

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	1
2	Коришћени алати	2
2.1	Скала	2
2.1.1	Функције вишег реда	3
2.2	Чизел	4
2.2.1	Генератори кола и условни изрази	4
3	Мреже за сортирање	7
3.1	Битоник сортер	8
4	Постојећа решења	13
4.1	Итеративна имплементација битоник сортера у Чизелу	13
4.1.1	Анализа рада итеративне имплементације	15
4.2	Рекурзивна имплементација у Пајтону	17
5	Рекурзивна имплементација битоник сортера у Чизелу - резултат рада	18
5.1	Опис програма	19
5.2	Тест за битоник сортер	20
5.3	Покретање програма	21
5.4	Приказ добијеног кола	22
6	Закључак	23
	Литература	24
	Списак скраћеница	25
	Списак табела	25
	Списак слика	26
	Прилог	27
	А. Итеративна имплементација - Чизел	27
	Б. Рекурзивна имплементација - Пајтон	29
	В. Рекурзивна имплементација у Чизелу - резултат рада	30
	Г. Програмски код за тестирање битоник сортера	33

1 Увод

У овом раду ће бити описана рекурзивна имплементација хардверске дигиталне мреже за сортирање битоник сортер. Имплементација је урађена коришћењем језика за опис хардвера Чизел (Chisel). Чизел је базиран на програмском језику Скала и омогућава једноставније дефинисање хардвера од других језика (као Верилог) и прегледнији програмски код.

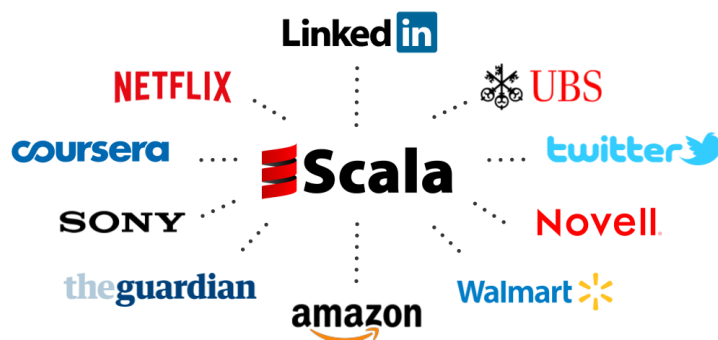
Занимљиво је да се у раду врши поређење са постојећим решењем где је битоник сортер имплементиран итеративно. Замена елемената се у том решењу врши инстанцирањем посебног модула који представља операцију поређења.

Логика решења приказаног у овом раду се базира на постојећој Пајтон имплементацији битоник сорта, али је постојало неколико ограничења да би се она применила у Чизелу. Такође, битно је разликовање променљивих или условних израза који припадају Скали и оних који припадају Чизела.

2 Коришћени алати

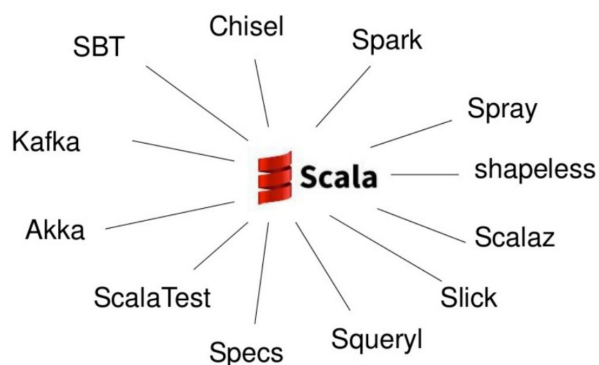
2.1 Скала

Скала (Scala) 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 (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 платформи и ради беспрекорно са свим Јава библиотекама. Примена конструкта функционалног програмирања омогућава брзо стварање 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: 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 `sumFactorials()` (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 (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 (Verilog) 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 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 поквареног дизајна хардвера у Чизелу може бити изазовно. Чизел често иде предалеко са тим да се пише сажет програмски код на уштрб безбедности. 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 УИ (IO) 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 Вериллогу, 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 преправља 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 f0fX = 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.f0fX := result
}

```


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.

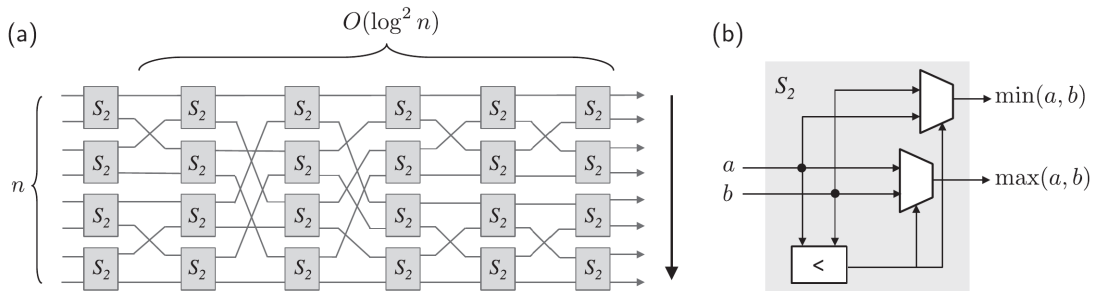
Имплементирањем дизајна са пропусном ширином w мањом од n (број речи који се сортира), цена и У/И пропусна способност система се смањују.

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* (*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 су 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 опеаратора 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

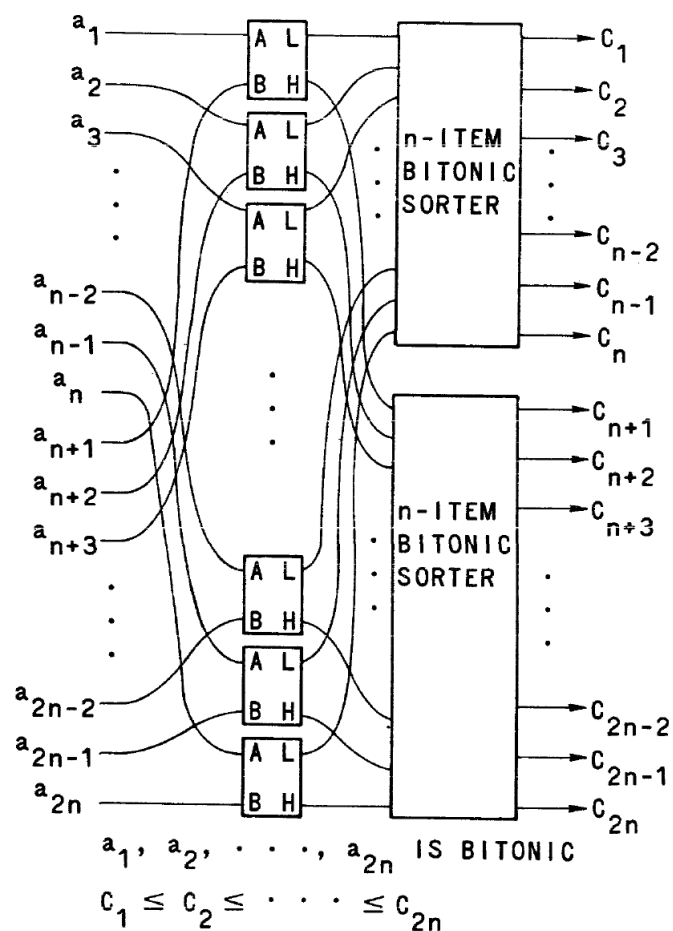
Низ бројева може да се назове битоник ако се састоји од два монотона низа, једног растућег и другог опадајућег. Такође, низ остаје битоник ако је подељен било где и два дела замењена.

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

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

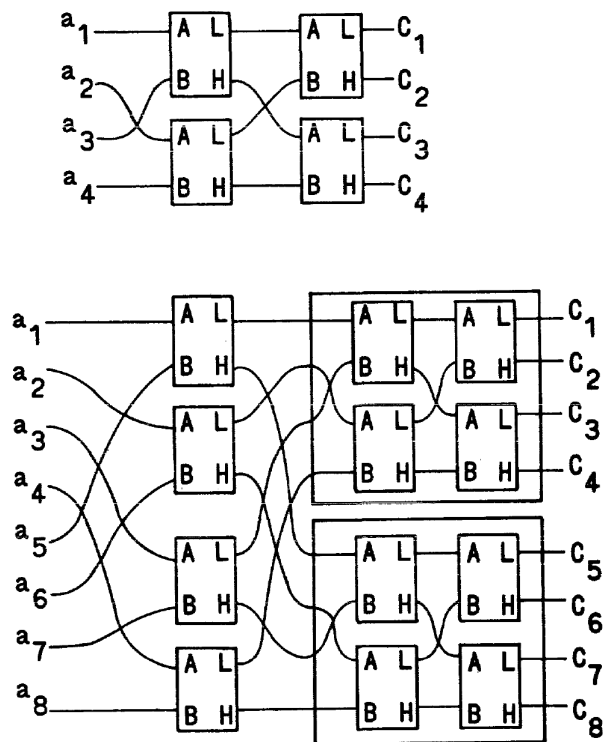
$$\max(a_1, a_{n+1}), \max(a_2, a_{n+2}), \dots, \max(a_n, a_{2n}) \quad (2)$$

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 кон од n елемената за поређење и два битоник сортера за n бројева. Елементи за операцију поређења формирају (1) и (2) и пошто су обе битоник потом су сортиране 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].



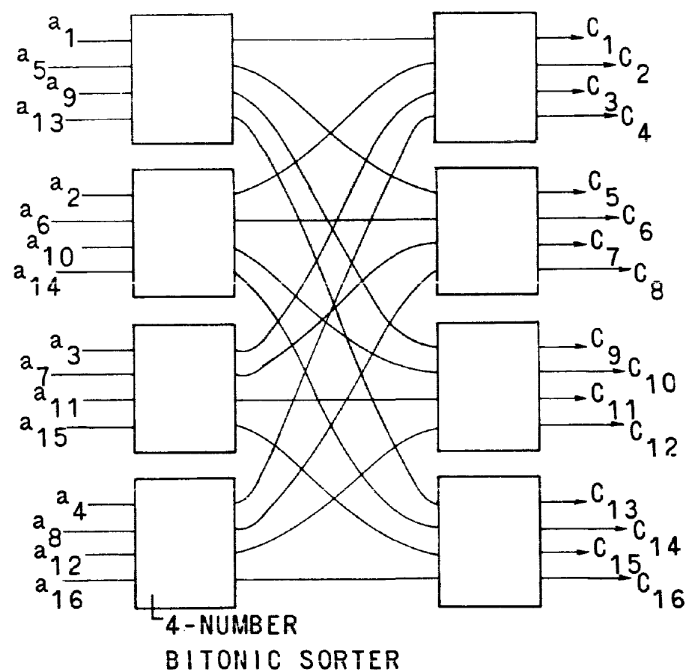
Слика 3.1.2: Итеративно правило за битоник сортере

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 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

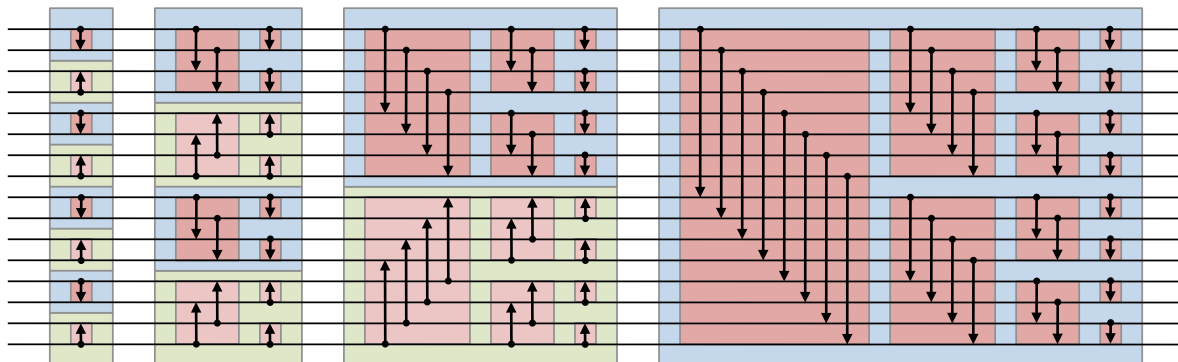
Велики битоник сортери могу бити конструисани из више мањих битоник сортера. На слици 3.1.4 је приказан битоник сортер за 16 бројева конструисаног од осам битоник сортера за 4 броја.



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

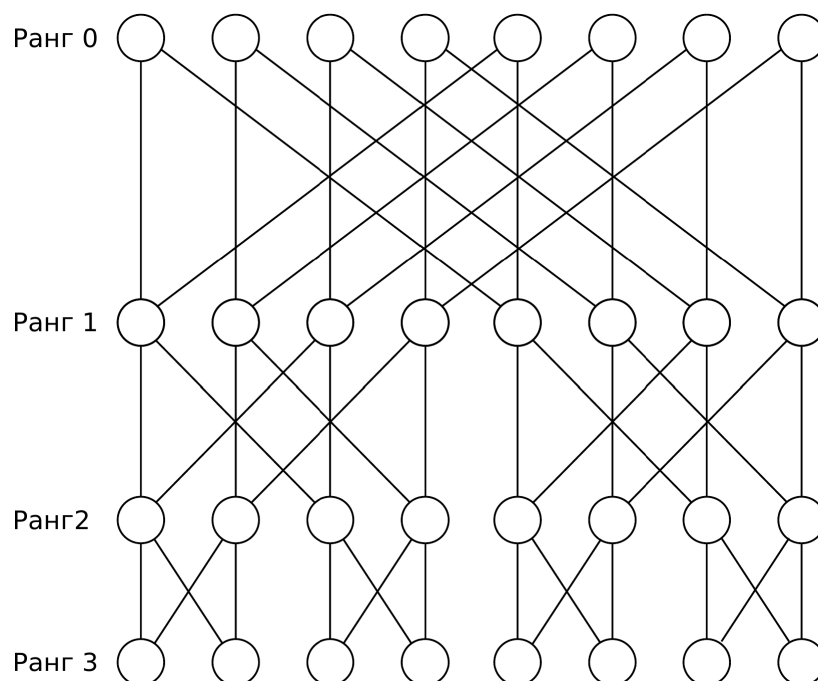
Следећа слика боље приказује како ради алгоритам. 16 бројева се налази на улазу на левој страни и прелази преко 16 хоризонталних жица до излаза на десној страни. Мрежа сортира елементе, тако да се највећи налази на дну.

Стрелице представљају операције поређења и показују ка већем броју. Уокверни 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 посматрају 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 на доле (плави) ил нагоре (зелени). Ова структура је позната као лептир (butterfly) 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 случају 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 парове 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].

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

Прилог садржи програмски код итеративне имплементације битоник сортера у Чизелу [10]. Овај 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.

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

```

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))
}

```

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)
}

```

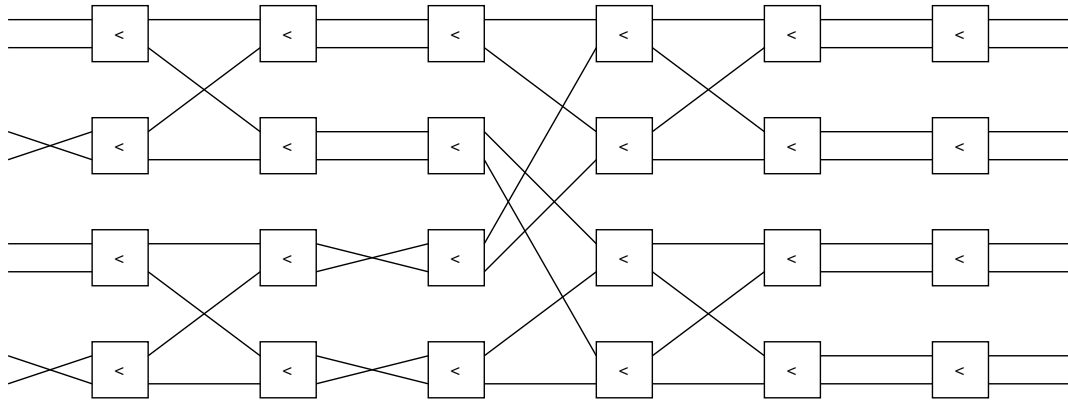

4.1.1 Анализа рада итеративне имплементације

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

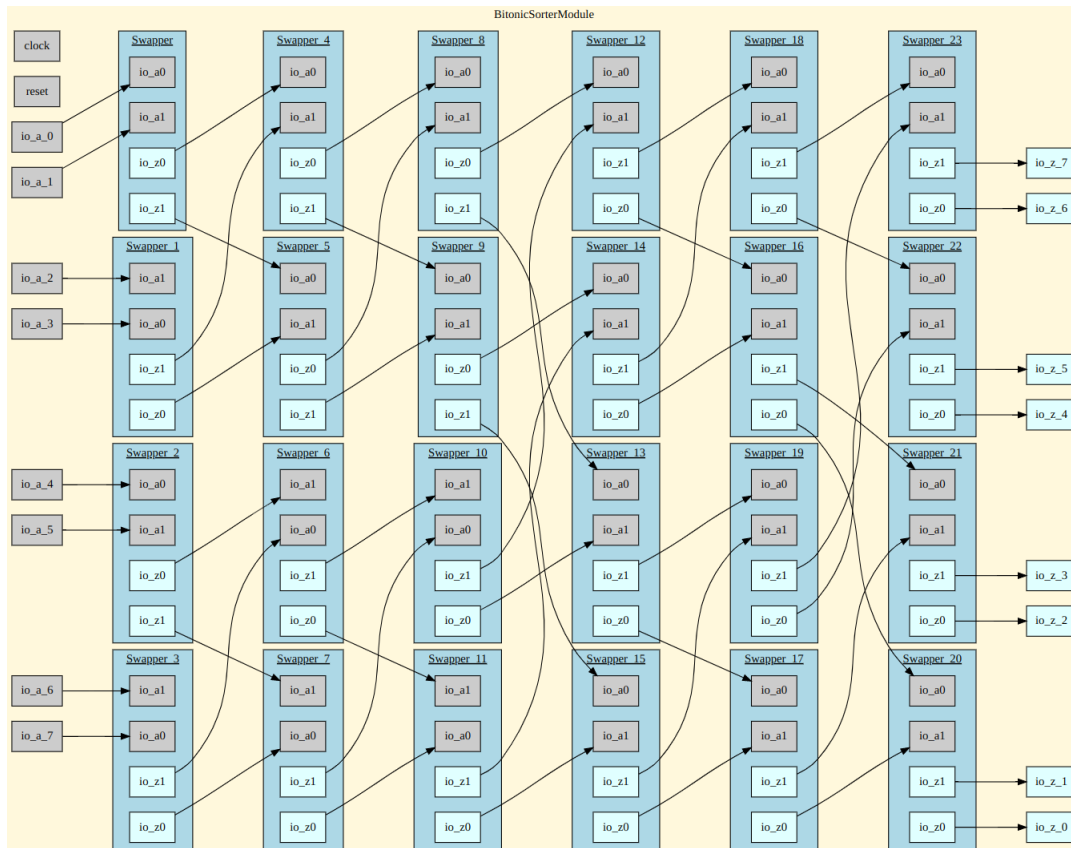
Ако бисмо посматрали пример да на улазу имамо низ од осам бројева, и ако бисмо генерисали парове индекса бројева унутар тог низа које је потребно поредити у сваком нивоу (табела 1), битоник мрежа за сортирање би изгледала као на слици 4.1.1.



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.

4.2 Рекурзивна имплементација у Пајтону

Прилог садржи програмски код рекурзивне имплементације битоник сортера у Пајтону [11].

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).
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.

5 Рекурзивна имплементација битоник сортера у Чизелу - резултат рада

Прилог садржи програмски код рекурзивне имплементације битоник сортера у Чизелу која је изложена у овом раду. 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 размивање начина рада итеративне имплементације где се прави посебна инстанца модула за замену два елемента `Swapper` и излаз додељује новом низу, који је повратна вредност функције `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()`.

```
if (length > 1) {
  val half = length / 2
  val indices = 0 until half
  val swappedSeq = indices.foldLeft(arr) { (seq, i) =>
    swapIfNecessary(seq, i, ascending, lt)
  }
```

Сада се долази до проблема саме замене елемената и на овом примеру се може видети како су Скала и Чизел сједињени. Наиме, функција `swapIfNecessary()` мора имати повратну вредност и не може се враћати унутар Чизел `when` блока, а потребно је користити `when` зато што вршимо замену на основу Чизел `Bool` вредности, и њему дефинисати вредност неког сигнала, притом Скала `if` услов се може само користи уз Скала `Boolean`.

```
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))
```

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.
 - Функција `bitonicMerge()` прима низ елемената `arr`, дужину низа `length`, информацију о редоследу сортирања `ascending`, 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, низ се дели на две половине и за сваки пар елемената `(i, i+length/2)` у низу се врши поређење и ако је потребно замењује се кроз модул `Passthrough`. Након што се обаве сви кораци, сортиран низ се враћа као излаз функције.
 - 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 редоследу; а потом се две половине спајају користећи `bitonicMerge()`. Након што се обаве сви кораци, сортиран низ се враћа као излаз функције.
 - Функција `sort()` проверава да ли је дужина улазног низа степен броја 2, и ако јесте, позива функцију `bitonicSort()` за извршење сортирања.

```
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
  }
}
```

3. Класа `SorterModuleIfc` дефинише интерфејс за модул битоник сортера. Као параметре прима број елемената за сортирање `n` и тип податка елемената `proto`.

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

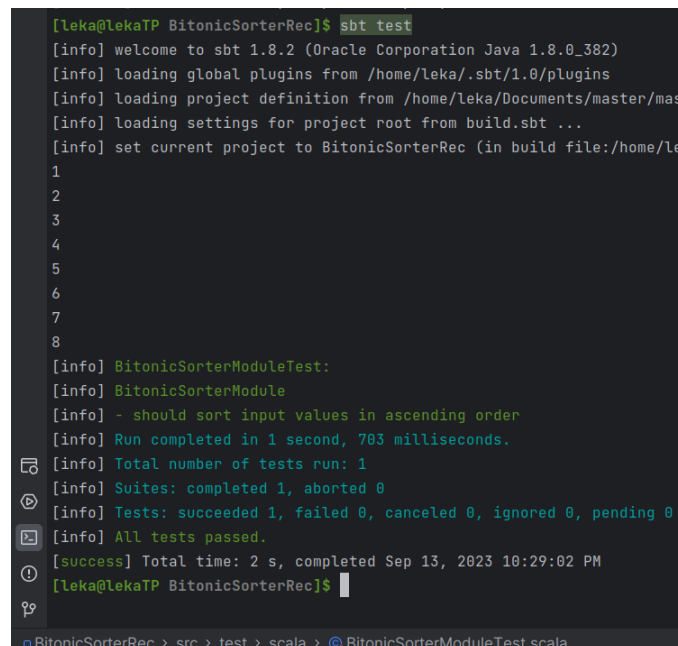
- Као улазне параметре прихвата број елемената за сортирање `n`, тип податка елемената `proto`, флег за редослед сортирања `ascending` и функцију за упоређивање `lt`.
- Креира интерну секвенцу `a` која се користи за сортирање. Елементи ове секвенце се изводе из улазних података `io.a`, обезбеђујући да дужина секвенце буде степен броја 2.
- Сортиран излаз се израчунава помоћу функције `BitonicSorter.sort()` и додељује се излазном порту `io.z`.

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

5. 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

Прилог садржи програмски код за тестирање описане рекурзивне имплементације битоник сортера. Овај Чизел тест проверава `BitonicSorterModule` путем давања улазних вредности, померајући такт и обезбеђујући да излазне вредности буду сортиране у растућем редоследу. За тестирање се користи `ChiselTest` и `ScalaTest`.



```
[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/mas  
[info] loading settings for project root from build.sbt ...  
[info] set current project to BitonicSorterRec (in build file:/home/leka/  
1  
2  
3  
4  
5  
6  
7  
8  
[info] BitonicSorterModuleTest:  
[info] BitonicSorterModule  
[info] - should sort input values in ascending order  
[info] Run completed in 1 second, 703 milliseconds.  
[info] Total number of tests run: 1  
[info] Suites: completed 1, aborted 0  
[info] Tests: succeeded 1, failed 0, canceled 0, ignored 0, pending 0  
[info] All tests passed.  
[success] Total time: 2 s, completed Sep 13, 2023 10:29:02 PM  
[leka@lekaTP BitonicSorterRec]$
```

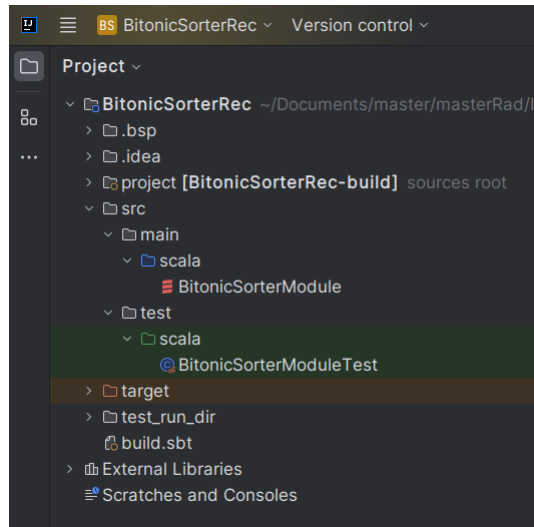
Слика 5.2.1: Извршавање теста

5.3 Покретање програма

У GitHub репозиторијуму *BitonicSorterRecursive*, наведеном у литератури [12], се налазе:

- Пројекат *BitonicSorterRec*, који се може отворити у IntelliJ-у
- L^AT_EX код овог рада и pdf датотека

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

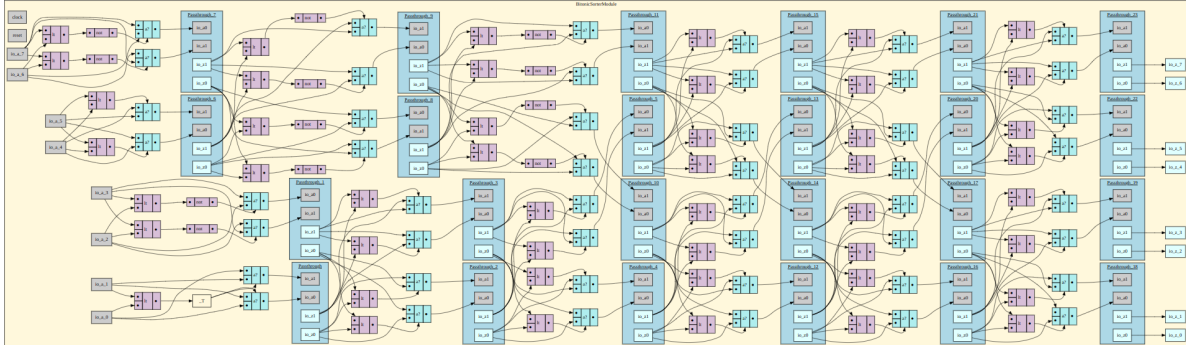


Слика 5.3.2: Структура пројекта

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



Слика 5.4.3: Приказ реурзивне 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.

6 Закључак

Током израде је било потребно разумети начин рада постојећих решења да би се превазишле одређене разлике и ограничења у програмирању коришћењем програмских језика и језика за опис хардвера. Чизел свакако олакшава превазилажење ових препрека, и донекле дозвољава апстрактније размишљање, јер се ипак овај претвара у Верилог, па се може рећи да је језик вишег нивоа.

Визуелно су приказана генерисана кола и може се видети да рекурзивно решење изнето у овом раду захтева више елемената у односу на итеративно. Оба решења имају врло сажет програмски код.

Литература

- [1] *Scala (programming language)*, приступљено (август 2023.) на [https://en.wikipedia.org/wiki/Scala_\(programming_language\)](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_\(programming_language\)](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] Steve Burns, *Bitonic Sorter*, приступљено (септембар 2023.) на <https://github.com/freechipsproject/ip-contributions/tree/master/src/main/scala/chisel/lib/bitonicsorter>
- [11] *Bitonic Sort*, приступљено (септембар 2023.) на <https://www.geeksforgeeks.org/bitonic-sort/>
- [12] *BitonicSorterRecursive*, приступљено (септембар 2023.) на <https://github.com/aleksavelickovic5762015/BitonicSorterRecursive.git>

Списак скраћеница

ФВР Функције вишег реда

Списак табела

1	Парови индекса бројева који се упоређују	15
---	--	----

Списак слика

2.1.1 Компаније које користе Скалу	2
2.1.2 Окружења која користе Скалу	2
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	7
3.1.2 Итеративно правило за битоник сортере	9
3.1.3 Конструкција битоник сортера за четири и осам бројева	10
3.1.4 Битоник сортер за 16 бројева конструисаног од осам битоник сортера за 4 броја	10
3.1.5 Приказ рада битоник алгоритма са 16 бројева на улазу	11
3.1.6 Топологија лептира	11
4.1.1 Битоник мрежа за сортирање - итеративна имплементација	15
4.1.2 Визуализација итеративне имплементације	16
5.2.1 Извршавање теста	20
5.3.2 Структура пројекта	21
5.4.3 Приказ реурзивне имплементације	22

Прилог

A. Итеративна имплементација - Чизел

У наставку је приказана итеративна имплементација битоник сортера [10].

```
// See README.md for license details.

package chisel3.lib.bitonicsorter

import chisel3._
import chisel3.stage.ChiselStage
import chisel3.util._

/**
 * Builds a mux whose that tests its two inputs
 * Construct hardware to swap inputs to outputs or pass them straight through
 * depending on comparison function
 * otherwise
 * @param proto The type generator for the swapper
 * @param lt The comparison function
 * @tparam T The type of the inputs and outputs, derived from proto
 */
class Swapper[T <: Data](proto: T, lt: (T, T) => Bool) extends Module {
  val io = IO(new Bundle {
    val a0 = Input(proto)
    val a1 = Input(proto)
    val z0 = Output(proto)
    val z1 = Output(proto)
  })
  when(lt(io.a1, io.a0)) {
    io.z0 := io.a1
    io.z1 := io.a0
  }.otherwise {
    io.z0 := io.a0
    io.z1 := io.a1
  }
}

/**
 * Bitonic Sorter is a factory for combinational sort hardware.
 * It requires that the elements to be sorted are a power of 2.
 *
 * @see https://en.wikipedia.org/wiki/Bitonic\_sorter
 */
object BitonicSorter {
  def apply[T <: Data](a: IndexedSeq[Option[T]], factory: () => Swapper[T]):
    IndexedSeq[Option[T]] = {
    assert((1 << log2Up(a.length)) == a.length)
  }
```

```

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))
    }
    }
    (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) }
    }
}

/**
 * Defines the interface used for the Bitonic Sort module
 * @param n      The number of elements
 * @param proto  An instance of the type of all elements
 * @tparam T     The type as derived from proto
 */
class SorterModuleIfc[T <: Data](val n: Int, proto: T) extends Module {
    val io = IO(new Bundle {
        val a = Input(Vec(n, proto.cloneType))
        val z = Output(Vec(n, proto.cloneType))
    })
}

/*
 * What follows are some example code generation calls.
 */

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))
}

```

```

}

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

//scalastyle:off magic.number
object BitonicSorterUInt8_384Driver extends App {
  (new ChiselStage).emitSystemVerilog(new BitonicSorterModule(384, UInt(8.W),
    (x: UInt, y: UInt) => x < y), args)
}

```

Б. Рекурзивна имплементација - Пајтон

У наставку је приказана рекурзивна имплементација битоник сортера у Пајтону [11].

```

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]

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)

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)

def sort(a,N, up):
    bitonicSort(a,0, N, up)

a = [3, 7, 4, 8, 6, 2, 1, 5]
n = len(a)
up = 1

sort(a, n, up)

```

V. Рекурзивна имплементација у Чизелу - резултат рада

У наставку је приказана рекурзивна имплементација битоник сортера у Чизелу која је резултат овог рада.

```
// See README.md for license details.

import chisel3._
import chisel3.stage.ChiselStage
import chisel3.util._

/**
 * Simple Passthrough module
 * @param proto The type generator for the swapper
 * @tparam T The type of the inputs and outputs, derived from proto
 */
class Passthrough[T <: Data](proto: T) extends Module {
  val io = IO(new Bundle {
    val a0 = Input(proto)
    val a1 = Input(proto)
    val z0 = Output(proto)
    val z1 = Output(proto)
  })

  io.z0 := io.a0
  io.z1 := io.a1
}

/**
 * Bitonic Sorter is a factory for combinational sort hardware.
 * It requires that the elements to be sorted are a power of 2.
 *
 * @see https://en.wikipedia.org/wiki/Bitonic\_sorter
 */
object BitonicSorter {
  def bitonicMerge[T <: Data](arr: IndexedSeq[Option[T]], length: Int,
    ascending: Bool, lt: (T, T) => Bool): IndexedSeq[Option[T]] = {
    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))
}

if (length > 1) {
    val half = length / 2
    val indices = 0 until half

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

    val left = bitonicMerge(swappedSeq.slice(0, half),
                            half, ascending, lt)
    val right = bitonicMerge(swappedSeq.slice(half, length),
                              half, ascending, lt)

    left ++ right
}
else {
    arr
}
}

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
    }
}

def sort[T <: Data](a: IndexedSeq[Option[T]], ascending: Bool = true.B,
                    lt: (T, T) => Bool): IndexedSeq[Option[T]] = {
    if (a.length <= 1) {
        // Nothing to sort, return the input array
        a
    }
    else {
        assert((1 << log2Up(a.length)) == a.length,
               "Array length must be a power of 2")
    }
}

```

```

        val sortedArray = bitonicSort(a, ascending, lt)
        sortedArray
    }
}

/**
 * Defines the interface used for the Bitonic Sort module
 * @param n      The number of elements
 * @param proto  An instance of the type of all elements
 * @tparam T     The type as derived from proto
 */
class SorterModuleIfc[T <: Data](val n: Int, proto: T) extends Module {
    val io = IO(new Bundle {
        val a = Input(Vec(n, proto.cloneType))
        val z = Output(Vec(n, proto.cloneType))
    })
}

/*
 * What follows are some example code generation calls.
 */

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

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

object BitonicSorterUInt8_384Driver extends App {
    (new ChiselStage).emitSystemVerilog(new BitonicSorterModule(384, UInt(8.W),
                                                                true.B, (x: UInt, y: UInt) => x < y), args)
}

```

Г. Програмски код за тестирање битоник сортера

У наставку је приказана програм за тестирање описане рекурзивне имплементације битоник сортера.

```
import chisel3._
import chiseltest._
import org.scalatest.flatspec.AnyFlatSpec

class BitonicSorterModuleTest extends AnyFlatSpec
    with ChiselScalatestTester {
  "BitonicSorterModule" should "sort input values in ascending order" in {
    test(new BitonicSorterModule(8, UInt(8.W), true.B,
      (x: UInt, y: UInt) => x < y)) { dut =>
      val input = Seq(5, 8, 3, 1, 6, 2, 7, 4).map(_.U)

      dut.io.a.zipWithIndex.foreach { case (inputPort, index) =>
        inputPort.poke(input(index))
      }

      dut.clock.step()

      // Check if the output values are in ascending order
      var prevValue = BigInt(0)
      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 ascending order.")
        prevValue = value
      }
    }
  }
}
```