PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA

SKRIPSI

untuk memenuhi salah satu persyaratan
mencapai derajat Sarjana S1

Disusun oleh:

MUHAMMAD IRFAN

16524100

Jurusan Teknik Elektro

Fakultas Teknologi Industri

Universitas Islam Indonesia

Yogyakarta

2020

LEMBAR PENGESAHAN

PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA

TUGAS AKHIR

Diajukan sebagai Salah Satu Syarat untuk Memperoleh
Gelar Sarjana Teknik
pada Program Studi Teknik Elektro
Fakultas Teknologi Industri
Universitas Islam Indonesia

Disusun oleh:

Muhammad Irfan
16524100

Yogyakarta, 20-juli-2020

Menyetujui,

Pembimbing

Dr. Eng. Hendra Setiawan, S.T., M.T
025200526

LEMBAR PENGESAHAN

SKRIPSI

PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA

Dipersiapkan dan disusun oleh:

Muhammad Irfan

16524100

Telah dipertahankan di depan dewan penguji

Pada tanggal: tanggal bulan tahun

Susunan dewan penguji

Ketua Penguji : Dr. Eng. Hendra Setiawan, S.T., M.T, _____

Anggota Penguji 1: Sisdarmanto Adinandra, ST, M.Sc., Ph.D, _____

Anggota Penguji 2: Ida Nurcahyani, ST, M.Eng, _____ _____

Skripsi ini telah diterima sebagai salah satu persyaratan untuk memperoleh gelar Sarjana

Tanggal: 24 Agustus 2020

Ketua Program Studi Teknik Elektro

Yusuf Aziz Amrullah,S.T., M.Sc., Ph.D

045240101

PERNYATAAN

Dengan ini Saya menyatakan bahwa:

1. Skripsi ini tidak mengandung karya yang diajukan untuk memperoleh gelar kesarjanaan di suatu Perguruan Tinggi, dan sepanjang pengetahuan Saya juga tidak mengandung karya atau pendapat yang pernah ditulis atau diterbitkan oleh orang lain, kecuali yang secara tertulis diacu dalam naskah ini dan disebutkan dalam daftar pustaka.
2. Informasi dan materi Skripsi yang terkait hak milik, hak intelektual, dan paten merupakan milik bersama antara tiga pihak yaitu penulis, dosen pembimbing, dan Universitas Islam Indonesia. Dalam hal penggunaan informasi dan materi Skripsi terkait paten maka akan diskusikan lebih lanjut untuk mendapatkan persetujuan dari ketiga pihak tersebut diatas.

Yogyakarta, 21 agustus 2020

Muhammad Irfan

KATA PENGANTAR

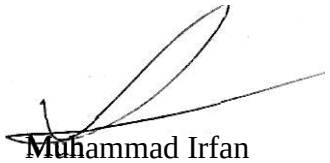
Atas izin Allah Subhanahu wa Ta'ala tugas akhir yang berjudul "PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA" dapat kami selesaikan melalui proses yang cukup panjang, tugas akhir ini disusun untuk memenuhi salah satu persyaratan untuk menyelesaikan program studi teknik elektro di Universitas Islam Indonesia.

Selama mengikuti pendidikan program studi teknik elektro sampai dengan proses penyelesaian tugas akhir, berbagai pihak telah memberikan fasilitas, membantu, membina dan membimbing penulis, untuk itu penulis mengucapkan terimakasih khususnya kepada:

1. Kedua orang tua yang telah memberikan dukungan dan do'a selama penulis menuntut ilmu di UII.
2. Bapak Dr.Eng., Hendra Setiawan, S.T., M.T. selaku dosen teknik elektro dan pembimbing tugas akhir.
3. Bapak Yusuf Aziz Amrullah,S.T., M.Sc., Ph.D selaku Ketua Jurusan Teknik Elektro, Universitas Islam Indonesia
4. Bapak dan Ibu dosen pengajar yang telah memberikan ilmunya kepada penulis.
5. Teman – teman teknik elektro angkatan 2016 dan Keluarga besar teknik elektro UII.

Penulis memohon maaf jika terdapat kesalahan dalam penulisan tugas akhir ini, semoga yang sedikit ini dapat bermanfaat dan dapat memberikan kontribusi dalam bidang keilmuan yang terkait.

Yogyakarta, 21 agustus 2020



Muhammad Irfan

ARTI LAMBANG DAN SINGKATAN

n	: Selisih dari hasil perhitungan kalkulator dengan hasil simulasi
E	: Persentase *Error* dari hasil simulasi dengan hasil kalkulator
Fm	: Frekuensi maksimal pada suhu tertinggi
P	: Hasil perhitungan akar kuadrat menggunakan
Q	: Hasil perhitungan akar kuadrat dari simulasi
Tm	: Delay terbesar atau terburuk antar *register*
dkk	: dan kawan-kawan
kbit	: kilo bit
ns	: *nanosecond*
DSP	: *Digital Signal Processor*
DTC	: *Direct Torque Control*
Fmax	: *Frequency maximum*
FPGA	: *Field programmable Gate Array*
FSM	: *Finite State machine*
HDL	: *Hardware Description Language*
I/O	: *Input/Output*
LE	: *Logic Element*
MHz	: *Megahertz*
PWM	: *Pulse Width Modulation*
VHDL	: *VHSIC Hardware Description Language*
VHSIC	:*Very High Speed Integrated Circuit*

ABSTRAK

Direct Torque Control (DTC) adalah metode kendali telah banyak digunakan dalam penggerak motor. Pada DTC terdapat perhitungan akar kuadrat yang membutuhkan proses perhitungan yang sangat cepat dan beroperasi pada frekuensi yang tinggi. Penerapan DTC pada FPGA diusulkan untuk mengatasi masalah tersebut. Pada penelitian ini, mengimplementasikan perhitungan akar kuadrat bilangan pecahan dengan metode *digit by digit non-restoring modified* pada FPGA. Tujuan dari penelitian ini untuk mengimplementasikan perhitungan akar kuadrat bilangan pecahan dan mengetahui hasil dari *resource* yang didapatkan pada implementasi tersebut. Proses perhitungan akar kuadrat menggunakan metode *digit by digit non-restoring modifed* dengan jumlah bit *input* 32-bit dan *output* 16-bit. Untuk menghemat jumlah *resource* yang digunakan, sistem ini dirancang dengan sederhana menggunakan *Finite State machine* (FSM). Pada FSM terdapat 2 *state* penting pada pengoperasian perhitungan akar kuadrat yaitu *state* penggeseran dan *state* pengoperasian. Proses verifikasi sistem ini dilakukan dalam dua tahap, yaitu verifikasi fungsional dengan aplikasi ModelSim-Altera dan verifikasi *hardware* menggunakan modul FPGA Cylcone IV EP4CE6E228N. Hasil pengujian menunjukkan bahwa *output* perhitungan akar kuadrat memiliki presisi yang tinggi dengan relsolusi 0,00390625. Sistem ini membutuhkan 157 *Logic Elements*, 120 register, 0 *multiplier* 9-bit, dan 151,59 MHz FMax.

Kata kunci: DTC, Akar Kuadrat, *Non-Restoring*, *Delay*, *Logic Element*, *Delay, VHDL*, FPGA, FSM.

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENDAHULUAN

1.1 Latar Belakang Masalah

Direct Torque Control (DTC) adalah metode kendali telah banyak digunakan dalam penggerak motor [1]. Populeritas DTC dikarenakan memiliki strukturnya yang sederhana, di mana tidak memerlukan pengkodean posisi dan pembangkit *Pulse Width Modulation* (PWM), dan keunggulan lainya memiliki respon torsi yang cepat [2]. DTC dapat diimplementasikan dalam mikrokontroler, *Digital Signal Processor* (DSP), *Field Programmable Gate Array* (FPGA) dan perangkat digital lain. Ketika di implementasikan ke mikrokontroler atau DSP kinerja pada DTC akan menurun dikarenakan ketidakmampuan untuk menghitung dengan sangat cepat dan beroperasi pada frekuensi *switching* yang tinggi, sehingga tidak memadai untuk meminimalkan keluaran torsi riak [3]–[5].

Oleh karena itu, penerapan DTC dalam FPGA diusulkan untuk mengatasi masalah ini. FPGA adalah salah satu perangkat umum yang digunakan dalam pembuatan prototipe modul pemrosesan digital, biayanya yang murah dan komputasi yang berkinerja tinggi [6].

Pada DTC dibutuhkannya kalkulator akar kuadrat untuk menghasilkan perkiraan fluks stator [7]. Akar kuadrat adalah operasi aritmatika penting dalam pemrosesan sinyal digital [8]. Tidak seperti operasi lain seperti penjumlahan, pengurangan dan perkalian, perhitungan akar kuadrat sangat sulit direalisasikan dalam FPGA, karena berbagai faktor seperti kompleksitas algoritme, waktu penyelesaian perhitungan, jumlah penggunaan sumber daya *hardware* dan juga konsumsi daya [9]. Ada beberapa algoritme yang dapat digunakan untuk membangun kalkulator akar kuadrat dalam FPGA, seperti *Rough Estimation*, metode *Babylonian*, metode *Taylor-Series expansion*, dan metode *digit by digit* [4]–[6], [10]–[12]. Pada penelitian ini metode yang digunakan adalah metode *digit by digit non restoring modified*, hal ini dianjurkan dari penelitian lain dikarenakan metode tersebut memiliki struktur yang sederhana sehingga menghasilkan komputasi yang lebih cepat dan mengkonsumsi sumber daya *hardware* lebih rendah [6].

Algoritme akar kuadrat tidak mudah diimplementasikan pada FPGA. Sudah ada beberapa penelitian yang berusaha mengimplementasikan algoritme akar kuadrat, seperti yang dilakukan penelitian [4]–[6], [13], [14]. Penelitian tersebut melakukan modifikasi algoritme perhitungan akar kuadrat menjadi lebih sederhana dan membandingkan dengan penelitian yang lain. Pada penelitian Tole Sutikno dan AZ Jidin dkk menggunakan metode *digit by digit non restoring modified* dan melakukan perbandingan jumlah *Logic Element* (LE) [4]–[6]. Sedangkan pada penelitian Samavi dkk dan AP Ramesh dkk melakukan perbandingan *delay* yang diperoleh [13], [14]. Namun dari

semua penelitian tersebut perhitungan akar kuadrat yang dilakukan hanya memberi nilai bilangan bulat sehingga mendapatkan hasil yang kurang presisi. Hasil dari perhitungan akar kuadrat bilangan bulat yang mengabaikan nilai pecahan dapat dilihat pada Tabel 1.1 sebagai berikut.

Tabel 1.1 Perhitungan Akar kuadrat Yang Mengabaikan Nilai Pecahan

A	$\sqrt{A}$	Hasil Perhitungan kalkulator	Selisih A dan Hasil Perhitungan kalkulator	*Persentase Error*
1	1	1	0	0%
2	1	1,41421356	0,41421356	29,28%
3	1	1,73205081	0,73205081	42,26%
4	2	2	0	0%
5	2	2,23606798	0,23606798	10,5%
6	2	2,44948974	0,44948974	18.35%
7	2	2,64575131	0,64575131	24,4%
8	2	2,82842712	0,82842712	29.28%

Tabel 1.1 adalah hasil perhitungan akar kuadrat yang mengabaikan bilangan pecahan. Nilai A adalah nilai yang dilakukan operasi akar kuadrat dan hasil dari akar kuadrat A mengabaikan nilai pecahannya, sehingga akar kuadrat dari 1, 2 dan 3 adalah 1 dan akar kuadrat dari 4, 5, 6, 7 dan 8 adalah 2 maka tidak ada perbedaan antara akar kuadrat dari 1, 2 dan 3 dan juga akar kuadrat dari 4, 5, 6, 7 dan 8. Hal ini menimbulkan persentase *Error* yang besar, persentase *Error* pada Tabel 1.1 yang dimaksud adalah persentase *Error* dari hasil perhitungan $\sqrt{A}$ dengan hasil nilai dari kalkulator. Pada akar kuadrat dari 3 memiliki persentase *Error* terbesar yaitu 42,26%.

Untuk mengurangi *Error* yang besar atau mendapatkan hasil presisi yang tinggi, perhitungan akar kuadrat akan melibatkan nilai pecahan. Pada penelitian ini akan dilakukan modifikasi algoritme perhitungan akar kuadrat yang melibatkan nilai pecahannya dengan algoritme yang sederhana. Sehingga perhitungan akar kuadrat yang terdapat di algoritme DTC akan menghasilkan respons kecepatan yang tinggi dan memiliki nilai yang lebih presisi.

1.2 Rumusan Masalah

Dari uraian di atas dapat dibuat rumusan masalah sebagai berikut:

1. Bagaimana mengimplementasikan perhitungan akar kuadrat bilangan pecahan pada FPGA?
2. Bagaimana unjuk kerja atau hasil pengujian perhitungan akar kuadrat bilangan pecahan pada FPGA?

1.3 Batasan Masalah

1. Algoritme *modified digit by digit non-restoring* untuk perhitungan akar kuadrat.
2. *Resource* yang terlibat dalam perancangan adalah *Logic Element, Register, Total Pins* dan Frekuensi tertinggi.

1.4 Tujuan Penelitian

1. Mengimplementasi perhitungan akar kuadrat bilangan pecahan pada FPGA.
2. Mendapatkan unjuk kerja atau hasil perhitungan akar kuadrat bilangan pecahan pada FPGA.

1.5 Manfaat Penelitian

1. Meningkatkan akurasi perhitungan akar kuadrat bilangan pecahan di perangkat FPGA.
2. Menambah khazanah keilmuan berkaitan dengan implementasi komputasi matematis pada FPGA.

BAB 2
TINJAUAN PUSTAKA

2.1 Studi Literatur

Perhitungan akar kuadrat menggunakan FPGA telah banyak dilakukan sebelumnya, seperti yang dilakukan oleh S. Samavi dkk [13]. Karena desain sirkuit terdahulu memiliki proses yang panjang dan banyak, sehingga S. Samavi dkk mencari proses desain yang sederhana secara signifikan. Pada *input bits* 4, 8 dan 16, S. Samavi dkk mendapatkan *delay* terburuk sebesar 6,485 ns, 10,023 ns dan 32.016 ns [13]. Hal ini merupakan hasil yang baik, karena desain sebelumnya memiliki hasil *delay* lebih besar dari pada desain S. Samavi dkk [13]. Mereka telah menyederhanakan penulisan algoritme *digit by digit non-restoring* sehingga mengurangi *delay* terburuk.

Pada tahun 2010, Tole Sutikno melakukan perbandingan metode antara metode *modified non-restoring digit by digit* dan metode operasi perhitungan akar kuadrat yang lain [5]. Penelitian tersebut mengimplementasi algoritme *non-restoring* yang dimodifikasi dengan 32 bit dan 64 bit menggunakan Altera APEX 20KE FPGA yang membutuhkan 256 *Logic Element* (LE) dan 1023 LE [5]. Algoritme *non-restoring* yang dioptimalkan mengurangi area *chip* dan *pipelining* sehingga meningkatkan kinerja kecepatan. Algoritme *restoring* menyimpan sisanya di setiap interasi maka membutuhkan LE lebih banyak dibandingkan dengan algoritme *non-restoring* yang dimodifikasi[15].

Pada tahun 2015, Addanki Purna Ramesh dan I. Jayaram Kumar melakukan penelitian dengan menerapkan akar kuadrat bilangan bulat dengan menggunakan metode *Square and Compare*, s*uccessive subtraction of odd integer"s method* dan modifikasi metode *non-restoring* yang diimplementasikan menggunakan bahasa program *Very High Speed Integrated Circuit (VHSIC) Hardware Description Language* (VHDL) dan biosintesis dengan menggunakan Xilinx12.1 [14]. Dengan *input* 16 bit mendapatkan hasil *delay* dari metode *Square and Compare* 82,941 ns, s*uccessive subtraction of odd integer"s method* 59,816 ns, dan modifikasi metode *non-restoring* 37,166 ns [14]. Hasil dari penelitian tersebut menunjukkan bahwa metode *non-restoring* yang dimodifikasi memiliki *delay* yang lebih sedikit [14].

Penelitian S. Samavi dkk dan AP Ramesh merupakan perbandingan metode *non-restoring* dengan metode lain, dan didapatkan hasil bahwa metode *non-restoring* memiliki *delay* yang lebih sedikit dibandingkan dengan metode lain [13], [14]. Pada penelitian-penelitian tersebut melakukan implementasi algoritme yang hanya berlaku pada bilangan bulat saja. Namun pada penelitian di atas tidak memecahkan perhitungan akar kuadrat bilangan desimal. Dengan adanya perhitungan

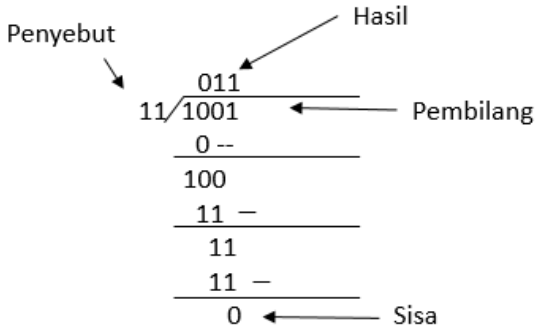
akar kuadrat desimal maka mendapatkan hasil yang lebih presisi. Penelitian ini akan memodifikasi metode sebelumnya untuk mendapatkan perhitungan akar kuadrat bilangan desimal agar mendapatkan hasil yang presisi dan efisien.

2.2 Tinjauan Teori

Perhitungan akar kuadrat bilangan biner memiliki berbagai macam metode. Yaitu metode *Rough Estimation*, metode *Babylonian*, algoritme ekspansi *Taylor-Series*, metode *Newton-Raphson*, dan algoritme berurutan (metode perhitungan *digit by digit*) [4]. Pada penelitian ini memakai metode *digit by digit* dengan algoritme *non-restoring* yang telah dimodifikasi. Konsep dasar dari metode ini seperti konsep pembagian bilangan biner, dimana dilakukannya operasi perhitungan *digit by digit* secara berurutan.

2.2.1 Pembagian dalam Bilangan Biner

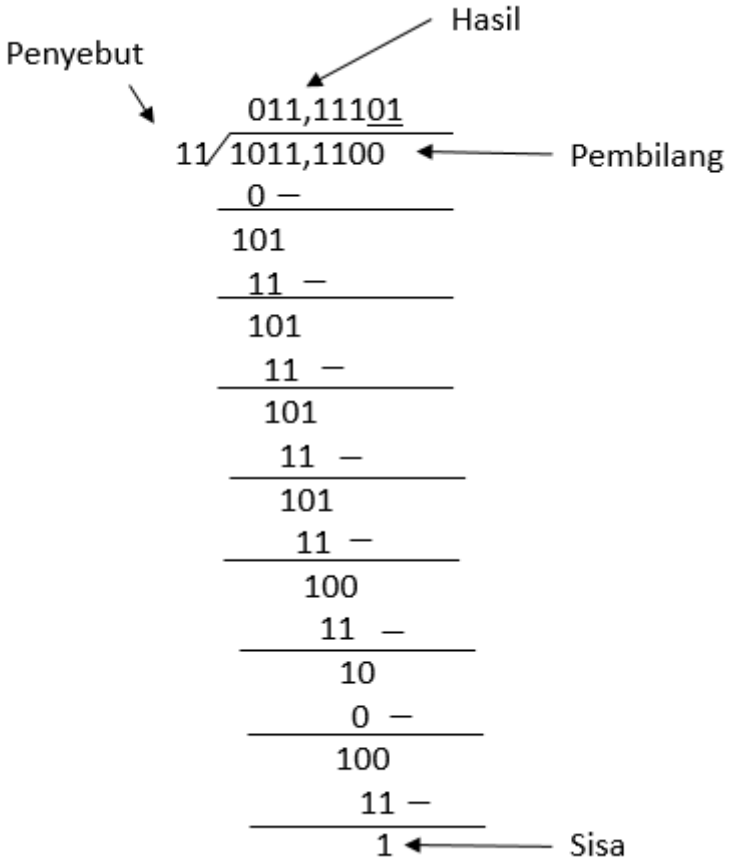
Pembagian bilangan biner pada dasarnya sama dengan pembagian bilangan desimal, yang membedakan ialah pada biner hanya mempunyai dua angka yaitu 0 dan 1. Cara pembagian bilangan biner adalah mengurangi biner yang mau dibagi mulai dari sebelah kiri dengan biner pembagi. Jika biner tersebut bisa dikurangi maka bernilai 1 pada hasil dan jika tidak bisa di kurangi maka bernilai 0. Pada Gambar 2.1 adalah contoh pembagian bilangan biner:



Gambar 2.1 Contoh operasi perhitungan pembagian biner

Gambar 2.1 adalah contoh pembagian dalam bilangan biner 1001 : 11 = 11 atau nilai desimalnya menjadi 9 : 3 = 3. Pada prinsipnya pengambilan biner setiap satu bit setiap operasi. Ketika pembilang lebih besar atau sama dengan dari penyebut maka dilakukan operasi pengurangan dan hasil pembagian bernilai 1. Pada sisa operasi ditambahkan bit berikutnya dan menjadi pembilang pada operasi selanjutnya. Jika pembilang lebih kecil dari pada penyebut maka tidak dilakukan operasi pengurangan dan hasil bit selanjutnya akan bernilai 0. Pembilang yang

tidak dilakukan pengurangan akan ditambah dengan bit selanjutnya dan menjadi pembilang pada operasi selanjutnya. Hal ini dilakukan hingga bit berikutnya tidak ada lagi.

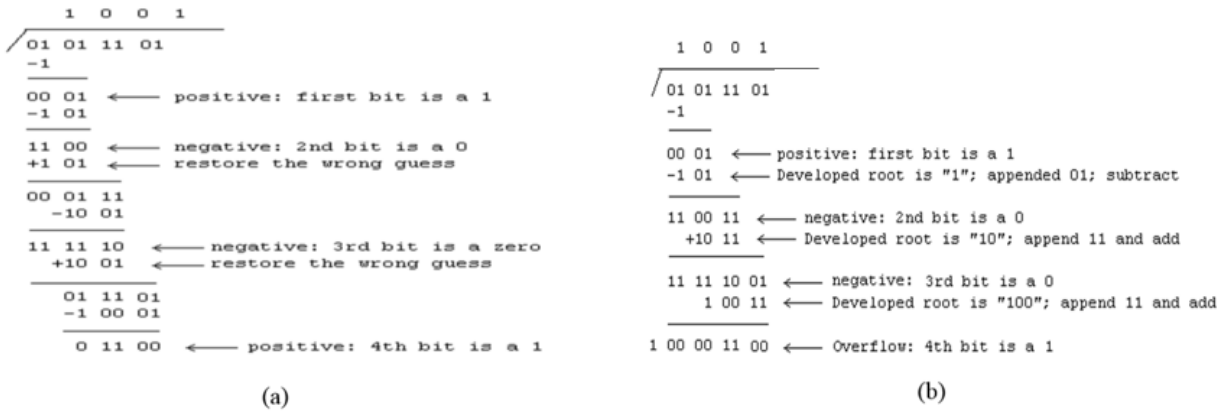


Gambar 2.2 Contoh operasi perhitungan pembagian biner dengan pecahan

Gambar 2.2 adalah operasi perhitungan pembagian biner pecahan. Pada prinsipnya sama dengan pembagian biner bilangan bulat hanya saja perhitungan ini tidak mengabaikan sisa pembagian sehingga menghasilkan nilai pecahan. Ketika nilai di sebelah kiri sudah tidak ada maka di tambakan nilai 0 supaya bisa dikurangi. Ketika nilai hasil pengurangan sama dengan yang sebelumnya maka bisa berhenti perhitungan karena pembagian akan berulang–ulang dengan nilai yang sama. Seperti gambar 2.2 mengoperasikan pembagian nilai biner 1011,11 : 11 yang hasilnya 11,11$\underline{10}$ atau 11,111010101010101... dan seterusnya, Karena melakukan pengulangan terus-menerus maka diberi tanda garis atas atau bawah untuk pengulangan.

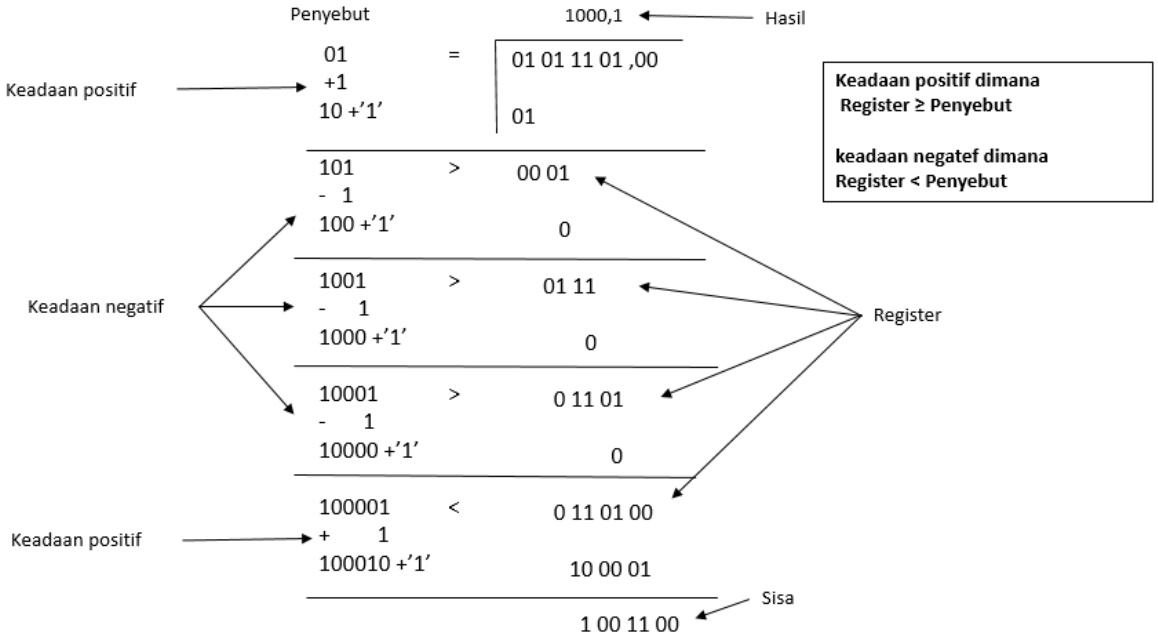
2.2.2 Perhitungan Akar Kuadrat dalam Bilangan Biner

Metode perhitungan akar kuadrat bilangan biner yang digunakan adalah metode *digit by digit*, metode ini memiliki 2 algoritme yang berbeda yaitu *restoring* dan *non-restoring*. Gambar 2.3 berikut ini adalah langkah-langkah metode *restoring* dan *non-restoring*.



Gambar 2.3 Contoh metode *digit by digit* perhitungan akar kuadrat (a) *algoritme restoring* (b) algoritme *non-restoring* [4]

Sedangkan yang digunakan peneliti ini adalah metode *modified digit by digit non-restoring*, metode ini memodifikasi algoritme *non-restoring*. Gambar 2.4 berikut ini adalah langkah-langkah metode *modified digit by digit non-restoring*.



Gambar 2.4 Contoh metode *digit by digit* perhitungan akar kuadrat algoritme *modified non-restoring*

Gambar 2.4 adalah contoh dari perhitungan akar kuadrat biner menggunakan algoritme *modified non-restoring*. Prinsip utama metode ini adalah pembilang akan dikelompokkan setiap 2 bit dan di kurang dengan penyebutnya. penyebut tersebut diawali dengan bit 01, pembilang dan penyebut akan berubah seiring keadaan sebelumnya. Penyebut akan menambahkan bit 1 kanan *Least Significant Bit*(LSB) setiap beroperasi. Ketika penyebut lebih kecil atau sama dengan pembilang maka akan dilakukan operasi, pembilang akan dikurang penyebut dan hasil akan bernilai 1, keadaan ini disebut keadaan positif. Penyebut akan ditambah dengan 01 kemudian

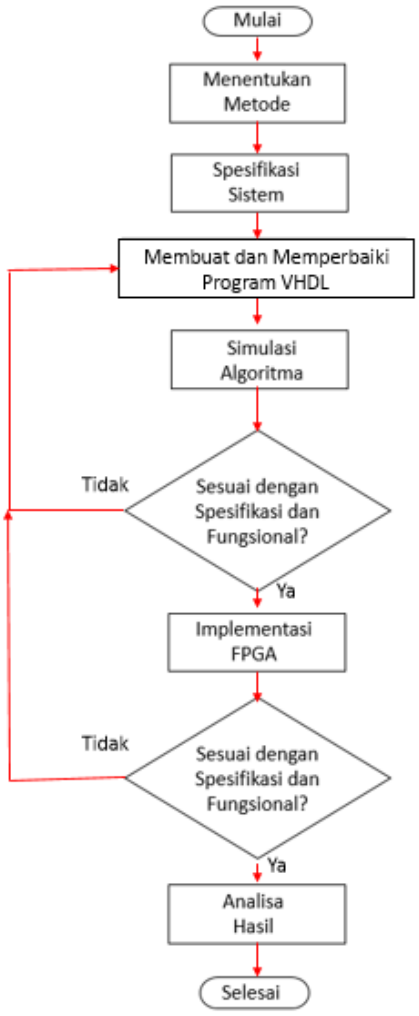
menambahkan bit bernilai 1 kanan LSB dan pembilang akan menambahkan 2 bit yang sudah di kelompok pada sebelah kanan LSB sisa dari pengurangan sebelumnya untuk operasi berikutnya. Jika pembilang lebih kecil dari pada penyebut maka tidak dilakukan operasi pengurangan dan hasil bernilai 0, keadaan ini disebut keadaan negatif. Penyebut akan di kurang dengan 01 kemudian menambahkan bit bernilai 1 kanan LSB dan pembilang akan menambahkan 2 bit yang sudah di kelompok pada sebelah kanan LSB penyebut sebelumnya untuk operasi berikutnya. Dari gambar di atas memberikan contoh perhitungan akar dari 93 atau nilai biner 01011101, sehingga di dapatkan akar dari 93 yaitu 9,64 atau nilai biner 1001,10

Proses pada Gambar 2.4 yaitu algoritme *modified non-restoring* lebih sederhana dibandingkan dengan Gambar 2.3 algoritme *restoring* dan *non-restoring*. Algoritme *restoring* dan *non-restoring* memiliki operasi pengurangan dan pertambahan sedangkan algoritme *modified non-restoring* hanya operasi pengurangan. Sehingga algoritme *modified non-restoring* lebih sedikit operasi perhitungan dibandingkan algoritme *restoring* dan *non-restoring*

BAB 3
METODOLOGI

3.1 Alur Penelitian

Proses penelitian yang dilakukan dijelaskan pada Gambar 3.1 yang merupakan diagram alir alur penelitian ini :



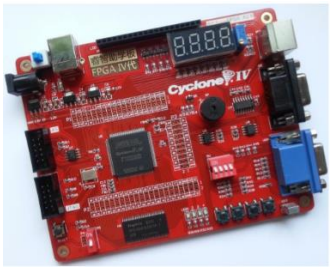
Gambar 3.1 Diagram Alir Penelitian

Gambar 3.1 adalah langkah-langkah penelitian dalam bentuk diagram alir. Penelitian ini dimulai dari menentukan spesifikasi sistem hingga analisa hasil. Pada proses spesifikasi yang bertujuan untuk menentukan target yang dicapai dalam penelitian ini. Selanjutnya membuat atau memodifikasi algoritme pemrograman VHDL dan dilakukan simulasi. Kemudian mengimplementasikan ke FPGA dan memperbaiki kesalahan pada implementasi FPGA. Setelah berhasil mengimplementasi ke FPGA maka dilakukan penyesuaian dengan spesifikasi, setelah sesuai kemudian dilakukan analisa hasil penelitian.

3.2 Spesifikasi Sistem

Sistem ini dirancang dengan spesifikasi sebagai berikut:

1. Komputasi algoritme menggunakan metode *modified digit by digit non-restoring*.
2. Target dari sistem ini adalah mendapatkan hasil perhitungan kuadrat bilangan bulat beserta pecahannya.
3. Pada sistem ini peneliti akan memakai jumlah bit *input* sebanyak 32 bit dengan resolusi biner sebesar 0,0000000000000001, jika dikonversikan desimal menjadi 0,0000152587890625.
4. Jumlah bit pada sisi *output* diperoleh dari setengah jumlah bit pada sisi *input* yaitu 16-bit dengan resolusi biner sebesar 0,00000001, jika dikonversikan desimal menjadi 0,00390625
5. Perangkat lunak yang digunakan pada penelitian ini adalah Altera Quartus Prime 16.1 *Lite Edition* dengan bahasa program VHDL.
6. Modul FPGA yang digunakan adalah *Board* Altera Cyclone IV modul dapat dilihat pada Gambar 3.2



Gambar 3.2 Tampilan pada *Board* Altera Cyclone IV

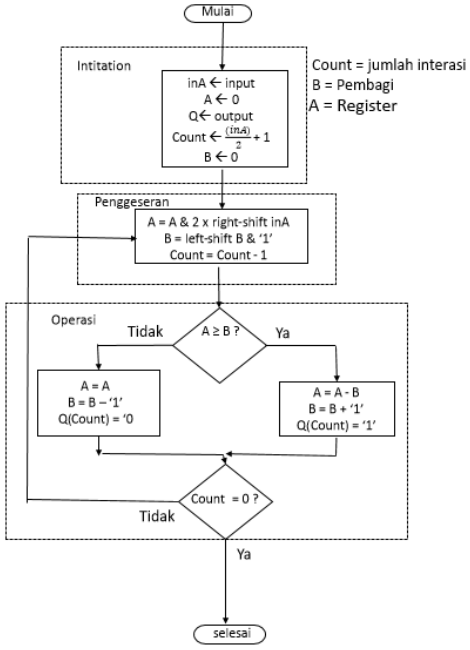
7. Spesifikasi dari FPGA Cyclone IV E seri EP4CE6E22C8 terdapat pada Tabel 3.1 sebagai berikut..

Tabel 3.1 Spesifikasi *Board* FPGA Cyclone IV E seri EP4CE6E22C8

Produk Atribut	**Nilai Atribut**
Manufacturer	Intel
Product Category	FPGA – *Field Programmable Gate Array*
Product	Cyclone IV E
Number of Logic Elements	6272
Number of Logic Array Blocks – LABs	392
Number of I/Os	91 *Input*/*Output* (I/O)
Operating Supply Voltage	1 V to 1,2V
Minimum Operating Temperature	0 C
Maximum Operating Temperature	+ 70 C
Series	EP4CE6E22C8 Cyclone IV E
Brand	Intel / Altera
Maximum Operating Frequency	200 MHz
Subcategory	*Programmable Logic ICs*
Total Memory	270 kbit

3.3 Perancangan Sistem/Simulasi/Metode Analisis

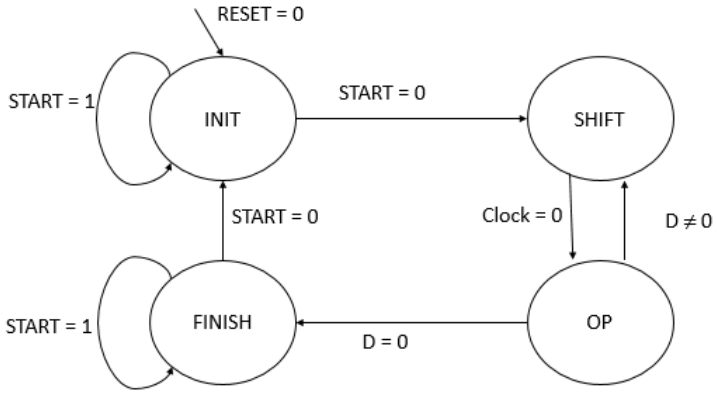
Perancangan diawali dengan pembuatan diagram alir algoritme *modified digit by digit non-restoring* yang ditunjukkan pada Gambar 3.3



Gambar 3.3 Diagram alir algoritme *modified digit by digit non-restoring*

Diagram alir pada Gambar 3.3 merupakan langkah–langkah cara kerja perhitungan akar kuadrat. Hal ini dibutuhkan untuk mengimplementasikan metode tersebut ke dalam bahasa pemrograman.

Pada diagram alir ini akan di konversikan *Finite State machine*(FSM), FSM terdiri dari seperangkat status, beberapa *input*, beberapa *output*, dan seperangkat aturan untuk berpindah dari satu keadaan ke keadaan lain atau *state to state*.



Gambar 3.4 Diagram *state machine*

Pada Gambar 3.4 merupakan diagram dari *state machine* yang digunakan untuk pemrograman. Setiap lingkaran yang diisikan nama di dalamnya merupakan suatu *state* atau

keadaan. Panah antar *state* merupakan pengubah dari suatu *state* ke *state* yang lain. Informasi pada tanda panah merupakan instruksi jika informasi tersebut tercapai maka *state* tersebut berpindah sesuai dengan tanda panah. Pada penelitian ini memakai 4 *state* seperti pada gambar 3.4 di atas, tanda panah yang datang dari "*NOWHERE*" mengarah ke INIT menunjukkan bahwa INIT adalah keadaan awal. Berikut adalah keterangan dari *state* yang digunakan.

- INIT

  State INIT merupakan keadaan awal, *state* ini tidak akan berpindah jika reset = '0' dan atau start = '1'. *State* ini memiliki fungsi untuk membaca nilai *input* baru dan men-reset semua *signal*, *input* dan *output*. *State* ini akan berpindah ke *state* SHIFT jika reset ='1' dan start = '0'.
- SHIFT

  State SHIFT merupakan keadaan selanjutnya dari INIT dan selalu berpindah ke *state* OP setiap *clock*. Pada *state* ini terjadinya penggeseran pada operasi perhitungan.
- OP

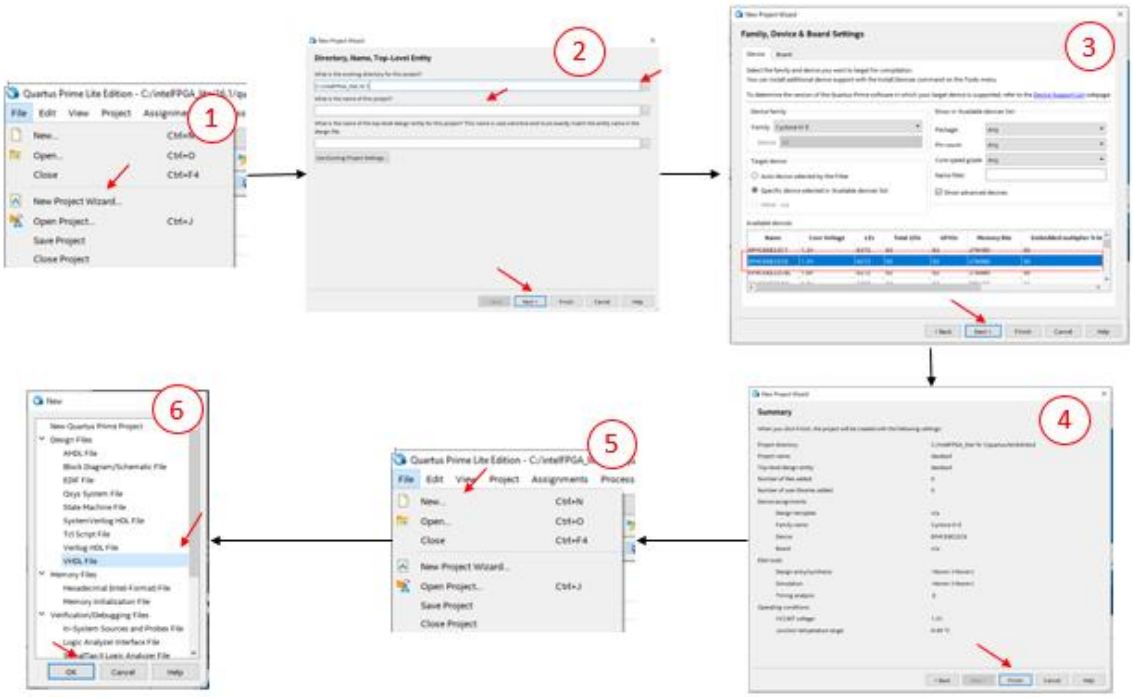
  State OP atau operasi merupakan inti dari pengoperasian perhitungan pada pemrograman. *State* ini merupakan jalannya perhitungan akar kuadrat dan sekaligus menghasilkan nilai *output* yang akan diperoleh. *State* ini akan mengurangi nilai D sampai D = '0' yang menunjukkan *state* ini akan berpindah ke *state* FINISH. Nilai D merupakan jumlah iterasi pada perhitungan akar kuadrat.
- FINISH

  State FINISH merupakan keadaan perhitungan akar kuadrat telah selesai. *State* ini ditandai dengan nilai D = '0'. *State* ini tidak akan berpindah jika nilai start ='1'. *State* ini akan berpindah ke INIT jika start ='0' dan membaca nilai *input* baru.

Selanjutnya penulisan program algoritme dengan cara *state machine* yang dilakukan sesuai dengan spesifikasi sistem. Penulisan dilakukan berulang jika terjadi *Error* dan tidak sesuai dengan spesifikasi sistem.

3.4 Penulisan Algoritme Menggunakan Bahasa VHDL

program ditulis menggunakan perangkat lunak Quartus prime 16.1 *Lite Edition*. Bahasa yang digunakan pada penulisan program adalah bahasa VHDL. Langkah-langkah pembuatan *project* pada perangkat lunak perangkat Quarus Prime 16.1 *Lite Edition* terdapat pada gambar 3.5 sebagai berikut.



Gambar 3.5 Membuat *project* di Quarus Prime 16.1 *Lite Edition* dengan bahasa program VHDL

Keterangan :

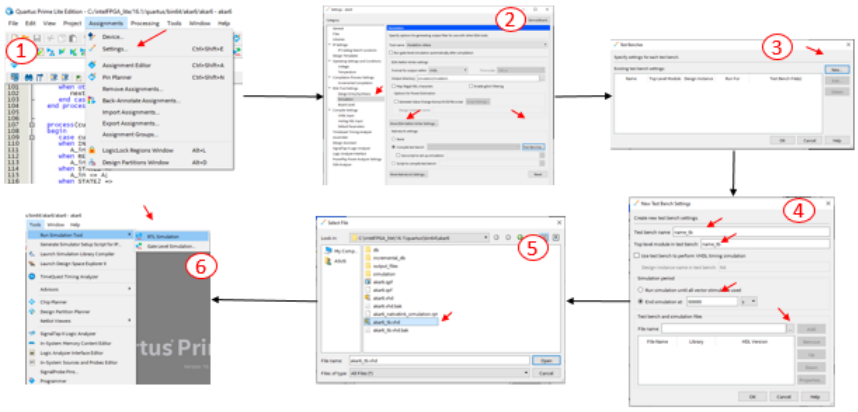
1) Setelah perangkat lunak Quartus Prime 16.1 *Lite Edition* terbuka, untuk membuat *project* baru dengan cara menekan pada menu *File* > *New Project Wizard* pada gambar nomor 1.
2) Setelah muncul jendela pada nomor 2 isi lokasi *direktori project* di lokasi yang diinginkan (dapat membuat folder), kemudian ini nama *project* yang diinginkan tidak mengandung spasi dan diawali dengan huruf, kemudian tekan *next* sampai muncul jendela pada gambar nomor 3.
3) Setelah muncul jendela nomor 3 isikan *device family* sesuai dengan FPGA yang digunakan, pada penelitian ini perangkat yang akan digunakan adalah Cyclone IV E dengan nama seri EP4CE6E22C8 seperti pada jendela nomor 3 kemudian tekan *next*.
4) Maka akan muncul tampilan *EDA Tool Setting*, kemudian tekan *next* sehingga diperoleh tampilan pada jendela nomor 4 yaitu bagian akhir/ *summary* dari *New Projet Wizard* lalu tekan *Finish*.
5) Selanjutnya untuk membuat VHDL *file* dapat dicari menekan *keybord* ctrl + N atau menekan menu *File* > *New*.. seperti pada gambar nomor 5.
6) Setelah itu maka muncul jendela seperti pada nomor 6, maka pilih *Design* VHDL *file* untuk bahasa program VHDL.

3.5 Verifikasi Sistem

Proses verifikasi sistem dilakukan dalam dua tahap, yaitu verifikasi fungsional pada simulasi program menggunakan aplikasi ModelSim-Altera dan verifikasi fungsional pada *hardware* dengan modul FPGA Altera Cyclone IV EP4CE6E22C8.

3.5.1 Verifikasi Fungsional Program

Verifikasi fungsional pada program dilakukan untuk melihat bahwa algoritme program yang disimulasikan sesuai dengan yang diinginkan. Hal ini dilakukan dengan membandingkan antara hasil dari hasil simulasi dengan perhitungan manual. Ketika ada perbedaan antara perhitungan manual dengan simulasi akan dilakukan perbaikan pada algoritme program dan dilakukan perbandingan kembali hingga hasil dari program sama dengan hasil dari perhitungan manual. Pengujian dilakukan menggunakan *Test Bench* agar dapat melakukan pengujian banyak *input* sehingga memudahkan dalam pengujian. Langkah-langkah simulasi dengan cara *Test Bench* ditunjukkan pada Gambar 3.6 pada berikut.



Gambar 3.6 Langkah-langkah simulasi menggunakan *Test Bench*

Keterangan:

1) Penelitian ini melakukan simulasi fungsional dengan menggunakan *Test Bench*, sehingga diperlukan penulisan program *Test Bench* terlebih dahulu. Setelah selesai penulisan tekan menu *Assigment* > *Settings*.. seperti pada Gambar 3.6 nomor 1.
2) Kemudian muncul jendela *settings*, pilih *Category Simulasion* lalu pilih *Compile Test Bench* dan tekan *Test Benches*,
3) Kemudian pada jendela *Test Benches* tekan *New*.
4) Selanjutnya isi nama dan *top level* pada *New Test Bench* dan isi panjang simulasi tersebut.
5) Kemudian masukan *file Test Bench* yang telah dibuat lalu tekan *OK* dan *Apply*.

6) Setelah sudah dimasukkan *file Test Bench* maka dilakukan simulasi, tekan menu *Tools* > *Run Simulation Tool* > *RTL Simulation.*

Setelah selesai melakukan pengujian maka dilakukan mencari selisih dan persentase *Error* dari hasil simulasi. Hal ini dilakukan agar mengetahui dampak penggunaan bilangan pecahan dalam perhitungan akar kuadrat. Cara untuk mencari selisih seperti pada persamaan (3.1):

$$n = P - Q \tag{3.1}$$

Dan untuk mencari persentase *Error* dari hasil simulasi seperti pada persamaan (3.2):

$$E = \frac{P-Q}{P} \times 100\% \tag{3.2}$$

Keterangan :

n : Selisih dari hasil perhitungan kalkulator dengan hasil simulasi

P : Hasil perhitungan akar kuadrat menggunakan kalkulator

Q : Hasil perhitungan akar kuadrat dari simulasi

E : Persentase *Error* dari hasil simulasi dengan hasil kalkulator

3.5.2 Verifikasi Fungsional *Hardware*

Sedangkan verifikasi fungsional pada *hardware* dilakukan penambahan algoritme untuk menyesuaikan I/O *board* pada FPGA. Algoritme tambahan tersebut adalah algoritme untuk menampilkan dan memasukan nilai I/O perhitungan akar kuadrat dari atau ke-*board* FPGA ke program utama. Penelitian ini menggunakan *port* reset sebagai reset, *port clock* sebagai *clock*. Pada modul FPGA terdapat 4 buah *port push button*, pada penelitian ini memakai semua *push button* yang dimiliki modul dan masing – masing *port push button* memiliki fungsi yang berbeda-beda yaitu *port* S1 *push button* sebagai start, *port* S2 *push button* untuk menambah nilai *input*, *port* S3 *push button* untuk mengurangi nilai *input*, *port* S4 *push button* untuk mengganti tampilan *seven segment* menjadi tampilan *input* atau *output*. Kemudian 4 buah *port seven segment* digunakan untuk melihat nilai *input* dan *output*, karena pada modul FPGA tersebut hanya memakai 4 buah *seven segment* maka ada keterbatasan dalam penampilan *input* dan *output*. 2 buah *seven segment* digunakan untuk menampilkan nilai bilangan bulat dan 2 buah *seven segment* digunakan untuk menampilkan pecahannya. Dan sebuah LED sebagai indikator proses telah selesai.

3.6 Evaluasi

Pada evaluasi dilakukan perbandingan dengan peneliti lain melibatkan hasil *resource* yang didapatkan yaitu perbandingan *Logic Element* dan *Delay* terbesar atau terburuk. Pada perbandingan *Logic Element* jumlah *input* yang digunakan adalah 64-bit. 62-bit, dan 32-bit. Jumlah *Logic Element* mempengaruhi kecepatan komputasi dan penggunaan sumber daya

perangkat keras sehingga lebih sedikit jumlah *Logic Element* yang di peroleh maka penggunaan sumber daya perangkat keras lebih sedikit. Untuk mengetahui total *Logic Element* yang digunakan dapat dilihat pada hasil *report* simulai.

Kemudian perbandingan *Delay* terbesar atau terburuk antar *register*. Pada perbandingan *Delay* terbesar jumlah *input* yang digunakan adalah 62-bit, 16-bit, 8-bit dan 4-bit. Jumlah *Delay* terbesar mempengaruhi kecepatan frekuensi *switching* sehingga jika lebih kecil *Delay* yang dimiliki maka lebih tinggi frekuensi *switching* yang dimana dibutuhkannya untuk penerapan DTC yang membutuhkan frekuensi *switching* yang tinggi. *Delay* tersebut dapat diketahui dari hasil *report* simulasi pada frekuensi terbesar pada suhu tertinggi. Untuk mencari nilai *Delay* dapat dilihat dari persamaan (3.3):

$$T_m = \frac{1}{F_m} \tag{3.3}$$

Keterangan:

Tm : *Delay* terbesar atau terburuk antar *register*

Fm : Frekuensi terbesar pada suhu tertinggi

BAB 4
HASIL DAN PEMBAHASAN

4.1 Hasil Perancangan Pada Quartus

Pada penulisan program menggunakan quartus dengan FSM menggunakan 4 *state*/keadaan yaitu INIT, SHIFT, OP dan FINISH1. Penulisan perintah *state* dapat dilihat pada Gambar 4.1

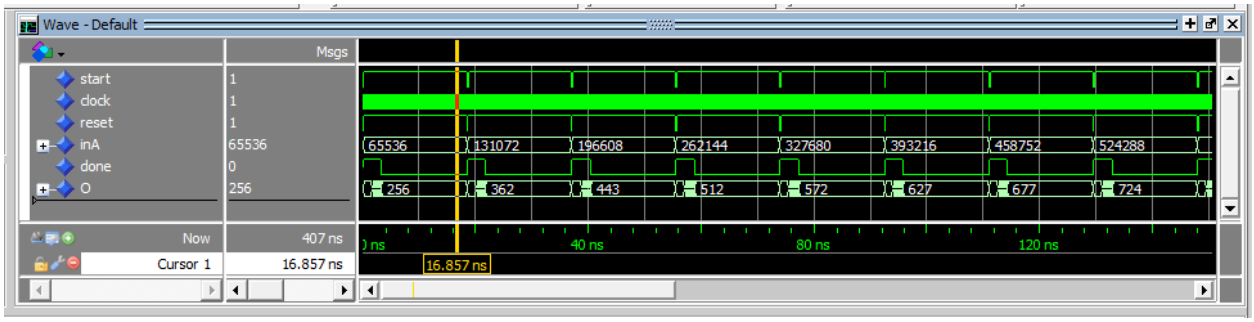
```
21    type states is (INIT, SHIFT, OP, FINISH1);
52    process (currentstate, start, D)
53    begin
54       case currentstate is
55       when INIT =>
56          if (start = '1') then
57             nextstate <= INIT;
58          else
59             nextstate <= SHIFT;
60          end if;
61       when SHIFT =>
62          nextstate <= OP;
63       when OP =>
64          if (D = 0) then
65             nextstate <= FINISH1;
66          else
67             nextstate <= SHIFT;
68          end if;
69       when FINISH1 =>
70          if (start = '0') then
71             nextstate <= INIT;
72          else
73             nextstate <= FINISH1;
74          end if;
75       when others =>
76          nextstate <= INIT;
77       end case;
78    end process;
79
```

Gambar 4.1 Algoritme *State machine*

Gambar 4.1 adalah bagian arsitektur FSM pada algoritme perhitungan akar kuadrat. Pada Gambar 4.1 dapat diketahui bahwa setiap *state* berpindah sesuai dengan aturannya, seperti pada *state* INIT berpindah ke *state* SHIFT jika start = 0 jika tidak maka tetap di INIT, *state* SHIFT berpindah setiap *clock*, *state* OP ke FINISH1 jika D = 0 dan jika tidak maka pindah ke SHIFT, dan *state* FINISH berpindah ke INIT jika start = 0 dan jika tidak maka maka tetap di INIT. Pada satu siklus ini hanya terdapat 1 *state* atau 1 keadaan setiap *clock.*

4.2 Verifikasi Fungsional Program dengan Simulasi

Proses verifikasi fungsional program dengan menggunakan *Test Bench* dengan 22 *input* simulasi yang berbeda. *Input* simulasi bernilai 1, 2, 3, 4, 5, 6, 7, 8, 9, 22, 39, 72, 137, 265, 534, 1063, 2120, 4233, 8201, 16406, 32807, dan 65535,99609375. Hasil simulasi seperti pada Gambar 4.2.



Gambar 4.2 Hasil Simulasi Menggunakan Perangkat Lunak MODELSim Altera

Gambar 4.2 adalah hasil simulasi yang menggunakan *Test Bench*. Dapat dilihat bahwa "inA" merupakan nilai *input* akar kuadrat dan "O" merupakan *output* akar kuadrat. Pada Gambar 4.2 menampilkan nilai I/O yang berbeda dengan tujuan hasil yang dicapai oleh peneliti dikarenakan sebagian bit digunakan untuk nilai pecahannya. Hal ini dilakukan karena pada simulasi tidak bisa menerjemahkan bit pecahannya. Sehingga dibutuhkannya penjabaran untuk mengetahui nilai sebenarnya pada simulasi. Nilai-nilai pada Gambar 4.2 dijabarkan pada Tabel 4.1.

Tabel 4.1 Penjabaran Hasil dari Simulasi

Input Simulasi	*Output Simulasi*	Hasil Kalkulator	Selisih	*Error* %
1,0	1	1	0	0%
2,0	1,4140625	1,414213562	$1{,}5106 \times 10^{4}$	0,01%
3,0	1,73046875	1,732050807	$1{,}58205 \times 10^{3}$	0.091%
4,0	2,00	2	0	0%
5,0	2,234375	2,236067977	$1{,}69297 \times 10^{3}$	0,075%
6,0	2,449218750	2,449489742	$2{,}7099 \times 10^{4}$	0.011%
7,0	2,64453125	2,645751311	$1{,}22006 \times 10^{3}$	0,046%
8,0	2,828125	2,828427124	$3{,}02124 \times 10^{4}$	0,01%
9,0	3	3	0	0%
22	4,6875	4,69041576	$2{,}91576 \times 10^{3}$	0,062 %
39	6,2421875	6,244998	$2{,}8105 \times 10^{3}$	0,045%
72	8,484375	8,48528137	$9{,}0637 \times 10^{4}$	0,01%
137	11,703125	11,7046999	$1{,}574942 \times 10^{3}$	0,013%
265	16,27734375	16,2788206	$1{,}476846 \times 10^{3}$	0,009%
534	23,10546875	23,10544	$2{,}9712665 \times 10^{3}$	0,012%
1063	32,6015625	32,8966026	$2{,}07135 \times 10^{3}$	0,006%
2120	46,04296875	46,0434577	$4{,}889828 \times 10^{4}$	0.001%
4233	65,05859375	65,0615094	$2{,}9156084 \times 10^{3}$	0.004%
8201	90,55859375	90,55859375	$7{,}790393 \times 10^{4}$	0.0008%
16406	128,08203125	128,085909	$3{,}8774207 \times 10^{3}$	0.003%
32807	181,125	181,127227	$2{,}0272488 \times 10^{3}$	0.0011%
65535,99609375	255,99609375	255,999992	$3{,}8986206 \times 10^{3}$	0.0015%

Tabel 4.1 adalah penjabaran dari simulasi pada Gambar 4.2 dan hasil perhitungan kalkulator. *input* simulasi merupakan penjabaran dari "inA" atau nilai *input* akar kuadrat dan *output* simulasi merupakan penjabaran dari "O" atau merupakan nilai *output* akar kuadrat pada *Test Bench*. Dari Tabel 4.1 diketahui sistem ini memiliki nilai *Error* yang kecil sehingga nilai *output* simulasi mendekati nilai perhitungan kalkulator. Persentase *Error* yang dimaksud adalah persentase *Error* dari hasil perhitungan simulasi yaitu *output* simulasi dengan perhitungan kalkulator. Pada tabel

diatas diketahui bahwa *Error* yang diperoleh bervariasi ada keterkaitannya nilai *Error* semakin kecil jika nilai *input* simulasi besar. *Error* terjadi karena terbatasnya nilai bit *output* simulasi pecahan, pada simulasi tersebut *Error* terbesar yang di dapatkan pada Tabel 4.1 adalah 0,091% dan selisih terbesar adalah 0,0038986206. *Error* terjadi karena nilai bit *output* simulasi memiliki keterbatasan yaitu 16-bit, maka pembagian bit *output* menjadi 8-bit untuk bilangan bulat dan 8-bit digunakan untuk pecahannya. Sehingga selisih terbesar yang akan didapatkan sistem adalah dengan bilangan biner 0,00000001 dan jika dikonversikan ke desimal menjadil 0,00390625. Total *resource* yang didapatkan pada sistem ini ditunjukan oleh Tabel 4.2.

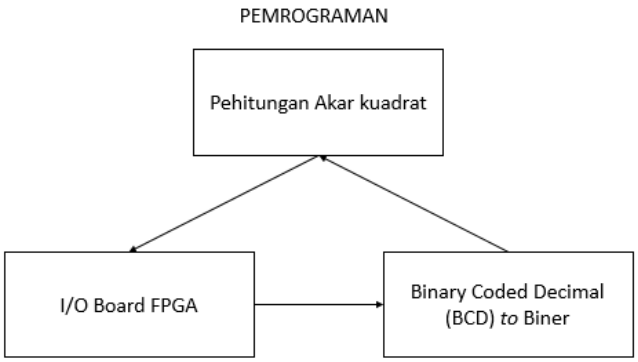
Tabel 4.2 Hasil *Resource*

Resource	Jumlah
Logic Element	157
Registers	120
Pins	52
Fmax	151,49 MHz

Pada tabel 4.2 adalah hasil *resource* pada program perhitungan akar kuadrat. Sistem ini tidak menggunakan *Memory bits* dan *Embedded multiplier* 9-bit *element*. Pin yang digunakan pada sistem ini adalah 1 pin start, 1 pin *clock*, 1 pin reset, 32 pin *input* simulasi, 16 pin *output* simulasi dan 1 pin *Enable.* Pada *Logic Element* yang didapatkan menghasilkan hasil yang bagus dikarenakan *Logic Element* yang digunakan sebanyak 3% dari total yang dimiliki *hardware*. hal ini merupakan penggunaan yang hemat sehingga dapat mengurangi penggunaan sumber daya pada FPGA. Pada Frekuensi maksimal yang ditampilkan pada Tabel 4.2 adalah frekuensi maksimal pada suhu tertinggi atau pada suhu 85°C.

4.3 Hasil Perancangan FPGA

Mengimplementasi perancangan ke FPGA untuk menampilkan hasil perhitungan akar kuadrat pada FPGA dibutuhkan program tambahan. Ada 2 pemrograman tambahan untuk mengimplementasikan perhitungan akar kuadrat yaitu I/O pada Board FPGA dan *Binary Coded Decimal* (BCD) *to* Biner. Keterkaitan kedua pemrograman tersebut dapat dilihat pada Gambar 4.3.



Gambar 4.3 Siklus Jalan I/O

Pada gambar 4.3 merupakan proses pemrograman berkerja. Masing – masing pemrograman memiliki fungsi yang bebeda-beda. Pada pemrograman I/O Board FPGA memberikan nilai *input* dalam bentuk BCD ke pemrograman BCD *to* Biner, dan menampilkan nilai *output* hasil simulai perhitungan akar kuadrat pada pemrograman perhitungan akar kuadrat. Sedangkan pemrograman BCD *to* Biner memiliki fungsi mengubah nilai *input* BCD dari pemrograman I/O *board* FPGA menjadi biner dan diteruskan ke pemrograman perhitungan akar kuadrat.

4.4 Verifikasi Fungsional *Hardware*

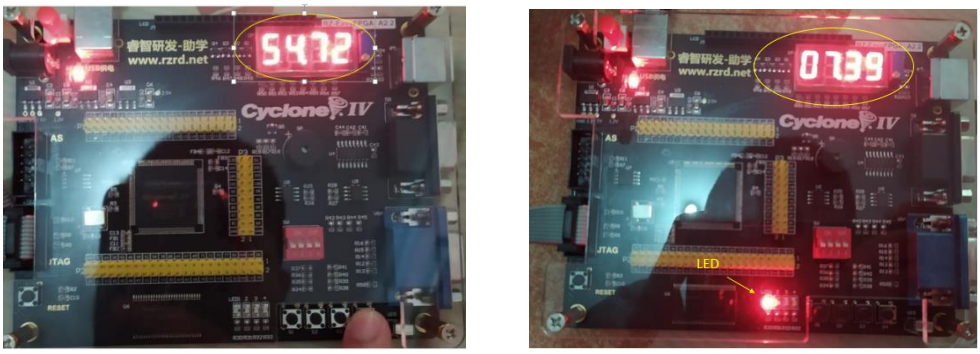
Proses verifikasi *hardware* menggunakan *port* pada *board* FPGA sebagai *input* yang digunakan adalah *clock*, *port* reset, dan 4 buah *port push button*, sedangkan untuk *output* adalah 4 buah *port seven segment* dan 1 buah *port* LED. Verifikasi dilakukan untuk mengetahui fungsi dari *port* tersebut sesuai dengan spesifikasi sistem. *Push button* pada *board* memiliki fungsi yang berbeda-beda, dapat dilihat fungsi *push button* pada Tabel 4.3.

Tabel 4.3 Kegunaan *Port Push button*

Port Push button	Fungsi
S1	*Counter up*
S2	*Counter Down*
S3	Start
S4	Mengganti penampilan *Seven segment*

Pada *push button* S1 dan S2 berfungsi untuk menaikan atau menurunkan nilai *input* yang dapat dilihat pada *seven segment*. Hal ini dilakukan secara bersamaan dengan menekan *push button* S4, fungsi dari *push button* S4 adalah menampilkan nilai *input*. Pada *push button* S3 berfungsi untuk memulai proses pada FPGA sehingga menampilkan nilai *output* baru pada *seven segment*.

Pada Bord sistem ini hanya memiliki 4 buah *seven segment* sehingga memiliki keterbatasan dalam menampilkan nilai *input* dan *output*. Dengan nilai *input* tertinggi 99,99 dan nilai *input* terendah 0,01. Berikut adalah proses verifikasi *hardware* pada Gambar 4.4.



Gambar 4.4 Verifikasi *Hardware*: gambar kiri menampilkan nilai *input* dan gambar kanan menampilkan nilai *output*

Pada Gambar 4.4 merupakan tampilan pada *board* nilai *input* dan nilai *output*. Pada indikator LED sebagai indikator bahwa proses telah selesai dan nilai *output* tertera di *seven segment*. Karena keterbatasan *seven segment* nilai *output* memiliki resolusi 0,01 dan akan terjadi *Error* dengan selisih kurang dari 0,01. Hal ini juga dikarenakan jumlah bit *output* pecahan yang seharusnya 8 bit menjadi 7 bit pada penampilan *seven segment*.

4.5 Perbandingan dengan Hasil Penelitian Lain

Perbandingan hasil penelitian lain dilakukan perubahan spesifikasi yang sesuai dengan spesifikasi penelitian yang ingin dibandingkan. Berdasarkan hasil kompilasi untuk mengimplementasikan perhitungan akar kuadrat dengan *input* 32-bit, 62-bit dan 64-bit menghasilkan *Logic Element* (LE) masing-masing sebesar 157, 248 dan 258. Hasil penelitian yang dilibatkan dalam penelitian ini adalah penelitian [5], [6]. Perbandingan penggunaan LE dengan penelitian lain ditunjukan pada Tabel 4.4.

Tabel 4.4 Perbandingan *Logic Element* (LE)

No	Metode	Penggunaan LE		
		64-bit	62-bit	32-bit
1	*Classical-NR* [5]	4092	-	1008
2	*Reduced-Area-NR* [5]	2464	-	632
3	*Modular-NR* [5]	2468	-	624
4	*Simple-X-Module* [5]	2488	-	648
5	Tole Sutikno [5]	1023	-	256
6	AZ Jidin dkk [6]	-	281	-
7	Penelitian ini	258	248	157

Berdasarkan pada pada bit *input* 64-bit, 62-bit, dan 32-bit penelitian ini menggunakan *Logic Element* yang terkecil dibandingkan dengan penelitian lain, sehingga lebih sedikit mengkonsumsi sumber daya, dan juga hal ini membuat perhitungan akar kuadrat pada DTC akang menjadi cepat.

Kemudian membandingkan jumlah *Delay* terbesar dengan penelitian lain. *Delay* terbesar yang dimaksud adalah *Delay* antar register. Hal ini juga didasarkan dengan nilai *input* yang sama dengan penelitian yang ingin dibandingkan. Hasil penelitian yang dilibatkan dalam penelitian ini adalah penelitian [6], [13], [14]. Perbandingan *Delay* terbesar atau terburuk dengan penelitian lain ditunjukan pada Tabel 4.5

Tabel 4.5 Perbandingan *Delay* Terbesar atau Terburuk

No	Aspek	Jumlah *Delay* Terbesar			
		62-bit	16-bit	8-bit	4-bit
1	S Samavi dkk [13]	-	32,023 ns	10,023 ns	6,485 ns
2	AP Ramesh dkk [14]	-	37,166 ns	9,215 ns	-
3	AZ Jidin dkk [6]	12,345 ns	-	-	-
4	Penelitian ini	8,349 ns	4,863 ns	4,133 ns	4,234 ns

Berdasarkan pada Tabel 4.5 membuktikan penelitian ini memiliki *Delay* terburuk yang lebih rendah dibandingkan dengan AZ Jidin dkk, S Samavi dkk dan AP Ramesh dkk. Hasil dari penelitian ini cukup memuaskan dikarenakan dibutuhkan *Delay* terburuk antar register yang bernilai kecil untuk penerapan DTC dalam frekuensi *switching* yang tinggi. Sehingga penelitinan ini memiliki hasil lebih unggul dibandingkan dengan penelitian lain. Hal ini dikarenakan sistem ini memaksimalkan kegunaan lebar bit pada pengoperasiannya.

BAB 5
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Kesimpulan dari penelitian ini adalah seebagai berikut:

1. Implementasi perhitungan akar kuadrat bilangan pecahan berjalan dengan baik, hasil dari implementasi tersebut sesuai dengan spesifikasi sistem yang telah ditentukan.
2. Implementasi perhitungan akar kuadrat dengan algoritme *modified digit by digit non-restoring* pada FPGA berhasil sesuai dengan yang diharapkan yang sederhana sehingga medapatkan nilai dengan sekali pengoperasian. Pada penelitian ini menggunakan algoritme modified *digit by digit non-restoring* yang dikonversikan ke *state machine* sehingga ada dua keadaan atau *state* yaitu penggeseran dan pengoperasian. Dengan menggunakan metode tersebut penelitian ini menggunakan lebih sedikit jumlah *Logic Element* yang digunakan dan *Delay* yang kecil.
3. Pada sistem ini peneliti akan memakai jumlah bit *input* sebanyak 32-bit dan *output* 16-bit, pada bit *input* bilangan bulat memakai 16 bit dan bit pecahan memakai 16 bit. Pada bit *output* bilangan bulat memakai 8 bit dan bit pecahan memakai 8 bit. Resolusi pada nilai *input* sebesar 0,0000152587890625 dan resolusi nilai *output* 0,00390625.
4. Sistem ini menghasilkan *resource* yaitu : 157 *Logic Element*, 120 *Registers*, pin yang digunakan 52 *pins*, 151,49 Mhz, 0 *total memory bits*, dan 0 *embedded multiplier 9-bit element.*
5. Pada tampilan *board* FPGA memiliki nilai *input* terbesar 99,99 dan terkecil 0,01 dengan resolusi *Error* terbesar 0,01.

5.2 Saran

1. Untuk memaksimalkan lebar bit *input* simulasi dapat dilakukan peletakan bit *input* secara fleksibel sehingga sekaligus memaksimalkan kegunaan lebar bit *output* simulasi.
2. Dapat menyederhanakan penulisan program dengan cara yang berbeda sehingga memperkecil *resource* yang didapatkan dari penelitian ini

DAFTAR PUSTAKA

[1] I. Takahashi and T. Noguchi, "A New Quick-Response and High-Efficiency Control Strategy of an Induction Motor," *IEEE Trans. Ind. Appl.*, vol. IA-22, no. 5, pp. 820–827, 1986.

[2] Y. Liu, Z. Zhu, D. H.-T. on I. Applications, and U. 2007, "Commutation-torque-ripple minimization in direct-torque-controlled PM brushless DC drives," *IEEE Trans. Ind. Appl.*, vol. 43, no. 4, pp. 1012–1021, 2007.

[3] T. Sutikno, N. R. Nik Idris, A. Z. Jidin, and A. Jidin, "A Model of FPGA-based Direct Torque Controller," *TELKOMNIKA Indones. J. Electr. Eng.*, vol. 11, no. 2, 2013.

[4] T. Sutikno, "An Efficient Implementation of the Non Restoring Square Root Algorithm in Gate Level," *Int. J. Comput. Theory Eng.*, vol. 3, no. 1, pp. 46–51, 2013.

[5] T. Sutikno, "An Optimized Square Root Algorithm for Implementation in FPGA Hardware," *TELKOMNIKA (Telecommunication Comput. Electron. Control.*, vol. 8, no. 1, p. 1, 2010.

[6] A. Z. Jidin, S. H. Mohamad, S. Ahmad, M. F. Yakub, N. A. N. Azlan, and A. Jidin, "Optimizing The Flux and Torque Estimator of DTC in FPGA by Using Low-Area Square Root Calculator," in *PECON 2016 - 2016 IEEE 6th International Conference on Power and Energy, Conference Proceeding*, 2017, pp. 517–521.

[7] T. Sutikno, N. R. N. Idris, A. Jidin, and M. N. Cirstea, "An Improved FPGA Implementation of Direct Torque Control for Induction Machines," *IEEE Trans. Ind. Informatics*, vol. 9, no. 3, pp. 1280–1290, 2013.

[8] K. N. Vijeyakumar, V. Sumathy, P. Vasakipriya, and A. Dinesh Babu, "FPGA Implementation of Low Power High Speed Square Root Circuits," in *2012 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC 2012*, 2012, pp. 1–5.

[9] S. Lachowicz and H. J. Pfleiderer, "Fast Evaluation of The Square Root and Other Nonlinear Functions in FPGA," in *Proceedings - 4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, 2008, pp. 474–477.

[10] M. D. Ercegovac, "On Digit-by-Digit Methods for Computing Certain Functions," in *Conference Record - Asilomar Conference on Signals, Systems and Computers*, 2007, pp. 338–342.

[11] O. Kosheleva, "Babylonian method of computing the square root: Justifications based on fuzzy techniques and on computational complexity," in *Annual Conference of the North American Fuzzy Information Processing Society - NAFIPS*, 2009, pp. 1–6.

[12] T. J. Kwon, J. Sondeen, and J. Draper, "Floating-Point Division and Square Root Implementation Using a Taylor-Series Expansion Algorithm," in *Proceedings of the 15th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2008*, 2008, pp. 702–705.

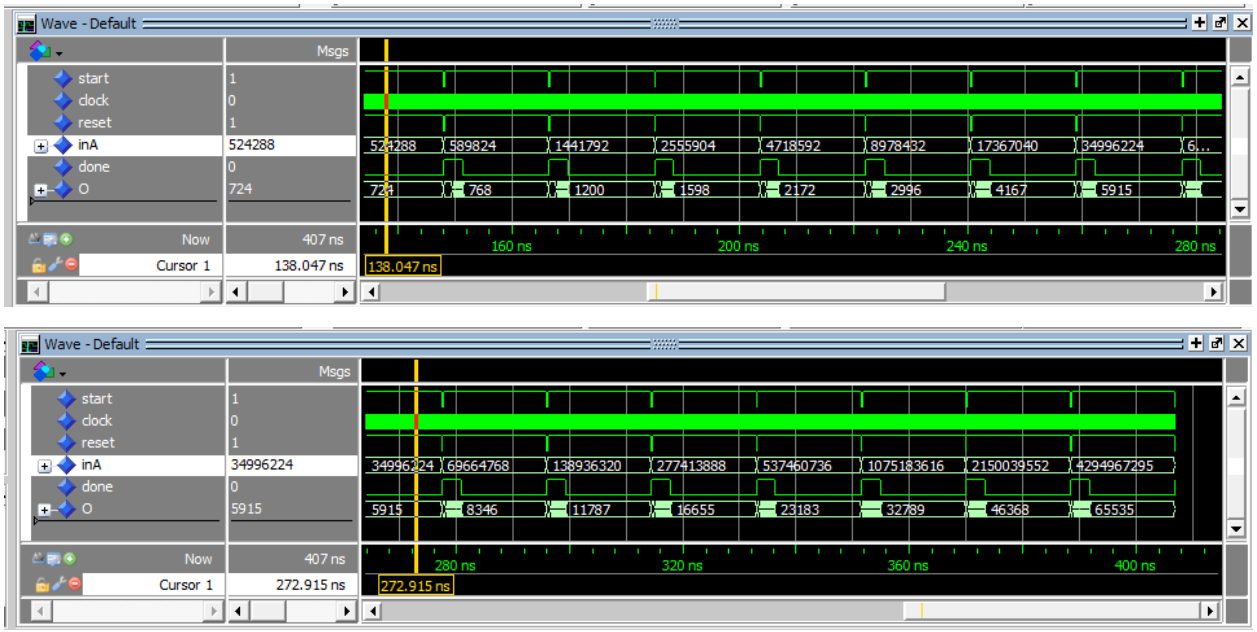
[13] S. Samavi, "Improving Array Structure of Non-Restoring Square Root Circuit," *Int. J. Eng. Sci.*, vol. 14, no. 1, pp. 1–14, 2003.

[14] A. P. Ramesh and I. J. Kumar, "Implementation of Integer Square Root," *Int. J. Eng. Sci. Innov. Technol.*, vol. 4, no. 1, pp. 105–113, 2015.

[15] A. Nanhe, G. Gawali, S. Ahire, and K. Sivasankaran, “Implementation of Fixed and Floating Point Square Root Using Nonrestoring Algorithm on FPGA,” *Int. J. Comput. Electr. Eng.*, vol. 5, no. 5, pp. 533–537, 2013.

LAMPIRAN

Berikut adalah hasil simulasi menggunakan perangkat MODELSim



Gambar 1 Hasil Simulasi yang Lain, Menggunakan Perangkat Lunak MODELSim

Berikut adalah lampiran dari sintaks program VHDL untuk perhitungan akar kuadrat.

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

entity akar is
  port (
        start, clock, reset  : in std_logic;
        inA                  :in std_logic_vector (31 downto 0);
        o                    : out std_logic_vector (15 downto 0);
        done                 : out std_logic
  );
end akar;

architecture Behavioral of akar is
  type states is (INIT, INIT1, SHIFT, OP, FINISH1);
  signal currentstate, nextstate: states;
  signal A, A_in: std_logic_vector (17 downto 0);
  signal A1, A1_in: std_logic_vector (31 downto 0);
  signal B, B_in: std_logic_vector (17 downto 0);
  signal D, D_in: integer;
  signal Z, Z_in: std_logic_Vector (15 downto 0);

begin
  process(clock, reset)
  begin
        if(reset = '0' )  then
              currentstate <= INIT;
              A <= (others => '0');
```

```
                A1 <= (others => '0');
                B <= (others => '0');
                D <= 15;
                Z <= (others => '0');
        elsif(clock'event and clock = '1') then
                currentstate <= nextstate;
                A <= A_in;
                A1 <= A1_in;
                B <= B_in;
                D <= D_in;
                Z <= Z_in;

        end if;
end process;

process (currentstate, start, D)
begin
        case currentstate is
        when INIT =>
                if (start = '1') then
                        nextstate <= INIT;
                else
                        nextstate <= INIT1;
                end if;
        when INIT1 =>
                if (start = '0') then
                        nextstate <= INIT1;
                else
                        nextstate <= SHIFT;
                end if;

        when SHIFT =>
                nextstate <= OP;
        when OP =>
                if (D = 0) then
                        nextstate <= FINISH1;
                else
                        nextstate <= SHIFT;
                end if;
        when FINISH1 =>
                if (start = '0') then
                        nextstate <= INIT;
                else
                        nextstate <= FINISH1;
                end if;
        when others =>
                nextstate <= INIT;
        end case;
end process;


process(currentstate, A1, A, B)
begin
        case currentstate is
        when SHIFT =>
                A_in <= A(15 downto 0) & A1(31 downto 30);
        when OP =>
                if (A < B) then
                A_in <= A;
        else
                A_in <= A - B;
                end if;
```

```
		when others =>
			A_in <= A;

		end case;
end process;

process(currentstate, B, A)
begin
		case currentstate is
		when SHIFT =>
			B_in <= B(16 downto 0) & '1';
		when OP =>
			if (A < B) then
			B_in <= B - "0000000000000001";
		else
			B_in <= B + "0000000000000001";
			end if;
		when others =>
			B_in <= B;

		end case;
end process;

process(currentstate, inA, A1)
begin
		case currentstate is
		when INIT =>
			A1_in <= inA;
		when SHIFT =>
			A1_in <= A1(29 downto 0) & "00";
		when OP =>
			A1_in <= A1;
		when others =>
			A1_in <= A1;
		end case;
end process;


process(currentstate, D)
begin
		case currentstate is
		when SHIFT =>
			D_in <= D;
		when OP =>
			D_in <= D - 1;
		when others =>
			D_in <= D;
		end case;
end process;

process(currentstate, Z, A, B)
begin
		case currentstate is
		when SHIFT=>
			Z_in <= Z;
		when OP =>
			if (A < B) then
			Z_in <=  Z(14 downto 0) & '0';
		else
			Z_in <= Z(14 downto 0) & '1';
			end if;
		when others =>
```

```
                        Z_in <= Z;
                end case;
    end process;

    o <= z;
    process (currentstate)
    begin
                case currentstate is
                when FINISH1 =>
                        done <= '0';
                when others =>
                        done <= '1';
                end case;
    end process;
end Behavioral;
```

PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA

SKRIPSI

untuk memenuhi salah satu persyaratan
mencapai derajat Sarjana S1

Disusun oleh:

MUHAMMAD IRFAN

16524100

Jurusan Teknik Elektro

Fakultas Teknologi Industri

Universitas Islam Indonesia

Yogyakarta

2020

LEMBAR PENGESAHAN

PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA

TUGAS AKHIR

Diajukan sebagai Salah Satu Syarat untuk Memperoleh
Gelar Sarjana Teknik
pada Program Studi Teknik Elektro
Fakultas Teknologi Industri
Universitas Islam Indonesia

Disusun oleh:

Muhammad Irfan
16524100

Yogyakarta, 20-juli-2020

Menyetujui,

Pembimbing

Dr. Eng. Hendra Setiawan, S.T., M.T
025200526

KATA PENGANTAR

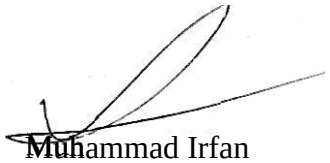
Atas izin Allah Subhanahu wa Ta'ala tugas akhir yang berjudul "PERHITUNGAN AKAR KUADRAT UNTUK FLUKS STATOR PADA *DIRECT TORQUE CONTROL* (DTC) DENGAN PERANGKAT FPGA" dapat kami selesaikan melalui proses yang cukup panjang, tugas akhir ini disusun untuk memenuhi salah satu persyaratan untuk menyelesaikan program studi teknik elektro di Universitas Islam Indonesia.

Selama mengikuti pendidikan program studi teknik elektro sampai dengan proses penyelesaian tugas akhir, berbagai pihak telah memberikan fasilitas, membantu, membina dan membimbing penulis, untuk itu penulis mengucapkan terimakasih khususnya kepada:

1. Kedua orang tua yang telah memberikan dukungan dan do'a selama penulis menuntut ilmu di UII.
2. Bapak Dr.Eng., Hendra Setiawan, S.T., M.T. selaku dosen teknik elektro dan pembimbing tugas akhir.
3. Bapak Yusuf Aziz Amrullah,S.T., M.Sc., Ph.D selaku Ketua Jurusan Teknik Elektro, Universitas Islam Indonesia
4. Bapak dan Ibu dosen pengajar yang telah memberikan ilmunya kepada penulis.
5. Teman – teman teknik elektro angkatan 2016 dan Keluarga besar teknik elektro UII.

Penulis memohon maaf jika terdapat kesalahan dalam penulisan tugas akhir ini, semoga yang sedikit ini dapat bermanfaat dan dapat memberikan kontribusi dalam bidang keilmuan yang terkait.

Yogyakarta, 21 agustus 2020



Muhammad Irfan

ARTI LAMBANG DAN SINGKATAN

n	: Selisih dari hasil perhitungan kalkulator dengan hasil simulasi
E	: Persentase *Error* dari hasil simulasi dengan hasil kalkulator
Fm	: Frekuensi maksimal pada suhu tertinggi
P	: Hasil perhitungan akar kuadrat menggunakan
Q	: Hasil perhitungan akar kuadrat dari simulasi
Tm	: Delay terbesar atau terburuk antar *register*
dkk	: dan kawan-kawan
kbit	: kilo bit
ns	: *nanosecond*
DSP	: *Digital Signal Processor*
DTC	: *Direct Torque Control*
Fmax	: *Frequency maximum*
FPGA	: *Field programmable Gate Array*
FSM	: *Finite State machine*
HDL	: *Hardware Description Language*
I/O	: *Input/Output*
LE	: *Logic Element*
MHz	: *Megahertz*
PWM	: *Pulse Width Modulation*
VHDL	: *VHSIC Hardware Description Language*
VHSIC	:*Very High Speed Integrated Circuit*

ABSTRAK

Direct Torque Control (DTC) adalah metode kendali telah banyak digunakan dalam penggerak motor. Pada DTC terdapat perhitungan akar kuadrat yang membutuhkan proses perhitungan yang sangat cepat dan beroperasi pada frekuensi yang tinggi. Penerapan DTC pada FPGA diusulkan untuk mengatasi masalah tersebut. Pada penelitian ini, mengimplementasikan perhitungan akar kuadrat bilangan pecahan dengan metode *digit by digit non-restoring modified* pada FPGA. Tujuan dari penelitian ini untuk mengimplementasikan perhitungan akar kuadrat bilangan pecahan dan mengetahui hasil dari *resource* yang didapatkan pada implementasi tersebut. Proses perhitungan akar kuadrat menggunakan metode *digit by digit non-restoring modifed* dengan jumlah bit *input* 32-bit dan *output* 16-bit. Untuk menghemat jumlah *resource* yang digunakan, sistem ini dirancang dengan sederhana menggunakan *Finite State machine* (FSM). Pada FSM terdapat 2 *state* penting pada pengoperasian perhitungan akar kuadrat yaitu *state* penggeseran dan *state* pengoperasian. Proses verifikasi sistem ini dilakukan dalam dua tahap, yaitu verifikasi fungsional dengan aplikasi ModelSim-Altera dan verifikasi *hardware* menggunakan modul FPGA Cylcone IV EP4CE6E228N. Hasil pengujian menunjukkan bahwa *output* perhitungan akar kuadrat memiliki presisi yang tinggi dengan relsolusi 0,00390625. Sistem ini membutuhkan 157 *Logic Elements*, 120 register, 0 *multiplier* 9-bit, dan 151,59 MHz FMax.

Kata kunci: DTC, Akar Kuadrat, *Non-Restoring*, *Delay*, *Logic Element*, *Delay, VHDL*, FPGA, FSM.

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENDAHULUAN

1.1 Latar Belakang Masalah

Direct Torque Control (DTC) adalah metode kendali telah banyak digunakan dalam penggerak motor [1]. Populeritas DTC dikarenakan memiliki strukturnya yang sederhana, di mana tidak memerlukan pengkodean posisi dan pembangkit *Pulse Width Modulation* (PWM), dan keunggulan lainya memiliki respon torsi yang cepat [2]. DTC dapat diimplementasikan dalam mikrokontroler, *Digital Signal Processor* (DSP), *Field Programmable Gate Array* (FPGA) dan perangkat digital lain. Ketika di implementasikan ke mikrokontroler atau DSP kinerja pada DTC akan menurun dikarenakan ketidakmampuan untuk menghitung dengan sangat cepat dan beroperasi pada frekuensi *switching* yang tinggi, sehingga tidak memadai untuk meminimalkan keluaran torsi riak [3]–[5].

Oleh karena itu, penerapan DTC dalam FPGA diusulkan untuk mengatasi masalah ini. FPGA adalah salah satu perangkat umum yang digunakan dalam pembuatan prototipe modul pemrosesan digital, biayanya yang murah dan komputasi yang berkinerja tinggi [6].

Pada DTC dibutuhkannya kalkulator akar kuadrat untuk menghasilkan perkiraan fluks stator [7]. Akar kuadrat adalah operasi aritmatika penting dalam pemrosesan sinyal digital [8]. Tidak seperti operasi lain seperti penjumlahan, pengurangan dan perkalian, perhitungan akar kuadrat sangat sulit direalisasikan dalam FPGA, karena berbagai faktor seperti kompleksitas algoritme, waktu penyelesaian perhitungan, jumlah penggunaan sumber daya *hardware* dan juga konsumsi daya [9]. Ada beberapa algoritme yang dapat digunakan untuk membangun kalkulator akar kuadrat dalam FPGA, seperti *Rough Estimation*, metode *Babylonian*, metode *Taylor-Series expansion*, dan metode *digit by digit* [4]–[6], [10]–[12]. Pada penelitian ini metode yang digunakan adalah metode *digit by digit non restoring modified*, hal ini dianjurkan dari penelitian lain dikarenakan metode tersebut memiliki struktur yang sederhana sehingga menghasilkan komputasi yang lebih cepat dan mengkonsumsi sumber daya *hardware* lebih rendah [6].

Algoritme akar kuadrat tidak mudah diimplementasikan pada FPGA. Sudah ada beberapa penelitian yang berusaha mengimplementasikan algoritme akar kuadrat, seperti yang dilakukan penelitian [4]–[6], [13], [14]. Penelitian tersebut melakukan modifikasi algoritme perhitungan akar kuadrat menjadi lebih sederhana dan membandingkan dengan penelitian yang lain. Pada penelitian Tole Sutikno dan AZ Jidin dkk menggunakan metode *digit by digit non restoring modified* dan melakukan perbandingan jumlah *Logic Element* (LE) [4]–[6]. Sedangkan pada penelitian Samavi dkk dan AP Ramesh dkk melakukan perbandingan *delay* yang diperoleh [13], [14]. Namun dari

semua penelitian tersebut perhitungan akar kuadrat yang dilakukan hanya memberi nilai bilangan bulat sehingga mendapatkan hasil yang kurang presisi. Hasil dari perhitungan akar kuadrat bilangan bulat yang mengabaikan nilai pecahan dapat dilihat pada Tabel 1.1 sebagai berikut.

Tabel 1.1 Perhitungan Akar kuadrat Yang Mengabaikan Nilai Pecahan

A	$\sqrt{A}$	Hasil Perhitungan kalkulator	Selisih A dan Hasil Perhitungan kalkulator	*Persentase Error*
1	1	1	0	0%
2	1	1,41421356	0,41421356	29,28%
3	1	1,73205081	0,73205081	42,26%
4	2	2	0	0%
5	2	2,23606798	0,23606798	10,5%
6	2	2,44948974	0,44948974	18.35%
7	2	2,64575131	0,64575131	24,4%
8	2	2,82842712	0,82842712	29.28%

Tabel 1.1 adalah hasil perhitungan akar kuadrat yang mengabaikan bilangan pecahan. Nilai A adalah nilai yang dilakukan operasi akar kuadrat dan hasil dari akar kuadrat A mengabaikan nilai pecahannya, sehingga akar kuadrat dari 1, 2 dan 3 adalah 1 dan akar kuadrat dari 4, 5, 6, 7 dan 8 adalah 2 maka tidak ada perbedaan antara akar kuadrat dari 1, 2 dan 3 dan juga akar kuadrat dari 4, 5, 6, 7 dan 8. Hal ini menimbulkan persentase *Error* yang besar, persentase *Error* pada Tabel 1.1 yang dimaksud adalah persentase *Error* dari hasil perhitungan $\sqrt{A}$ dengan hasil nilai dari kalkulator. Pada akar kuadrat dari 3 memiliki persentase *Error* terbesar yaitu 42,26%.

Untuk mengurangi *Error* yang besar atau mendapatkan hasil presisi yang tinggi, perhitungan akar kuadrat akan melibatkan nilai pecahan. Pada penelitian ini akan dilakukan modifikasi algoritme perhitungan akar kuadrat yang melibatkan nilai pecahannya dengan algoritme yang sederhana. Sehingga perhitungan akar kuadrat yang terdapat di algoritme DTC akan menghasilkan respons kecepatan yang tinggi dan memiliki nilai yang lebih presisi.

1.2 Rumusan Masalah

Dari uraian di atas dapat dibuat rumusan masalah sebagai berikut:

1. Bagaimana mengimplementasikan perhitungan akar kuadrat bilangan pecahan pada FPGA?
2. Bagaimana unjuk kerja atau hasil pengujian perhitungan akar kuadrat bilangan pecahan pada FPGA?

1.3 Batasan Masalah

1. Algoritme *modified digit by digit non-restoring* untuk perhitungan akar kuadrat.
2. *Resource* yang terlibat dalam perancangan adalah *Logic Element, Register, Total Pins* dan Frekuensi tertinggi.

1.4 Tujuan Penelitian

1. Mengimplementasi perhitungan akar kuadrat bilangan pecahan pada FPGA.
2. Mendapatkan unjuk kerja atau hasil perhitungan akar kuadrat bilangan pecahan pada FPGA.

1.5 Manfaat Penelitian

1. Meningkatkan akurasi perhitungan akar kuadrat bilangan pecahan di perangkat FPGA.
2. Menambah khazanah keilmuan berkaitan dengan implementasi komputasi matematis pada FPGA.

BAB 2
TINJAUAN PUSTAKA

2.1 Studi Literatur

Perhitungan akar kuadrat menggunakan FPGA telah banyak dilakukan sebelumnya, seperti yang dilakukan oleh S. Samavi dkk [13]. Karena desain sirkuit terdahulu memiliki proses yang panjang dan banyak, sehingga S. Samavi dkk mencari proses desain yang sederhana secara signifikan. Pada *input bits* 4, 8 dan 16, S. Samavi dkk mendapatkan *delay* terburuk sebesar 6,485 ns, 10,023 ns dan 32.016 ns [13]. Hal ini merupakan hasil yang baik, karena desain sebelumnya memiliki hasil *delay* lebih besar dari pada desain S. Samavi dkk [13]. Mereka telah menyederhanakan penulisan algoritme *digit by digit non-restoring* sehingga mengurangi *delay* terburuk.

Pada tahun 2010, Tole Sutikno melakukan perbandingan metode antara metode *modified non-restoring digit by digit* dan metode operasi perhitungan akar kuadrat yang lain [5]. Penelitian tersebut mengimplementasi algoritme *non-restoring* yang dimodifikasi dengan 32 bit dan 64 bit menggunakan Altera APEX 20KE FPGA yang membutuhkan 256 *Logic Element* (LE) dan 1023 LE [5]. Algoritme *non-restoring* yang dioptimalkan mengurangi area *chip* dan *pipelining* sehingga meningkatkan kinerja kecepatan. Algoritme *restoring* menyimpan sisanya di setiap interasi maka membutuhkan LE lebih banyak dibandingkan dengan algoritme *non-restoring* yang dimodifikasi[15].

Pada tahun 2015, Addanki Purna Ramesh dan I. Jayaram Kumar melakukan penelitian dengan menerapkan akar kuadrat bilangan bulat dengan menggunakan metode *Square and Compare*, s*uccessive subtraction of odd integer"s method* dan modifikasi metode *non-restoring* yang diimplementasikan menggunakan bahasa program *Very High Speed Integrated Circuit (VHSIC) Hardware Description Language* (VHDL) dan biosintesis dengan menggunakan Xilinx12.1 [14]. Dengan *input* 16 bit mendapatkan hasil *delay* dari metode *Square and Compare* 82,941 ns, s*uccessive subtraction of odd integer"s method* 59,816 ns, dan modifikasi metode *non-restoring* 37,166 ns [14]. Hasil dari penelitian tersebut menunjukkan bahwa metode *non-restoring* yang dimodifikasi memiliki *delay* yang lebih sedikit [14].

Penelitian S. Samavi dkk dan AP Ramesh merupakan perbandingan metode *non-restoring* dengan metode lain, dan didapatkan hasil bahwa metode *non-restoring* memiliki *delay* yang lebih sedikit dibandingkan dengan metode lain [13], [14]. Pada penelitian-penelitian tersebut melakukan implementasi algoritme yang hanya berlaku pada bilangan bulat saja. Namun pada penelitian di atas tidak memecahkan perhitungan akar kuadrat bilangan desimal. Dengan adanya perhitungan

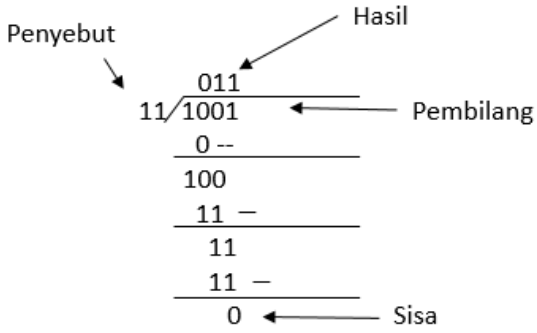
akar kuadrat desimal maka mendapatkan hasil yang lebih presisi. Penelitian ini akan memodifikasi metode sebelumnya untuk mendapatkan perhitungan akar kuadrat bilangan desimal agar mendapatkan hasil yang presisi dan efisien.

2.2 Tinjauan Teori

Perhitungan akar kuadrat bilangan biner memiliki berbagai macam metode. Yaitu metode *Rough Estimation*, metode *Babylonian*, algoritme ekspansi *Taylor-Series*, metode *Newton-Raphson*, dan algoritme berurutan (metode perhitungan *digit by digit*) [4]. Pada penelitian ini memakai metode *digit by digit* dengan algoritme *non-restoring* yang telah dimodifikasi. Konsep dasar dari metode ini seperti konsep pembagian bilangan biner, dimana dilakukannya operasi perhitungan *digit by digit* secara berurutan.

2.2.1 Pembagian dalam Bilangan Biner

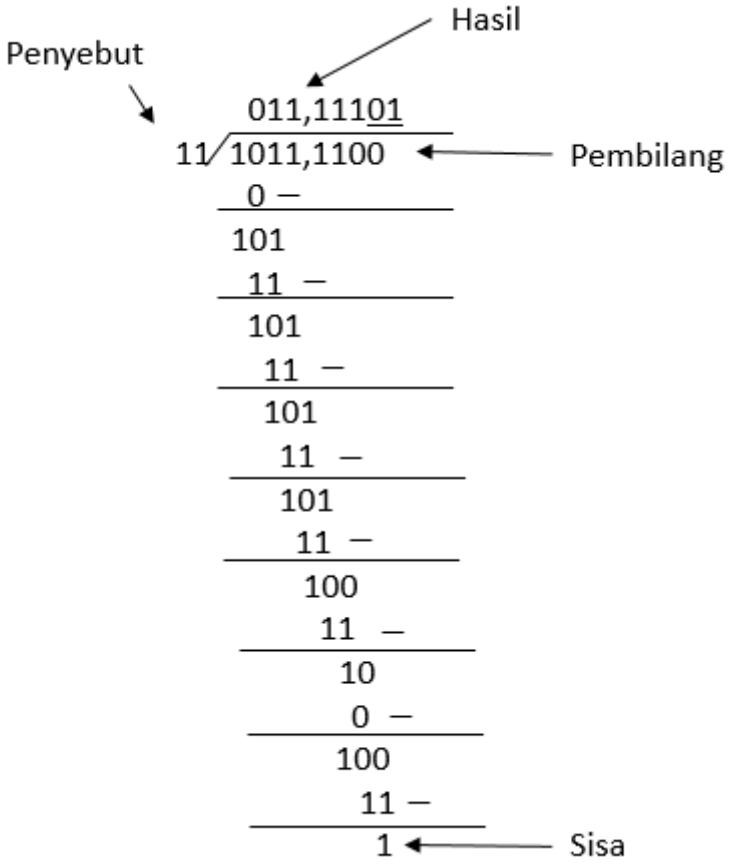
Pembagian bilangan biner pada dasarnya sama dengan pembagian bilangan desimal, yang membedakan ialah pada biner hanya mempunyai dua angka yaitu 0 dan 1. Cara pembagian bilangan biner adalah mengurangi biner yang mau dibagi mulai dari sebelah kiri dengan biner pembagi. Jika biner tersebut bisa dikurangi maka bernilai 1 pada hasil dan jika tidak bisa di kurangi maka bernilai 0. Pada Gambar 2.1 adalah contoh pembagian bilangan biner:



Gambar 2.1 Contoh operasi perhitungan pembagian biner

Gambar 2.1 adalah contoh pembagian dalam bilangan biner 1001 : 11 = 11 atau nilai desimalnya menjadi 9 : 3 = 3. Pada prinsipnya pengambilan biner setiap satu bit setiap operasi. Ketika pembilang lebih besar atau sama dengan dari penyebut maka dilakukan operasi pengurangan dan hasil pembagian bernilai 1. Pada sisa operasi ditambahkan bit berikutnya dan menjadi pembilang pada operasi selanjutnya. Jika pembilang lebih kecil dari pada penyebut maka tidak dilakukan operasi pengurangan dan hasil bit selanjutnya akan bernilai 0. Pembilang yang

tidak dilakukan pengurangan akan ditambah dengan bit selanjutnya dan menjadi pembilang pada operasi selanjutnya. Hal ini dilakukan hingga bit berikutnya tidak ada lagi.

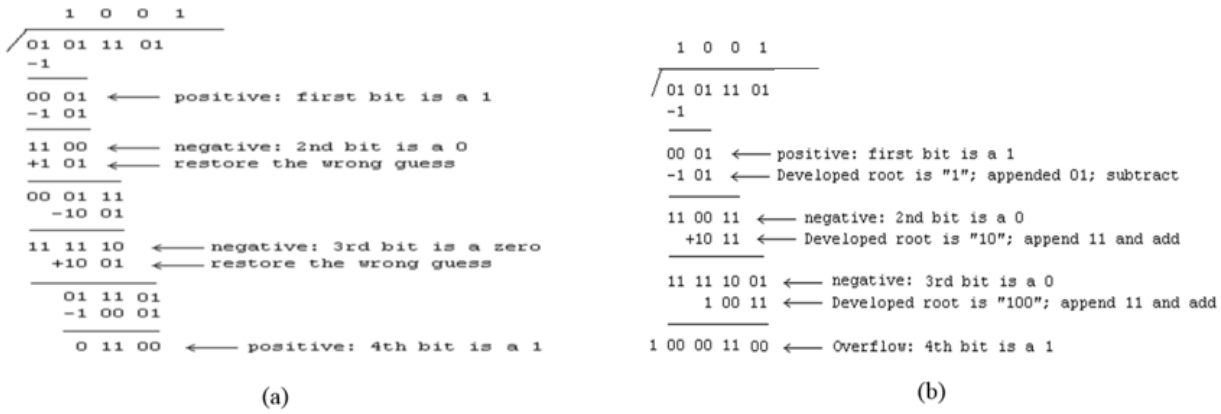


Gambar 2.2 Contoh operasi perhitungan pembagian biner dengan pecahan

Gambar 2.2 adalah operasi perhitungan pembagian biner pecahan. Pada prinsipnya sama dengan pembagian biner bilangan bulat hanya saja perhitungan ini tidak mengabaikan sisa pembagian sehingga menghasilkan nilai pecahan. Ketika nilai di sebelah kiri sudah tidak ada maka di tambakan nilai 0 supaya bisa dikurangi. Ketika nilai hasil pengurangan sama dengan yang sebelumnya maka bisa berhenti perhitungan karena pembagian akan berulang–ulang dengan nilai yang sama. Seperti gambar 2.2 mengoperasikan pembagian nilai biner 1011,11 : 11 yang hasilnya 11,11$\underline{10}$ atau 11,111010101010101... dan seterusnya, Karena melakukan pengulangan terus-menerus maka diberi tanda garis atas atau bawah untuk pengulangan.

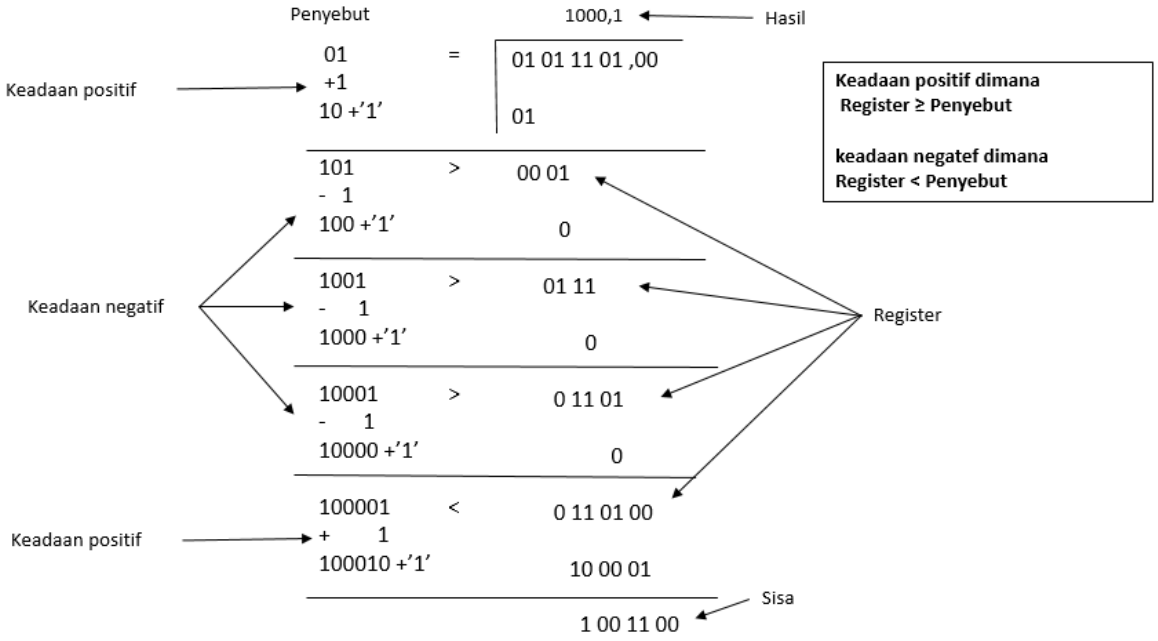
2.2.2 Perhitungan Akar Kuadrat dalam Bilangan Biner

Metode perhitungan akar kuadrat bilangan biner yang digunakan adalah metode *digit by digit*, metode ini memiliki 2 algoritme yang berbeda yaitu *restoring* dan *non-restoring*. Gambar 2.3 berikut ini adalah langkah-langkah metode *restoring* dan *non-restoring*.



Gambar 2.3 Contoh metode *digit by digit* perhitungan akar kuadrat (a) *algoritme restoring* (b) algoritme *non-restoring* [4]

Sedangkan yang digunakan peneliti ini adalah metode *modified digit by digit non-restoring*, metode ini memodifikasi algoritme *non-restoring*. Gambar 2.4 berikut ini adalah langkah-langkah metode *modified digit by digit non-restoring*.



Gambar 2.4 Contoh metode *digit by digit* perhitungan akar kuadrat algoritme *modified non-restoring*

Gambar 2.4 adalah contoh dari perhitungan akar kuadrat biner menggunakan algoritme *modified non-restoring*. Prinsip utama metode ini adalah pembilang akan dikelompokkan setiap 2 bit dan di kurang dengan penyebutnya. penyebut tersebut diawali dengan bit 01, pembilang dan penyebut akan berubah seiring keadaan sebelumnya. Penyebut akan menambahkan bit 1 kanan *Least Significant Bit*(LSB) setiap beroperasi. Ketika penyebut lebih kecil atau sama dengan pembilang maka akan dilakukan operasi, pembilang akan dikurang penyebut dan hasil akan bernilai 1, keadaan ini disebut keadaan positif. Penyebut akan ditambah dengan 01 kemudian

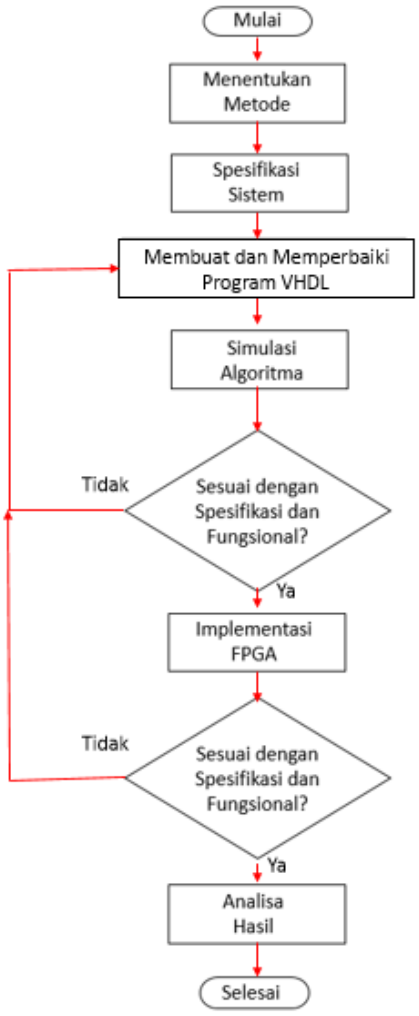
menambahkan bit bernilai 1 kanan LSB dan pembilang akan menambahkan 2 bit yang sudah di kelompok pada sebelah kanan LSB sisa dari pengurangan sebelumnya untuk operasi berikutnya. Jika pembilang lebih kecil dari pada penyebut maka tidak dilakukan operasi pengurangan dan hasil bernilai 0, keadaan ini disebut keadaan negatif. Penyebut akan di kurang dengan 01 kemudian menambahkan bit bernilai 1 kanan LSB dan pembilang akan menambahkan 2 bit yang sudah di kelompok pada sebelah kanan LSB penyebut sebelumnya untuk operasi berikutnya. Dari gambar di atas memberikan contoh perhitungan akar dari 93 atau nilai biner 01011101, sehingga di dapatkan akar dari 93 yaitu 9,64 atau nilai biner 1001,10

Proses pada Gambar 2.4 yaitu algoritme *modified non-restoring* lebih sederhana dibandingkan dengan Gambar 2.3 algoritme *restoring* dan *non-restoring*. Algoritme *restoring* dan *non-restoring* memiliki operasi pengurangan dan pertambahan sedangkan algoritme *modified non-restoring* hanya operasi pengurangan. Sehingga algoritme *modified non-restoring* lebih sedikit operasi perhitungan dibandingkan algoritme *restoring* dan *non-restoring*

BAB 3
METODOLOGI

3.1 Alur Penelitian

Proses penelitian yang dilakukan dijelaskan pada Gambar 3.1 yang merupakan diagram alir alur penelitian ini :



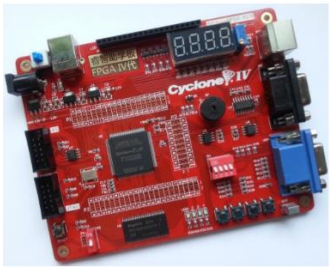
Gambar 3.1 Diagram Alir Penelitian

Gambar 3.1 adalah langkah-langkah penelitian dalam bentuk diagram alir. Penelitian ini dimulai dari menentukan spesifikasi sistem hingga analisa hasil. Pada proses spesifikasi yang bertujuan untuk menentukan target yang dicapai dalam penelitian ini. Selanjutnya membuat atau memodifikasi algoritme pemrograman VHDL dan dilakukan simulasi. Kemudian mengimplementasikan ke FPGA dan memperbaiki kesalahan pada implementasi FPGA. Setelah berhasil mengimplementasi ke FPGA maka dilakukan penyesuaian dengan spesifikasi, setelah sesuai kemudian dilakukan analisa hasil penelitian.

3.2 Spesifikasi Sistem

Sistem ini dirancang dengan spesifikasi sebagai berikut:

1. Komputasi algoritme menggunakan metode *modified digit by digit non-restoring*.
2. Target dari sistem ini adalah mendapatkan hasil perhitungan kuadrat bilangan bulat beserta pecahannya.
3. Pada sistem ini peneliti akan memakai jumlah bit *input* sebanyak 32 bit dengan resolusi biner sebesar 0,0000000000000001, jika dikonversikan desimal menjadi 0,0000152587890625.
4. Jumlah bit pada sisi *output* diperoleh dari setengah jumlah bit pada sisi *input* yaitu 16-bit dengan resolusi biner sebesar 0,00000001, jika dikonversikan desimal menjadi 0,00390625
5. Perangkat lunak yang digunakan pada penelitian ini adalah Altera Quartus Prime 16.1 *Lite Edition* dengan bahasa program VHDL.
6. Modul FPGA yang digunakan adalah *Board* Altera Cyclone IV modul dapat dilihat pada Gambar 3.2



Gambar 3.2 Tampilan pada *Board* Altera Cyclone IV

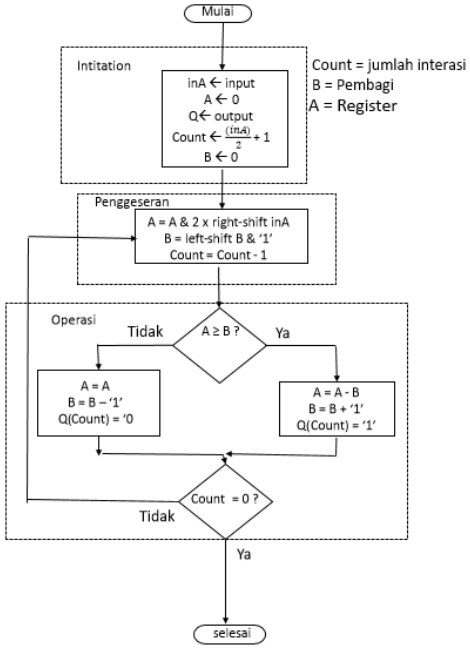
7. Spesifikasi dari FPGA Cyclone IV E seri EP4CE6E22C8 terdapat pada Tabel 3.1 sebagai berikut..

Tabel 3.1 Spesifikasi *Board* FPGA Cyclone IV E seri EP4CE6E22C8

Produk Atribut	**Nilai Atribut**
Manufacturer	Intel
Product Category	FPGA – *Field Programmable Gate Array*
Product	Cyclone IV E
Number of Logic Elements	6272
Number of Logic Array Blocks – LABs	392
Number of I/Os	91 *Input*/*Output* (I/O)
Operating Supply Voltage	1 V to 1,2V
Minimum Operating Temperature	0 C
Maximum Operating Temperature	+ 70 C
Series	EP4CE6E22C8 Cyclone IV E
Brand	Intel / Altera
Maximum Operating Frequency	200 MHz
Subcategory	*Programmable Logic ICs*
Total Memory	270 kbit

3.3 Perancangan Sistem/Simulasi/Metode Analisis

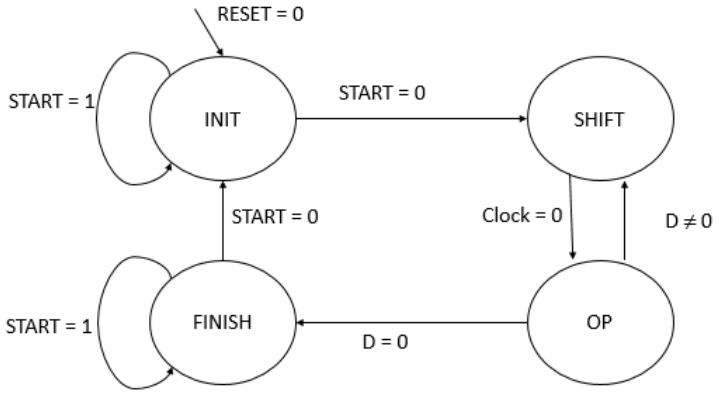
Perancangan diawali dengan pembuatan diagram alir algoritme *modified digit by digit non-restoring* yang ditunjukkan pada Gambar 3.3



Gambar 3.3 Diagram alir algoritme *modified digit by digit non-restoring*

Diagram alir pada Gambar 3.3 merupakan langkah–langkah cara kerja perhitungan akar kuadrat. Hal ini dibutuhkan untuk mengimplementasikan metode tersebut ke dalam bahasa pemrograman.

Pada diagram alir ini akan di konversikan *Finite State machine*(FSM), FSM terdiri dari seperangkat status, beberapa *input*, beberapa *output*, dan seperangkat aturan untuk berpindah dari satu keadaan ke keadaan lain atau *state to state*.



Gambar 3.4 Diagram *state machine*

Pada Gambar 3.4 merupakan diagram dari *state machine* yang digunakan untuk pemrograman. Setiap lingkaran yang diisikan nama di dalamnya merupakan suatu *state* atau

keadaan. Panah antar *state* merupakan pengubah dari suatu *state* ke *state* yang lain. Informasi pada tanda panah merupakan instruksi jika informasi tersebut tercapai maka *state* tersebut berpindah sesuai dengan tanda panah. Pada penelitian ini memakai 4 *state* seperti pada gambar 3.4 di atas, tanda panah yang datang dari "*NOWHERE*" mengarah ke INIT menunjukkan bahwa INIT adalah keadaan awal. Berikut adalah keterangan dari *state* yang digunakan.

- INIT

  State INIT merupakan keadaan awal, *state* ini tidak akan berpindah jika reset = '0' dan atau start = '1'. *State* ini memiliki fungsi untuk membaca nilai *input* baru dan men-reset semua *signal*, *input* dan *output*. *State* ini akan berpindah ke *state* SHIFT jika reset ='1' dan start = '0'.
- SHIFT

  State SHIFT merupakan keadaan selanjutnya dari INIT dan selalu berpindah ke *state* OP setiap *clock*. Pada *state* ini terjadinya penggeseran pada operasi perhitungan.
- OP

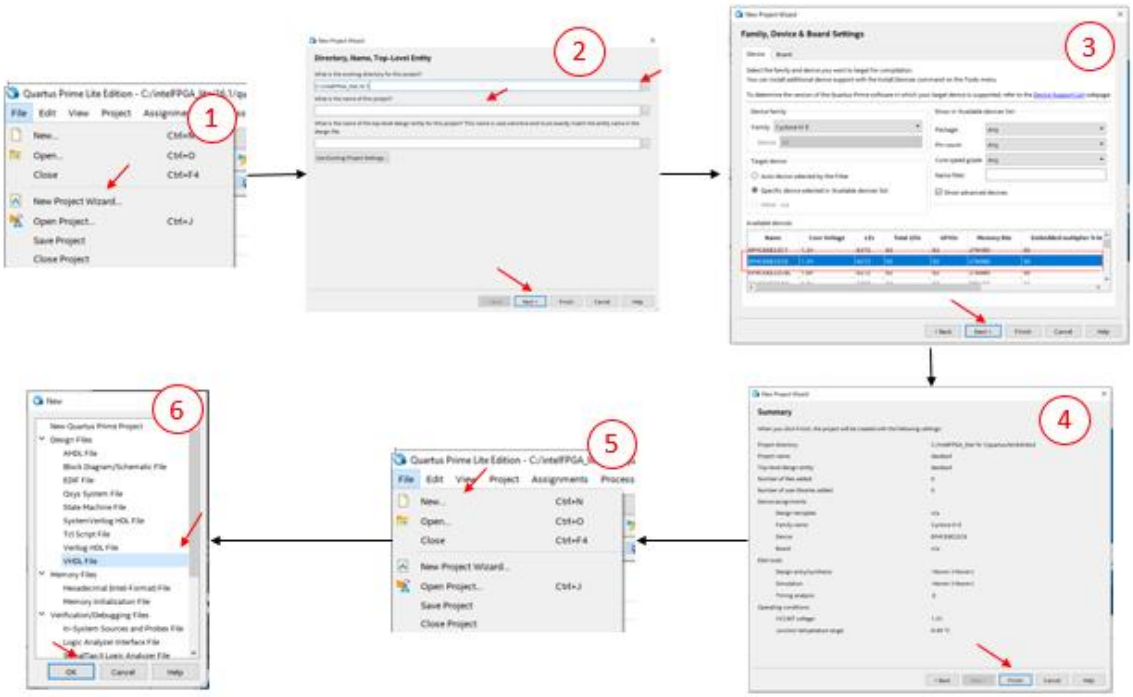
  State OP atau operasi merupakan inti dari pengoperasian perhitungan pada pemrograman. *State* ini merupakan jalannya perhitungan akar kuadrat dan sekaligus menghasilkan nilai *output* yang akan diperoleh. *State* ini akan mengurangi nilai D sampai D = '0' yang menunjukkan *state* ini akan berpindah ke *state* FINISH. Nilai D merupakan jumlah iterasi pada perhitungan akar kuadrat.
- FINISH

  State FINISH merupakan keadaan perhitungan akar kuadrat telah selesai. *State* ini ditandai dengan nilai D = '0'. *State* ini tidak akan berpindah jika nilai start ='1'. *State* ini akan berpindah ke INIT jika start ='0' dan membaca nilai *input* baru.

Selanjutnya penulisan program algoritme dengan cara *state machine* yang dilakukan sesuai dengan spesifikasi sistem. Penulisan dilakukan berulang jika terjadi *Error* dan tidak sesuai dengan spesifikasi sistem.

3.4 Penulisan Algoritme Menggunakan Bahasa VHDL

program ditulis menggunakan perangkat lunak Quartus prime 16.1 *Lite Edition*. Bahasa yang digunakan pada penulisan program adalah bahasa VHDL. Langkah-langkah pembuatan *project* pada perangkat lunak perangkat Quarus Prime 16.1 *Lite Edition* terdapat pada gambar 3.5 sebagai berikut.



Gambar 3.5 Membuat *project* di Quarus Prime 16.1 *Lite Edition* dengan bahasa program VHDL

Keterangan :

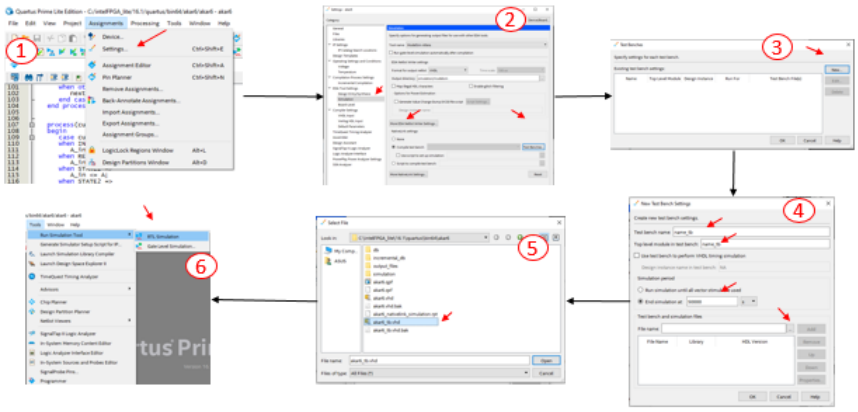
1) Setelah perangkat lunak Quartus Prime 16.1 *Lite Edition* terbuka, untuk membuat *project* baru dengan cara menekan pada menu *File* > *New Project Wizard* pada gambar nomor 1.
2) Setelah muncul jendela pada nomor 2 isi lokasi *direktori project* di lokasi yang diinginkan (dapat membuat folder), kemudian ini nama *project* yang diinginkan tidak mengandung spasi dan diawali dengan huruf, kemudian tekan *next* sampai muncul jendela pada gambar nomor 3.
3) Setelah muncul jendela nomor 3 isikan *device family* sesuai dengan FPGA yang digunakan, pada penelitian ini perangkat yang akan digunakan adalah Cyclone IV E dengan nama seri EP4CE6E22C8 seperti pada jendela nomor 3 kemudian tekan *next*.
4) Maka akan muncul tampilan *EDA Tool Setting*, kemudian tekan *next* sehingga diperoleh tampilan pada jendela nomor 4 yaitu bagian akhir/ *summary* dari *New Projet Wizard* lalu tekan *Finish*.
5) Selanjutnya untuk membuat VHDL *file* dapat dicari menekan *keybord* ctrl + N atau menekan menu *File* > *New*.. seperti pada gambar nomor 5.
6) Setelah itu maka muncul jendela seperti pada nomor 6, maka pilih *Design* VHDL *file* untuk bahasa program VHDL.

3.5 Verifikasi Sistem

Proses verifikasi sistem dilakukan dalam dua tahap, yaitu verifikasi fungsional pada simulasi program menggunakan aplikasi ModelSim-Altera dan verifikasi fungsional pada *hardware* dengan modul FPGA Altera Cyclone IV EP4CE6E22C8.

3.5.1 Verifikasi Fungsional Program

Verifikasi fungsional pada program dilakukan untuk melihat bahwa algoritme program yang disimulasikan sesuai dengan yang diinginkan. Hal ini dilakukan dengan membandingkan antara hasil dari hasil simulasi dengan perhitungan manual. Ketika ada perbedaan antara perhitungan manual dengan simulasi akan dilakukan perbaikan pada algoritme program dan dilakukan perbandingan kembali hingga hasil dari program sama dengan hasil dari perhitungan manual. Pengujian dilakukan menggunakan *Test Bench* agar dapat melakukan pengujian banyak *input* sehingga memudahkan dalam pengujian. Langkah-langkah simulasi dengan cara *Test Bench* ditunjukkan pada Gambar 3.6 pada berikut.



Gambar 3.6 Langkah-langkah simulasi menggunakan *Test Bench*

Keterangan:

1) Penelitian ini melakukan simulasi fungsional dengan menggunakan *Test Bench*, sehingga diperlukan penulisan program *Test Bench* terlebih dahulu. Setelah selesai penulisan tekan menu *Assigment* > *Settings*.. seperti pada Gambar 3.6 nomor 1.
2) Kemudian muncul jendela *settings*, pilih *Category Simulasion* lalu pilih *Compile Test Bench* dan tekan *Test Benches*,
3) Kemudian pada jendela *Test Benches* tekan *New*.
4) Selanjutnya isi nama dan *top level* pada *New Test Bench* dan isi panjang simulasi tersebut.
5) Kemudian masukan *file Test Bench* yang telah dibuat lalu tekan *OK* dan *Apply*.

6) Setelah sudah dimasukkan *file Test Bench* maka dilakukan simulasi, tekan menu *Tools* > *Run Simulation Tool* > *RTL Simulation.*

Setelah selesai melakukan pengujian maka dilakukan mencari selisih dan persentase *Error* dari hasil simulasi. Hal ini dilakukan agar mengetahui dampak penggunaan bilangan pecahan dalam perhitungan akar kuadrat. Cara untuk mencari selisih seperti pada persamaan (3.1):

$$n = P - Q \tag{3.1}$$

Dan untuk mencari persentase *Error* dari hasil simulasi seperti pada persamaan (3.2):

$$E = \frac{P-Q}{P} \times 100\% \tag{3.2}$$

Keterangan :

n : Selisih dari hasil perhitungan kalkulator dengan hasil simulasi

P : Hasil perhitungan akar kuadrat menggunakan kalkulator

Q : Hasil perhitungan akar kuadrat dari simulasi

E : Persentase *Error* dari hasil simulasi dengan hasil kalkulator

3.5.2 Verifikasi Fungsional *Hardware*

Sedangkan verifikasi fungsional pada *hardware* dilakukan penambahan algoritme untuk menyesuaikan I/O *board* pada FPGA. Algoritme tambahan tersebut adalah algoritme untuk menampilkan dan memasukan nilai I/O perhitungan akar kuadrat dari atau ke-*board* FPGA ke program utama. Penelitian ini menggunakan *port* reset sebagai reset, *port clock* sebagai *clock*. Pada modul FPGA terdapat 4 buah *port push button*, pada penelitian ini memakai semua *push button* yang dimiliki modul dan masing – masing *port push button* memiliki fungsi yang berbeda-beda yaitu *port* S1 *push button* sebagai start, *port* S2 *push button* untuk menambah nilai *input*, *port* S3 *push button* untuk mengurangi nilai *input*, *port* S4 *push button* untuk mengganti tampilan *seven segment* menjadi tampilan *input* atau *output*. Kemudian 4 buah *port seven segment* digunakan untuk melihat nilai *input* dan *output*, karena pada modul FPGA tersebut hanya memakai 4 buah *seven segment* maka ada keterbatasan dalam penampilan *input* dan *output*. 2 buah *seven segment* digunakan untuk menampilkan nilai bilangan bulat dan 2 buah *seven segment* digunakan untuk menampilkan pecahannya. Dan sebuah LED sebagai indikator proses telah selesai.

3.6 Evaluasi

Pada evaluasi dilakukan perbandingan dengan peneliti lain melibatkan hasil *resource* yang didapatkan yaitu perbandingan *Logic Element* dan *Delay* terbesar atau terburuk. Pada perbandingan *Logic Element* jumlah *input* yang digunakan adalah 64-bit. 62-bit, dan 32-bit. Jumlah *Logic Element* mempengaruhi kecepatan komputasi dan penggunaan sumber daya

perangkat keras sehingga lebih sedikit jumlah *Logic Element* yang di peroleh maka penggunaan sumber daya perangkat keras lebih sedikit. Untuk mengetahui total *Logic Element* yang digunakan dapat dilihat pada hasil *report* simulai.

Kemudian perbandingan *Delay* terbesar atau terburuk antar *register*. Pada perbandingan *Delay* terbesar jumlah *input* yang digunakan adalah 62-bit, 16-bit, 8-bit dan 4-bit. Jumlah *Delay* terbesar mempengaruhi kecepatan frekuensi *switching* sehingga jika lebih kecil *Delay* yang dimiliki maka lebih tinggi frekuensi *switching* yang dimana dibutuhkannya untuk penerapan DTC yang membutuhkan frekuensi *switching* yang tinggi. *Delay* tersebut dapat diketahui dari hasil *report* simulasi pada frekuensi terbesar pada suhu tertinggi. Untuk mencari nilai *Delay* dapat dilihat dari persamaan (3.3):

$$T_m = \frac{1}{F_m} \tag{3.3}$$

Keterangan:

Tm : *Delay* terbesar atau terburuk antar *register*

Fm : Frekuensi terbesar pada suhu tertinggi

BAB 4
HASIL DAN PEMBAHASAN

4.1 Hasil Perancangan Pada Quartus

Pada penulisan program menggunakan quartus dengan FSM menggunakan 4 *state*/keadaan yaitu INIT, SHIFT, OP dan FINISH1. Penulisan perintah *state* dapat dilihat pada Gambar 4.1

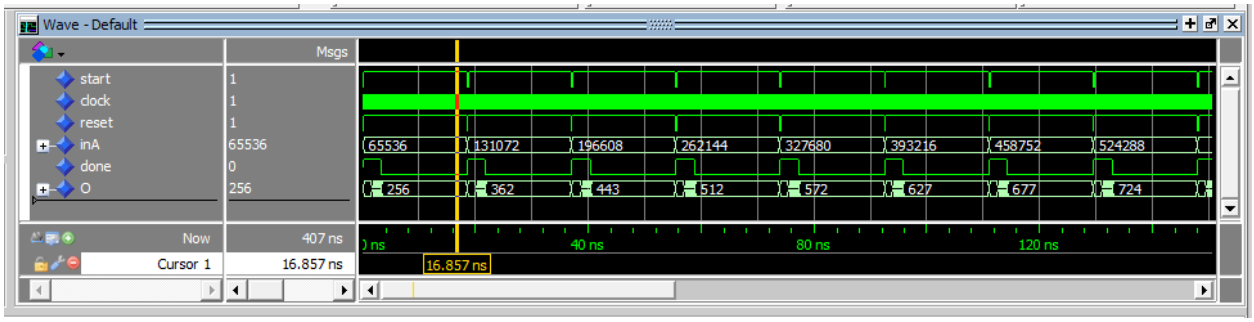
```
21    type states is (INIT, SHIFT, OP, FINISH1);
52    process (currentstate, start, D)
53    begin
54       case currentstate is
55       when INIT =>
56          if (start = '1') then
57             nextstate <= INIT;
58          else
59             nextstate <= SHIFT;
60          end if;
61       when SHIFT =>
62          nextstate <= OP;
63       when OP =>
64          if (D = 0) then
65             nextstate <= FINISH1;
66          else
67             nextstate <= SHIFT;
68          end if;
69       when FINISH1 =>
70          if (start = '0') then
71             nextstate <= INIT;
72          else
73             nextstate <= FINISH1;
74          end if;
75       when others =>
76          nextstate <= INIT;
77       end case;
78    end process;
79
```

Gambar 4.1 Algoritme *State machine*

Gambar 4.1 adalah bagian arsitektur FSM pada algoritme perhitungan akar kuadrat. Pada Gambar 4.1 dapat diketahui bahwa setiap *state* berpindah sesuai dengan aturannya, seperti pada *state* INIT berpindah ke *state* SHIFT jika start = 0 jika tidak maka tetap di INIT, *state* SHIFT berpindah setiap *clock*, *state* OP ke FINISH1 jika D = 0 dan jika tidak maka pindah ke SHIFT, dan *state* FINISH berpindah ke INIT jika start = 0 dan jika tidak maka maka tetap di INIT. Pada satu siklus ini hanya terdapat 1 *state* atau 1 keadaan setiap *clock.*

4.2 Verifikasi Fungsional Program dengan Simulasi

Proses verifikasi fungsional program dengan menggunakan *Test Bench* dengan 22 *input* simulasi yang berbeda. *Input* simulasi bernilai 1, 2, 3, 4, 5, 6, 7, 8, 9, 22, 39, 72, 137, 265, 534, 1063, 2120, 4233, 8201, 16406, 32807, dan 65535,99609375. Hasil simulasi seperti pada Gambar 4.2.



Gambar 4.2 Hasil Simulasi Menggunakan Perangkat Lunak MODELSim Altera

Gambar 4.2 adalah hasil simulasi yang menggunakan *Test Bench*. Dapat dilihat bahwa "inA" merupakan nilai *input* akar kuadrat dan "O" merupakan *output* akar kuadrat. Pada Gambar 4.2 menampilkan nilai I/O yang berbeda dengan tujuan hasil yang dicapai oleh peneliti dikarenakan sebagian bit digunakan untuk nilai pecahannya. Hal ini dilakukan karena pada simulasi tidak bisa menerjemahkan bit pecahannya. Sehingga dibutuhkannya penjabaran untuk mengetahui nilai sebenarnya pada simulasi. Nilai-nilai pada Gambar 4.2 dijabarkan pada Tabel 4.1.

Tabel 4.1 Penjabaran Hasil dari Simulasi

Input Simulasi	*Output Simulasi*	Hasil Kalkulator	Selisih	*Error* %
1,0	1	1	0	0%
2,0	1,4140625	1,414213562	$1{,}5106 \times 10^{4}$	0,01%
3,0	1,73046875	1,732050807	$1{,}58205 \times 10^{3}$	0.091%
4,0	2,00	2	0	0%
5,0	2,234375	2,236067977	$1{,}69297 \times 10^{3}$	0,075%
6,0	2,449218750	2,449489742	$2{,}7099 \times 10^{4}$	0.011%
7,0	2,64453125	2,645751311	$1{,}22006 \times 10^{3}$	0,046%
8,0	2,828125	2,828427124	$3{,}02124 \times 10^{4}$	0,01%
9,0	3	3	0	0%
22	4,6875	4,69041576	$2{,}91576 \times 10^{3}$	0,062 %
39	6,2421875	6,244998	$2{,}8105 \times 10^{3}$	0,045%
72	8,484375	8,48528137	$9{,}0637 \times 10^{4}$	0,01%
137	11,703125	11,7046999	$1{,}574942 \times 10^{3}$	0,013%
265	16,27734375	16,2788206	$1{,}476846 \times 10^{3}$	0,009%
534	23,10546875	23,10544	$2{,}9712665 \times 10^{3}$	0,012%
1063	32,6015625	32,8966026	$2{,}07135 \times 10^{3}$	0,006%
2120	46,04296875	46,0434577	$4{,}889828 \times 10^{4}$	0.001%
4233	65,05859375	65,0615094	$2{,}9156084 \times 10^{3}$	0.004%
8201	90,55859375	90,55859375	$7{,}790393 \times 10^{4}$	0.0008%
16406	128,08203125	128,085909	$3{,}8774207 \times 10^{3}$	0.003%
32807	181,125	181,127227	$2{,}0272488 \times 10^{3}$	0.0011%
65535,99609375	255,99609375	255,999992	$3{,}8986206 \times 10^{3}$	0.0015%

Tabel 4.1 adalah penjabaran dari simulasi pada Gambar 4.2 dan hasil perhitungan kalkulator. *input* simulasi merupakan penjabaran dari "inA" atau nilai *input* akar kuadrat dan *output* simulasi merupakan penjabaran dari "O" atau merupakan nilai *output* akar kuadrat pada *Test Bench*. Dari Tabel 4.1 diketahui sistem ini memiliki nilai *Error* yang kecil sehingga nilai *output* simulasi mendekati nilai perhitungan kalkulator. Persentase *Error* yang dimaksud adalah persentase *Error* dari hasil perhitungan simulasi yaitu *output* simulasi dengan perhitungan kalkulator. Pada tabel

diatas diketahui bahwa *Error* yang diperoleh bervariasi ada keterkaitannya nilai *Error* semakin kecil jika nilai *input* simulasi besar. *Error* terjadi karena terbatasnya nilai bit *output* simulasi pecahan, pada simulasi tersebut *Error* terbesar yang di dapatkan pada Tabel 4.1 adalah 0,091% dan selisih terbesar adalah 0,0038986206. *Error* terjadi karena nilai bit *output* simulasi memiliki keterbatasan yaitu 16-bit, maka pembagian bit *output* menjadi 8-bit untuk bilangan bulat dan 8-bit digunakan untuk pecahannya. Sehingga selisih terbesar yang akan didapatkan sistem adalah dengan bilangan biner 0,00000001 dan jika dikonversikan ke desimal menjadil 0,00390625. Total *resource* yang didapatkan pada sistem ini ditunjukan oleh Tabel 4.2.

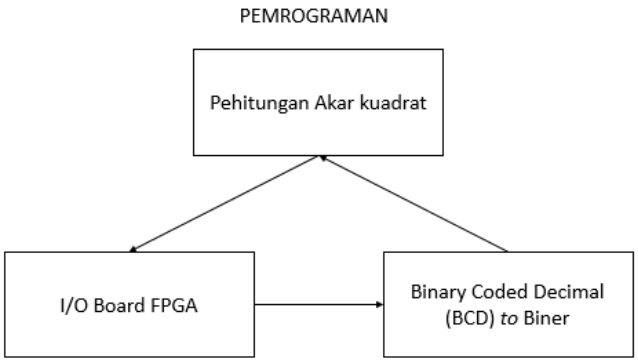
Tabel 4.2 Hasil *Resource*

Resource	Jumlah
Logic Element	157
Registers	120
Pins	52
Fmax	151,49 MHz

Pada tabel 4.2 adalah hasil *resource* pada program perhitungan akar kuadrat. Sistem ini tidak menggunakan *Memory bits* dan *Embedded multiplier* 9-bit *element*. Pin yang digunakan pada sistem ini adalah 1 pin start, 1 pin *clock*, 1 pin reset, 32 pin *input* simulasi, 16 pin *output* simulasi dan 1 pin *Enable.* Pada *Logic Element* yang didapatkan menghasilkan hasil yang bagus dikarenakan *Logic Element* yang digunakan sebanyak 3% dari total yang dimiliki *hardware*. hal ini merupakan penggunaan yang hemat sehingga dapat mengurangi penggunaan sumber daya pada FPGA. Pada Frekuensi maksimal yang ditampilkan pada Tabel 4.2 adalah frekuensi maksimal pada suhu tertinggi atau pada suhu 85°C.

4.3 Hasil Perancangan FPGA

Mengimplementasi perancangan ke FPGA untuk menampilkan hasil perhitungan akar kuadrat pada FPGA dibutuhkan program tambahan. Ada 2 pemrograman tambahan untuk mengimplementasikan perhitungan akar kuadrat yaitu I/O pada Board FPGA dan *Binary Coded Decimal* (BCD) *to* Biner. Keterkaitan kedua pemrograman tersebut dapat dilihat pada Gambar 4.3.



Gambar 4.3 Siklus Jalan I/O

Pada gambar 4.3 merupakan proses pemrograman berkerja. Masing – masing pemrograman memiliki fungsi yang bebeda-beda. Pada pemrograman I/O Board FPGA memberikan nilai *input* dalam bentuk BCD ke pemrograman BCD *to* Biner, dan menampilkan nilai *output* hasil simulai perhitungan akar kuadrat pada pemrograman perhitungan akar kuadrat. Sedangkan pemrograman BCD *to* Biner memiliki fungsi mengubah nilai *input* BCD dari pemrograman I/O *board* FPGA menjadi biner dan diteruskan ke pemrograman perhitungan akar kuadrat.

4.4 Verifikasi Fungsional *Hardware*

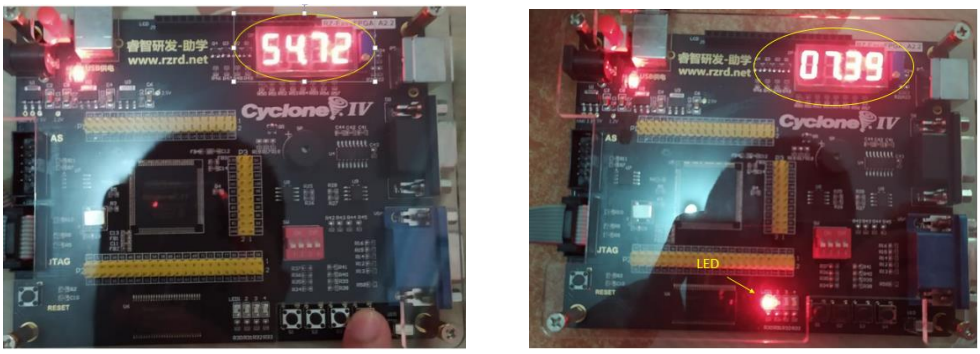
Proses verifikasi *hardware* menggunakan *port* pada *board* FPGA sebagai *input* yang digunakan adalah *clock*, *port* reset, dan 4 buah *port push button*, sedangkan untuk *output* adalah 4 buah *port seven segment* dan 1 buah *port* LED. Verifikasi dilakukan untuk mengetahui fungsi dari *port* tersebut sesuai dengan spesifikasi sistem. *Push button* pada *board* memiliki fungsi yang berbeda-beda, dapat dilihat fungsi *push button* pada Tabel 4.3.

Tabel 4.3 Kegunaan *Port Push button*

Port Push button	Fungsi
S1	*Counter up*
S2	*Counter Down*
S3	Start
S4	Mengganti penampilan *Seven segment*

Pada *push button* S1 dan S2 berfungsi untuk menaikan atau menurunkan nilai *input* yang dapat dilihat pada *seven segment*. Hal ini dilakukan secara bersamaan dengan menekan *push button* S4, fungsi dari *push button* S4 adalah menampilkan nilai *input*. Pada *push button* S3 berfungsi untuk memulai proses pada FPGA sehingga menampilkan nilai *output* baru pada *seven segment*.

Pada Bord sistem ini hanya memiliki 4 buah *seven segment* sehingga memiliki keterbatasan dalam menampilkan nilai *input* dan *output*. Dengan nilai *input* tertinggi 99,99 dan nilai *input* terendah 0,01. Berikut adalah proses verifikasi *hardware* pada Gambar 4.4.



Gambar 4.4 Verifikasi *Hardware*: gambar kiri menampilkan nilai *input* dan gambar kanan menampilkan nilai *output*

Pada Gambar 4.4 merupakan tampilan pada *board* nilai *input* dan nilai *output*. Pada indikator LED sebagai indikator bahwa proses telah selesai dan nilai *output* tertera di *seven segment*. Karena keterbatasan *seven segment* nilai *output* memiliki resolusi 0,01 dan akan terjadi *Error* dengan selisih kurang dari 0,01. Hal ini juga dikarenakan jumlah bit *output* pecahan yang seharusnya 8 bit menjadi 7 bit pada penampilan *seven segment*.

4.5 Perbandingan dengan Hasil Penelitian Lain

Perbandingan hasil penelitian lain dilakukan perubahan spesifikasi yang sesuai dengan spesifikasi penelitian yang ingin dibandingkan. Berdasarkan hasil kompilasi untuk mengimplementasikan perhitungan akar kuadrat dengan *input* 32-bit, 62-bit dan 64-bit menghasilkan *Logic Element* (LE) masing-masing sebesar 157, 248 dan 258. Hasil penelitian yang dilibatkan dalam penelitian ini adalah penelitian [5], [6]. Perbandingan penggunaan LE dengan penelitian lain ditunjukan pada Tabel 4.4.

Tabel 4.4 Perbandingan *Logic Element* (LE)

No	Metode	Penggunaan LE		
		64-bit	62-bit	32-bit
1	*Classical-NR* [5]	4092	-	1008
2	*Reduced-Area-NR* [5]	2464	-	632
3	*Modular-NR* [5]	2468	-	624
4	*Simple-X-Module* [5]	2488	-	648
5	Tole Sutikno [5]	1023	-	256
6	AZ Jidin dkk [6]	-	281	-
7	Penelitian ini	258	248	157

Berdasarkan pada pada bit *input* 64-bit, 62-bit, dan 32-bit penelitian ini menggunakan *Logic Element* yang terkecil dibandingkan dengan penelitian lain, sehingga lebih sedikit mengkonsumsi sumber daya, dan juga hal ini membuat perhitungan akar kuadrat pada DTC akang menjadi cepat.

Kemudian membandingkan jumlah *Delay* terbesar dengan penelitian lain. *Delay* terbesar yang dimaksud adalah *Delay* antar register. Hal ini juga didasarkan dengan nilai *input* yang sama dengan penelitian yang ingin dibandingkan. Hasil penelitian yang dilibatkan dalam penelitian ini adalah penelitian [6], [13], [14]. Perbandingan *Delay* terbesar atau terburuk dengan penelitian lain ditunjukan pada Tabel 4.5

Tabel 4.5 Perbandingan *Delay* Terbesar atau Terburuk

No	Aspek	Jumlah *Delay* Terbesar			
		62-bit	16-bit	8-bit	4-bit
1	S Samavi dkk [13]	-	32,023 ns	10,023 ns	6,485 ns
2	AP Ramesh dkk [14]	-	37,166 ns	9,215 ns	-
3	AZ Jidin dkk [6]	12,345 ns	-	-	-
4	Penelitian ini	8,349 ns	4,863 ns	4,133 ns	4,234 ns

Berdasarkan pada Tabel 4.5 membuktikan penelitian ini memiliki *Delay* terburuk yang lebih rendah dibandingkan dengan AZ Jidin dkk, S Samavi dkk dan AP Ramesh dkk. Hasil dari penelitian ini cukup memuaskan dikarenakan dibutuhkan *Delay* terburuk antar register yang bernilai kecil untuk penerapan DTC dalam frekuensi *switching* yang tinggi. Sehingga penelitinan ini memiliki hasil lebih unggul dibandingkan dengan penelitian lain. Hal ini dikarenakan sistem ini memaksimalkan kegunaan lebar bit pada pengoperasiannya.

BAB 5
KESIMPULAN DAN SARAN

5.1 Kesimpulan

Kesimpulan dari penelitian ini adalah seebagai berikut:

1. Implementasi perhitungan akar kuadrat bilangan pecahan berjalan dengan baik, hasil dari implementasi tersebut sesuai dengan spesifikasi sistem yang telah ditentukan.
2. Implementasi perhitungan akar kuadrat dengan algoritme *modified digit by digit non-restoring* pada FPGA berhasil sesuai dengan yang diharapkan yang sederhana sehingga medapatkan nilai dengan sekali pengoperasian. Pada penelitian ini menggunakan algoritme modified *digit by digit non-restoring* yang dikonversikan ke *state machine* sehingga ada dua keadaan atau *state* yaitu penggeseran dan pengoperasian. Dengan menggunakan metode tersebut penelitian ini menggunakan lebih sedikit jumlah *Logic Element* yang digunakan dan *Delay* yang kecil.
3. Pada sistem ini peneliti akan memakai jumlah bit *input* sebanyak 32-bit dan *output* 16-bit, pada bit *input* bilangan bulat memakai 16 bit dan bit pecahan memakai 16 bit. Pada bit *output* bilangan bulat memakai 8 bit dan bit pecahan memakai 8 bit. Resolusi pada nilai *input* sebesar 0,0000152587890625 dan resolusi nilai *output* 0,00390625.
4. Sistem ini menghasilkan *resource* yaitu : 157 *Logic Element*, 120 *Registers*, pin yang digunakan 52 *pins*, 151,49 Mhz, 0 *total memory bits*, dan 0 *embedded multiplier 9-bit element.*
5. Pada tampilan *board* FPGA memiliki nilai *input* terbesar 99,99 dan terkecil 0,01 dengan resolusi *Error* terbesar 0,01.

5.2 Saran

1. Untuk memaksimalkan lebar bit *input* simulasi dapat dilakukan peletakan bit *input* secara fleksibel sehingga sekaligus memaksimalkan kegunaan lebar bit *output* simulasi.
2. Dapat menyederhanakan penulisan program dengan cara yang berbeda sehingga memperkecil *resource* yang didapatkan dari penelitian ini

DAFTAR PUSTAKA

[1] I. Takahashi and T. Noguchi, "A New Quick-Response and High-Efficiency Control Strategy of an Induction Motor," *IEEE Trans. Ind. Appl.*, vol. IA-22, no. 5, pp. 820–827, 1986.

[2] Y. Liu, Z. Zhu, D. H.-T. on I. Applications, and U. 2007, "Commutation-torque-ripple minimization in direct-torque-controlled PM brushless DC drives," *IEEE Trans. Ind. Appl.*, vol. 43, no. 4, pp. 1012–1021, 2007.

[3] T. Sutikno, N. R. Nik Idris, A. Z. Jidin, and A. Jidin, "A Model of FPGA-based Direct Torque Controller," *TELKOMNIKA Indones. J. Electr. Eng.*, vol. 11, no. 2, 2013.

[4] T. Sutikno, "An Efficient Implementation of the Non Restoring Square Root Algorithm in Gate Level," *Int. J. Comput. Theory Eng.*, vol. 3, no. 1, pp. 46–51, 2013.

[5] T. Sutikno, "An Optimized Square Root Algorithm for Implementation in FPGA Hardware," *TELKOMNIKA (Telecommunication Comput. Electron. Control.*, vol. 8, no. 1, p. 1, 2010.

[6] A. Z. Jidin, S. H. Mohamad, S. Ahmad, M. F. Yakub, N. A. N. Azlan, and A. Jidin, "Optimizing The Flux and Torque Estimator of DTC in FPGA by Using Low-Area Square Root Calculator," in *PECON 2016 - 2016 IEEE 6th International Conference on Power and Energy, Conference Proceeding*, 2017, pp. 517–521.

[7] T. Sutikno, N. R. N. Idris, A. Jidin, and M. N. Cirstea, "An Improved FPGA Implementation of Direct Torque Control for Induction Machines," *IEEE Trans. Ind. Informatics*, vol. 9, no. 3, pp. 1280–1290, 2013.

[8] K. N. Vijeyakumar, V. Sumathy, P. Vasakipriya, and A. Dinesh Babu, "FPGA Implementation of Low Power High Speed Square Root Circuits," in *2012 IEEE International Conference on Computational Intelligence and Computing Research, ICCIC 2012*, 2012, pp. 1–5.

[9] S. Lachowicz and H. J. Pfleiderer, "Fast Evaluation of The Square Root and Other Nonlinear Functions in FPGA," in *Proceedings - 4th IEEE International Symposium on Electronic Design, Test and Applications, DELTA 2008*, 2008, pp. 474–477.

[10] M. D. Ercegovac, "On Digit-by-Digit Methods for Computing Certain Functions," in *Conference Record - Asilomar Conference on Signals, Systems and Computers*, 2007, pp. 338–342.

[11] O. Kosheleva, "Babylonian method of computing the square root: Justifications based on fuzzy techniques and on computational complexity," in *Annual Conference of the North American Fuzzy Information Processing Society - NAFIPS*, 2009, pp. 1–6.

[12] T. J. Kwon, J. Sondeen, and J. Draper, "Floating-Point Division and Square Root Implementation Using a Taylor-Series Expansion Algorithm," in *Proceedings of the 15th IEEE International Conference on Electronics, Circuits and Systems, ICECS 2008*, 2008, pp. 702–705.

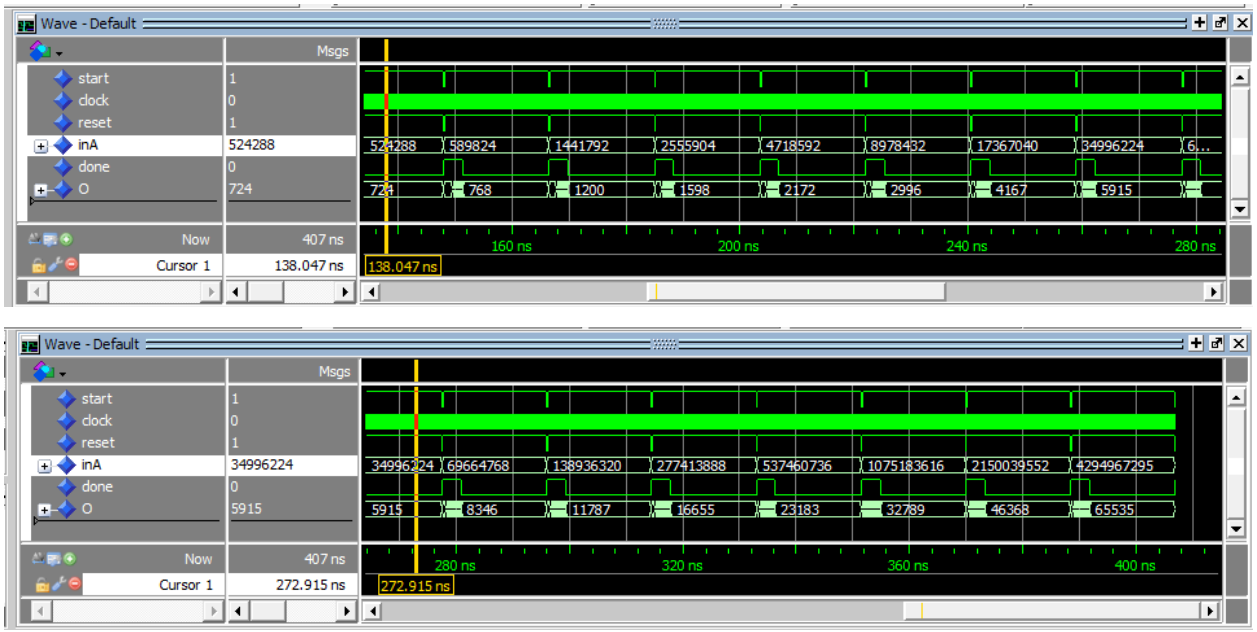
[13] S. Samavi, "Improving Array Structure of Non-Restoring Square Root Circuit," *Int. J. Eng. Sci.*, vol. 14, no. 1, pp. 1–14, 2003.

[14] A. P. Ramesh and I. J. Kumar, "Implementation of Integer Square Root," *Int. J. Eng. Sci. Innov. Technol.*, vol. 4, no. 1, pp. 105–113, 2015.

[15] A. Nanhe, G. Gawali, S. Ahire, and K. Sivasankaran, “Implementation of Fixed and Floating Point Square Root Using Nonrestoring Algorithm on FPGA,” *Int. J. Comput. Electr. Eng.*, vol. 5, no. 5, pp. 533–537, 2013.

LAMPIRAN

Berikut adalah hasil simulasi menggunakan perangkat MODELSim



Gambar 1 Hasil Simulasi yang Lain, Menggunakan Perangkat Lunak MODELSim

Berikut adalah lampiran dari sintaks program VHDL untuk perhitungan akar kuadrat.

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

entity akar is
  port (
        start, clock, reset  : in std_logic;
        inA                  :in std_logic_vector (31 downto 0);
        o                    : out std_logic_vector (15 downto 0);
        done                 : out std_logic
  );
end akar;

architecture Behavioral of akar is
  type states is (INIT, INIT1, SHIFT, OP, FINISH1);
  signal currentstate, nextstate: states;
  signal A, A_in: std_logic_vector (17 downto 0);
  signal A1, A1_in: std_logic_vector (31 downto 0);
  signal B, B_in: std_logic_vector (17 downto 0);
  signal D, D_in: integer;
  signal Z, Z_in: std_logic_Vector (15 downto 0);

begin
  process(clock, reset)
  begin
        if(reset = '0' )  then
              currentstate <= INIT;
              A <= (others => '0');
```

```
                A1 <= (others => '0');
                B <= (others => '0');
                D <= 15;
                Z <= (others => '0');
        elsif(clock'event and clock = '1') then
                currentstate <= nextstate;
                A <= A_in;
                A1 <= A1_in;
                B <= B_in;
                D <= D_in;
                Z <= Z_in;

        end if;
end process;

process (currentstate, start, D)
begin
        case currentstate is
        when INIT =>
                if (start = '1') then
                        nextstate <= INIT;
                else
                        nextstate <= INIT1;
                end if;
        when INIT1 =>
                if (start = '0') then
                        nextstate <= INIT1;
                else
                        nextstate <= SHIFT;
                end if;

        when SHIFT =>
                nextstate <= OP;
        when OP =>
                if (D = 0) then
                        nextstate <= FINISH1;
                else
                        nextstate <= SHIFT;
                end if;
        when FINISH1 =>
                if (start = '0') then
                        nextstate <= INIT;
                else
                        nextstate <= FINISH1;
                end if;
        when others =>
                nextstate <= INIT;
        end case;
end process;


process(currentstate, A1, A, B)
begin
        case currentstate is
        when SHIFT =>
                A_in <= A(15 downto 0) & A1(31 downto 30);
        when OP =>
                if (A < B) then
                A_in <= A;
        else
                A_in <= A - B;
                end if;
```

```
		when others =>
			A_in <= A;

		end case;
end process;

process(currentstate, B, A)
begin
		case currentstate is
		when SHIFT =>
			B_in <= B(16 downto 0) & '1';
		when OP =>
			if (A < B) then
			B_in <= B - "0000000000000000001";
		else
			B_in <= B + "0000000000000000001";
			end if;
		when others =>
			B_in <= B;

		end case;
end process;

process(currentstate, inA, A1)
begin
		case currentstate is
		when INIT =>
			A1_in <= inA;
		when SHIFT =>
			A1_in <= A1(29 downto 0) & "00";
		when OP =>
			A1_in <= A1;
		when others =>
			A1_in <= A1;
		end case;
end process;


process(currentstate, D)
begin
		case currentstate is
		when SHIFT =>
			D_in <= D;
		when OP =>
			D_in <= D - 1;
		when others =>
			D_in <= D;
		end case;
end process;

process(currentstate, Z, A, B)
begin
		case currentstate is
		when SHIFT=>
			Z_in <= Z;
		when OP =>
			if (A < B) then
			Z_in <=  Z(14 downto 0) & '0';
		else
			Z_in <= Z(14 downto 0) & '1';
			end if;
		when others =>
```

```
                        Z_in <= Z;
                end case;
    end process;

    o <= z;
    process (currentstate)
    begin
                case currentstate is
                when FINISH1 =>
                        done <= '0';
                when others =>
                        done <= '1';
                end case;
    end process;
end Behavioral;
```