Input(SInt(32.W))
    val fOfX = Output(SInt(32.W))
  })
  val result = Wire(SInt(32.W))
  val square = Wire(SInt(32.W))
  square := io.x * io.x
  when(io.select === 0.U) {
```

```
    result := (square - (2.S * io.x)) + 1.S
  }.elsewhen(io.select === 1.U) {
    result := (2.S * square) + (6.S * io.x) + 3.S
  }.otherwise {
    result := (4.S * square) - (10.S * io.x) - 5.S
  }

  io.fOfX := result
}
```

2.2.2 Регистри

Основни елемент у Чизелу за чување стања је регистар означен са `Reg`. Он чува своју излазну вредност до следеће растуће ивице тактног циклуса, након које узима вредност са улаза. Чизел има имплицитно дефинисан такт који користи сваки регистар у дизајну, па није потребно дефинисати исти тактни сигнал изнова [6].

Следећи програмски код приказује имплементацију модула који узима улаз, повећава му вредност за 1, а потом додељује ову вредност улазу регистра. (Имплицитни такт може бити „прегажен" за дизајн са више тактних сигнала.)

```
class RegisterModule extends Module {
  val io = IO(new Bundle {
    val in  = Input(UInt(12.W))
    val out = Output(UInt(12.W))
  })
  val register = Reg(UInt(12.W))
  register := io.in + 1.U
  io.out := register
}
test(new RegisterModule) { c =>
  for (i <- 0 until 100) {
    c.io.in.poke(i.U)
    c.clock.step(1)
    c.io.out.expect((i + 1).U) } }
println("SUCCESS!!")
```

Регистар се креира позивом функције `Reg(tpe)`, где је `tpe` променљива која кодира тип регистра који желимо. У овом примеру, `tpe` је `UInt` дужине 12 бита. Између позива функције `poke()` и `expect()`, се налази позив функције `step(1)`. Ово обавештава тестни оквир да „окине" један тактни сигнал, што ће довести до преноса улазних података из регистра на његов излаз. Позив функције step(n)+ ће откуцати тактни сигнал `n` пута. Тестери који тестирају комбинаторну логику не позивају `step()`. Ово је зато што позив функције `poke()` на улазу одмах прослеђује ажуриране вредности кроз комбинаторну логику. Позив функције `step()` је потребан само за ажурирање елемента стања у секвенцијалној логици.

Блок кода испод ће приказати део Верилог код који је генерисан помоћу `RegisterModule`. Променљива register се приказује као `reg [11:0]`, као што се и

очекивало. Постоји блок секције `ifdef Randomize` који иницијализује регистар на неку случајну вредност пре почетка симулације. Регистар се ажурира на позитивну ивицу такта. Позивом `getVerilog(new RegisterModule)` добија се 60 линија Верилог програмског кода.

```
module RegisterModule(
  input         clock,
  input         reset,
  input  [11:0] io_in,
  output [11:0] io_out
);
`ifdef RANDOMIZE_REG_INIT
  reg [31:0] _RAND_0;
`endif // RANDOMIZE_REG_INIT
  reg [11:0] register; // @[cmd2.sc 7:21]
  assign io_out = register; // @[cmd2.sc 9:10]
  always @(posedge clock) begin
    register <= io_in + 12'h1; // @[cmd2.sc 8:21]
  end
  ...
```

Претходно дефинисани модул се може скратити коришћењем `RegNext`, па би излаз модула био дефинисан као `io.out := RegNext(io.in + 1.U)`. Верилог код оваквог модула би изгледао готово исто као у претходном примеру.

Регистар у модулу `RegisterModule` је иницијализован случајним подацима за потребе симулације. Осим ако није другачије наведено, регистри немају вредност ресетовања (или ресет). Начин за креирање регистра који се ресетује на одређену вредност је коришћењем функције `RegInit`. Па би излазна вредност модула била дефинисана као:

```
val register = RegInit(0.U(12.W))
register := io.in + 1.U
io.out := register
```

Верилог код се добије из овако дефинисаног модула имао био блок који врши проверу `if(reset)` где се регистар ресетује на 0. Овај блок се налази унутар блока `always @(posedge clock)`. Регистар се опет иницијализује на насумичну вредност, пре него што се изврши ресет.

Регистри су веома слични жицама (`wire`) у Чизелу по питању контроле тока. Могу им се додељивати вредности употребом `when`, `elsewhen` и `otherwise`.

```
val max = RegInit(0.U(10.W))
  when (io.in > max) {
    max := io.in
  }
```

Операције над регистрима се изводе над његовим излазом, и врсте операција зависе од тога ког је типа регистар. Ако је је регистар типа `UInt` и дефинисан као

`val reg: UInt = Reg(UInt(4.W))`, то значи да су могуће све операције које дозвољава тип `UInt`, као `+`, `-` ...

2.2.3 Типови и оператори

Потребно је правити разлику између Скала и Чизел типова. На следећем примеру се може видети нпр. разлика између `UInt` из Чизела и `Int` из Скале. У реду је доделити `a` вредност `0.U`, а ако би се доделила вредност `0` дошло би до грешке.

```
val a = Wire(UInt(4.W))
a := 0.U  // legal
a := 0    // illegal
```

Чизел тип `Bool` и Скала тип `Boolean` се разликују. Код условних израза ово долази до изражаја, па ја у случају услова `when` могуће користити само `Bool`, а у `if` је могуће поставити као услов `Boolean`; мешање доводи до грешке.

Такође, није дозвољено комбиновање типова нпр. `1.U + 1`. Операције одузимања или множења се изводе на неозначеним бројевима нпр. `io.out_add := 1.U * 4.U`.

Чизел има уграђене операторе `Mux` (мултиплексер) и `Cat` (надовезивање), где би у наредном примеру `Mux` вратио `3.U` уколико је `s` тачно, а `Cat` би извршио надовезивање бинарних репрезентација бројева и излаз је 101.

```
io.out_mux := Mux(s, 3.U, 0.U)
io.out_cat := Cat(2.U, 1.U)
```

2.3 Јупитер свеска и Докер

Током савладавања Чизела кроз приручник (Chisel Bootcamp), наведеном у литератури [6], било је потребно користи за његово покретање платформу Докер (Docker), а сам приручник је приказан у Јупитер свесци (Jupyter notebook). Јупитер свеска омогућава извршавање делова кода и није потребно извршавати целу скрипту сваки пут. Развој имплементације изложене у овом раду је започет у оквиру овог туторијала, али у случају грешке поруке нису биле довољно јасне (или програм није терминирао), па је било неопходно извршавати команде локално кроз `sbt`.

2.4 Intellij и SBT

Развој рекурзивне имплементације је рађен у IntelliJ развојном окружењу, а за компајлирање и тестирање програма је коришћен алат SBT - Scala (Simple) Build Tool, где се команде покрећу у терминалу директоријума пројекта (нпр. `sbt`).

2.5 Quartus Prime Lite

Синтеза кола имплементираних у Верилогу је рађена у софтверу Quartus Prime Lite компаније Intel, који се користи за дизајнирање програмабилних логичких кола. Поред саме синтезе, коришћене су функционалности софтвера као RTL View који омогућава приказ слике добијеног кола и симулација уз помоћ EDA алата Questa. Коришћене су бесплатне верзије ових алата.

3 Мреже за сортирање

Сортирање је одувек био фундаментални проблем у рачунарству и детаљно је изучаван. Постоје разни методи сортирања за софтверске и хардверске имплементације. У случају хардверских, постоји компромис између пропусне способности и цене (тј. логике и меморије за сортирање n елемената). Два популарна решења су битоник мрежа за сортирање и линеарни сортер. За листу дужине n, битоник сортер захтева $\mathcal{O}(n \log^2 n)$ логике и меморије и може да сортира n елемената по циклусу; линеарни сортер захтева $\mathcal{O}(n)$ логике и меморије и сортира n елемената за n циклуса [7].

Сада ће бити дефинисани неопходни кључни термини за описивање својстава различитих решења.

Дефиниција 3.1 (Пропуштање)**.** Замислимо хардверски дизајн који мора да прими n речи података (и да врати n речи података на излазу). Ако прими (и врати) w речи по тактном циклусу током n/w узастопних циклуса, ово се назива *пропуштање* (*streaming*), где је w пропусна ширина.

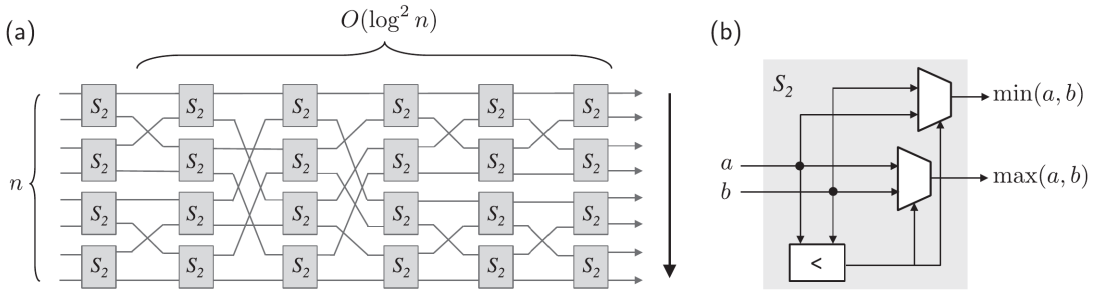
Имплементирањем дизајна са пропусном ширином w мањом од n (број речи који се сортира), цена и улазно-излазна пропусна способност система се смањују.

Дефиниција 3.2 (Потпуно пропуштање)**.** Дизајн који константно може да прима улазни ток података без икаквог времена чекања између два улазна скупа се назива *потпуно пропустан* (*fully streaming*) дизајн.

Потпуно пропусна мрежа за сортирање пропусне ширине w ће имати пропусну способност w речи по тактном циклусу. Обично се може очекивати да потпуно пропустан дизајн захтева више ресурса од дизајна која није потпуно пропустан, зато што за исто време обрађује више података.

Мреже за сортирање паралелно обрађују листу n улазних елемената кроз више нивоа тако да је на последњем нивоу улазна листа сортирана. Током сваког нивоа паралелно се изводе поређења и пермутације. Број нивоа који је потребан да се добије сортирана листа и број поређења по нивоу одређују цену мреже. Број улазних вредности је степен броја два: $n = 2^t$.

Слика 3.1 показује *битоник* (енг. *bitonic*) мрежу за сортирање [8]. Има осам улазних елемената. Оператор S_2 сортира два улаза у растућем редоследу. Мреже за сортирање могу да постигну веома велику пропусност, зато што су потпуно пропусне, где је ширина протока n. Обично су повезане тако да се одржи разумна брзина такта и тада је потребно $\mathcal{O}(n \log^2 n)$ флип-флопова за меморију.



Слика 3.1: (a) Битоник мрежа за сортирање за улазну величину скупа $n = 8$ (b) Двоулазни сортер S_2 је градивна јединица у мрежама за сортирање

Као што се може приметити на слици 3.1, битоник мрежа за сортирање има константну структуру и може се конструисати рекурзивно. Сортира се n елемената у $\log_2(n)(\log_2(n)+1)/2$ нивоа са $n/2$ паралелних S_2 операција, па је потребно $\mathcal{O}(n\log(n)^2)$ операција поређења. Кенет Едвард Бечер (Kenneth Edward Batcher) је објавио овај алгоритам 1968. године, а такође је развио и *пар-непар* (*odd-even*) мреже за сортирање, које имају мање константну структуру, али захтевају мало мање операција поређења. Пар-непар и битоник мреже за сортирање су остале најпопуларнија решења због своје једноставности и скоро оптималне цене и перформанси.

Линеарни сортер је линеарни низ n чворова који сортирају n вредности, где n није ограничено да мора да буде степен броја два. Сваки чвор се састоји од флип-флопа и оператора поређења. Асимптотско време и логичка цена линеарних сортера је $\mathcal{O}(n)$ и оне су потпуно пропусне са пропусном ширином 1.

3.1 Битоник сортер

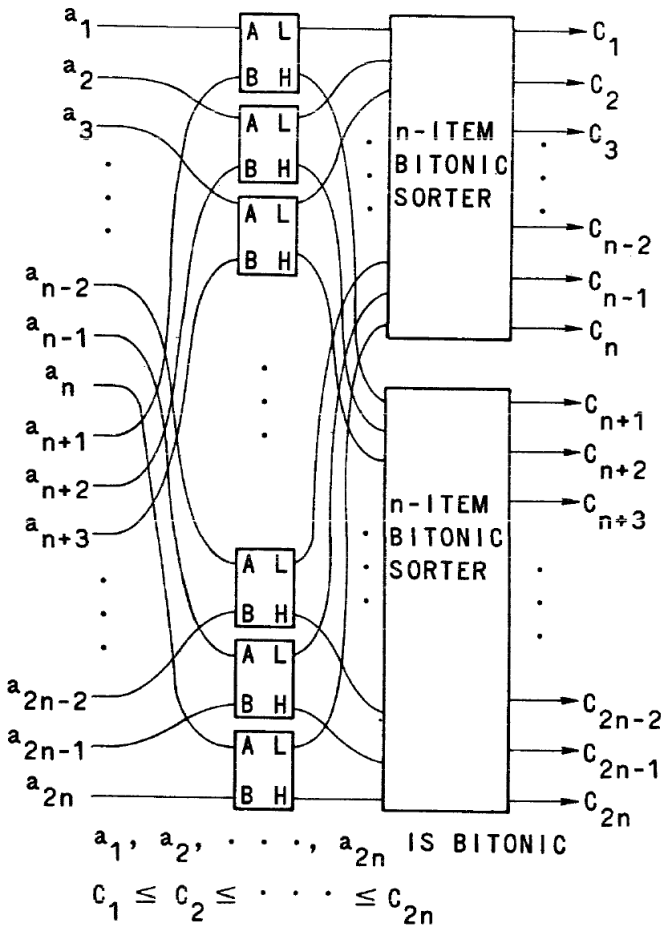
Низ бројева може да се назове битоник ако се састоји од два монотона низа, једног растућег и другог опадајућег. Такође, низ остаје битоник ако је подељен било где и два дела замењена.

Ако је низ $2n$ бројева $a_1, a_2, \ldots, a_{2n}$ битоник и ако направимо два низа од n бројева

$$min(a_1, a_{n+1}), min(a_2, a_{n+2}), \ldots, min(a_n, a_{2n}) \tag{1}$$

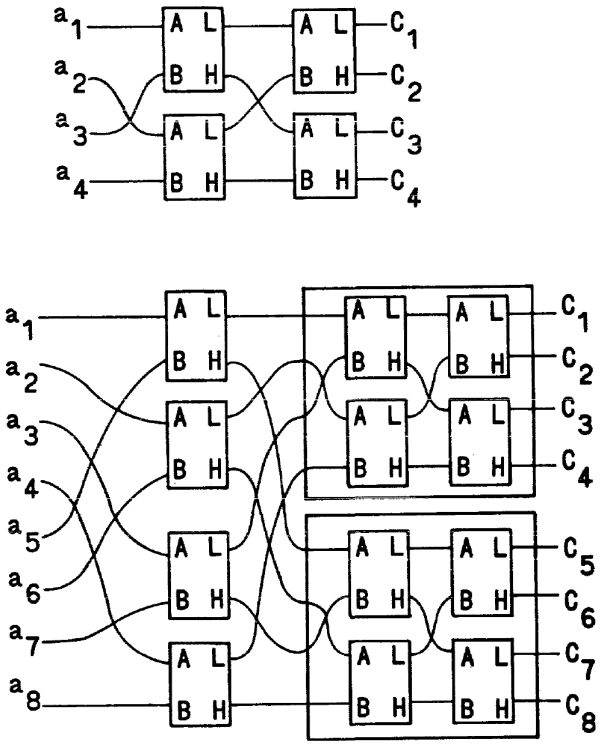
$$max(a_1, a_{n+1}), max(a_2, a_{n+2}), \ldots, max(a_n, a_{2n}) \tag{2}$$

онда су оба ова низа битоник и ниједан број из (1) није већи ни од једног броја из низа (2). Одавде се долази до итеративног правила приказаног на слици 3.1.2. Битоник сортер за $2n$ бројева може бити конструисан од n елемената за поређење и два битоник сортера за n бројева. Елементи за операцију поређења формирају два низа (1) и (2) и пошто су оба битоник сортирају се са два битоник сортера за n бројева. Пошто ниједан број из (1) није већи ни од једног броја из низа (2) излаз из једног битоник сортера је доња половина сортираног низа, а излаз другог сортера је горња половина [8].



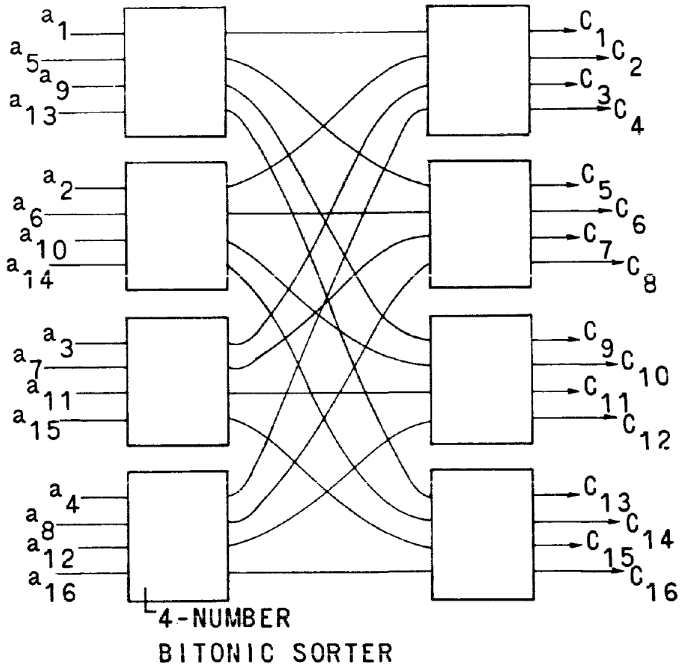
Слика 3.1.2: Итеративно правило за битоник сортере

Битоник сортер за два броја је само један елемент за поређење и коришћењем итеративног правила битоник сортери за 2^p бројева могу бити конструисани за било које p. Слика 3.1.3 приказује приказује сортере за четири и осам бројева. Битоник сортер за 2^p бројева захтева p нивоа по 2^{p-1} елемената, укупно $p2^{p-1}$. Може да се искористи као мрежа за спајање за било које две листе укупне дужине 2^p.



Слика 3.1.3: Конструкција битоник сортера за четири и осам бројева

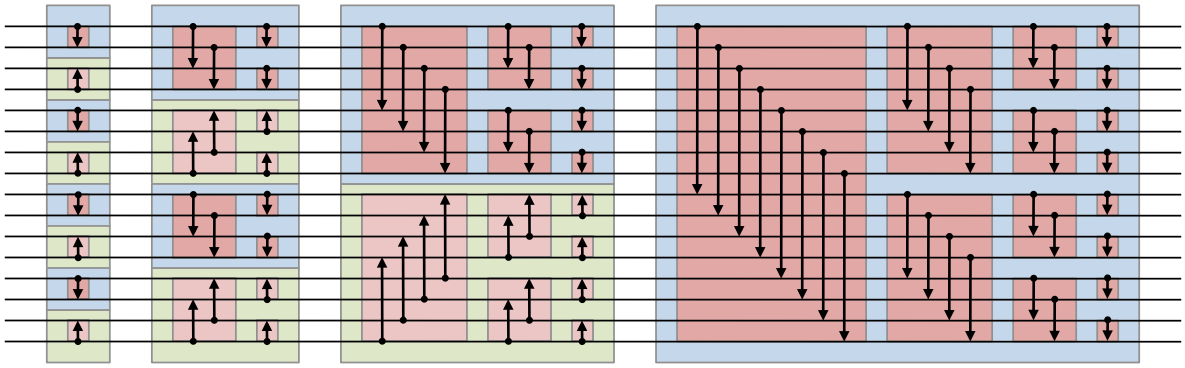
Велики битоник сортери могу бити конструисани из више мањих битоник сортера. На слици 3.1.4 је приказан битоник сортер за 16 бројева конструисаног од осам битоник сортера за 4 броја.



Слика 3.1.4: Битоник сортер за 16 бројева конструисаног од осам битоник сортера за 4 броја

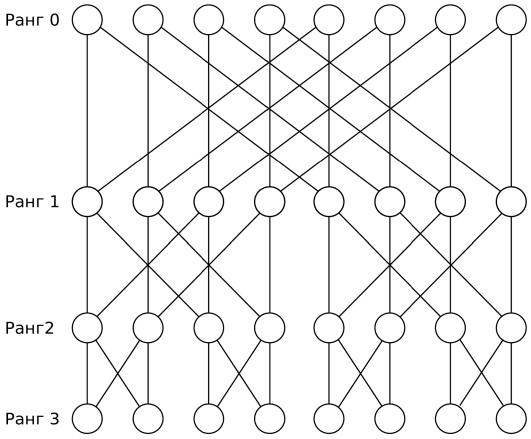
Следећа слика боље приказује како ради алгоритам. 16 бројева се налази на улазу на левој страни и прелазе преко 16 хоризонталних жица до излаза на десној страни. Мрежа сортира елементе, тако да се највећи налази на дну.

Стрелице представљају операције поређења и показују ка већем броју. Уоквирени правоугаоници у боји немају утицај на рад алгоритма, само олакшавају разумевање.



Слика 3.1.5: Приказ рада битоник алгоритма са 16 бројева на улазу

Ако се осматрају црвени правоугаоници, може се приметити да се сваки улаз из горње половине пореди са одговарајућим улазом који се налази у доњој половини правоугаоника, где су стрелице усмерене надоле ако су у питању тамноцрвени правоугаоници или нагоре код светлоцрвених. Ако бројеви на улазу формирају битоник низ, онда ће се на излазу наћи два битоник низа. Горња и доња половина излаза ће бити битоник, где ће сваки број горње половине бити мањи (или једнак) од сваког из доње (у тамноцрвеним) или обрнуто (у светлоцрвеним) [9].



Слика 3.1.6: Топологија лептира

Црвени правоугаоници формирају плаве и зелене. Црвени примају цео улазни низ, потом половину резултата, потом половину тих резултата итд. Све стрелице показују надоле (плави) ил нагоре (зелени). Ова структура је позната као лептир (buterfly) мрежа. Топологија лептира је приказана на слици 3.1.6. Ако се на улазу у овакав правоугаоник нађе битнок низ, онда ће излаз бити сортиран у растућем поретку у слчају плавих правоугаоника или опадајућем код зелених. Када се број налази на улазу у зелени или плави правоугаоник, онда ће први црвени да га сортира у праву половину листе. Потом пролази кроз мањи црвени правоугаоник који га

сортира у праву четвртину листе унутар те половине итд. Излаз из црвеног или плавог правоугаоника је потпуно сортиран.

Коришћењем плавих и зелених правоугаоника се може направити цела мрежа. Излаз из сваког зеленог и плавог правоугаоника је сортиран низ, па ће излаз сваког пара суседних низова бити битоник, зато што је горњи плави, а доњи зелени. Свака колона плавих и зелених правоугаоника узима N сортираних низова и спаја их у паровове тако да се добије $N/2$ битоник низова, који се потом сортирају унутар правоугаоника те колоне да се добије $N/2$ низова. Овај процес почиње од сортиране листе која садржи један елемент и наставља се кроз све колоне док се у последњој колони не споје у једну сортирану листу. Пошто је последњи ниво плави, коначна листа ће имати највећи елемент на дну. [9].

3.2 Други сортери

Стоун [10] је развио битоник мреже за сортирање са константном геометријом (1971). Ово значи да су две суседне фазе поређења од n/2 паралелних поређења увек повезане истом пермутацијом - такозваним савршеним мешањем. Тако се мрежа може пресавити (fold) тако да се користи само једна фаза поређења и мешања, кроз коју се подаци крећу $\mathcal{O}(\log_2 n)$ пута. Смер поређења се мења током фаза, што захтева мали додатни простор за контролну логику.

Слични напори да се пресавије или регулише битоник мреже за сортирање предложени су касније. Дауд и други [12] предлажу периодичну балансирану мрежу за сортирајућу која постиже различиту грануларност поновне употребе са блоком од $\log n$ фаза који се може користити $\log n$ пута. С друге стране, Лајер и други [14] предлажу битоник мрежу за сортирање коју чини $\log_2 n$ фаза које више пута круже са подацима унутар сваке фазе (1989).

Ограничење већине паралелних архитектура за сортирање са фиксним рачунарским током, као што су мреже за сортирање, је да морају увек сортирати улазне сетове исте величине. Решење за ово је адекватно додавање улазног сета тако да се елементи који се сортирају налазе на врху излазне листе. Лајер и Пфлајдерер [13] предложили су архитектуру која може подржавати различите величине улаза, до фиксног максимума (2004). Прилагодљива логика се користи за промену неопходних мешања у зависности од ширине улаза. Други приступ, представљен у раду Занг и Женг [15], користи серијску мрежу за сортирање са фиксном величином улазног сета `m` и скупом од `m` редова, сваки од којих може садржавати `n/m` елемената (2000). Контролна логика је потребна да адекватно подржи мрежу за сортирање. Предложене су и мрежу за сортирање са бољом асимптотском ценом. Посебно, Ајтаи и други [16] представљају AKC сортирајућу мрежу, која има оптималну асимптотску цену од $\mathcal{O}(n \log n)$ (1983). На жалост, ови резултати имају чисто теоријску вредност, јер велика скривена константа чини ове мреже неупотрбљивим у пракси.

Друга мрежа за сортирање која је популарна због своје високе регуларности и једноставности је пар-непар мрежа за сортирање са премештањем (transposition) [17], која међутим захтева $\mathcal{O}(n)$ фаза и $\mathcal{O}(n^2)$ операција поређења (1968). Као и у случају Стоун-а [10], она се може пресавити тако да се користи само једна фаза $\mathcal{O}(n)$ пута.

На крају, пар-непар мрежа за сортирање и битоник мреже за сортирање остају најпопуларнија решења због своје једноставности и близине оптималној цени и перформансама.

4 Постојећа решења

Сада ће бити приказане две имплементације које су биле основа за израду оне која је приказана у овом раду.

4.1 Итеративна имплементација битоник сортера у Чизелу

Као основа за израду рекурзивног решења изложеног у овом раду је послужила је итеративна имплементација наведена у литератури [18]. Програмски код дефинише хардверски модул за извршавање битоник сортирања на броју елемената који је степен броја 2 користећи `Swapper` модуле за поређење елемената. Дати примери показују како користити битоник сортер модул за конкретне типове података и величине. Сада следи објашњење програмског кода:

1. Увозе се Чизел библиотеке и дефинише се пакет за битоник сортирање

2. Класа `Swapper` дефинише хардверски модул за замену два улазна елемента на основу функције поређења. Овај модул има два улаза (`a0` и `a1`) и два излаза (`z0` и `z1`).

```
  when(lt(io.a1, io.a0)) {
    io.z0 := io.a1
    io.z1 := io.a0
  }.otherwise {
    io.z0 := io.a0
    io.z1 := io.a1
  } }
```

3. Креира се објекат `BitonicSorter`, који је фабрика за креирање битоник сортер хардверских модула. Он прима `IndexedSeq` (Indexed Sequence) која садржи опционе улазне елементе `a` и функцију `factory()` за креирање `Swapper` модула. Функција проверава да ли је број елемената за сортирање степен броја 2 и затим врши сортирање елемената користећи битоник алгоритам за сортирање.

```
def insertSorter(a: IndexedSeq[Option[T]], lo: Int, hi: Int):
  IndexedSeq[Option[T]] = { (a(lo), a(hi)) match {
    case (_, None)         => a
    case (None, Some(aH)) => a.updated(lo, Some(aH))
                              .updated(hi, None)
    case (Some(aL), Some(aH)) =>
      val m = Module(factory())
      m.io.a0 := aL
      m.io.a1 := aH
      a.updated(lo, Some(m.io.z0))
        .updated(hi, Some(m.io.z1)) } }
```

- `insertSorter` је помоћна функција која врши сортирање уметањем у опсегу елемената у a задатим са `lo` и `hi`. Она врши замену елемената користећи `Swapper` модуле на основу функције поређења `lt`.
- Главна логика сортирања се извршава у угњежденој петљи, где `i` представља битну позицију која се разматра, `j` варира од `i` до `0`, и `k0` и `k1` итерирају кроз елементе који требају да се пореде.
- Унутар петље се проверава да ли је потребна замена елемената користећи `Swapper` модуле на основу битних позиција и функције поређења.
- Резултат је сортирана `IndexedSeq` опционих елемената.

```
(for {
      i <- 0 until log2Up(a.length)
      j <- i to 0 by -1
      k0 <- a.indices by (2 << j)
      k1 <- 0 until 1 << j
    } yield {
      val lo = k0 + k1
      val hi = lo + (1 << j)
      if ((lo >> (i + 1)) % 2 == 0) (lo, hi) else (hi, lo)
    }).foldLeft(a) { case (s, (l, h)) => insertSorter(s, l, h) }
```

4. Класа `SorterModuleIfc` дефинише интерфејс за битоник сортер модул. Прима број елемената `n` и прототип `proto` за елементе.

5. Дефинише се класа `BitonicSorterModule` која проширује `SorterModuleIfc`. Ова класа представља битоник сортер модул и прима број елемената `n`, прототип `proto`, и функцију поређења `lt`.

   - Креира се `IndexedSeq` опционих елемената `a`, иницијализујући га улазним елементима `io.a`. Величина `a` је постављена на следећи степен броја 2 који је већи или једнак `io.a.length`.
   - Сортирају се елементи `a` користећи фабрику `BitonicSorter`, наводећи прототип и функцију поређења. Резултат је сортиран `IndexedSeq` опционих елемената.

     ```
     class BitonicSorterModule[T <: Data](n: Int, proto: T,
           lt: (T, T) => Bool) extends SorterModuleIfc(n, proto) {
     private val a = IndexedSeq.tabulate(1 << log2Up(io.a.length))
                       { i => if (i < n) Some(io.a(i)) else None }
     io.z := VecInit(BitonicSorter(a, () =>
         new Swapper(proto.cloneType, lt)).slice(0, n).map(_.get))
     }
     ```

   - Сортирани елементи се додељују излазу `io.z`.

6. Програмски код садржи примере позива кода за генерисање за битоник сортер модула. Дата су два примера за сортирање 64 и 384 елемената 8-битних неозначених целих бројева (`UInt8`) користећи `BitonicSorterModule`.

Ови примери генеришу SystemVerilog излаз користећи `ChiselStage` из Чизела и прослеђују тип и функцију поређења као аргументе.

```
object BitonicSorterUInt8_64Driver extends App {(new ChiselStage)
          .emitSystemVerilog(new BitonicSorterModule(64, UInt(8.W),
                                  (x: UInt, y: UInt) => x < y), args)
}
```

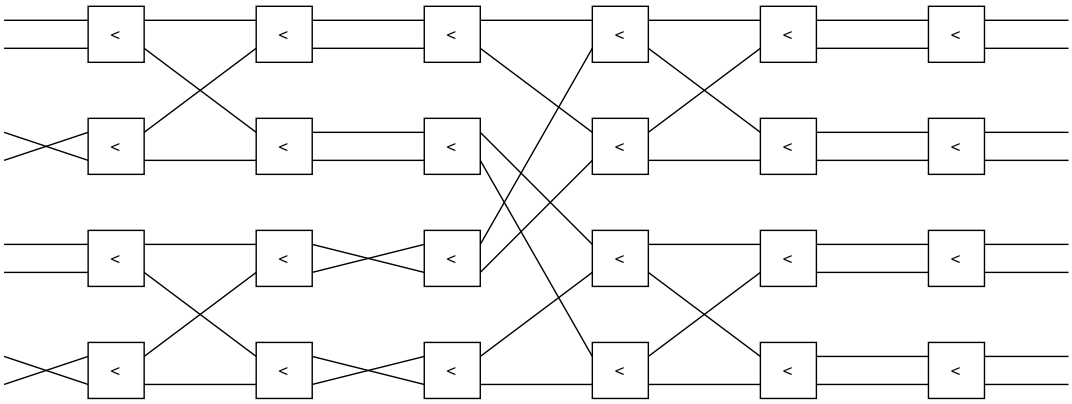
4.1.1 Анализа рада итеративне имплементације

Као што је претходно назначено, главна логика сортирања извршава се у угњежденој петљи, која је изнад приказана.

l	h	l	h	l	h	l	h	l	h	l	h
0	1	0	2	0	1	0	4	0	2	0	1
3	2	1	3	2	3	1	5	1	3	2	3
4	5	6	4	5	4	2	6	4	6	4	5
7	6	7	5	7	6	3	7	5	7	6	7

Табела 1: Парови индекса бројева који се упоређују

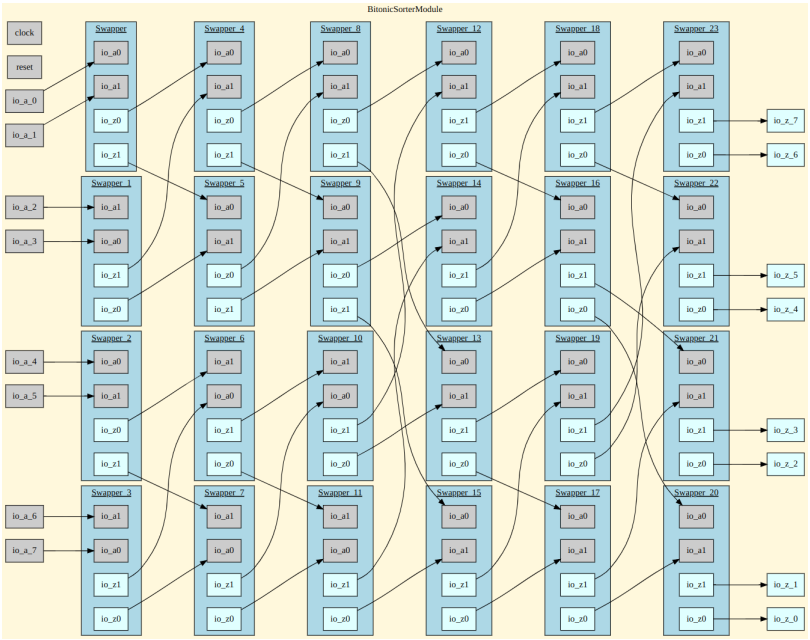
Ако бисмо посматрали пример да на улазу имамо низ од осам бројева, и ако бисмо генерисали парове индекса бројева унутар тог низа које је потребно поредити у сваком нивоу (табела 1), битоник мрежа за сортирање би изгледала као на слици 4.1.1.



Слика 4.1.1: Битоник мрежа за сортирање - итеративна имплементација

Занимљиво је приметити да је ова мрежа другачија од оне приказане на слици 3.1. Има исти број нивоа паралелног извршавања операција поређења, али се везе између неких нивоа разликују.

На слици 4.1.2 је приказана визуализација кола извршавањем команде `visualize(() => new BitonicSorterModule(8, UInt(8.W), ...))`



Слика 4.1.2: Визуализација итеративне имплементације

Може се приметити да су парови који се пореде исти као у табели 1.

4.1.2 Карактеристике генерисаног кола

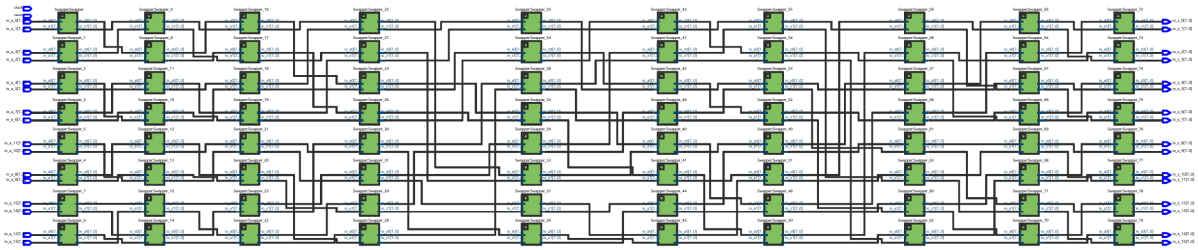
Позивом `getVerilog(new BitonicSorterModule(...)` је добијен Верилог програмски код ове имплементације. Синтеза и симулација рада битоник сортера је урађена у софтверу Quartus Prime Lite, а FPGA уређај коришћен за синтезу је Cyclone V 5CGXFC9E7F35C8.

Број ел.	Ширина	LUT	Pin
8	8	347	130
8	16	673	258
8	32	1669	514
16	8	929	258
16	16	2154	514

Табела 2: Простор који синтеза заузима за различите величине улаза

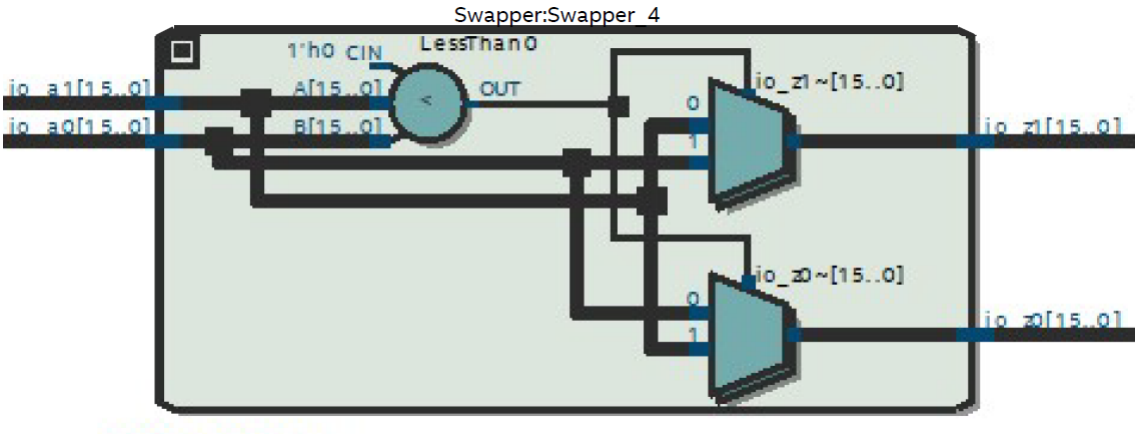
Извршена је синтеза за ону величину улаза и излаза колико овај модел има пинова да подржи (616). Табела 2 приказује колико простора синтеза кода заузима простора на чипу, за различите величине улаза (и излаза). За различит број елемената на улазу и ширину података приказане су у колонама: LUT се односи на број искоришћених Look-Up табела и Pin представља искоришћен број пинова потребних за У/И.

Слика 4.1.3 приказује генерисано коло за сортирање 16 8-битних елемената на улазу. Може се приметити да коло значајно расте са порастом броја елемената за сортирање.



Слика 4.1.3: Слика добијеног кола за сортирање 16 елемената

Следећа слика приказује један `Swapper` модул.



Слика 4.1.4: Слика добијеног кола за сортирање 16 елемената

4.2 Рекурзивна имплементација у Пајтону

Логика рекурзивног решења у Чизелу изложеног у овом раду се базира на имплементацији у Пајтону, наведеној у литератури [19].

Сада следи објашњење програмског кода:

1. `compAndSwap(a, i, j, dire)` функција служи за упоређивање и замену два елемента у низу a на позицијама `i` и `j` у зависности од смера сортирања (уколико је `dire` једнак 1, онда сортира у растућем редоследу, а уколико је 0, онда сортира у опадајућем редоследу).

```
def compAndSwap(a, i, j, dire):
  if (dire==1 and a[i] > a[j]) or (dire==0 and a[i] < a[j]):
      a[i],a[j] = a[j],a[i]
```

2. `bitonicMerge(a, low, cnt, dire)` функција рекурзивно сортира битонички низ у растућем или опадајућем редоследу, у зависности од вредности параметра `dire`. Низ се сортира почевши од позиције `low`, а параметар `cnt` представља број елемената који се сортирају.

```
def bitonicMerge(a, low, cnt, dire):
  if cnt > 1:
    k = cnt//2
    for i in range(low , low+k):
        compAndSwap(a, i, i+k, dire)
    bitonicMerge(a, low, k, dire)
    bitonicMerge(a, low+k, k, dire)
```

3. `bitonicSort(a, low, cnt, dire)` функција прво производи битонички низ рекурзивним сортирањем његових две половине у супротним редоследима, а затим позива функцију `bitonicMerge` да их уједини у исти редослед.

```
def bitonicSort(a, low, cnt,dire):
  if cnt > 1:
    k = cnt//2
    bitonicSort(a, low, k, 1)
    bitonicSort(a, low+k, k, 0)
    bitonicMerge(a, low, cnt, dire)
```

4. Из функције `sort(a, N, up)` се позива сортирање целокупног низа `a` дужине `N` у растућем или опадајућем редоследу, у зависности од вредности параметра `up`.

5. Главни део програма почиње креирањем низа `a` који треба сортирати и подешава величину низа `n` и смер сортирања `up`. Затим се позива функција `sort(a, n, up)` да изврши сортирање низа, и на крају се исписује сортиран низ.

4.3 Имплементација битоник сортер у Верилогу

У овом поглављу ће бити описана имплементација битоник сортера у Верилогу, наведена у литератури [20]. Описан је програмски код решења и наведене су карактеристике добијеног кола.

Пројекат се састоји од 4 модула `bitonic_node`, `bitonic_comp`, `bitonic_block` и `bitonic_sort`, где је `bitonic_sort` на највишем нивоу:

1. Модул `bitonic_node`:
    - Представља основну јединицу битоник мреже за сортирање.
    - Користи се за поређење и размену две вредности.
    - Параметри:
        - `DATA_WIDTH`: Представља ширину података тј. број битова бинарне репрезентације једног елемента.
        - `ORDER`: Редослед овог чвора у битоник мрежи.
        - `POLARITY`: Одређује редослед (улазан или силазан).
        - `SIGNED`: Одређује да ли су бројеви означени или неозначени.
    - Улази:
        - `clk`: Сигнал такта.
        - `data_in`: Улазни подаци овог чвора.
    - Излази:
        - `data_out`: Излазни подаци из овог чвора.
    - Унутар овог модула има више инстанци компоненте `bitonic_comp` које извршавају стварно поређење и замену.
    - Користи `generative` петље да креира и повеже ове инстанце на основу наведених параметара.
    - Основни циљ модула је да усмери улазне податке кроз битоник мрежу за сортирање и произведе сортиране излазе.
2. Модул `bitonic_comp`:
    - Овај представља компаратор и логику размене за две вредности података.
    - Параметри:
        - `DATA_WIDTH`: Представља ширину података
        - `POLARITY`: Одређује редослед (улазан или силазан).
        - `SIGNED`: Одређује да ли су бројеви означени или неозначени.
    - Улази:
        - `CLK`: Сигнал такта.
        - `A`: Прва улазна вредност.
        - `B`: Друга улазна вредност.
    - Излази:
        - `H`: Виша од две улазне вредности.

- `L`: Нижа од две улазне вредности.
- `O`: Излаз који означава да ли је `A` мање од `B`.

- Овај модул пореди улазне вредности `A` и `B` и одређује која је виша `H` и која је нижа `L`.
- Такође, производи излаз `O` који указује да ли је `A` мање од `B`.
- Поларитет компаратора (узлазан или силазан) контролише се параметром `POLARITY`.
- Модул користи регистре за чување резултата и ажурира их на сваком такту.

3. Модул `bitonic_block`:

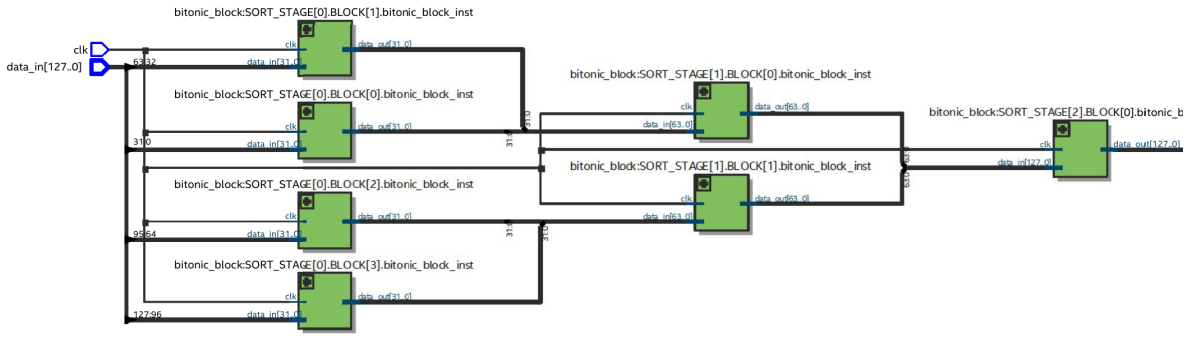
- Представља степен (ниво) у битоник мрежи за сортирање
- Параметри:
  - `DATA_WIDTH`: Представља ширину података.
  - `ORDER`: Редослед овог степена у битоник мрежи.
  - `POLARITY`: Одређује редослед (улазан или силазан) степена.
  - `SIGNED`: Одређује да ли су бројеви означени или неозначени.
- Улази:
  - `clk`: Сигнал такта.
  - `data_in`: Улазни подаци за овај степен.
- Излази:
  - `data_out`: Излазни подаци из овог степена.
- Унутар овог модула има више инстанци модула `bitonic_node` које чине битоник мрежу овог степена.
- Користи `generative` петље да креира и повеже ове инстанце на основу наведених параметара.
- Основни циљ овог модула је управљати радом битоник мреже за дати степен.

4. Модул `bitonic_sort`:

- Ово је модул на највишем нивоу који представља потпуну битоник мрежу за сортирање.
- Параметри:
  - `DATA_WIDTH`: Представља ширину података.
  - `CHAN_NUM`: Одређује број елемената које треба сортирати.
  - `DIR`: Одређује смер сортирања (узлазан или силазан).
  - `SIGNED`: Одређује да ли су бројеви означени или неозначени.
- Улази:
  - `clk`: Сигнал такта.
  - `data_in`: Улазни подаци које требас сортирати.

- Излази:
  - `data_out`: Сортирани излазни подаци.
- Овај модул је улазна тачка за битоник мрежу за сортирање.
- Он управља сортирањем података делећи га на степене и блокове користећи модуле `bitonic_block` и `bitonic_node`.
- `generative` петље се користе у стварању хијерархијске структуре неопходне за битоник сортирање.
- Сортирани излаз доступан је на порту `data_out`.

Укратко, ови модули заједно раде на имплементацији битоник мреже која може сортирати више канала података у узлазном или силазном редоследу, зависно од конфигурационих параметара.



Слика 4.3.5: Слика добијеног кола - RTL View

Слика 4.3.5 је добијена коришћењем RTL View унутар алата Quartus Prime Lite и представља сортер осам 16-битних бројева добијен овом имплементацијом у Верилогу. Може се приметити да се мрежа састоји из три нивоа. Број нивоа се рачуна унутар `bitonic_sort`. У овом случају параметар `CHAN_NUM` износи 8 па је број нивоа у мрежи 3, у случају да је потребно сортирати 16 елемената број нивоа би био 4. Приказан је део програмског код модула `bitonic_sort`. Број блокова у нивоу се дефинише у параметру `BLOCKS`.

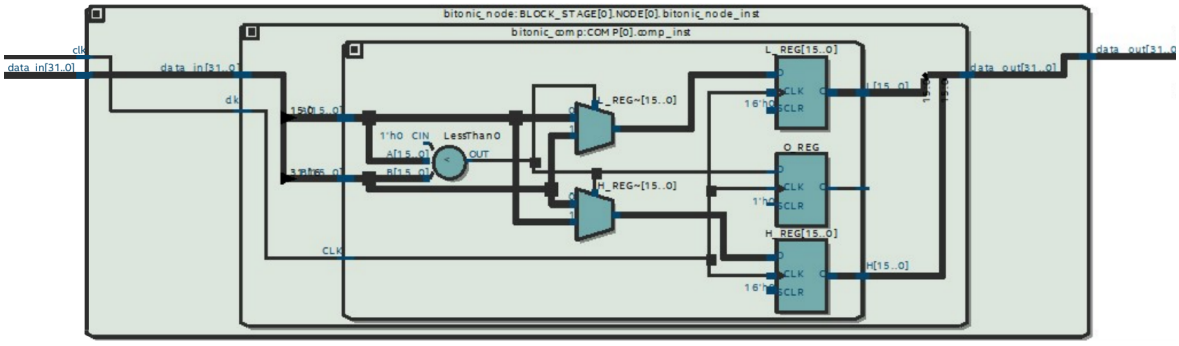
```
localparam CHAN_ACT = 2**$clog2(CHAN_NUM);
localparam STAGES = $clog2(CHAN_ACT);
generate for (stage = 0; stage < STAGES; stage = stage + 1)
  begin: SORT_STAGE
        localparam BLOCKS = CHAN_ACT / 2**(stage+1);
        localparam BLOCK_ORDER = stage;

        wire [STAGE_DATA_WIDTH-1:0]stage_data_in;
        wire [STAGE_DATA_WIDTH-1:0]stage_data_out;
        assign stage_data_in = stage_data[stage];
        assign stage_data[stage + 1] = stage_data_out;
        for (block = 0; block < BLOCKS; block = block + 1) begin: BLOCK
```

Модул `bitonic_block` (из првог нивоа) је приказан на слици 4.3.6. Следеће линије програмског код део су `bitonic_block` модула, параметар је `ORDER` једнак трунутном нивоу унутар мреже, па ће нпр. број инстанци `bitonic_node`-а које су део `bitonic_block`-а другом нивоу мреже бити $2^0 + 2^1 = 3$ (постоје два нивоа унутар блока).

```
localparam STAGES = ORDER + 1;
generate for (stage = 0; stage < STAGES; stage = stage + 1)
 begin: BLOCK_STAGE
        localparam NODES = 2**stage;
        localparam NODE_ORDER = STAGES - stage - 1;
```

Број инстанци `bitonic_comp` унутар једног `bitonic_node`-а је већи што је он ближи улазу, а тај број је дефинисан као 2 на степен колико је сам `bitonic_node` удаљен од излаза из блока, нпр. онај до излаза ће имати $2^0 = 1$ инстанцу.



Слика 4.3.6: Приказ подмодула у добијеном колу - RTL View

4.3.1 Карактеристике генерисаног кола

Синтеза и симулација рада битоник сортера је урађена у Quartus Prime Lite, а FPGA уређај коришћен за синтезу је Cyclone V 5CGXFC9E7F35C8.

Број ел.	Ширина	LUT	Reg	Pin
8	8	325	455	129
8	16	620	867	257
8	32	1217	1602	513
16	8	1100	1609	257
16	16	2067	2788	513

Табела 3: Простор који синтеза заузима за различите величине улаза

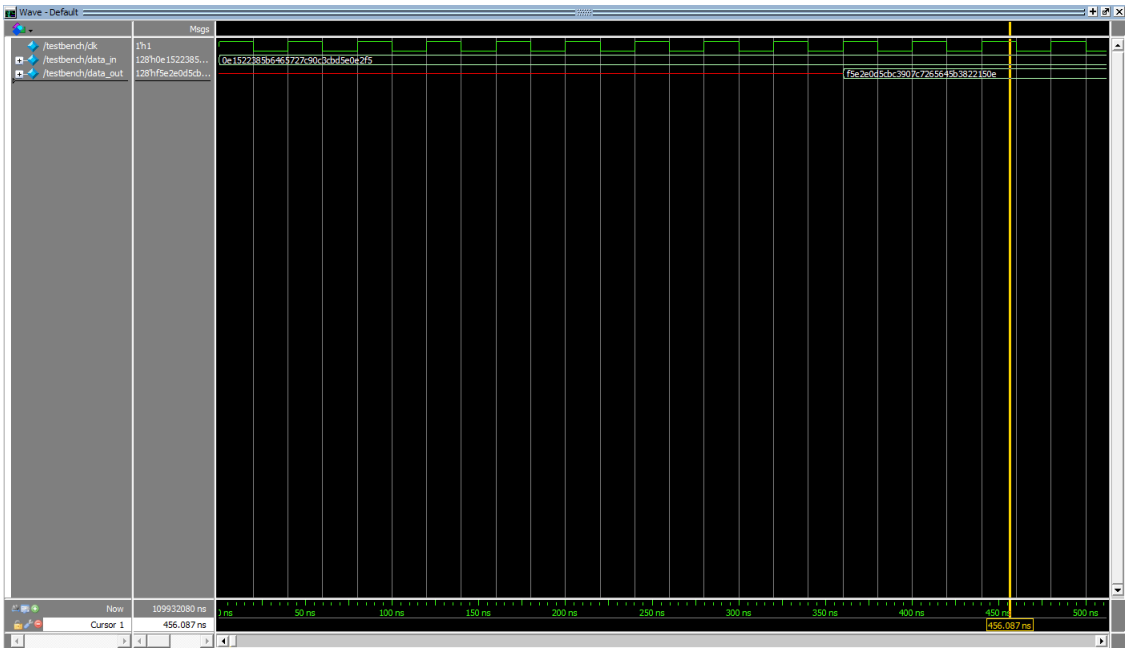
Табела 3 приказује колико простора синтеза кода заузима простора на чипу, за различите величине улаза (и излаза). За различит број елемената на улазу и ширину података приказане су у колонама: LUT се односи на број искоришћених Look-Up табела, колона Reg представља укупан број потребних регистара, а Pin је искоришћен број пинова потребних за У/И.

Табела 4 приказује број тактних циклуса потребних да се добије сортиран низ на излазу кола. Симулација је извршена за различите величину улаза (и излаза) и различите низове (насумични, скоро сортирани и обрнуто сортирани низови).

Број ел.	Ширина	Насумични	Скоро с.	Обрнуто с.
8	8	6	6	6
8	16	6	6	6
8	32	6	6	6
16	8	10	10	10
16	16	10	10	10

Табела 4: Број тактних циклуса за различите низове и величину улаза

Следећа слика приказује симулацију улазног низа од 16 бројева коришћењем EDA алата Questa у оквиру Quartus Prime Lite.



Слика 4.3.7: Симулација добијеног кола за сортирање 16 елемената

4.4 Имплементација битоник сортера у Верилогу - комбинациона

Сада ће бити описана једноставна имплементација битоник сортера у Верилогу [21], коло које се добије је секвенцијално, а број елемената који се сортирају је фиксан (16) и ширина података је 16. Сада ће бити описан програмски код:

1. Модул `BN` има два 16-битна улаза `a` и `b` и два 16-битна излаза, `min` и `max`. Излазне вредности се дефинишу као:

```
assign min = (a < b) ? a : b;
assign max = (a < b) ? b : a;
```

2. Модул `BN1` је сличан `BN`, али мења редослед излаза; прво `max`, па `min`.

3. Модул `sorting` је главни модул за сортирање који сортира 8 16-битних вредности. Улази су `a1` до `a8`, излази `b1` до `b8`. Може се рећи да је ова имплементација „хардкодована" репрезентација битоник мреже за сортирање као на сликама 3.1 или 4.1.3. Сваки ниво мреже се експлицитно дефинише и спаја жицама. Иако једноставно решење, није параметризовано и може се користити само за фиксан број елемената на улазу.

```
BN m21(a1,a2,mnn1,mxx1);
BN1 m22(a3,a4,mnn2,mxx2);
BN m23(a5,a6,mnn3,mxx3);
BN1 m24(a7,a8,mnn4,mxx4);
```

4.4.1 Карактеристике добијеног кола

Извршена је синтеза и симулација овог кола. Следећа табела описује простор који генерисано коло заузима на чипу.

Број ел.	Ширина	LUT	Pin
16	8	929	256
16	16	2154	512

Табела 5: Простор који синтеза заузима за различите величине улаза

На слици добијеној коришћењем RTL view из алата Quartus, може се приметити да се заправо ради о истом решењу као што је претходно описана итеративна у Чизелу (осим што `BN` и `BN1` имају окренуте излазе).

5 Рекурзивна имплементација битоник сортера у Чизелу - резултат рада

Две имплементације описане у претходном поглављу су коришћене као основа за израду ове, где је прва коришћена као референца за синтаксу Чизела, а друга као логичка основа. Оно што је било изазовно је размевање начина рада итеративне имплементације где се прави посебна инстанца модула за замену два елемента `Swapper` и излаз додељује новом низу, који је повратна вредност функцкије `insertSorter()`. То значи да приступ примењен у рекурзивној Пајтон имплементацији није могућ а да коло истовремено буде секвенцијално, јер се ту мења улазни низ. Потребно је било написати рекурзивну имплементацију у Чизелу где функције `sort`, `bitonicSort` и `bitonicMerge` као повратну вредност имају низ тј. `IndexedSeq[Option[T]]`, а рекурзивним позивима функција се прослеђују само делови улазног низа.

Да би се генерисала нова листа са замењеним (свапованим) елементима у одговарајућем редоследу, у `bitonicMerge()` функцији, искоришћена је `foldLeft()` функција, која извршава функцију `swapIfNecessary()`. Потом се долази до проблема саме замене елемената и на овом примеру се може видети како су Скала и Чизел сједињени и зашто је бити јасно разграничити типове података.

На крају, прави се инстанца једноставног модула `Passthrough` који само прослеђује вредности са улаза, и излаз се додељује новој листи, а на улазу се у зависности од `swapNeeded` елементи прослеђују у обрнутом или истом редоследу. Ако би се слично као у итеративној имплементацији прослеђивала `Bool` вредност `swapNeeded` модулу `Swapper`, Чизел би пријавио грешку да не види ову вредност из модула `Swapper`, па је на овај начин решен тај проблем.

5.1 Опис програма

Сада ће бити описан програмски код имплементације изложене у овом раду:

1. `Passthrough` је једноставан модул који има два улазна порта `a0` и `a1` и два излазна порта `z0` и `z1`. Прихвата два податка истог типа `T` и директно их рутира на излазне портове без измене.

2. Објекат `BitonicSorter` дефинише функције за имплементацију битоник сорт алгоритма: `bitonicMerge()`, `bitonicSort()` и `sort()`.

3. Функција `bitonicMerge()` прима низ елемената `arr`, дужину низа `length`, информацију о редоследу сортирања `ascending`, и функцију за упоређивање два елемента `lt()`.

    - Унутрашња функција `swapIfNecessary()` прихвата низ типа `IndexedSeq[Option[T]]`, целобројни параметар `i`, логичку вредност `ascending` и функцију за поређење `lt()` као параметре. Ова функција је одговорна за замену два елемента у улазном низу ако је то потребно на основу функције поређења и флега ascending. Елементи се замењују кроз модул `Passthrough` и то зато што је било потребно користити услов `when` јер се проверава вредност `swapNeeded` која је типа `Chisel.Bool`.

На улазе овог модула се прослеђују вредности које су замењене или не, а излази `m.io.a0` и `m.io.a1` се уписују излазној секвенци.

```
def swapIfNecessary(seq: IndexedSeq[Option[T]], i: Int,
 ascending: Bool, lt: (T, T) => Bool):IndexedSeq[Option[T]]={
  val j = i + length / 2
  val si = arr(i).getOrElse(0.U.asTypeOf(arr(i).get))
  val sj = arr(j).getOrElse(0.U.asTypeOf(arr(j).get))
  val m = Module(new Passthrough(si.cloneType))
  val swapNeeded = lt(sj, si) === ascending
  when(swapNeeded) {
    m.io.a0 := sj
    m.io.a1 := si
  }.otherwise {
    m.io.a0 := si
    m.io.a1 := sj
  }
  seq.updated(i, Some(m.io.z0)).updated(j, Some(m.io.z1))
}
```

- Ако је дужина низа `arr` 1 или мања, низ се сматра већ сортираним и враћа се без промена.
- Ако је дужина низа већа од 1, низ се дели на две половине и за сваки пар елемената `(i, i+length/2)` се врши позив функције `swapIfNecessary()` над овим индексима коришћењем `foldLeft()`.

```
val swappedSeq = indices.foldLeft(arr) { (seq, i) =>
  swapIfNecessary(seq, i, ascending, lt) }
```

- Након што се обаве сви кораци, сортиран низ се враћа као излаз функције.

4. Функција `bitonicSort()` прима низ елемената `arr` који треба сортирати, информацију о редоследу сортирања `ascending` - да ли је сортирање у растућем редоследу или не, и функцију за упоређивање два елемента `lt`.

   - Ако је дужина низа `arr` 1 или мања, низ се сматра већ сортираним и враћа се без промена
   - Ако је дужина низа већа од 1, врше се рекурзивни позиви половина низа у растућем и опадајућем редпоследу; а потом се две половине спајају користећи `bitonicMerge()`.
   - Након што се обаве сви кораци, сортиран низ се враћа као излаз функције.

```
def bitonicSort[T<:Data](arr: IndexedSeq[Option[T]], ascending: Bool,
                         lt: (T, T) => Bool): IndexedSeq[Option[T]] = {
  val n = arr.length
  if (n > 1) {
    val half = n / 2
    val left = bitonicSort(arr.slice(0, half), true.B, lt)
```

```
    val right = bitonicSort(arr.slice(half, n), false.B, lt)
    val merged = bitonicMerge(left ++ right, n, ascending, lt)
    merged
  } else {
    arr } }
```

5. Функција `sort()` проверава да ли је дужина улазног низа 1 и ако јесте враћа га као излазну вредност, а ако је већа од 1 позива функцију `bitonicSort()` за извршење сортирања.

6. Класа `SorterModuleIfc` дефинише интерфејс за модул битоник сортера. Као параметре прима број елемената за сортирање `n` и тип података елемената `proto`. Овде су дефинисани улаз `a` и излаз `z`.

7. Класа `BitonicSorterModule` представља стварни модул битоник сортера који би се користио у хардверском дизајну.

   - Као улазне параметре прихвата број елемената за сортирање `n`, тип података елемената `proto`, флег за редослед сортирања `ascending` и функцију за упоређивање `lt`.
   - Проверава се да ли је улзани низ степен броја 2 и избацује изузетак ако није испуњен овај услов.
   - Креира интерну секвенцу `a` која се користи за сортирање. Елементи ове секвенце се изводе из улазних података `io.a`, обезбеђујући да дужина секвенце буде степен броја 2.

     ```
     IndexedSeq.tabulate(if (n <= 1) 1 else 1<<log2Up(io.a.length)) {
       i => if (i < n) Some(io.a(i)) else None }
     ```

   - Сортиран излаз се израчунава помоћу функције `BitonicSorter.sort()` и додељује се излазном порту `io.z`.

8. Примери генерисања кода `BitonicSorterUInt8_64Driver` и `BitonicSorterUInt8_384Driver` демонстрирају како користити модул битоник сортера са конкретним конфигурацијама, тако што се креирају инстанце `BitonicSorterModule` са различитим параметрима (нпр. сортирају 64 елемента типа `UInt(8)` у растућем редоследу) и генеришу SystemVerilog код користећи `ChiselStage`.

5.2 Тест за битоник сортер

Сада ће бити описано тестирање рекурзивне имплементације битоник сортера, где се користе библиотеке `ChiselTest` и `ScalaTest`.

1. Класа `BitonicSorterModuleTest` је главна тест класа која проширује `AnyFlatSpec` из `ScalaTest` и `ChiselScalatestTester`. Користи се за тестирање `BitonicSorterModule`-а.

```
class BitonicSorterModuleTest extends AnyFlatSpec
                                    with ChiselScalatestTester {
def generateRandomInput(size: Int, len: Int): Seq[UInt] = {
  Seq.fill(len)(BigInt(size, Random).U(size.W))
}
```

2. Функција `generateRandomInput` генерише случајни низ улазних вредности `UInt`. Прима два параметра: `size` (број битова сваког елемента) и `len` (број елемената у низу). Користи `Seq.fill()` да креира низ случајних `UInt` вредности са наведеним величинама.

```
val lengths = Seq(1, 2, 4, 8, 16)
val randomIndex = Random.nextInt(lengths.length)
val length = lengths(randomIndex)
test(new BitonicSorterModule(length, UInt(bitSize.W), false.B,
                             (x: UInt, y: UInt) => x < y)) { dut =>
  val input = generateRandomInput(bitSize, length)
  dut.io.a.zipWithIndex.foreach { case (inputPort, index) =>
    inputPort.poke(input(index))}
```

3. Тестни случајеви:

   - Први тест случај проверава да ли `BitonicSorterModule` исправно сортира улазне вредности у растућем редоследу за различите величине података (8, 16, 32 и 64). Генерише случајне низове улазних вредности различитих дужина (1, 2, 4, 8 и 16) за сваку ширине података и проверава да ли излазне вредности одговарају растућем редоследу.

```
// Check if the output values are in descending order
val veryLargeNumber = BigInt("92233720368547758075555")
var prevValue = veryLargeNumber
dut.io.z.foreach { outputPort: UInt =>
  val value = outputPort.peek().litValue //.toInt
  //println(value)
  outputPort.expect(value.U)
  assert(value <= prevValue,
            "Output values are not in descending order.")
  prevValue = value }
```

- Други тест случај је сличан првом, али проверава да ли модул исправно сортира улазне вредности у опадајућем редоследу. Такође тестира за различите ширине података и случајне дужине улаза.

```
it should """not allow construction of Arrays whose length
        is not a power of 2""" in {
  assertThrows[IllegalArgumentException] {
          new BitonicSorterModule(7,
            UInt(8.W), true.B, (x: UInt, y: UInt) => x < y) }
```

- Трећи тест случај проверава да ли конструктор `BitonicSorterModule` баца `IllegalArgumentException` када се покуша креирање инстанце са дужином улаза која није степен броја 2.

```
[leka@lekaTP BitonicSorterRec]$ sbt test
[info] welcome to sbt 1.8.2 (Oracle Corporation Java 1.8.0_382)
[info] loading global plugins from /home/leka/.sbt/1.0/plugins
[info] loading project definition from /home/leka/Documents/master/masterRad/Idea
[info] loading settings for project root from build.sbt ...
[info] set current project to BitonicSorterRec (in build file:/home/leka/Document
[info] compiling 1 Scala source to /home/leka/Documents/master/masterRad/IdeaProj
[warn] 4 feature warnings; re-run with -feature for details
[warn] one warning found
[info] BitonicSorterModuleTest:
[info] BitonicSorterModule
[info] - should sort input values in ascending order
[info] BitonicSorterModule
[info] - should sort input values in descending order
[info] - should not allow construction of Arrays whose length is not a power of 2
[info] Run completed in 2 seconds, 975 milliseconds.
[info] Total number of tests run: 3
[info] Suites: completed 1, aborted 0
[info] Tests: succeeded 3, failed 0, canceled 0, ignored 0, pending 0
[info] All tests passed.
```

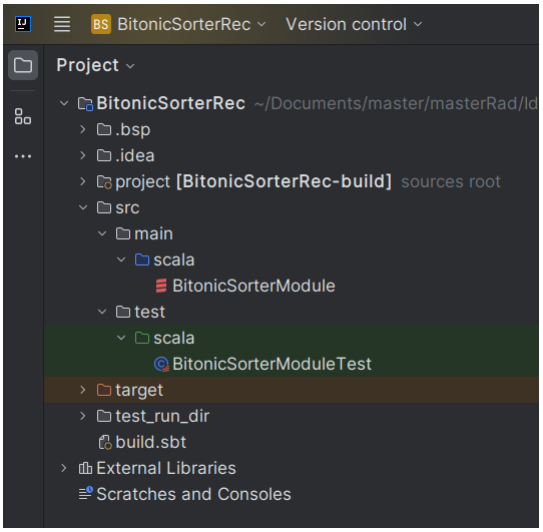
Слика 5.2.1: Извршавање теста

5.3 Покретање програма

У GitHub репозиторијуму *BitonicSorterRecursive*, наведеном у литератури [22], се налазе:

- Пројекат *BitonicSorterRec*, који се може отворити у Intellij-у
- LaTeX код овог рада и pdf датотека

Покретање пројекта се врши извршавањем команде `sbt test` у директоријуму пројекта.

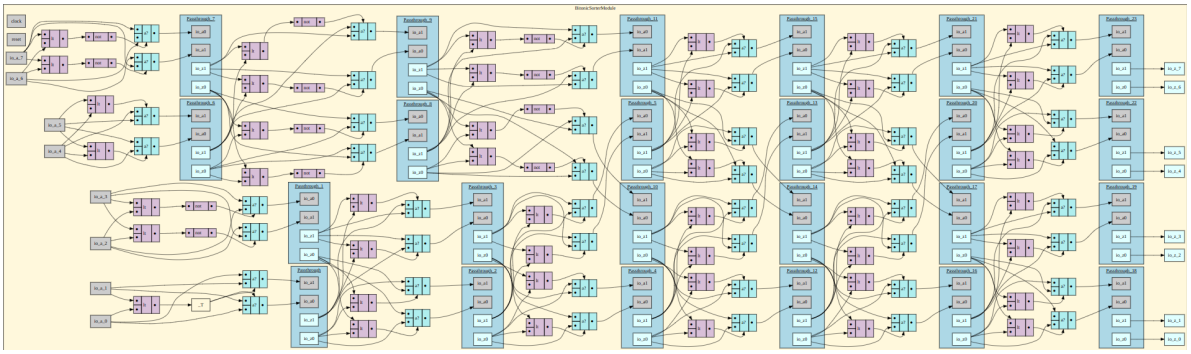


Слика 5.3.2: Структура пројекта

Слика 5.3.2 приказује структуру пројекта, `BitonicSorterModule.scala` се налази у директоријуму `/src/main/scala/`, а `BitonicSorterModuleTest.scala` у `/src/test/scala/`. Слика 5.2.1 приказује резултат извршавања теста.

5.4 Приказ добијеног кола

Слика 5.4.3 прикатује генерисано коло овако имплементираног битоник сортера и на први поглед изгледа сложеније него коло имплементирано итеративним приступом.



Слика 5.4.3: Приказ рекурзивне имплементације

Програмски код итеративне имплементације модул `Swapper` независно врши замену елемената, док су овде у замену укључени модул `Passthrough`, функција поређења `lt()`, `ascending` и може се видети негација `not`; али су то су елементи који се понављају.

5.5 Карактеристике генерисаног кола

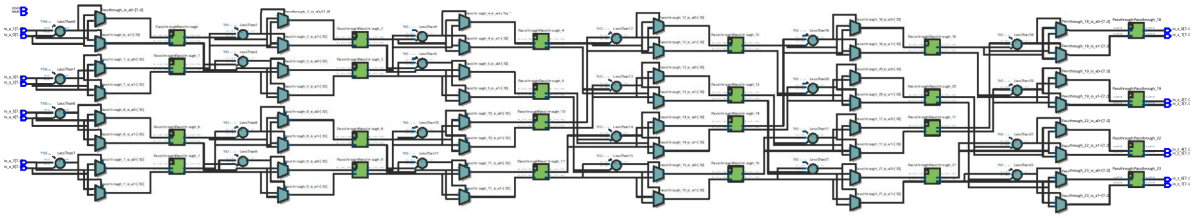
Позивом `getVerilog(new BitonicSorterModule(...)` је добијен Верилог програмски код рекурзивне имплементације. Синтеза и симулација рада битоник сортера је урађена у софтверу Quartus Prime Lite, резултати су приказани у наредној табели.

Број ел.	Ширина	LUT	Pin
8	8	347	130
8	16	676	258
8	32	1669	514
16	8	931	258
16	16	2155	514

Табела 6: Простор који синтеза заузима за различите величине улаза

Може се приметити да су вредности у колонама LUT и Pin готово исте као у случају синтезе кола претходно описане итеративне имплементације у Чизелу.

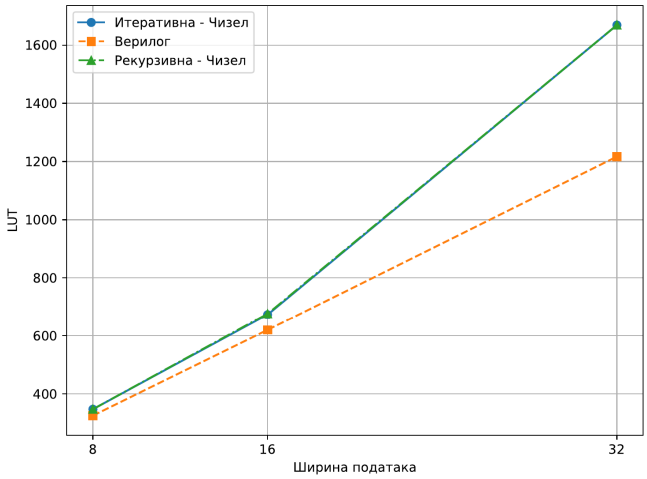
Слика 5.5.4 приказује генерисано коло за сортирање 8 8-битних елемената, иако коло изгледа сложено, заправо се из табеле 6 види заузумају исто простора на чипу.



Слика 5.5.4: Слика добијеног кола за сортирање 8 елемената

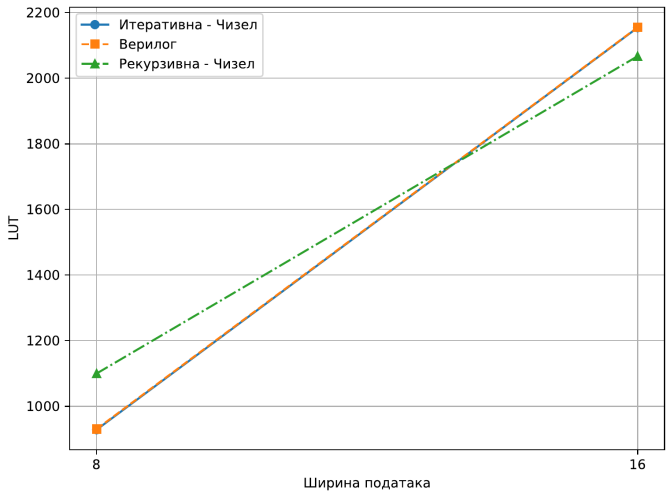
5.5.1 Поређење са претходним имплементацијама

Следећи график приказује број Look-up табела потребних за синтезу приказаних имплементација на FPGA уређају Cyclone V 5CGXFC9E7F35C8 када је број елемената за сортирање 8.



Слика 5.5.5: LUT - дужина низа 8

График приказан на следећој слици приказује број Look-up табела када је дужина низа 16.



Слика 5.5.6: LUT - дужина низа 16

Следећа табела приказује разлику у величини резултујућег Верилог кода ове имплементације у односу на приказано итеративно решење, зависно до величине улаза.

Број ел.	Ширина	Рекурзивна и.	Итеративна
8	8	356	327
8	16	368	327
8	32	368	327
16	8	1158	1023
16	16	1158	1023

Табела 7: Простор који синтеза заузима за различите величине улаза

Занимљиво је приметити да се у коду рекурзивне имплементације оператори примењују директно над жицама (`wire`), и због тога се добије сложена шема кола после синтезе приказана на сликама 5.5.4 и 5.4.3.

```
wire  _T_2 = io_a_3 < io_a_2; // @[cmd4.sc 1:101]
wire  _T_3 = ~_T_2; // @[cmd2.sc 36:35]
wire  _T_4 = Passthrough_1_io_z0 < Passthrough_io_z0;
    // @[cmd4.sc 1:101]
...
assign Passthrough_io_a0 = _T ? io_a_1 : io_a_0;
    // @[cmd2.sc 37:24 cmd2.sc 38:17 cmd2.sc 41:17]
assign Passthrough_io_a1 = _T ? io_a_0 : io_a_1;
    // @[cmd2.sc 37:24 cmd2.sc 39:17 cmd2.sc 42:17]
...
Passthrough Passthrough ( // @[cmd2.sc 35:21]
  .io_a0(Passthrough_io_a0),
  .io_a1(Passthrough_io_a1),
  .io_z0(Passthrough_io_z0),
  .io_z1(Passthrough_io_z1)
);
```

На основу ових исечака програмског кода може се закључити зашто резултује компликованом шемом кола.

Број ел.	Ширина	Вер.	Рек.	Итер.
8	8	530.08	530.06	531.20
8	16	535.70	535.81	536.66
8	32	546.12	546.50	545.98
16	8	535.41	535.40	535.59
16	16	546.31	545.60	546.36

Табела 8: Простор који синтеза заузима за различите величине улаза

Табела 8 приказује потрошњу енергије изражену у [mW], за рекурзивну, итеративну и имплементацију у Верилогу.

6 Закључак

Чизел нуди могућност дизајнирања хардвера са начином размишљања као да је у питању програмирање у језику вишег нивоа. Самим тим програмирање је лакше, док се добија сличан код као да се директно програмира у језику за дизајн хардвера.

Током израде је било потребно разумети начин рада постојећих решења да би се превазишле разлике и ограничења у програмирању коришћењем програмских језика и језика за опис хардвера. Рекурзивна имплементација битоник сортера приказана у овом раду заузима исти простор на чипу као итеративна, иако шема на први поглед делује компликованије; а сам програмски код је интуитивнији. Код свих имплементација се може видети раст броја елемената у колу са порастом броја елемената за сортирање.

Добијена имплементација није преведена до фазе рутирања и временске анализе. У будућности може се анализирати на колико фреквенцији ова имплементација може да ради, као и који делови се могу оптимизовати. Добијено коло изгледа сложено и шема би била прегледнија ако би имплементација била састављена од подмодула, а не појединачних оператора. Простор за унапређење би био да се додатно анализира битоник сорт алгоритам и уоче још неки начини рекурзивне имплементације чиме би се добила јаснија шема резултујућег кола.

Литература

[1] *Scala (programming language)*, приступљено (август 2023.) на https://en.wikipedia.org/wiki/Scala_(programming_language)

[2] Martin Odersky, Lex Spoon, and Bill Venners, "Programming in Scala, Third Edition", Artima, 2016.

[3] Martin Odersky, *Higher-order functions*, [Интернет курс] *Functional Programming Principles in Scala*, приступљено (август 2023.) на https://www.coursera.org/learn/scala-functional-programming/lecture/V5crF/lecture-2-1-higher-order-functions

[4] *Chisel (programming language)*, приступљено (август 2023.) на https://en.wikipedia.org/wiki/Chisel_(programming_language)

[5] Stephen Twigg, "Flexible FFT Optimization and RTL Generation in the Chisel Hardware Design Language", [Мастер рад, Универзитет у Калифорнији, Беркли], 2015. http://www2.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-256.html

[6] *Chisel Bootcamp*, приступљено (август 2023.) на https://github.com/freechipsproject/chisel-bootcamp.git

[7] Marcela Zuluaga, Peter Milder, and Markus Püschel, "Streaming sorting networks", ACM Trans. Des. Autom. Electron. Syst. 21, 4, чланак 55, 2016. DOI:http://dx.doi.org/10.1145/2854150

[8] K. E. Batcher, "Sorting networks and their applications", ACM, 1968. DOI:http://dx.doi.org/10.1145/1468075.1468121

[9] *Bitonic sorter*, приступљено (август 2023.) на https://en.wikipedia.org/wiki/Bitonic_sorter

[10] H. S. Stone, "Parallel processing with the perfect shuffle", IEEE Transactions on Computers 20, 2, 153–161, 1971. DOI:http://dx.doi.org/10.1109/T-C.1971.223205

[11] Gianfranco Bilardi, Franco P. Preparata "An architecture for bitonic sorting with optimal VLSI performance", IEEE Transactions on Computers, 1984. DOI:http://dx.doi.org/10.1109/TC.1984.5009338

[12] Martin Dowd, Yehoshua Perl, Larry Rudolph, Michael Saks, "The periodic balanced sorting network", Journal of the ACM 36, 4, 738–757, 1989. DOI:http://dx.doi.org/10.1145/76359.76362

[13] Christophe Layer, Hans-Jörg Pfleiderer "A reconfigurable recurrent bitonic sorting network for concurrently accessible data", Field Programmable Logic and Application, 648–657, 2004. DOI:http://dx.doi.org/10.1007/978-3-540-30117-2_66

[14] Christophe Layer, Daniel Schaupp, Hans-Jörg Pfleiderer “Area and throughput aware com-parator networks optimization for parallel data processing on FPGA”, Proceedings of the Interna-tional Symposium on Circuits and Systems, IEEE 405–408, 2007 DOI:http://dx.doi.org/10.1109/ISCAS.2007.378475

[15] Y. Zhang, S. Q. Zheng “An efficient parallel VLSI sorting architecture”, VLSI Design 11, 2, 137–147, 2000. DOI:http://dx.doi.org/10.1155/2000/14617

[16] M. Ajtai, J. Komlós, and E. Szemerédi “ An O(N log N) sorting network.”, VLSI Design 11, 2, 137–147, 2000. DOI:http://dx.doi.org/10.1155/2000/14617

[17] Donald E. Knuth, “The Art of Computer Programming: Sorting and Searching”, Addison-Wesley, 1968.

[18] Steve Burns, *Bitonic Sorter*, приступљено (септембар 2023.) на https://github.com/freechipsproject/ip-contributions/tree/master/src/main/scala/chisel/lib/bitonicsorter

[19] *Bitonic Sort*, приступљено (септембар 2023.) на https://www.geeksforgeeks.org/bitonic-sort/

[20] *Bitonic sorter - Verilog*, приступљено (септембар 2023.) на https://github.com/mcjtag/bitonic_sorter.git

[21] *Bitonic sorter (Combinational) - Verilog*, приступљено (септембар 2023.) на https://digitalsystemdesign.in/bitonic-sorter/

[22] *BitonicSorterRecursive*, приступљено (септембар 2023.) на https://github.com/aleksavelickovic5762015/BitonicSorterRecursive.git

Списак скраћеница

ФВР Функције вишег реда

Списак табела

Списак слика