UNIVERSITAS INDONESIA

DESAIN FUNCTION GENERATOR BERBASIS PLD (FPGA)

SKRIPSI

Diajukan sebagai salah satu syarat untuk memperoleh gelar sarjana sains

DONY HARRIS SAROSO
0304020205

FAKULTAS MATEMATIKA DAN ILMU PENGETAHUAN ALAM
PROGRAM STUDI FISIKA
PEMINATAN FISIKA INSTRUMENTASI
DEPOK
MEI 2009

HALAMAN PERNYATAAN ORISINALITAS

Skripsi ini adalah hasil karya saya sendiri,

dan semua sumber baik yang dikutip maupun dirujuk

telah saya nyatakan dengan benar

Nama : Dony Harris Saroso

NPM : 0304020205

Tanda tangan :

Tanggal : 14 Juni 2009

HALAMAN PENGESAHAN

Skripsi ini diajukan oleh	:
Nama	: Dony Harris Saroso
NPM	: 0304020205
Program Studi	: Fisika Instrumentasi
Judul Skripsi	: Desain Function Generator Berbasis PLD (FPGA)

Telah berhasil dipertahankan di hadapan Dewan Penguji dan diterima sebagai bagian persyaratan yang diperlukan untuk memperoleh gelar Sarjana Sains pada Program Studi Fisika, Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Indonesia

Dewan Penguji

Pembimbing	: Dr. Prawito	(……)
Pembimbing	: Lingga Hermanto, M.Si	(……)
Penguji	: Dr. Santoso S	(……)
Penguji	: Dr. B.E.F da Silva	(……)

Ditetapkan di : Depok

Tanggal : 14 Juni 2009

KATA PENGANTAR

Segala puji dan syukur Penulis panjatkan kehadirat Allah SWT, atas berkat rahmat, nikmat dan karunia-Nyalah Penulis dapat menyelesaikan skripsi ini. Penulisan skripsi ini dilakukan dalam rangka memenuhi salah satu syarat untuk memperoleh gelar sarjana sains Fakultas Matematika dan Ilmu Pengetahuan Alam, Universitas Indonesia.

Pada kesempatan ini, Penulis ingin menyampaikan penghargaan dan rasa terima kasih Penulis kepada semua pihak yang telah membantu dalam proses penyusunan skripsi ini. Dengan ketulusan hati Penulis menyampaikan rasa syukur Penulis kepada Allah SWT, dengan telah memberikan nikmat yang tak terhitung jumlahnya pada Penulis hingga saat ini, juga Nabi Muhammad SAW, serta tak lupa rasa terima kasih Penulis tujukan kepada:

1. Keluarga besar dari Penulis, kedua orangtua Penulis yang selalu memberikan segala bantuan
2. Dr. Prawito dan Lingga Hermanto, M.Si sebagai pembimbing skripsi ini, yang banyak memberikan masukan yang sangat bermanfaat dan membantu Penulis.
3. Teman-teman Fisika angkatan 2004, Sugi, Zamroni, Budi, dan yang tak mungkin dapat disebutkan satu persatu.

Serta kepada seluruh pihak yang tidak mungkin dapat disebutkan semuanya namun memberikan kontribusi yang cukup berarti pada penyusunan skripsi ini.

Akhir kata, Penulis hanya mampu berdoa dan berharap, semoga Allah SWT membalas segala kebaikan semua pihak yang telah membantu. Semoga skripsi ini membawa manfaat bagi pengembangan ilmu.

Depok, Juni 2009

Dony Harris Saroso

HALAMAN PERNYATAAN PERSETUJUAN PUBLIKASI TUGAS AKHIR UNTUK KEPENTINGAN AKADEMIS

Sebagai sivitas akademik Universitas Indonesia, saya yang bertanda tangan di bawah ini:

Nama : Dony Harris Saroso

NPM : 0304020205

Program Studi : Fisika Instrumentasi

Departemen : Fisika

Fakultas : Matematika dan Ilmu Pengetahuan Alam

Jenis Karya : Skripsi

demi pengembangan ilmu pengetahuan, menyetujui untuk memberikan kepada Universitas Indonesia **Hak Bebas Noneksklusif (*NON-exclusif Royalty-Free Right*)** atas karya ilmiah saya yang berjudul:

Desain Function Generator Berbasis PLD (FPGA)

Beserta perangkat yang ada (jika diperlukan). Dengan Hak Bebas Royalti Noneksklusif ini Universitas Indonesia berhak menyimpan, mengalihmedia/format-kan, mengelola dalam bentuk pangkalan data (*database*), merawat, dan memublikasikan tugas akhir saya selama tetap mencantumkan nama saya sebagai penulis/pencipta dan sebagai pemilik Hak Cipta.

Demikian pernyataan ini saya buat dengan sebenarnya.

Dibuat di : Depok

Pada Tanggal : 14 Juni 2009

Yang menyatakan

(Dony Harris Saroso)

ABSTRAK

Nama : Dony Harris Saroso
Program Studi : Fisika
Judul : Desain Function Generator Berbasis PLD (FPGA)

FPGA (Field-Programmable Gate Array) memiliki banyak sekali aplikasi, salah satunya adalah frequency generator. Frekuensi generator ini menghasilkan gelombang kotak dengan rentang frekuensi sebesar 1Hz-100MHz dengan resolusi 1Hz dengan menggunakan teknik Direct Digital Synthesis (DDS). FPGA ini diprogram menggunakan bahasa VHDL (VHSIC Hardware Description Language). Di dalam FPGA diprogram sebuah ”soft” mikrokontroler untuk interface dengan pengguna dalam mengatur frekuensi yang diinginkan dan juga untuk mengkalkulasi bilangan-bilangan untuk mengontrol frekuensi. Frekuensi dapat diatur dengan sebuah rotary encoder / switch, dan ditampilkan di layar LCD.

Kata kunci: FPGA, function generator, DDS, VHDL

ABSTRACT

Name : Dony Harris Saroso
Departement : Fisika
Judul : PLD (FPGA) Based Function Generator

FPGA (Field-Programmable Gate Array) has many applications, one of the example is function generator. This function generator produced 1Hz-100MHz square wave with resolution of 1Hz, using Direct Digital Synthesis (DDS) technique. This FPGA programmed using VHDL (VHSIC Hardware Description Language). Inside the FPGA a soft microcontroller is programmed, to provide the human interface to frequency and perform high precision calculation in frequency synthesizer. The frequency can be with a rotary encoder / switch, and displayed in LCD.

Keywords: FPGA, function generator, DDS, VHDL

DAFTAR ISI

DAFTAR GAMBAR

DAFTAR TABEL

BAB 1
PENDAHULUAN

1.1 Latar Belakang

Dalam dunia elektronik, telah lumrah digunakan embedded system dengan hanya menggunakan mikrokontroller single-chip. Mikrokontroler ini memiliki kapabilitas memproses (processing capability) 8 / 16 / 32-bit, dan juga peripheral set seperti ADC, timer/counter, dan network: I^2C, CAN, SPI, dan UART. Untuk kebanyakan aplikasi, mikrokontroller ini sudah mencukupi. Sedangkan untuk aplikasi dimana dibutuhkan kecepatan yang lebih dan peripheral tambahan, digunakan FPGA (Field-Programmable Gate Array).

FPGA pada awalnya direncakan untuk digunakan sebagai *glue logic* (kumpulan logika-logika yang dapat diatur). Dewasa ini FPGA telah merambah ke bidang-bidang yang sebelumnya tidak direncanakan digunakan untuk FPGA. FPGA memiliki banyak sekali aplikasi, salah satunya adalah function generator.

Function generator ini berfungsi untuk menghasilkan sinyal / gelombang (kotak, segitiga, sinus) dengan nilai frekuensi yang bisa diatur. Untuk mengetahui respon frekuensi suatu alat elektronik digunakan function generator. Dari respon frekuensi ini bisa ditindaklanjuti oleh desain, testing, troubleshooting, dan perbaikan alat elektronik tersebut. Terdapat banyak sekali tipe-tipe function generator, dengan tujuan dan aplikasi yang berbeda-beda. Pada umumnya hampir tak ada alat yang sesuai untuk semua aplikasi.

Function generator diaplikasikan pada FPGA dengan menggunakan metode Direct Digital Synthesis (DDS). DDS adalah metode membuat gelombang (secara digital) dengan frekuensi yang bermacam-macam, dari sumber frekuensi tunggal (crystal oscillator).

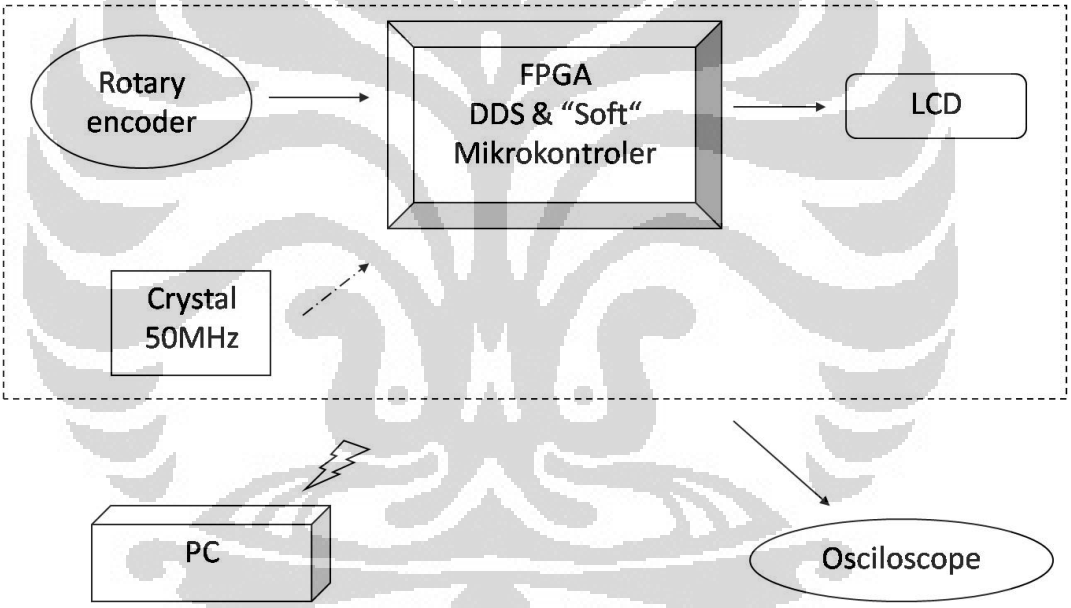
1.2 Tujuan Penelitian

Adapun tujuan penelitian ini adalah :

1. Memahami Function Generator
2. Memahami cara kerja FPGA, baik pemrogamannya maupun perangkat kerasnya
3. Mempelajari dan mengembangkan sistem

1.3 Pembatasan Masalah

Batasan masalah yang dibahas dalam tulisan ini mengenai cara kerja alat, pembuatan alat, pembuatan program, pengambilan data dari hasil yang didapatkan.



Gambar 1.1 Blok Fungsi Kerja Rangkaian Sistem

1.4 Metode Penelitian

Metode penelitian yang akan dilakukan terdiri dari beberapa tahap antara lain :

1. Studi Literatur

   Metode Studi Literatur ini digunakan penulis untuk memperoleh teori-teori dasar sebagai sumber dan acuan dalam penulisan skripsi. Informasi dan pustaka yang berkaitan dengan masalah ini diperoleh dari

literatur, penjelasan yang diberikan dosen pembimbing, rekan-rekan mahasiswa, internet, *data sheet* dan buku-buku yang berhubungan dengan tugas akhir penulis.

2. Perancangan dan Pembuatan Alat

   Perancangan alat merupakan tahap awal penulis untuk mencoba, memahami, menerapkan dan menggabungkan semua literatur yang telah diperoleh dan dipelajari untuk melengkapi sistem serupa yang pernah dikembangkan, sehingga untuk selanjutnya penulis dapat merealisasikan system sesuai dengan tujuan.

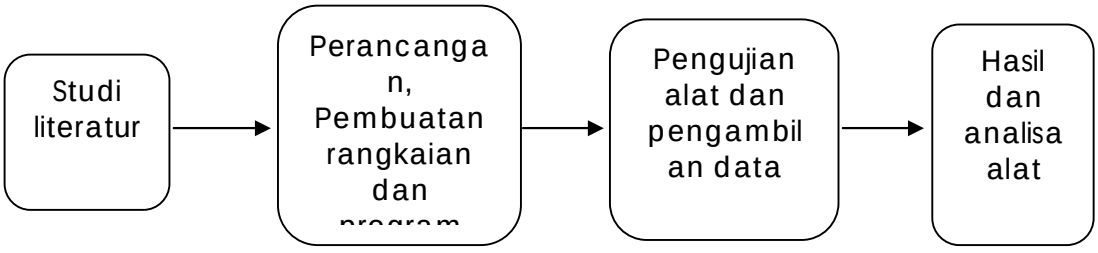
3. Pengujian Sistem

   Pengujian sistem ini berkaitan dengan pengujian alat serta pengambilan data dari alat yang telah dibuat. Pengujian ini dilakukan untuk mengetahui karakteristik dari masing-masing alat, sehingga dapat diketahui bagaimana kinerja alat dan sejauh mana tingkat keakuratan dari alat yang telah dibuat.

4. Metode Analisis

   Metode ini merupakan pengamatan terhadap data yang telah diperoleh dari pengujian alat serta pengambilan data. Setelah itu dilakukan penganalisisan sehingga dapat ditarik kesimpulan dan saran – saran untuk pengembangan lebih lanjut.

   Berikut ini adalah diagram langkah-langkah yang yang akan dilakukan dalam penelitian ini :



Gambar 1.2 Diagram langkah-langkah penelitian

1.5 Sistematika Penulisan

Sistematika penulisan skripsi ini terdiri dari bab-bab yang memuat beberapa sub-bab. Untuk memudahkan pembacaan dan pemahaman maka penulisan skripsi ini ini terdiri atas 5 bab dan secara garis besar dapat diuraikan sebagai berikut :

BAB I PENDAHULUAN

Pendahuluan berisi latar belakang, permasalahan, batasan masalah, tujuan penulisan, metode penulisan dan sistematika penulisan dari skripsi ini.

BAB II TEORI DASAR

Teori Dasar berisi landasan teori sebagai hasil dari studi literatur yang berhubungan dengan perancangan dan pembuatan alat (hardware) serta pembuatan program (software).

BAB III RANGKAIAN DAN RANCANGAN PROGRAM

Pada bab ini akan dijelaskan sistem kerja keseluruhan dari semua perangkat control (hardware) dan program penghubung (software) yang terlibat.

BAB IV PENGUJIAN SISTEM DAN ANALISA

Bab ini menjelaskan tentang unjuk kerja alat sebagai hasil dari perancangan sistem. Pengujian akhir ini dilakukan dengan menyatukan seluruh bagian dari sistem sehingga dapat diketahui apakah sistem dapat berfungsi dengan baik. Setelah sistem dapat bekerja dengan baik maka dilakukan pengambilan data untuk menentukan kapabilitas dari sistem yang dibangun.

BAB V KESIMPULAN DAN SARAN

Bab penutup ini berisi kesimpulan penulis yang diperoleh berdasarkan pengujian sistem dan pengambilan data selama penelitian

berlangsung, selain itu penutup juga berisikan tentang saran-saran dari penulis untuk mendapatkan hasil yang lebih baik dalam pengembangan lebih lanjut dari penelitian ini baik dari perangkat keras (hardware) maupun perangkat lunak (software).

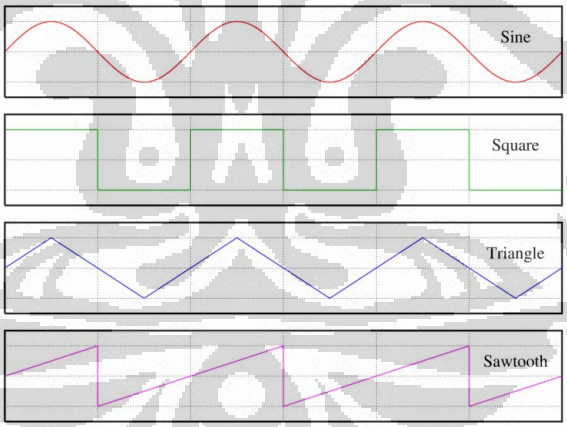
BAB 2
TEORI DASAR

Pada bab ini berisi pembahasan teori dasar yang berhubungan dengan penelitian yang dilakukan. Teori dasar yang akan dibahas pada bab ini yaitu penjelasan perangkat keras (hardware) yang digunakan.

2.1 FUNCTION GENERATOR

Function generator (frequency / waveform generator) adalah alat penguji elektronik atau software yang digunakan untuk menghasilkan gelombang / sinyal elektronik, baik yang berulang maupun yang tidak, digital maupun analog.Function generator terdiri dari electronic oscillator, rangkaian yang dapat menghasilkan gelombang yang repetitif.

Function generator modern menggunakan digital signal processing untuk men-synthesize gelombang, lalu ke DAC, untuk menghasilkan output analog.



Gambar 2.1 Gelombang sinus, segitiga, kotak, dan gergaji.

Sumber : http://commons.wikimedia.org/wiki/File:Waveforms.png

Function generator biasanya digunakan dalam hal perbaikan dan desain rangkaian elektronik, dimana digunakan untuk menstimulasi / mengetes rangkaian. Alat seperti osiloskop digunakan untuk mengukur output rangkaian (wikipedia, 2009).

Gambar 2.2 Contoh Function Generator

Sumber : http://en.wikipedia.org/wiki/File:DDS_function_generator.jpg

2.2 FPGA (FIELD-PROGRAMMABLE GATE ARRAY)

FPGA adalah komponen / alat (berbasis) semikonduktor yang dapat dikonfigurasikan – oleh karena itu dinamakan "field-programmable". FPGA diprogram dengan menggunakan diagram rangkaian logika atau menggunakan HDL (hardware description language), untuk menetapkan bagaimana chip-nya bekerja.

Gambar 2.3 Contoh FPGA

Sumber : http://en.wikipedia.org/wiki/File:Altera_StratixIVGX_FPGA.jpg

FPGA memiliki komponen gerbang terprogram (programmable logic) dan sambungan terprogram. Komponen gerbang terprogram yang dimiliki meliputi jenis gerbang logika biasa (AND, OR, XOR, NOT) maupun jenis fungsi matematis dan kombinatorik yang lebih kompleks (decoder, adder, subtractor, multiplier, dll). Blok-blok komponen di dalam FPGA bisa juga mengandung

elemen memori (register) mulai dari flip-flop sampai pada RAM (Random Access Memory).

Pengertian Terprogram (programmable) dalam FPGA adalah mirip dengan interkoneksi saklar dalam breadboard yang bisa diubah oleh pembuat desain. Dalam FPGA, interkoneksi ini bisa diprogram kembali oleh pengguna maupun pendesain di dalam lab atau lapangan (field). Oleh karena itu jajaran gerbang logika (Gate Array) ini disebut field-programmable. Jenis gerbang logika yang bisa diprogram meliputi semua gerbang dasar untuk memenuhi kebutuhan yang manapun.

Secara umum FPGA akan lebih lambat jika dibandingkan dengan jenis chip yang lain seperti pada chip Application-Specific Integrated Circuit (ASIC). Hal ini karena FPGA menggunakan power/daya yang besar bentuk desain yang kompleks. Beberapa kelebihan dari FPGA antara lain adalah harga yang murah, bisa diprogram mengikuti kebutuhan, dan kemampuan untuk di program kembali untuk mengkoreksi adanya bugs. Jenis FPGA dengan harga murah biasanya tidak bisa diprogram dan dimodifikasi setelah proses desain dibuat (fixed-version). Chip FPGA yang lebih kompleks dapat diperoleh dari jenis FPGA yang dikenal dengan CPLD (Complex-Programmable Logic Device).

Alur kerja yang umum dalam memprogram FPGA:

- Menggunakan komputer untuk mendeskripsikan fungsi logika yang diinginkan. Bisa dengan menggambar skematiknya atau menuliskan programnya.
- Menyusun (compile) fungsi logika, menggunakan software yang disediakan oleh vendor FPGA, lalu membuat file biner yang dapat diunduh ke dalam FPGA.
- Menghubungkan kabel dari komputer ke FPGA, dan mengunduh file biner ke FPGA.

2.2.1 Sejarah

Industri FPGA berkembang / berasal dari pengembangan PROM (programmable read only memory) dan PLD (programmable logic devices). PROM dan PLD memiliki kemampuan dapat diprogram di dalam pabrik maupun di lapangan (field programmable), tetapi antara programmable logic dan logic gates disambung secara langsung.

Pendiri Xilinx, Ross Freeman dan Bernard Vonderschmitt, menemukan FPGA komersial pertama pada tahun 1985 - XC2064. XC2064 ini memiliki gerbang dan interkoneksi yang programmable - sebuah awal bagi teknologi dan pasar yang baru. XC2064 ini terdiri dari 64 CLB (configurable logic blocks), dengan 3-input LUT (lookup tables). Lebih dari 20 tahun kemudian, Ross Freeman masuk ke dalam National Inventor's Hall of Fame untuk penemuannya ini.

Beberapa konsep dasar dan teknologi dalam industri untuk logic arrays, gates, dan logic blocks dipatenkan oleh David W. Page dan LuVerne R. Peterson pada tahun 1985.

Pada akhir 1980an, Naval Surface Warfare Department melakukan eksperimen yang dilakukan oleh Steve Casselman, yaitu mengembangkan komputer yang terdiri dari 600,000 reprogrammable gates. Sistem ini dipatenkan tahun 1992.

Xilinx tumbuh dengan cepat dari tahun 1985 sampai tengah-1990an. Tahun 1990an merupakan periode yang explosif untuk FPGAs, baik untuk kerumitan maupun volume produksinya. Pada awal 1990an, FPGA diutamakan digunakan untuk telekomunikasi dan networking. Pada tahun 2000an, FPGA masuk ke dalam bidang konsumer, automotif, dan aplikasi industri (wikipedia, 2009).

2.2.1.1 Pengembangan Dewasa Ini

Tren pada saat ini adalah mengembangkan arsitektur FPGA lebih jauh dengan menggabungkan logic blocks dan interconnect dari FPGA tradisional dengan embedded microprocessor dan peripheral yang beruhubungan, untuk membentuk "system on a programmable chip". Hal ini dilakukan oleh Ron Perlof dan Hana Potash dari Burroughs Advanced Systems Group, yang menggabungkan arsitektur CPU yang reconfigurable pada single chip yang disebut SB24 pada tahun 1982.

Alternatif dari pemakaian hard-macro processors yaitu dengan memamakai "soft" processor cores yang diimplementasikan pada logika FPGA (wikipedia, 2009).

2.2.1.2 Tahun Ke Tahun

1. Gerbang
   - 1987: 9,000, Xilinx
   - 1992: 600,000, Naval Surface Warfare Department
   - 2000an: Jutaan

2. Pasar
   - 1985: FPGA komersial pertama ditemukan oleh Xilinx
   - 1987: $14juta
   - 1993: >$385juta
   - 2005: $1.9 miliar
   - 2010 estimates: $2.75 miliar

3. Desain FPGA
   - Awal: 10,000
   - 2005: 80,000
   - 2008:90,000
   - 2010 (estimasi): 110,000

2.2.2 Perbandingan FPGA

2.2.2.1 FPGA & ASIC

Menurut sejarah jika dibandingkan dengan ASIC (application-specific integrated circuit) lainnya, FPGA itu lebih lambat, kurang efisien dalam hal energi, dan secara umum kurang memiliki kemampuan (less functionality). Kombinasi dari jumlah, improvisasi fabrikasi, riset dan pengembangan, serta kemampuan I/O dari superkomputer, telah mengecilkan gap performa antara ASIC dan FPGA.

Keuntungan FPGA yaitu *time to market* yang lebih pendek, kemampuan utnuk di reprogram di lapangan, dan non-recurring engineering costs yang lebih kecil. Pabrik-pabrik juga dapat mengembangkan hardware mereka pada FPGA, lalu membuat versi finalnya setelah desainnya disempurnakan.

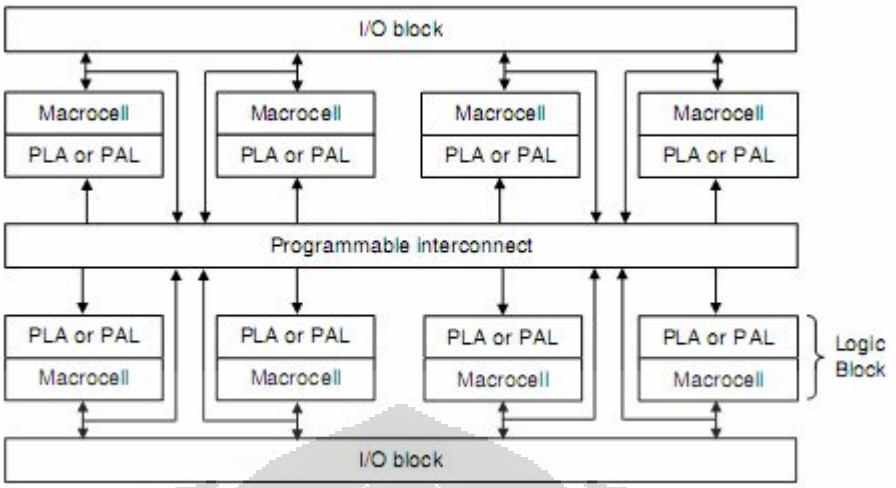
Xilinx mengklaim bahwa beberapa dinamika pasar dan teknologi telah mengubah paradigma ASIC/FPGA:

- Harga IC yang naik secara agresif
- Kompleksitas ASIC telah meningkatkan waktu dan biaya pengembangan
- Berkurangnya SDM R&D
- Peningkatan revenue losses untuk time-to-market yang lambat
- Batasan finansial dalam krisis ekonomi telah menggerakkan teknologi low-cost

Tren ini membuat FPGA menjadi alternatif yang lebih baik daripada ASIC (wikipedia, 2009).

2.2.2.1 FPGA & CPLD

Perbedaan utama antara CPLD dan FPGA adalah arsitekturnya. CPLD memiliki struktur yang terdiri dari programmable sum-of-products logic arrays. Hasilnya adalah kurang fleksibel, dengan keuntungannya yaitu waktu delay yang bisa diprediksi dan rasio logic-to-interconnect yang lebih tinggi. Arsitektur FPGA, didominasi oleh interconnect. Hasilnya adalah jauh lebih fleksibel tetapi juga makin kompleks untuk didesain (wikipedia, 2009).



Gambar 2.4 Struktur CPLD

Sumber : FPGA-based Implementation of Signal Processing Systems

2.2.3 Aplikasi

Aplikasi dari FPGA mencakup digital signal processing, software-defined radio, aerospace dan defense systems, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, computer hardware emulation dan area lainnya.

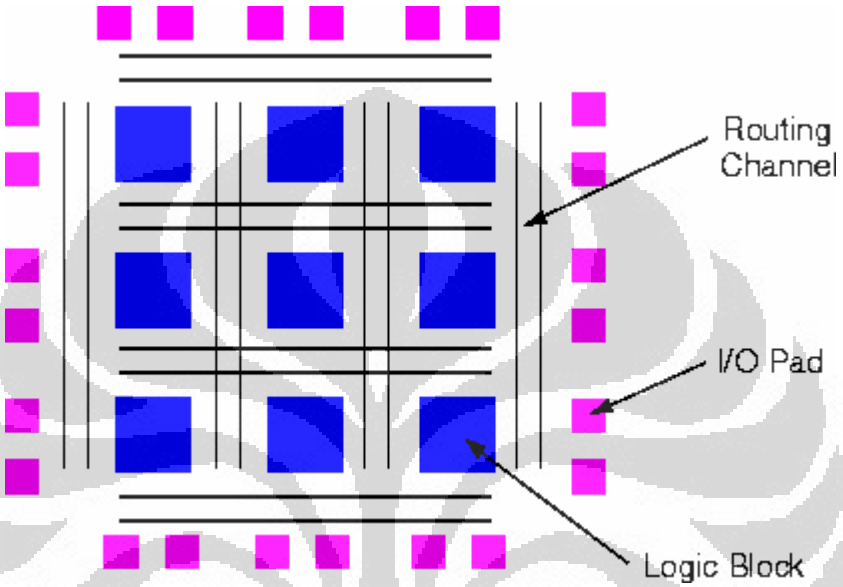
FPGA awalnya dibuat sebagai kompetitor CPLD. Setelah ukuran, kapabilitas, dan kecepatannya meningkat, keduanya mulai mengambil alih fungsi yang lebih besar, dimana sebagian sekarang dipasarkan sebagai full systems on chips (SoC).

FPGA secara khusus diaplikasikan di banyak area atau algoritma yang memakai banyak parallelism yang ditawarkan oleh arsitektur FPGA tersebut. Sebagai contoh yaitu code breaking dari algoritma cryptographic. FPGA juga digunakan pada aplikasi high performance, sebagai pengganti microprocessor.

Secara tradisional, FPGA telah digunakan pabrik-pabrik untuk dimana volume produksi, dari chip mereka, relatif kecil. Dalam hal ini, biaya yang dikeluarkan pabrik dalam biaya hardware per unit untuk programmable chip, lebih terjangkau daripada pengembangan pada pembuatan ASIC (wikipedia, 2009).

2.2.4 Arsitektur FPGA

Arsitektur FPGA paling umum adalah terdiri atas susunan dari CLB (configurable logic blocks), pad I/O, dan routing channel. Blok logika FPGA (model awal / klasik) terdiri dari 4-input lookup table (LUT), dan a flip-flop. Dewasa ini, pabrikan FPGA telah mulai mengganti dengan 6-input LUT dalam komponen performa tinggi mereka.

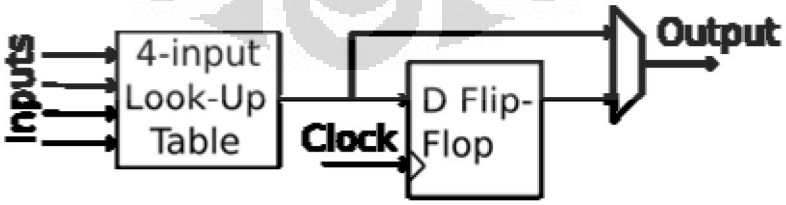


Gambar 2.5 Struktur Umum FPGA

Sumber : Introduction to Embedded Systems Using Field-Programmable Gate Array

2.2.4.1 Logic Block / Cell

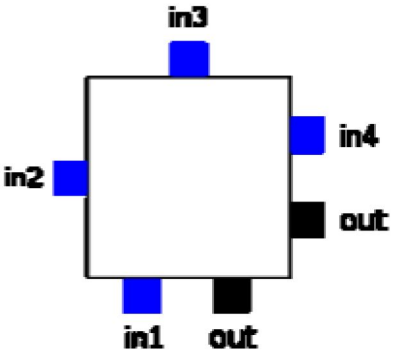
Blok logika memiliki 4 input untuk LUT dan 1 input clock. Signal clock biasanya dihubungkan via routing tersendiri pada FPGA umumnya.



Gambar2.6 Blok Logika Umum

Sumber : http://en.wikipedia.org/wiki/File:Logic_block2.svg

Sebagai contoh lokasi pin-pin blok logika, ditunjukkan pada gambar berikut.

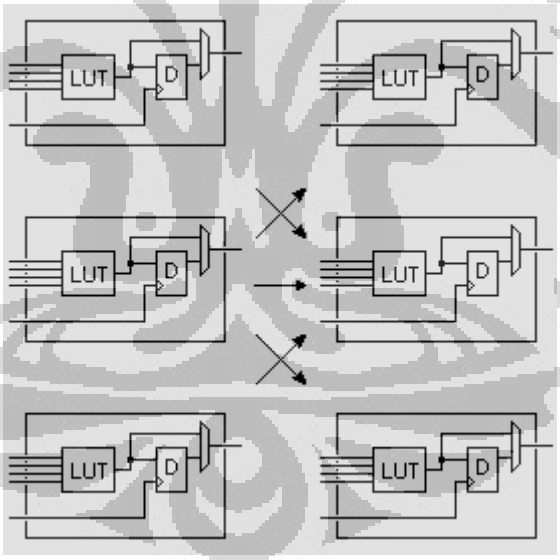


Gambar2.7 Contoh Lokasi Pin-Pin Blok Logika

Sumber : http://en.wikipedia.org/wiki/File:Logic_block_pins.svg

2.2.4.2 Routing Channel / Interconnect

Tiap pin output blok logika bisa dihubungkan ke wiring segment manapun pada channel yang di dekatnya. Dengan cara yang sama, sebuah pad I/O bisa dihubungkan ke wiring segment manapun juga.



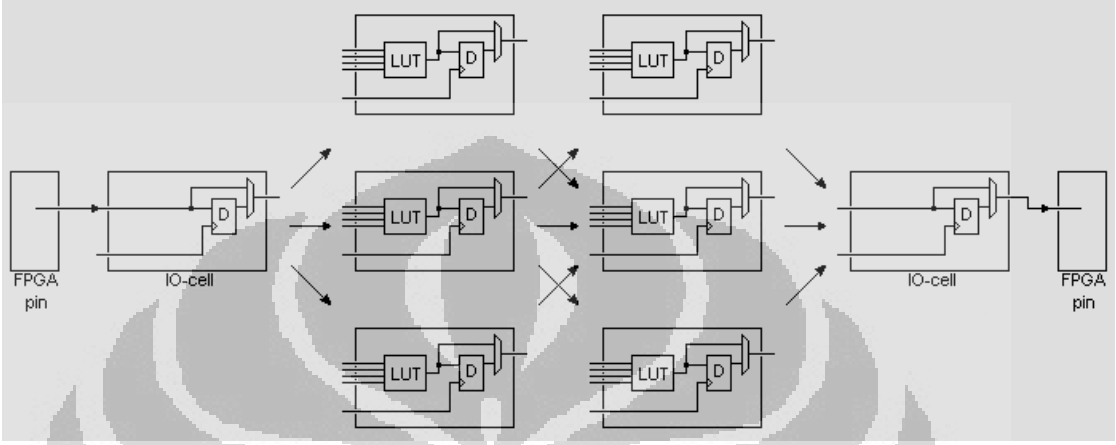
Gambar 2.8 Interconnect

Sumber : http://www.fpga4fun.com/images/LogicCells.gif

Tiap-tiap logic-cell bisa dihubungkan melalui interconnect (kabel / mux). Tiap cell hanya berfungsi sedikit, tapi bila digabungkan, fungsi logika yang kompleks dapat dibuat.

2.2.4.3 IO-Cells

Kabel interconnect juga bisa dihubungkan dengan I/O cell yang terhubung dengan pin-pin tertentu di FPGA.

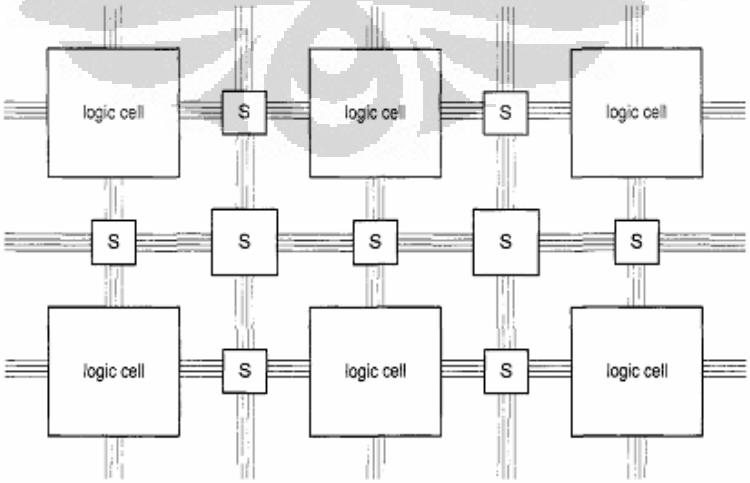


Gambar 2.9 Interconnect I/O

Sumber : http://www.fpga4fun.com/images/LogicCells_IOs.gif

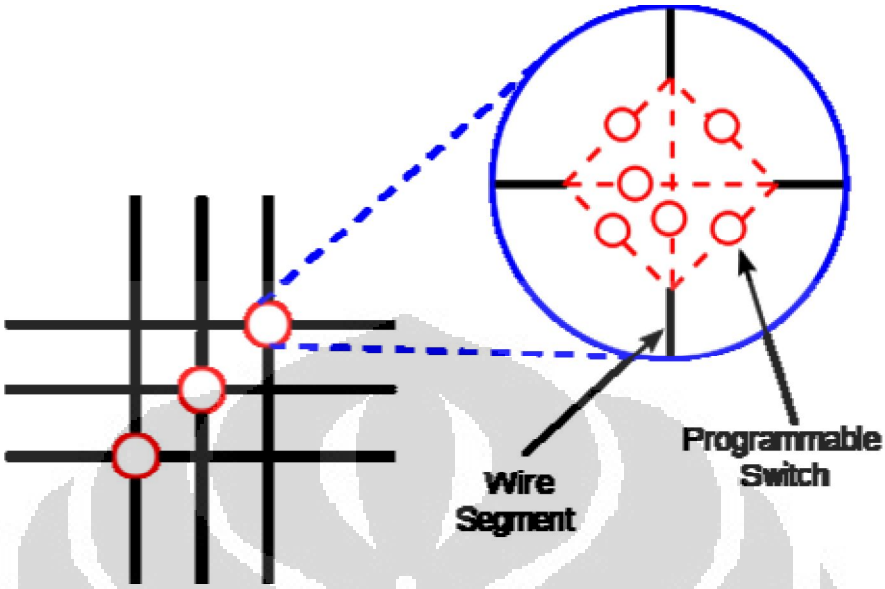
2.2.4.3 Switch Box

Dimana pun channel vertical dan horizontal berpotongan, maka akan terdapat switch box. Terdapat 3 programmable switch yang dapat menghubungkan suatu kabel dengan 3 kabel lain di dekatnya.



Gambar 2.10 Struktur konseptual dari FPGA (*ket : S = Switch Box*)

Sumber : Digital Systems Design With FPGAs and CPLDs



Gambar 2.11 Switch box topology

Sumber : http://en.wikipedia.org/wiki/File:Switch_box.svg

Keluarga FPGA modern saat telah ditingkatkan kapabilitasnya menuju kemampuan yang lebih tinggi. Contoh-contohnya adalah multipliers, generic DSP blocks, embedded processors, high speed IO logic dan embedded memori.

2.2.5 Desain dan Programming FPGA

Tiap vendor FPGA biasanya menyediakan software mendesain dan memprogram FPGA. Untuk mendefinisikan perilaku dari FPGA, pengguna menggunakan HDL (hardware description language) atau dengan skematik. Bahasa (HDL) yang umum digunakan adalah VHDL dan Verilog.

HDL lebih bermanfaat ketika bekerja dengan struktur yang besar, karena akan lebih mudah menetapkannya secara numerik daripada jika harus menggambar setiap bagian. Di sisi lain, skematik digunakan untuk visualisasi dari suatu desain. Lalu dengan menggunakan electronic design automation tool, sebuah netlist dapat dibuat. Netlist ini “dimasukkan” ke dalam FPGA dengan proses yang dinamakan place-and-route, biasanya dapat dilakukan oleh software FPGA yang

bersangkutan. Pengguna akan dapat mem-validasi map-nya, hasil place-and-route dengan analisa timing, melakukan simulasi, dan metode verifikasi lainnya. Setelah validasi ini selesai, dapat dibuat file biner-nya untuk mengatur FPGA. Langkah selanjutnya yaitu source file ini "dimasukkan" ke dalam FPGA via serial interface (JTAG) atau melalui external memori seperti EEPROM.

Untuk memudahkan desain sistem yang kompleks di FPGA, telah terdapat library yang berisi fungsi dan rangkaian kompleks yang telah didefinisikan (predefined) yang telah dites dan dioptimalkan untuk mempercepat proses desain. Rangkaian yang telah didefinisikan ini disebut IP (intellectual property) cores, dan disediakan oleh vendor-vendor FPGA dan suplier IP pihak-ketiga (jarang yang gratis, dan biasanya dirilis dibawah lisensi tertentu) (wikipedia, 2009).

2.2.6. Tipe Proses Teknologi

- SRAM – berdasarkan teknologi static memory. In-system programmable dan re-programmable. Memerlukan external boot devices. CMOS.
- Antifuse - One-time programmable. CMOS.
- EPROM - Erasable Programmable Read-Only Memory. Biasanya one-time programmable. Bisa dihapus dengan menggunakan ultraviolet (UV). CMOS.
- EEPROM - Electrically Erasable Programmable Read-Only Memory. Bisa dihapus. Beberapa, dapat mendukung in-system programmed. CMOS.
- Flash - Flash-erase EPROM. Bisa dihapus. Beberapa, dapat mendukung in-system programmed. Biasanya, flash cell lebih kecil dari EEPROM cell yang sama maka memiliki biaya lebih murah tuntuk diproduksi. CMOS.
- Fuse - One-time programmable. Bipolar.

2.2.6. Pabrikan Utama

Xilinx dan Altera adalah penguasa pangsa pasar FPGA dan merupakan rival lama. Mereka, menguasai sekitar 80% pasar, dengan pasar Xilinx lebih dari 50%.

Xilinx menyediakan design software Windows dan Linux secara gratis, sementara Altera hanya menyediakan gratis untuk Windows; Solaris dan Linux hanya dapat diperoleh via rental.

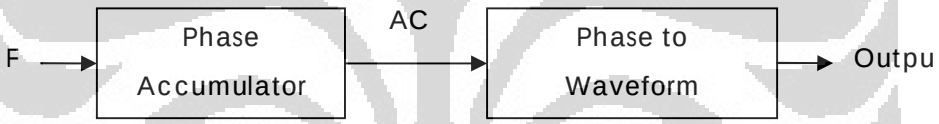
Kompetitor lainnya yaitu Lattice Semiconductor (SRAM), Actel (antifuse, flash-based, mixed-signal), dan QuickLogic (handheld focused) (wikipedia, 2009).

2.3 DIRECT DIGITAL SYNTHESIS

DDS adalah metode elektronik untuk menciptakan frekuensi / gelombang (arbitrary) dari sebuah sumber frekuensi tunggal.

Aplikasi dari DDS adalah : function generator, mixer, modulator, dan sound synthesizer.

Rangkaian DDS terdiri dari kontroler elektronik, RAM (random-access memory), referensi frekuensi (biasanya crystal oscillator), counter, dan DAC (digital-to-analog converter). Ada 2 langkah untuk membuat alat ini bekerja, yaitu langkah programming dan langkah running.



Gambar 2.12 Dasar DDS

2.3.1 Programming

Dalam langkah ini, kontroler elektronik mengisi memori dengan data. Tiap data adalah binary word yang merepresentasikan amplitudo dari signal pada waktu tertentu. Susunan data ini kemudian dibentuk menjadi tabel amplitudo. Sebagai contoh, jika setengah bagian pertama dari tabel berisi nilai 100% dan setengahnya lagi berisi nol, maka data akan merepresentasikan gelombang kotak. Bentuk gelombang lainnya bisa dibuat dengan mengubah data.

2.3.2 Running

Dalam langkah ini, counter (selanjutnya disebut phase accumulator) diinstruksikan untuk berjalan setiap peningkatan pulsa yang masuk dari referensi frekuensi. Output dari phase accumulator digunakan untuk memilih data sesuai dari tabel.kemudian, DAC mengubah data ini menjadi gelombang analog.

Untuk mendapatkan gelombang periodik, rangkaian diatur agar setiap (data) yang melewati tabel itu sama waktunya dengan perioda dari gelombang.sebagai contoh, jika referensi frekuensi 1 MHz, dan the tabel mengandung 1000 isian, maka jika melewati semua ini dengan kenaikan fase 1 maka waktunya 1000 / 1 MHz = 1 ms, maka frekuensi gelombang output akan menjadi 1/(1 ms) = 1 kHz.

Sistem dapat menghasilkan frekuensi yang lebih tinggi dengan menambah kenaikan fase sehingga counter-nya dapat melewati tabel dengan lebih cepat. Dalam contoh sebelumnya, kenaikan fasenya adalah 1, maka jika diubah menjadi 2, akan menggandakan frekuensi output. Untuk menghasilkan kontrol frekuensi yang lebih baik, kenaikan fase dapat diset menjadi, sebagai contoh, 10. Hal ini dapat mengakibatkan kenaikan / penurunan frekuensi yang kecil / sedikit. Sebagai contoh, jika kenaikan fasenya diubah menjadi 11 maka akan membuat frekuensi meningkat 10%, dan mengubahnya menjadi 9 akan menguranginya 10% juga. Jika menginginkan kenaikan / penurunan frekuensi yang lebih presisi, bit yang tinggi diperlukan pada counternya.

2.4 VHDL

VHDL adalah akronim dari VHSIC Hardware Description Language. VHSIC merupakan akronim dari Very High Speed Integrated Circuits. Biasanya digunakan di FPGA dan ASIC pada EDA (electronic design automation) dari rangkaian digital.

VHDL dapat digunakan sebagai dokumentasi, verifikasi, dan synthesis dari digital design. Hal ini merupakan salah satu fitur penting pada VHDL, karena dengan 1 kode VHDL yang sama secara teoritis dapat mencapai tiga tujuan tersebut, yang berakibat kepada penghematan. Untuk mencapai tujuan tersebut, VHDL dapat digunakan untuk mendeskripsikan hardware dengan 3 pendekatan yang berbeda. Pendekatan tersebut adalah structural, data flow, dan behavioral. Kebanyakan digunakan campuran ketiga metode ini.

2.4.1 Sejarah

VHDL adalah standar yang dikembangkan oleh IEEE (Institute of Electrical and Electronics Engineers). Standar ini diberi nama IEEE 1076. Bahasa ini telah

mengalami beberapa revisi. Versi awal yaitu versi 1987 (IEEE 1076-1987), kadang hanya disebut VHDL'87.

Berikut ini merupkan revisi-revisi yang telah terjadi :

- 1076-1987 : revisi standardisasi pertama dari ver 7.2 dari bahasa ini oleh United States Air Force.
- 1076-1993 : improvisasi yang signifikan, berasal dari feedback selama beberapa tahun. Versi yang paling banyak digunakan vendor-vendor.
- 1076-2000 : revisi minor. Dilakukan untuk memenuhi standar IEEE yang mengharuskan standar harus di revisi setiap 5 tahun agar tetap update.
- 1076-200x : nama yang diberikan untuk revisi yang terjadi dari tahun 2003 hingga sekarang ini (2008). Versi non-resmi terakhir yaitu 1076-2008, yang draftnya masih diedit oleh IEEE.

2.4.2 Desain

Ada 3 cara untuk mendeskripsikan hardware VHDL yaitu structural, data flow, dan behavioral.

2.4.2.1 Structural

Untuk membuat desain yang lebih dapat dimengerti dan dipelihara, biasanya desain dibagi kedalam beberapa blok. Blok-blok ini lalu dihubungkan sehingga didapat desain yang utuh. Setiap bagian dalam desain VHDL dapat dianggap sebuah blok. Setiap blok disebut *entity*. Entity mendeskripsikan interface pada blok dan mendeskripsikan bagaimana blok beroperasi. Deskripsi interface merupakan spesifikasi input dan output pada blok. Deskripsi dari operasi adalah seperti skematik blok.

Berikut merupakan contoh *entity declaration* pada VHDL.

```
entity latch is
  port (s,r: in bit;
        q,nq: out bit);
end latch;
```

Baris pertama menunjukkan definisi dari suatu entity, yang bernama *latch*. Baris terakhir merupakan akhir dari definisi. Baris diantaranya disebut *port clause*, yang mendeskripsikan interface. Port clause mengandung daftar dari deklarasi interface. Setiap deklarasi *interface* mendefinisikan satu atau lebih signal input atau outputs.

Setiap deklarasi mengandung daftar nama, mode, dan tipe. Dalam contoh diatas, kedua signal input didefinisikan, *s* dan *r*. Bagian kiri merupakan nama dari signal, dan bagian kanan merupakan mode dan tipe dari signal. Bagian mode menspesifikasikan apakah itu input, output, atau keduanya. Bagian tipe menspesifikasikan jenis signal. Signal *s* dan *r* merupakan mode *in* (input) dan bertipe *bit* (biner). Signal *q* dan *nq* mode *out* (outputs) dan tipe *bit*. Setiap port clause digunakan titik koma (semicolon) di akhirnya.

Bagian kedua yaitu mendeskripsikan bagaimana desain beroperasi. Didefinisikan dalam deklarasi *architecture*. Berikut merupakan contoh *architecture declaration*.

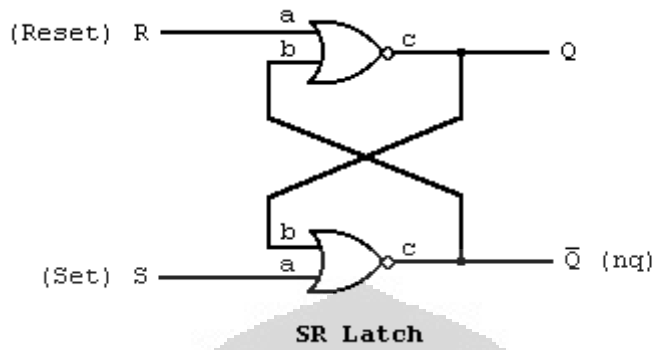
```
architecture dataflow of latch is
  signal q0 : bit := '0';
  signal nq0 : bit := '1';
begin
  q0<=r nor nq0;
  nq0<=s nor q0;

  nq<=nq0;
  q<=q0;
end dataflow;
```

Baris pertama mengindikasikan definisi dari arsitektur baru yaitu yang bernama *dataflow* dan merupakan bagian dari entity bernama *latch*. Maka arsitektur ini mendeskripsikan operasi dari entity *latch*. Baris diantara begin dan end mendeskripsikan operasi dari latch.

Sebelumnya telah dibuat blok-blok, maka selanjutnya adalah menghubungkannya.

```
entity latch is
  port (s,r: in bit;
        q,nq: out bit);
end latch;
```

Untuk pendekatan struktural, kita asumsikan sebuah entity bernama *nor_gate* telah didefinisikan. Maka skematiknya seperti berikut ini



Gambar 2.13 SR Latch dengan jalur

Dapat dispesifikasikan koneksi yang sama dengan skematik menggunakan aristektur berikut ini

```
architecture structure of latch is
  component nor_gate
    port (a,b: in bit;
          c: out bit);
  end component;
begin
  n1: nor_gate
    port map (r,nq,q);
  n2: nor_gate
    port map (s,q,nq);
end structure;
```

Antara baris pertama dan kata *begin* merupakan deklarasi komponen. Deklarasi komponen ini mendeskripsikan interface dari entity *nor_gate* yang akan digunakan sebagai komponen dari (atau bagian dari) desain. Antara kata *begin* dan *end*, dua baris pertama dan dua baris kedua mendefinisikan dua *component instance*.

Terdapat perbedaan penting antara entity, komponen, dan *component instance* di dalam VHDL. Entity mendeskripsikan desain interface, sedangkan komponen mendeskripsikan interface dari sebuah entity yang digunakan sebagai sebuah *instance* (atau sub-block), dan *component instance* merupakan salinan dari komponen yang telah dihubungkan ke part-part dan signal-signal yang lain. Dapat

dibuat perbandingan dengan proses desain bread board dengan part-part tertentu. Entity dan architecture adalah seperti *data book* yang mendeskripsikan interface dan skematik yang menjelaskan bagaimana part-part bekerja. Komponen adalah seperti list pin-pin yang mendeskripsikan bagaiman part-part harus disambungkan. Component instance adalah seperti part-part dalam bread board, yang dapat dioperasikan secara independen.

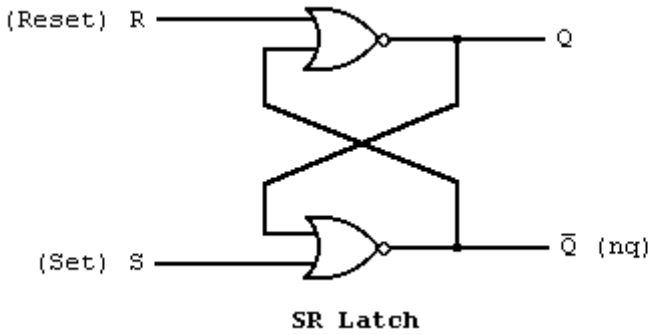
Dalam contoh sebelumnya, komponen *nor_gate* memiliki 2 input (*a* dan *b*) dan sebuah output (*c*). Ada 2 *instance* dari komponen *nor_gate* dalam arsitektur ini yang berhubungan dengan 2 simbol nor dalam skematik. *Instance* pertama merepresentasikan nor gate bagian atas dari skematik. Baris pertama dari *component instantiation* (baris setelah kata begin) yaitu pemberian nama pada instance yaitu *n1*, dan menspesifikasikan bahwa *n1* merupakan instance dari komponen *nor_gate*. Baris kedua mendeskripsikan bagaimana komponen dikoneksikan menggunakan *port map clause*. *Port map clause* menspesifikasikan signal apa yang akan dikoneksikan ke interface dari komponen dalam urutan yang sama seperti yang tertulis pada deklarasi komponen. Interface dispesifikasikan dengan berurutan *a*, *b*, dan *c*, sehingga instance menghubungkan *r* ke *a*, *nq* ke *b*, dan *q* ke *c*. Hal ini sama dengan skematik yang tertera. Instance kedua, diberi nama *n2*, yang menghubungkan *s* ke *a*, *q* ke *b*, dan *nq* ke *c*.

Deskripsi struktural dari suatu desain merupakan deskripsi tertulis (textual) dari skematik. List dari komponen dan koneksinya, dalam bahasa apapun, kadang disebut netlist. Deskripsi struktural dari suatu desain VHDL merupakan salah satu dari banyak arti dari netlist.

2.4.2.2 Data Flow

Dalam pendekatan data flow, rangkaian dideskripsikan dengan mengindikasikan bagaimana input dan output dari komponen built-in primitive (misal gerbang *and*) yang dihubungkan bekerja. Dengan kata lain, dideskripsikan bagaimana signal (data) ditransmisikan di dalam rangkaian.

Dibawah ini merupakan gambar skematik SR (set-reset) latch.



Gambar 2.14 SR Latch

Maka entity-nya akan seperti di bawah ini.

```
entity latch is
  port (s,r : in bit;
        q,nq : out bit);
end latch;

architecture dataflow of latch is
begin
  q<=r nor nq;
  nq<=s nor q;
end dataflow;
```

Seperti sebelumnya, entity mendeskripsikan desain interface. Terdapat 4 signal *s*,*r*,*q*, dan *nq*. Signal, seperti sebelumnya juga, dimodelkan dengan tipe data *bit*, yang merepresentasikan 2 level logika.

Bagian architecture mendeskripsikan operasi internal dari desain. Dalam pendekatan data flow, diindikasikan bagaimana data bertransmisi dari input ke output. Dalam VHDL, hal ini bisa dicapai dengan signal assignment statement. Pada contoh diatas terdapat 2 signal assignment statement.

Signal assignment statement mendeskripsikan bagaimana data bertransmisi dari signal di bagian kanan dari operator <= ke signal bagian kiri. Dapat dilihat pada contoh signal assignment pertama, bagaimana data dari signal *r* dan *nq* bertransmisi melalui gerbang *nor* untuk menentukan nilai dari signal *q*. Gerbang *nor* merepresentasikan komponen built-in yang disebut *operator*, oleh karena *operator* mengoperasikan suatu data untuk membuat data baru. Signal assignment

kedua, sama dengan yang pertama, mengindikasikan signal *nq* dibuat dari data (*s* and *q*) yang melewati (atau diproses by) operator *nor*.

Bagian kanan dari operator <= diberi nama *expression*. Nilai dari *expression* ditentukan dengan mengevaluasi *expression*-nya. Evaluasi dari expression ini dilakukan dengan mensubstitusi nilai dari signal pada bagian *expression* dan dihitung (computing) hasilnya dari tiap operator.

Standar dari VHDL tidak hanya mendeskripsikan bagaimana desain dispesifikasikan, tapi juga bagaimana desain diinterpretasikan. Hal ini merupakan tujuan dari sebuah standar, sehingga bisa disetujui tentang pengertian dari sebuah desain. It is important to understand how a VHDL simulator interprets a design because that dictates what the "correct" interpretation is according to the standard (Hopefully, simulators are not all 100% correct).

Skema yang digunakan untuk memodelkan desain VHDL disebut *discrete event time simulation*. Saat nilai sinyal berubah, dapat dikatakan sebuah *event* telah terjadi pada sinyal tersebut. Jika data bertransmisi dari sinyal A ke B, dan sebuah *event* telah terjadi pada sinyal A, maka perlu ditentukan kemungkinan nilai baru B. Hal ini merupakan dasar dari *discrete event time simulation*. Nilai-nilai dari sinyal hanya diperbaharui ketika terjadi peristiwa tertentu dan terjadi di waktu tertentu (diskrit).

Sejak satu *event* menyebabkan *event* lainnya, simulasi berproses dalam beberapa putaran. Simulator menyimpan daftar kegiatan yang harus diproses. Dalam setiap satu putaran, semua peristiwa dalam daftar diproses, *event* baru yang diproduksi akan disimpan dalam daftar terpisah (dan akan dijadwalkan) untuk diproses di putaran berikutnya. Tugas masing-masing sinyal dievaluasi sekali, ketika simulasi dimulai untuk menentukan nilai awal dari masing-masing sinyal.

Berikut ini adalah penjelasan bagaimana hasil simulasi kegiatan waktu untuk contoh SR latch sebelumnya (Lihat Gambar 2.13 SR Latch).

Esensi dari operasi internal dari latch terdiri dari 2 statement dibawah ini.

```
q<=r nor nq;
nq<=s nor q;
```

Karena data bertransmisi dari *r* dan *nq* ke *q*, maka *q* bergantung pada *r* dan *nq*. Secara umum, diberikan assignment apapun, sinyal di samping kiri operator <= bergantung pada semua sinyal yang muncul pada sisi kanan. Jika suatu sinyal

yang bergantung pada sinyal lain, telah terjadi suatu *event*, maka ekspresi di sinyal kembali dievaluasi. Jika hasil evaluasi berbeda dengan nilai sinyal, *event* akan dijadwalkan (ditambahkan ke daftar *event* untuk diproses) untuk memperbarui sinyal dengan nilai baru. Jadi, jika sebuah *event* terjadi pada *r* atau *nq*, maka operator *nor* dievaluasi, dan jika hasilnya berbeda dibandingkan dengan nilai *q*, maka *event* tersebut akan dijadwalkan untuk memperbarui *q*.

Misalnya pada saat tertentu selama simulasi dari contoh SR latch, nilai-nilai yang ada sinyal *s* ='0 ', *r* ='0', *q* ='1 ', dan *nq* ='0'. Sekarang seandainya nilai sinyal *r* berubah (karena terjadi beberapa *event* external) menjadi '1'. Karena *q* tergantung *r*, harus kembali dievaluasi ekspresi *r nor nq*, yang sekarang berubah menjadi '0'. Karena nilai *q* harus diubah menjadi '0', *event* yang baru akan dijadwalkan pada sinyal *q*. Saat putaran berikutnya, *event* yang dijadwalkan untuk *q*, diproses dan nilai *q* diperbarui menjadi '0'. Selain itu, sejak *nq* bergantung *q*, ekspresi *s nor q* harus dievaluasi kembali. Hasil ekspresi adalah '1 ', sehingga suatu *event* dijadwalkan untuk memperbarui nilai *nq*. Sat putaran berikutnya, ketika *event* di *nq* diproses, ekspresi untuk *q* akan dievaluasi lagi karena bergantung pada *nq*. Bagaimanapun, hasil dari ekspresi akan '0 'dan tidak ada *event* baru akan dijadwalkan karena q sudah '0'. Sehingga, tidak ada aktivitas baru yang dijadwalkan, dan tidak ada lagi *event* yang akan terjadi secara internal di latch.Sekarang, seandainya *event* eksternal menyebabkan *r* kembali ke nilai'0'.

Karena *q* tergantung *r*, *r nor nq* akan dievaluasi lagi. Hasil dari ekspresi ini adalah '0' dan *q* sudah '0', jadi tidak ada *event* yang dijadwalkan. Seperti yang dapat dilihat, hal ini adalah model SR latch yang benar yang diharapkan. Bila sinyal *r* menjadi aktif ('1') output dari latch di-*reset*, dan bila *r* menjadi tidak aktif ('0') hasilnya tetap tidak berubah.

Simulasi putaran terakhir ini dijelaskan dalam ringkasan berikut.

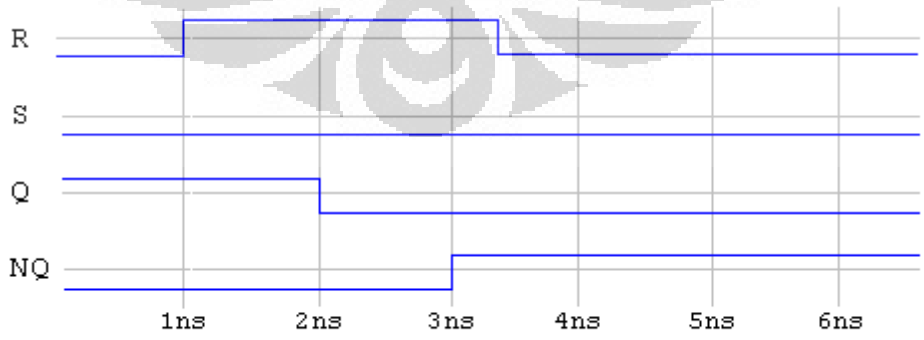
start	: r='0',s='0',q='1',nq='0'	
round 1	: r='1',s='0',q='1',nq='0',	nilai '0' di q.
round 2	: r='1',s='0',q='0',nq='0',	nilai '1' di nq.
round 3	: r='1',s='0',q='0',nq='1',	tak ada *event* .
round 4	: r='0',s='0',q='0',nq='1',	tak ada *event*.

Contoh dari bagian terakhir menunjukkan bagaimana simulasi fungsional (*functional simulation*) terjadi. Disebut *functional simulation* karena model simulasi ini hanya mendesain fungsi tanpa pertimbangan waktu. Hal ini kontras dengan simulasi waktu (*timing simulation*), yang memodelkan *delay* internal yang terjadi nyata di sirkuit. Akan dijelaskan bagaimana VHDL dapat digunakan untuk memodelkan waktu *delay* untuk membuat simulasi waktu.

Akan didiskusikan dua model *delay* yang digunakan dalam VHDL. Yang pertama adalah yang disebut *inertial delay model*. Model ini ditentukan dengan menambahkan *after clause* ke signal assignment statement. Misalnya, terjadi perubahan pada masukan dari pintu gerbang *nor* dan menyebabkan keluaran berubah setelah delay 1ns. Untuk memodelkan *delay* ini dalam contoh SR latch, kita bisa menggantikan dua *signal assignments* dengan dua pernyataan di bawah ini.

```
q<=r nor nq after 1ns;
nq<=s nor q after 1ns;
```

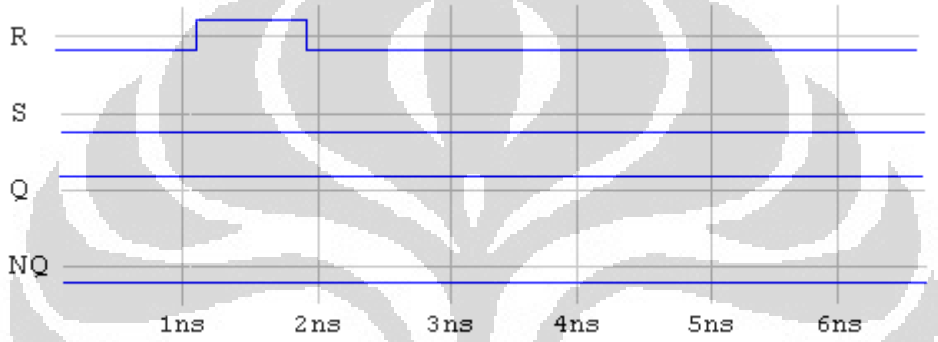
Selama simulasi, katakan sinyal *r* berubah dan akan menyebabkan sinyal *q* berubah, yang akan terjadi setelah 1ns, bukan pada saat putaran berikutnya. Dengan demikian, simulator harus menjaga nilai waktu. Bila tidak ada lagi ada *event* yang terjadi untuk diproses, waktu diperbarui ke waktu awal *event* dan semua *event* yang dijadwalkan untuk waktu akan diproses. Diagram waktu untuk ini untuk SR latch yang dimodifikasi sebagai berikut.



Gambar 2.15 Timing Diagram Simulasi Inertial Delay

Dapat dilihat bahwa perubahan tidak terjadi dalam *q* sampai 1ns setelah *r* berubah. Demikian pula perubahan dalam *nq* tidak terjadi setelah 1ns setelah *q* berubah.

Namun biasanya, bila komponen memiliki internal *delay* dan masukan berubah dalam waktu kurang dari waktu *delay* ini, maka tidak ada perubahan output yang akan terjadi. Hal ini juga terjadi pada kasus *inertial delay model.* Berikut merupakan diagram waktu dengan menggunakan model inertial keterlambatan, jika pulsa '1' pada sinyal *r* kurang dari 1ns.



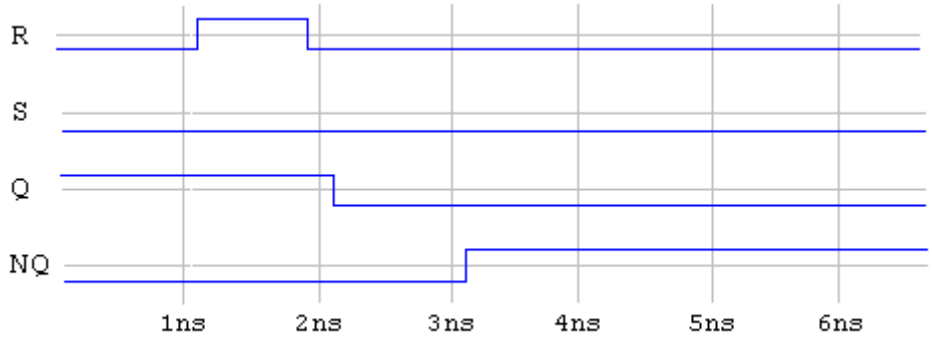
Gambar 2.16 Timing Diagram Simulasi Inertial Delay (gagal)

Nilai q tidak berubah karena perubahan dalam r tidak cukup lama. Dengan kata lain, perubahan dalam r tidak mendapatkan cukup inertia.

Meskipun sering keterlambatan inertial dikehendaki, kadang-kadang semua perubahan pada input harus memiliki efek pada output. Sebagai contoh, sebuah *bus* yang mengalami keterlambatan waktu, tetapi tidak akan "menyerap" pulsa singkat seperti *inertial delay model*. Maka, VHDL menyediakan *transport delay model*. *Transport delay model* in hanya menunda perubahan dalam output pada saat setelah ditetapkan. Dapat dipilih *transport delay model* alih-alih *inertial delay model* dengan menambahkan kata kunci *transport* ke *signal assignment statement*.

```
q<=transport r nor nq after 1ns;
nq<=transport s nor q after 1ns;
```

Jika transport delay model yang digunakan, hasil pada simulasi yang sama ditampilkan pada diagram timing diagram berikut.



Gambar 2.17 Timing Diagram Simulasi Transport Delay

Pada seluruh contoh sinyal pada bagian sebelumnya adalah merupakan jenis bit. VHDL menyediakan beberapa jenis lainnya. Sering kali digunakan beberapa bit sinyal bersamaan untuk mewakili nomor biner dalam desain. VHDL menyediakan mekanisme untuk menentukan jenis baru yang merupakan kumpulan beberapa item data yang sama jenisnya. Jenis tipe ini disebut *array*. Ada tipe array standar yang disebut *bit_vector* yang merupakan kumpulan bit. Contoh berikut mendemonstrasikan bagaimana *bit_vector* jenis dapat digunakan untuk mendefinisikan *1-to-4-line demultiplexer*.

```
entity demux is
  port (e: in bit_vector (3 downto 0);      -- enables for each
output
        s: in bit_vector (1 downto 0);      -- select signals
        d: out bit_vector (3 downto 0));    -- four output signals
end demux;

architecture rtl of demux is
  signal t : bit_vector(3 downto 0);        -- an internal signal
begin
  t(3)<=s(1) and s(0);
  t(2)<=s(1) and not s(0);
  t(1)<=not s(1) and s(0);
  t(0)<=not s(1) and not s(0);
  d<=e and t;
end rtl;
```

Komentar dapat ditambahkan pada akhir suatu pernyataan VHDL atau pada baris diawali dengan simbol (--).

Dapat dilihat bagaimana *bit_vector* digunakan dalam definisi dari sebuah sinyal. Defini dari *s* menunjukkan bahwa *s* adalah *bit_vector* dan bagian (1 downto 0) menetapkan bahwa sinyal *s* berisi dua bit bernilai 1 sampai 0. Demikian pula, *d* dan *e* adalah array 4 bit benilai dari 3 sampai 0. Lalu, lihat juga sinyal, seperti *t*, dapat dinyatakan dalam suatu arsitektur yang tidak terlihat dari luar *entity* ini. Sinyal internal ini dibuat dengan deklarasi sinyal (*signal declarations*) seperti contoh. *Signal declarations* berisi kata kunci *signal* diikuti dengan daftar nama-nama sinyal yang dibuat, dan diikuti oleh jenis sinyal. Dapat dilihat juga, arsitektur merujuk kepada bit *s* dan *t* dengan nomor. Dua bit dalam s adalah angka 1 dan 0, sehingga mereka disebut sebagai s(1) dan s(0). Tugas sinyal terakhir menunjukkan bahwa operasi dapat dilakukan pada seluruh array data sekaligus. Pernyataan ini setara dengan empat pernyataan berikut ini.

```
d(3)<=e(3) and t(3);
d(2)<=e(2) and t(2);
d(1)<=e(1) and t(1);
d(0)<=e(0) and t(0);
```

Setiap item dalam data array disebut *element*. Jumlah elemen dalam sinyal dari jenis array ditunjukkan oleh kisaran yang mengikuti nama jenis. Elemen diberi nomor sesuai dengan rentangnya, dan setiap elemen dirujuk sebagai individu oleh nomor. Operasi dapat dilakukan pada array secara keseluruhan (yang berlaku untuk setiap elemen dalam array), atau dapat dilakukan dengan menggunakan setiap elemen array, independen dari yang lain.

Ketika operasi dilakukan pada seluruh vektor, maka vektor harus memiliki nomor yang sama dari elemen. Jika tidak, simulator akan melaporkan kesalahan dan menghentikan simulasi. Dalam suatu operasi antara vektor, elemen dicocokkan seperti nomor dari kiri ke kanan. Jadi, jika variabel v1 memiliki elemen 0 sampai 1 dan variabel v2 memiliki unsur 1 sampai 0. Maka :

```
v1:=v2;
```

akan menghubungkan v2(1) ke v1(0) dan v2(0) ke v1(1).

Standar jenis lainnya yaitu waktu (*time*). Jenis ini digunakan untuk mewakili nilai-nilai dari waktu. Telah digunakan nilai konstan dari jenis ini di dalam after

clause. Waktu adalah contoh dari jenis fisik (*physical type*). Semua nilai-nilai fisik memiliki dua jenis komponen, nomor dan nama unit (*unit name*). Jenis waktu meliputi standar unit berikut nama sec (detik), ms (milliseconds), us (microseconds), ns (nanoseconds), ps (picoseconds), dan fs (femtoseconds). Ada beberapa jenis standar lainnya dalam VHDL termasuk jenis integers dan angka riil. Ini adalah waktu yang disebutkan dalam bagian yang berhubungan dengan perilaku keterangan.

Dalam bagian sebelumnya disebutkan beberapa jenis operator yang tersedia dalam VHDL. Ada juga beberapa operator built-in yang dapat digunakan.

Operator logika, NOT, AND, OR, NAND, NOR, dan XOR dapat digunakan dengan semua jenis *bit* atau *bit_vector*. Misalnya, "00101001" *xor* "11100101" hasilnya "11001100".

Sebagai catatan, sebagaimana '0 'dan '1' mewakili nilai *bit*, nilai-nilai *bit_vector* dapat ditulis dalam VHDL sebagai daftar nilai dalam tanda kutip ganda (“). Misalnya, jika d adalah *bit_vector(1 to 4)*, maka pernyataannya adalah d(1) ='1', d(2) ='1', d(3) ='0', dan d(4) ='0'.

```
d<="1100";
```

Heksadesimal juga dapat digunakan dengan cara pintas seperti dalam contoh berikut.

```
d<=X"C";
```

Karena C adalah heksadesimal dari 12, yang dalam biner adalah 1100, pernyataan ini setara dengan yang sebelumnya. X di depan menunjukkan bahwa angka dalam heksadesimal sebagai pengganti biner normal. Operator aljabar umum yang tersedia untuk integer, seperti +, -, * (perkalian), dan / (pembagian). Walaupun operasi ini tidak built-in untuk bit_vectors, tetapi sering disediakan di library yang ada pada software VHDL. Mereka digunakan dengan bit_vectors dengan menginterpretasikannya sebagai representasi biner dari integer, yang dapat ditambahkan, dikurangkan, dikalikan, atau dibagi.

Juga tersedia standar operator penghubung yang biasa, yaitu =, / =, <, <=, > dan >= (tanda / menandakan tidak sama). Hasil dari semua operator tersebut adalah nilai boolean (TRUE atau FALSE). Argumen dari operator = dan /= dapat berupa jenis apa pun. Argumen dari operator <, <=, > dan >= mungkin jenis scalar

apapun (integer, real, dan physical types) atau jenis bit_vector. Jika suatu argument bit_vectors, maka argumen harus sama panjang dan hasilnya adalah TRUE jika hubungan yang benar sesuai untuk setiap elemen array argument.

Operator & adalah operator VHDL built-in yang melakukan penggabungan (*concatenation*) dari *bit_vector*. Sebagai contoh, dengan deklarasi berikut:

```
signal a: bit_vector (1 to 4);
signal b: bit_vector (1 to 8);
```

Pernyataan berikut ini akan menghubungkan *a* ke setengah kanan dari *b* dan membuat setengah kiri dari konstan b '0'.

```
b<="0000" & a;
```

Operator & akan menambahkan *a* ke ujung dari "0000" untuk membentuk hasil 8 bits.

2.4.2.2 Behavioral

Pendekatan *behavioral* yang memodelkan komponen perangkat keras berbeda dari dua metode yang sebelumnya, yang mana tidak mencerminkan bagaimana desain diimplementasikan. Seperti pada dasarnya black box. Pendekatan *behavioral* secara akurat memodelkan apa yang terjadi pada masukan dan keluaran dari kotak hitam, tapi apa yang di dalam kotak (cara kerjanya) tidak menjadi soal. Deskripsi *behavioral* yang biasanya digunakan dalam dua cara pada VHDL. Pertama, dapat digunakan untuk memodelkan komponen kompleks yang akan menjemukan jika menggunakan pemodelan lainnya. Sebagai contoh kasus misalnya, jika ingin mensimulasikan pengoperasian desain custom (buatan sendiri) yang terhubung ke part-part lain seperti microprocessor. Dalam kasus ini, microprocessor sangatlah kompleks dan operasi internal tidak menjadi soal (hanya perilaku eksternal yang penting), sehingga akan digunakan pendekatan *behavioral*. Kedua, kemampuan *behavioral* dari VHDL dapat menjadi lebih tepat guna (*powerful*) dan lebih sesuai untuk beberapa desain. Dalam hal ini deskripsi *behavioral* akan menyiratkan beberapa struktur pelaksanaan.

Penjelasan *behavioral* didukung oleh pernyataan proses (*process statement*). *Process statement* dapat muncul di dalam sebuah arsitektur deklarasi, seperti halnya *signal assignment statement*. Isi dari pernyataan proses dapat mencakup pernyataan berurut (*sequential statement*) seperti yang ditemukan dalam bahasa

pemrograman. Pernyataan ini digunakan untuk menghitung keluaran dari proses dari masukan. Pernyataan berurut seringkali lebih tepat guna, tapi kadang-kadang tidak koresponden langsung ke pelaksanaan perangkat keras. Proses ini juga dapat berisi *signal assignments* untuk menentukan output dari proses.

Contoh pertama berikut adalah proses pernyataan sepele dan biasanya tidak akan dilakukan dalam proses pernyataan. Namun, dapat dimungkinkan untuk memeriksa *process statement* tanpa belajar *sequential statement* dahulu.

```
compute_xor: process (b,c)
begin
  a<=b xor c;
end process;
```

Bagian pertama `compute_xor:` digunakan untuk nama proses. Bagian ini opsional. Berikut adalah kata kunci *process* yang dimulai dengan definisi dari suatu proses. Setelah itu adalah daftar sinyal dalam kurung, disebut *sensitivity list*. Karena *process statement* dari isi mungkin tidak memiliki karakteristik struktural yang ada, tidak ada cara untuk mengetahui jika proses harus dievaluasi ulang untuk memperbarui output. Sinyal *sensitivity list* digunakan untuk menentukan sinyal mana yang akan menyebabkan proses kembali dievaluasi. Kapanpun ada *event* terjadi pada salah satu sinyal dalam *sensitivity list*, proses ini kembali dievaluasi. Sebuah proses dievaluasi dengan melakukan setiap pernyataan yang ada. Pernyataan ini (isi dari proses) muncul di antara kata *begin* dan *end*.

Contoh proses ini berisikan satu pernyataan, yaitu *signal assignment*. Tidak seperti *signal assignment* yang muncul di luar *process statement*, sinyal ini hanya dievaluasi saat *event* terjadi pada sinyal dalam proses *sensitivity list*, tanpa memperdulikan sinyal yang muncul pada sisi kanan operator <=. Ini penting artinya untuk memastikan sinyal yang tepat berada di *sensitivity list*. Pernyataan di dalam isi dari proses yang dilakukan (atau dilaksanakan) dengan urutan dari pertama ke yang terakhir. Bila *statement* terakhir telah dilaksanakan, proses ini selesai dan dikatakan *suspended*. Ketika sebuah *event* terjadi pada sinyal dalam *sensitivity list*, proses yang akan dilanjutkan dan *statement* akan dijalankan dari atas ke bawah lagi. Setiap proses dijalankan sekali pada awal simulasi untuk menentukan nilai awal dari output.

Ada dua jenis objek yang digunakan untuk menyimpan data. Jenis pertama, yang digunakan terutama di dalam deskripsi *structural* dan *data flow*, adalah sinyal. Kedua, yang hanya dapat digunakan dalam (statement) proses ini disebut variabel. Variabel berlaku seperti yang diharapkan dalam bahasa pemrograman perangkat lunak, yang jauh berbeda dengan perilaku dari sinyal.

Meskipun variabel mewakili data seperti sinyal, mereka tidak memiliki atau menyebabkan aktivitas, dan diubah dengan cara yang berbeda. Variabel dapat diubah dengan *variable assignment*. Misalnya,

```
a:=b;
```

akan memberikan nilai b ke a. Nilai disalin ke *a* dengan segera. Karena variabel hanya dapat digunakan dalam proses, penugasan pernyataan mungkin hanya muncul dalam proses. Tugas dilakukan saat proses dijalankan, seperti yang dijelaskan pada bagian sebelumnya.

Contoh berikut menunjukkan bagaimana variabel digunakan dalam proses.

```
count: process (x)
  variable cnt : integer := -1;
begin
  cnt:=cnt+1;
end process;
```

Deklarasi variabel muncul sebelum kata *begin* dari proses pernyataan, seperti pada contoh. Deklarasi variabel sama dengan deklarasi sinyal kecuali kata kunci *variable* yang digunakan sebagai pengganti *signal*. Deklarasi dalam contoh ini termasuk juga bagian opsional, yang menentukan nilai awal dari variabel, ketika sebuah simulasi dimulai. Bagian inisialisasi termasuk penambahan : = dan beberapa konstanta ekspresi setelah bagian deklarasi. . Bagian inisialisasi ini juga dapat dimasukkan dalam deklarasi sinyal. Variabel *cnt* dinyatakan sebagai tipe integer. Tipe integer mewakili tipe integer negatif dan positif.

Proses pada contoh berisi satu pernyataan, yaitu *assignment statement*. Assignment ini menghitung nilai *cnt* tambah satu dan langsung menyimpan nilai baru ini dalam variabel *cnt*. Dengan demikian, *cnt* akan bertambah satu setiap kali proses dijalankan. Ingat, dari bahasan sebelumnya, sebuah proses akan dijalankan sekali di awal simulasi, dan kemudian pada setiap kali sebuah *event* terjadi pada sinyal dalam *sensitivity list*. Karena nilainya diinisialisasi ke -1, dan proses

dijalankan sekali sebelum awal simulasi, maka nilai *cnt* akan 0 ketika simulasi dimulai. Setelah simulasi dimulai, *cnt* akan bertambah setiap kali sinyal *x* berubah, karena *x* ada di dalam *sensitivity list*. Jika *x* merupakan sinyal *bit*, maka proses ini akan menghitung jumlah *rise and fall edge* yang terjadi pada sinyal *x*.

Ada beberapa pernyataan yang hanya dapat digunakan di dalam isi suatu proses. Pernyataan ini disebut *sequential statements* karena dijalankan secara berurutan. Maksudnya, satu setelah yang lainnya seperti penulisannya dari bagian atas isi proses ke bagian bawah.

Contoh pertama mengilustrasikan pernyataan *if* dan merupakan atribut VHDL yang umum digunakan.

```
count: process (x)
  variable cnt : integer :=0 ;
begin
  if (x='1' and x'last_value='0') then
    cnt:=cnt+1;
  end if;
end process;
```

Statemen *if* ini memiliki dua komponen utama, bagian kondisi (*condition*) dan pernyataan (*statement*). Sebuah kondisi adalah ekspresi boolean apa saja (ekspresi yang mengevaluasi ke TRUE dan FALSE). *Condition* pada contoh menggunakan atribut *last_value*, yang digunakan untuk menentukan nilai yang terakhir yang sinyal punya. Atribut dapat digunakan untuk mendapatkan banyak tambahan informasi tentang sinyal. Nilai dari atribut untuk suatu sinyal diperoleh dengan menentukan nama sinyal, diikuti dengan (') dan nama atribut yang dikehendaki. Dengan demikian, kondisi dalam contoh adalah benar hanya jika pada saat ini nilai *x* adalah '1' dan nilai sebelumnya adalah '0'. Karena pernyataan ini hanya akan dieksekusi ketika sebuah *event* terjadi pada *x* (yaitu x baru saja berubah), kondisi ini akan benar bila terjadi rising edge pada *x* (karena *x* baru saja berubah, yang sebelumnya adalah'0 'dan sekarang adalah '1'). Bagian *statement* hanyalah berupa daftar dari *sequential statements* yang muncul di antara kata kunci *then* dan *end if*.

Pelaksanaan dari statement *if* diawali dengan mengevaluasi *condition*. Jika *condition* dievaluasikan nilai TRUE maka pernyataan-pernyataan dalam statement body akan dijalankan. Jika tidak, eksekusi akan terus berjalan sampai *end if* dan

statement *if* akan diabaikan. Dengan demikian, *assignment statement* dalam contoh ini dijalankan setiap kali ada *rising edge* di sinyal *x*, dan menghitung jumlah *rising edge*.

Contoh bentuk umum yang lain dari *if statement*

```
...
if (inc='1') then
  cnt:=cnt+1;
else
  cnt:=cnt-1;
end if;
...
```

Bentuk ini memiliki dua statement. Jika kondisi adalah TRUE, baris pertama dari daftar pernyataan dijalankan (antara *then* dan *else*) dan baris kedua dari daftar pernyataan (antara *else* dan *end if*) tidak dijalankan. Sebaliknya, baris kedua dari daftar pernyataan dijalankan dan yang pertama tidak. Dengan demikian, dalam contoh ini *cnt* akan bertambah jika *inc* adalah '1' dan akan berkurang jika sebaliknya.

Pernyataan terakhir adalah *loop statement*. Akan dijelaskan satu bentuk lingkaran pernyataan, yang sering disebut *for statement*. *For statement* digunakan untuk melaksanakan daftar *statement* beberapa kali. Contoh berikut menggunakan *loop statement* untuk menghitung paritas genap dari *bit vector*.

```
signal x : bit_vector (7 downto 0);
...
process (x)
  variable p : bit;
begin
  p:='0'
  for i in 7 downto 0 loop
    p:=p xor x(i);
  end loop;
end process;
```

Sinyal *x* ini merupakan sinyal 8 *bit* yang mewakili sebuah *byte*. Variabel *p* digunakan untuk menghitung paritas *byte* ini. Bagian pertama dari *loop* (*i in 7 downto 0*) disebut *parameter specification*. *Parameter specification* menentukan

berapa kali *loop* akan dijalankan dan akan membuat variabel sementara. Dimulai dengan nama sementara variabel yang akan dibuat, dalam hal ini adalah *i*. Diikuti oleh kata kunci *in* dan kemudian rentang nilai seperti yang telah kita lihat sebelumnya. *Loop* dijalankan sekali untuk setiap nilai dalam kisaran tertentu. Nilai dari variabel sementara diberikan salah satu nilai dalam rentang nilainya setiap kali *loop* dijalankan. Dalam contoh ini, *assignment* yang akan dijalankan pertama *i* = 7 lalu *i* = 6, dan kemudian *i* = 5, dan seterusnya sampai ke 0. Loop statement ini bertindak sama dengan pernyataan seperti ini

```
p:='0';
p:=p xor x(7);
p:=p xor x(6);
p:=p xor x(5);
p:=p xor x(4);
p:=p xor x(3);
p:=p xor x(2);
p:=p xor x(1);
p:=p xor x(0);
```

Perhatikan bagaimana variabel sementara *i* digunakan dalam *loop* beroperasi pada berbagai elemen-elemen vektor x setiap kali *loop* dijalankan. Hal ini merupakan penggunaan loop statement yang sangat umum. Meskipun *loop* hanya satu pernyataan, dimungkinkan ada banyak pernyataan dalam *loop*.

Bagian ini berisi informasi penting tentang penggunaan sinyal dalam proses pernyataan. Persoalan yang penting adalah untuk menghindari kebingungan tentang perbedaan antara bagaimana *signal assignment* dan *variable assignment* berperilaku dalam *process statement*. Harus diingat *signal assignment*, jika apapun, hanya menjadwal acara yang terjadi pada sinyal dan tidak memiliki efek langsung. Bila proses ini dilanjutkan, ia melaksanakan dari atas ke bawah dan tidak ada *event* yang diproses sampai setelah proses selesai. Artinya, jika sebuah *event* dijadwalkan pada sinyal selama pelaksanaan proses, sinyal ini dapat diolah setelah proses sebelumnya selesai. Berikut merupakan contoh dari perilaku ini. Dalam proses berikut dua peristiwa dijadwalkan pada sinyal *x* dan *z*.

```
...
signal x,y,z : bit;
...
```

```
process (y)
begin
  x<=y;
  z<=not x;
end process;
```

Jika sinyal y berubah maka *event* akan dijadwalkan pada *x* agar sama dengan *y*. Selain itu, sebuah *event* dijadwalkan pada *z* agar berkebalikan dari *x*. Pertanyaannya adalah, akankah nilai *z* berlawanan dengan *y*? Tentu saja, jawabannya tidak, karena bila *statement* kedua dijalankan, *event* di x belum diproses, dan *event* yang dijadwalkan pada z akan menjadi berlawanan dari nilai x sebelum proses dimulai.

Hal ini belum tentu merupakan perilaku yang intuitif dan karena variabel beroperasi secara berbeda. Misalnya, dalam

```
process (y)
variable x,z : bit;
begin
  x:=y;
  z:=not x;
end process;
```

Nilai dari variabel *z* akan berlawanan dengan *y* karena nilai dari *x* berubah dengan segera.

Dalam sebagian besar bahasa pemrograman ada mekanisme untuk mencetak teks pada monitor dan mendapatkan masukan dari pengguna melalui keyboard. Walaupun simulator akan membolehkan pengguna memonitor sinyal dan nilai variabel dalam desain, tetapi baik juga untuk meng-output informasi tertentu selama simulasi. Hali ini tidak diberikan sebagai fitur dalam bahasa VHDL, tetapi sebagai *library* standar yang berada pada setiap sistem bahasa VHDL. Dalam VHDL, *common code* dapat dimasukkan ke dalam file terpisah untuk digunakan dalam berbagai desain. *Common code* ini disebut *library*. Untuk menggunakan perpustakaan yang menyediakan kapabilitas input dan output harus ditambahkan statemen

```
use textio.all;
```

di sebelum setiap arsitektur yang menggunakan input dan output. Nama *library*-nya adalah *textio* dan *statement* ini menunjukkan bahwa ingin digunakan semua

library textio. Perlu diketahui bahwa meskipun bukan bagian dari bahasa, *library* adalah standar dan akan sama apapun alat-alat VHDL yang digunakan.

Teks adalah input dan output menggunakan textio melalui variabel dari tipe *line*. Karena variabel digunakan untuk *textio*, input dan output dilakukan dalam proses. Prosedur *outputting information* adalah menempatkannya ke dalam bentuk teks ke dalam variabel tipe *line* dan kemudian meminta baris tersebut menjadi output. Hal ini ditunjukkan dalam contoh berikut.

```
use textio.all;
architecture behavior of check is
begin
  process (x)
    variable s : line;
    variable cnt : integer:=0;
  begin
    if (x='1' and x'last_value='0') then
      cnt:=cnt+1;
      if (cnt>MAX_COUNT) then
        write(s,"Counter overflow - ");
        write(s,cnt);
        writeline(output,s);
      end if;
    end if;
  end process;
end behavior;
```

Fungsi *write* yang digunakan untuk menambahkan informasi teks di akhir baris variabel yang kosong saat simulator diinisialisasi. Fungsi inimembutuhkan dua argumen, yang pertama adalah nama dari baris yang akan ditambahkan, dan yang kedua adalah informasi yang akan ditambahkan. Pada contoh, *s* diatur ke "counter overflow - ", dan kemudian nilai *cnt* dikonversi ke teks dan ditambahkan ke akhir teks. Fungsi yang *writeline* mengeluarkan nilai terbaru ke baris di monitor, dan mengosongkan baris untuk digunakan kembali. Argumen pertama dari fungsi *writeline* hanya menunjukkan bahwa teks harus dioutput ke layar. Jika MAX_COUNT merupakan konstanta sebesar 15 dan lebih dari 15 *rising edges* terjadi pada sinyal x, maka pesan

```
Counter overflow - 16
```

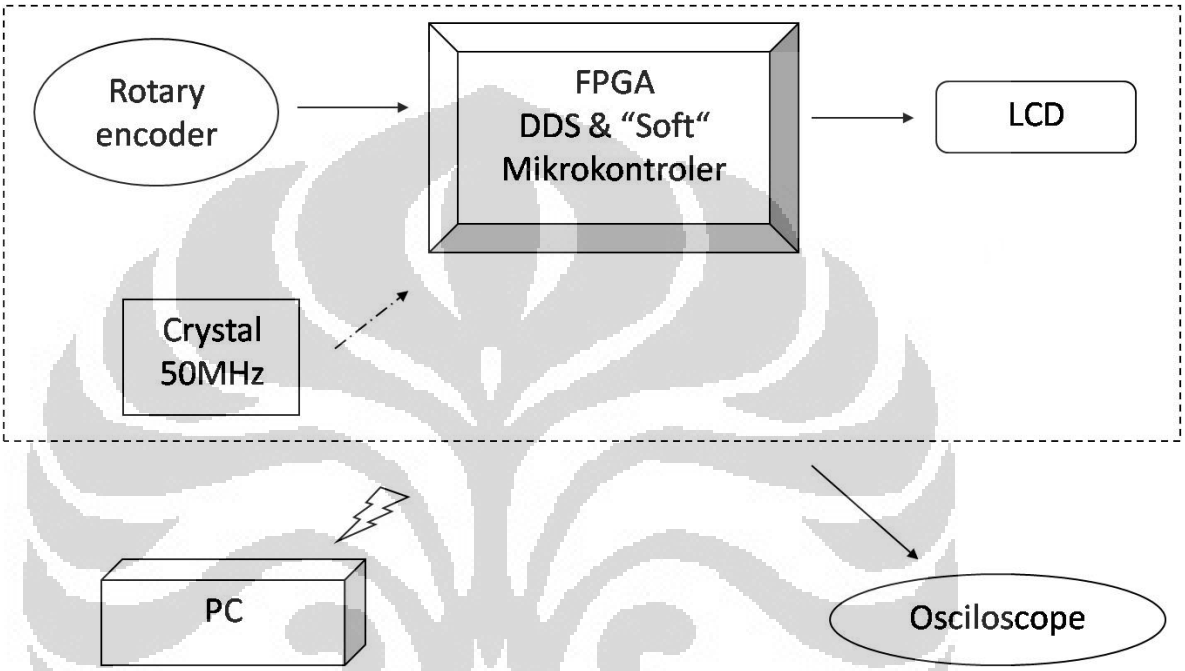
akan tertulis di layar.

Statemen write juga dapat digunakan untuk menambah nilai konstanta dan nilai variabel dan sinyal dari jenis *bit, bit_vector*, waktu, integer, dan riil.

BAB 3
RANGKAIAN DAN RANCANGAN PROGRAM

3.1 Blok Diagram

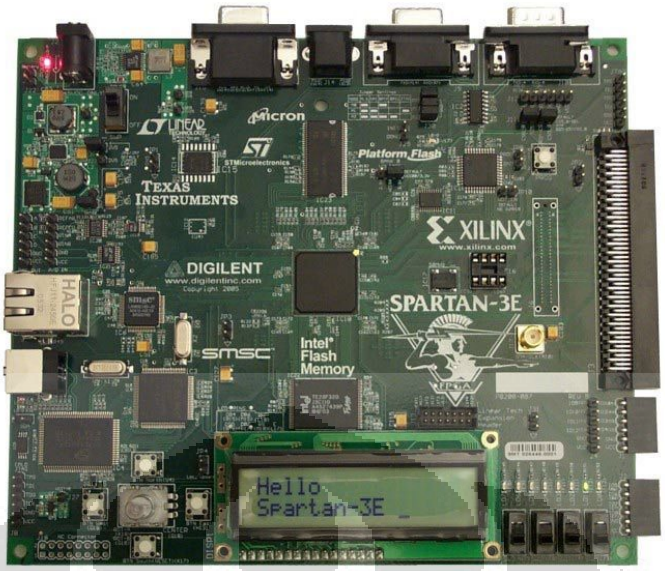
Berikut ini gambar Blok Diagram yang digunakan



Gambar 3.1 Blok Diagram

3.2 FPGA Spartan-3E Starter Kit Board

FPGA yang dipakai adalah Spartan-3E Starter Kit Board. Berikut merupakan bagian-bagian yang ada yang dipakai dalam skripsi ini.



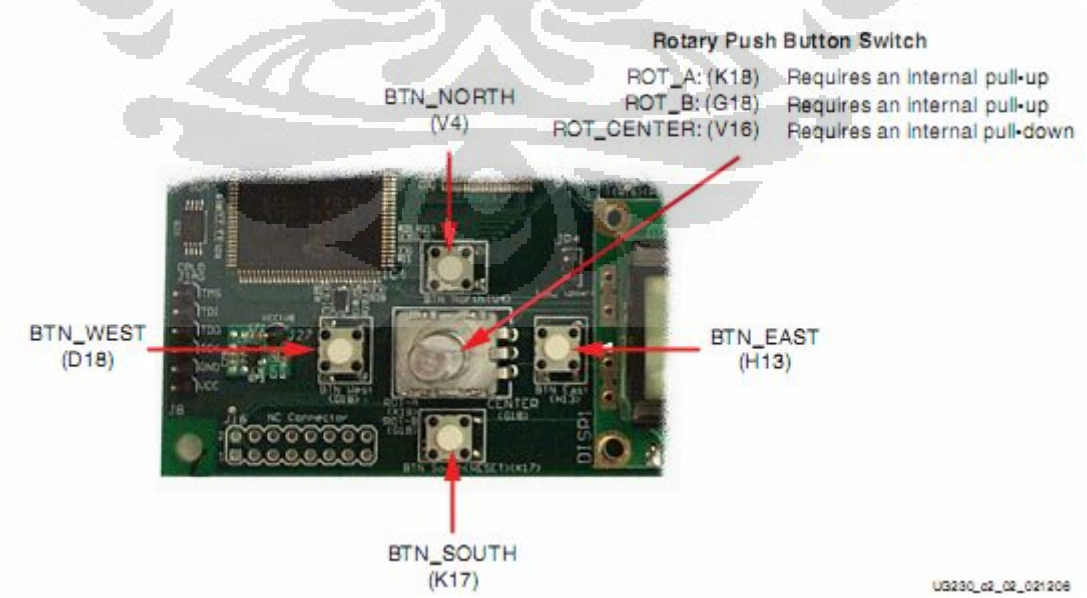
Gambar 3.2 Spartan-3E Starter Kit Board

Sumber : Frequency Generator for Spartan-3E Starter Kit

3.2.1 Rotary Push-Button Switch

3.2.1.1 Lokasi dan Label

Terletak di tengah-tengah antara keempat push-button switch, seperti ditunjukkan Gambar. Switch ini menghasilkan 3 output. Output dua shaft encoder yaitu ROT_A dan ROT_B. Sedangkan untuk center push-button switch yaitu ROT_CENTER.



Gambar 3.3 Push-Button Switch dan Rotary Push-Button Switch

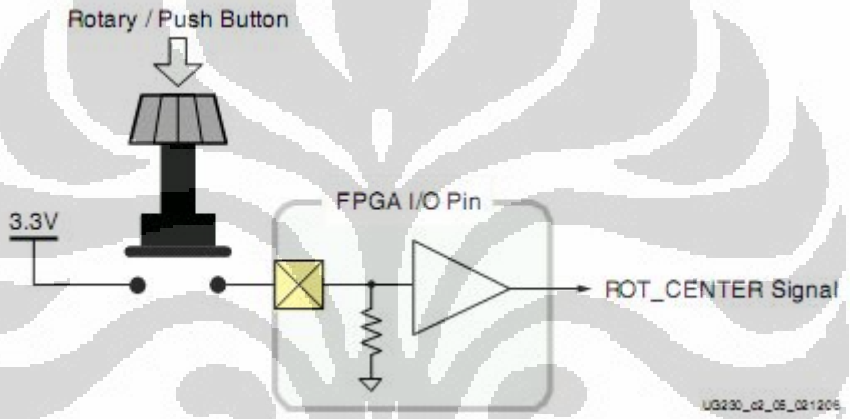
Sumber : Spartan-3E FPGA Family : Complete Data Sheet

3.2.1.2 Operasi

Rotary push-button switch mengintegrasikan 2 fungsi yang berbeda. Saat switch shaft berotasi, maka akan keluar output tertentu. Shaft juga bisa ditekan, seperti push-button switch.

3.2.1.2.1 Push-Button Switch

Dengan menekan knob rotary/push-button switch, maka akan menghubungkan pin FPGA yang berhubungan dengan tegangan 3.3V, seperti ditunjukkan pada Gambar. Untuk mengubah ke mode active low, digunakan resistor pull-down internal di dalam pin FPGA.



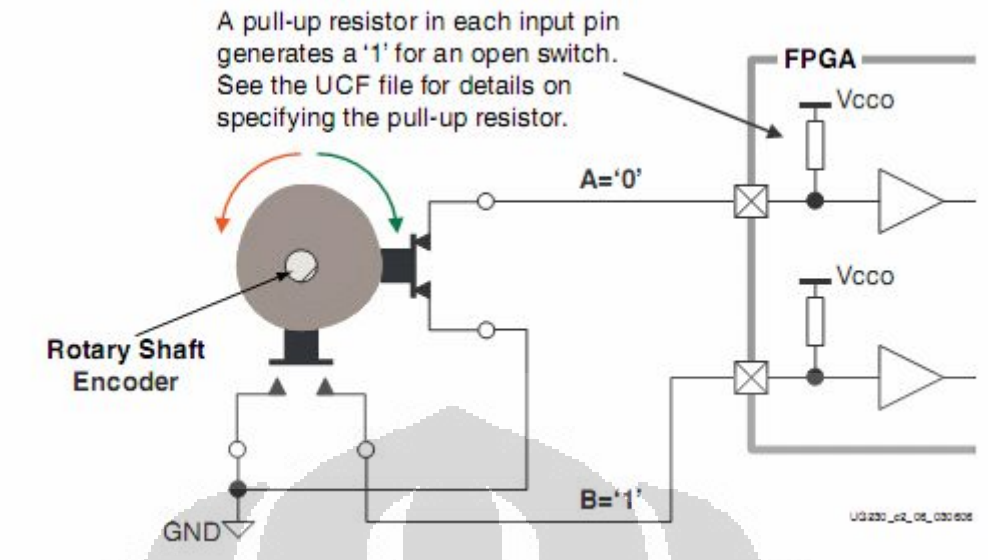
Gambar 3.4 Push-Button Switch dengan Resistor Pull-up pada Pin Input FPGA

Sumber : Spartan-3E FPGA Family : Complete Data Sheet

3.2.1.2.2 Rotary Shaft Encoder

Secara prinsip, rotary shaft encoder bertindak seperti poros, terhubunngkan dengan central shaft. Dengan memutar shaft akan mengoperasikan 2 push-button switch, seperti Gambar.

Salah satu switch terbuka sebelum yang lain, bergantung pada rotasi. Begitu juga saat rotasi berlanjut, salah satu switch menutup sebelum yang lain. Saat shaft stasioner, disebut juga detent position, kedua switch tertutup.



Gambar 3.5 Prinsip Rotary shaft encoder

Sumber : Spartan-3E FPGA Family : Complete Data Sheet

Saat switch terhubung ke ground, akan menghasilkan logika Low. Saat switch terbuka, pull-up resistor di dalam pin FPGA akan meneruskan signal logika High.

3.2.1.3 UCF Location Constraints

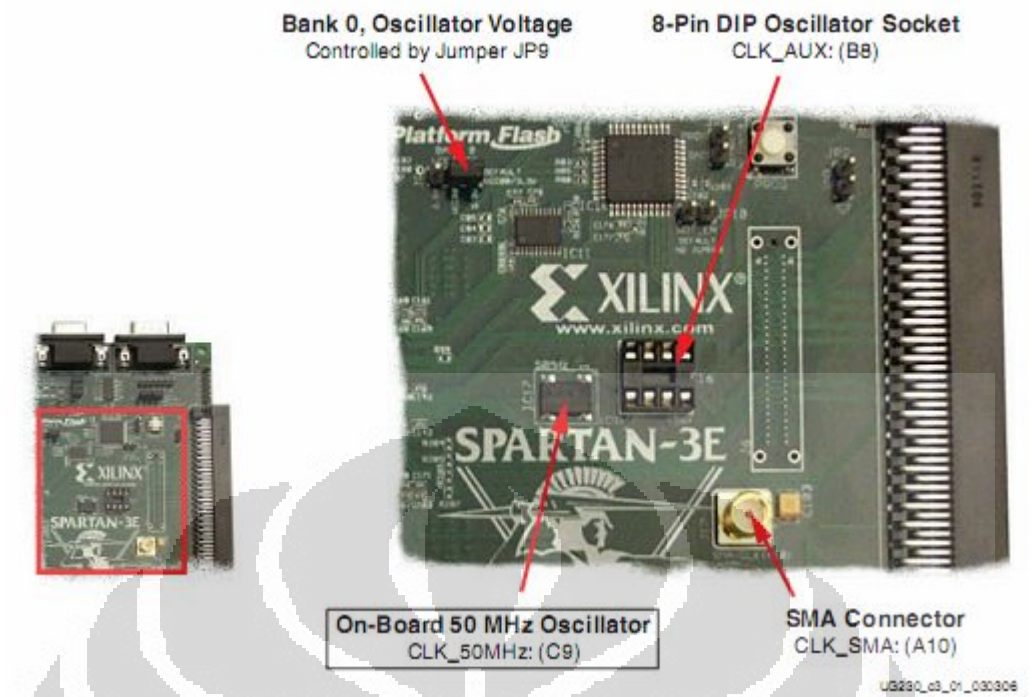
Di bawah ini merupakan definisi UCF constraint untuk rotary push-button switch, termasuk pengaturan pin I/O dan I/O standard, serta mendefinisikan resistor pull-down atau pull-up.

```
NET "ROT_A"      LOC = "K18" | IOSTANDARD = LVTTL | PULLUP   ;
NET "ROT_B"      LOC = "G18" | IOSTANDARD = LVTTL | PULLUP   ;
NET "ROT_CENTER" LOC = "V16" | IOSTANDARD = LVTTL | PULLDOWN ;
```

3.2.2. Clock

Seperti ditunjukkan di Gambar, di Spartan-3E Starter Kit terdapat 3 sumber input input clock.

- Terdapat 50 MHz clock oscillator on-board.
- Clock bisa disuplai lewat SMA-style connector. Di Spartan-3E Starter Kit ini bisa juga menyuplai signal clock atau high-speed signal lainnya lewat SMA-style connector ini.
- Terdapat juga socket 8-pin DIP-style untuk tambahan clock oscillator.



Gambar 3.6 Letak Clock

Sumber : Spartan-3E FPGA Family : Complete Data Sheet

3.2.2.1 Koneksi Clock

Tiap input clock terhubung dengan global buffer input di I/O Bank 0. Seperti ditunjukkan di Tabel, tiap input clock juga terhubung dengan DCM yang berhubungan.

Clock Input	FPGA Pin	Global Buffer	Associated DCM
CLK_50MHZ	C9	GCLK10	DCM_X0Y1
CLK_AUX	B8	GCLK8	DCM_X0Y1
CLK_SMA	A10	GCLK7	DCM_X1Y1

Tabele 3.1 Input Clock, Pin, Global Buffers dan DCM

3.2.2.2 Vol tage Control

Tegangan untuk semua I/O pins di I/O Bank 0 dikontrol oleh jumper JP9. Maka, clock-clock ini juga dikontrol oleh jumper JP9. Default-nya, JP9 diset 3.3V. On-board oscillator bekerja di tegangan 3.3V dan mungkin tidak akan bekerja sesuai harapan jika JP9 diset 2.5V.

3.2.2.3 50 MHz On-Board Oscillator

50 MHz on-board oscillator bekerja dengan 40% sampai 60% output duty cycle. Oscillator ini memiliki keakuratan ±2500 Hz atau ±50 ppm.

3.2.2.4 UCF Constraints

Clock input membutuhkan 2 tipe constraint. Constrain lokasi mendefinisikan I/O pin dan I/O standards. Constrain period mendefinisikan periode (tentunya juga frekuensi) dan duty cycle of signal clock.

3.2.2.4.1 Lokasi

Di bawah ini merupakan UCF constraint untuk ketiga input clock, termasuk pengaturan I/O pin dan I/O standard. Dalam pengaturan ini jumper JP9 diset 3.3V. Jika JP9 ingin diset 2.5V, maka IOSTANDARD dapat diubah.

```
NET "CLK_50MHZ" LOC = "C9"  | IOSTANDARD = LVCMOS33 ;
NET "CLK_SMA"   LOC = "A10" | IOSTANDARD = LVCMOS33 ;
NET "CLK_AUX"   LOC = "B8"  | IOSTANDARD = LVCMOS33 ;
```

3.2.2.4.1 Clock Period

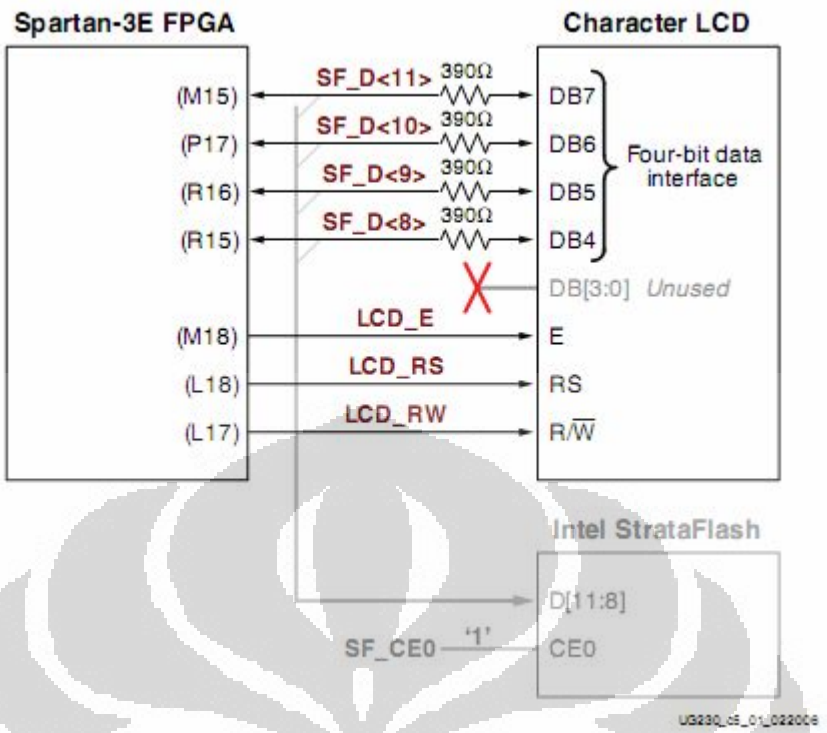
Untuk mengatur duty cycle, antara 40% sampai 60%, dapat dilihat dibawah ini.

```
# Define clock period for 50 MHz oscillator
NET "CLK_50MHZ" PERIOD = 20.0ns HIGH 40%;
```

3.2.3. LCD

Spartan-3E FPGA Starter Kit sudah menyediakan 16x2 karakter LCD (16 = kolom dan 2 = baris). FPGA mengontrol LCD via 4-bit data interface seperti ditunjukkan Gambar. Walaupun LCD men-suport 8-bit data interface, Starter Kit ini menggunakan 4-bit data interface supaya compatible dengan Xilinx development board lainnya dan untuk meminimalisasi total pin.

Processor PicoBlaze mengontrol display timing plus konten display-nya.



Gambar 3.7 Interface LCD

Sumber : Spartan-3E FPGA Family : Complete Data Sheet

3.2.3.1 Character LCD Interface Signals

Tabel di bawah ini menunjukkan interface character LCD dan signal.

Signal Name	FPGA Pin	Function	
SF_D<11>	M15	Data bit DB7	Shared with StrataFlash pins SF_D <11:8>
SF_D<10>	P17	Data bit DB6	
SF_D<9>	R16	Data bit DB5	
SF_D<8>	R15	Data bit DB4	
LCD_E	M18	Read/Write Enable Pulse 0: Disabled 1: Read/Write operation enabled	
LCD_RS	L18	Register Select 0: Instruction register during write operations. Busy Flash during read operations 1: Data for read or write operations	

LCD_RW	L17	Read/Write Control 0: WRITE, LCD accepts data 1: READ, LCD presents data

Tabel 3.2 Character LCD Interface

Sumber : Spartan-3E FPGA Family : Complete Data Sheet

3.2.3.1 Voltage Compatibility

LCD disuplai oleh tegangan +5V. Signal I/O FPGA berada di tegangan 3.3V. Akan tetapi LCD dapat menerima 5V TTL signal level dan output 3.3V LVCMOS yang disediakan FPGA memenuhi syarat 5V TTL tersebut.

Resistor 390Ω mencegah overstressing pada pin FPGA dan StrataFlash I/O saat LCD men-drive logika High.

3.2.3.3 Interaksi dengan Intel StrataFlash

Seperti ditunjukkan Gambar, data signal LCD juga di-share dengan data line StrataFlash SF_D<11:8>.

SF_CE0	SF_BYTE	LCD_RW	Operation
1	X	X	StrataFlash disabled. Full read/write access to LCD.
X	X	0	LCD write access only. Full access to StrataFlash.
X	0	X	StrataFlash in byte-wide (x8) mode. Upper address lines are not used. Full access to both LCD and StrataFlash.

Tabel 3.3 Kontrol Interaksi LCD dan StrataFlash

Sumber : Spartan-3E FPGA Family : Complete Data Sheet

3.2.3.3 UCF Location Constraint

Di bawah ini merupakan UCF constraint untuk LCD, termasuk pengaturan I/O pin dan I/O standard.

```
NET "LCD_E"   LOC = "M18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
NET "LCD_RS"  LOC = "L18" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
NET "LCD_RW"  LOC = "L17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
```

```
# The LCD four-bit data interface is shared with the StrataFlash.
NET "SF_D<8>"  LOC = "R15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
NET "SF_D<9>"  LOC = "R16" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
NET "SF_D<10>" LOC = "P17" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
NET "SF_D<11>" LOC = "M15" | IOSTANDARD = LVCMOS33 | DRIVE = 4 | SLEW =
SLOW ;
```

3.4 Perancangan Perangkat Lunak

Bahasa pemrograman yang dipakai adalah VHDL. Penjelasan mengenai VHDL dapat dilihat di Bab 2.

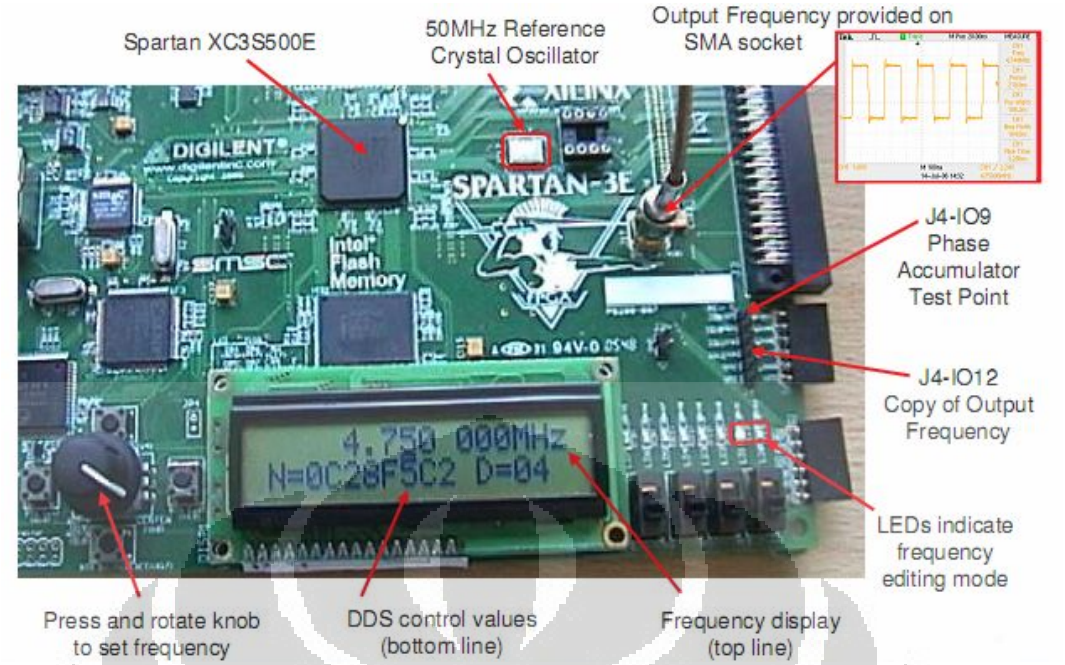
3.4.1 Penjelasan Umum

Program ini menyediakan frequency generator dengan resolusi 1Hz dengan frekuensi maksimum 100MHz (dengan clock internal 200MHz).Oscillator on board 50MHz digunakan sebagai referensi untuk rangkaian Direct Digital Synthesis (DDS) yang mengeluarkan frekuensi antara 6,25MHz sampai 12,5MHz. Lalu frekuensi ini dikalikan (multiplied) oleh DCM dengan menggunakan teknik khusus untuk mereduksi cycle to cycle jitter yang dapat terjadi. Akhirnya, counter dan multiplexer digunakan untuk membagi (divide) menjadi frekuensi output yang diinginkan. PicoBlaze digunakan sebagai human interface pada rotary knob dan display LCD yang digunakan untuk mengeset frekuensi yang diinginkan. PicoBlaze juga melakukan beberapa kalkulasi presisi tinggi untuk menghasilkan nilai-nilai pengontrol yang diperlukan di dalam rangkaian DDS.

Program dapat dilihat pada bagian Lampiran.

Port	Kegunaan
J4-IO9	Tes point untuk phase accumulator
SMA Socket	Output frekuensi
J4-IO12	Copy dari output frekuensi

Tabel 3.4 Port Output



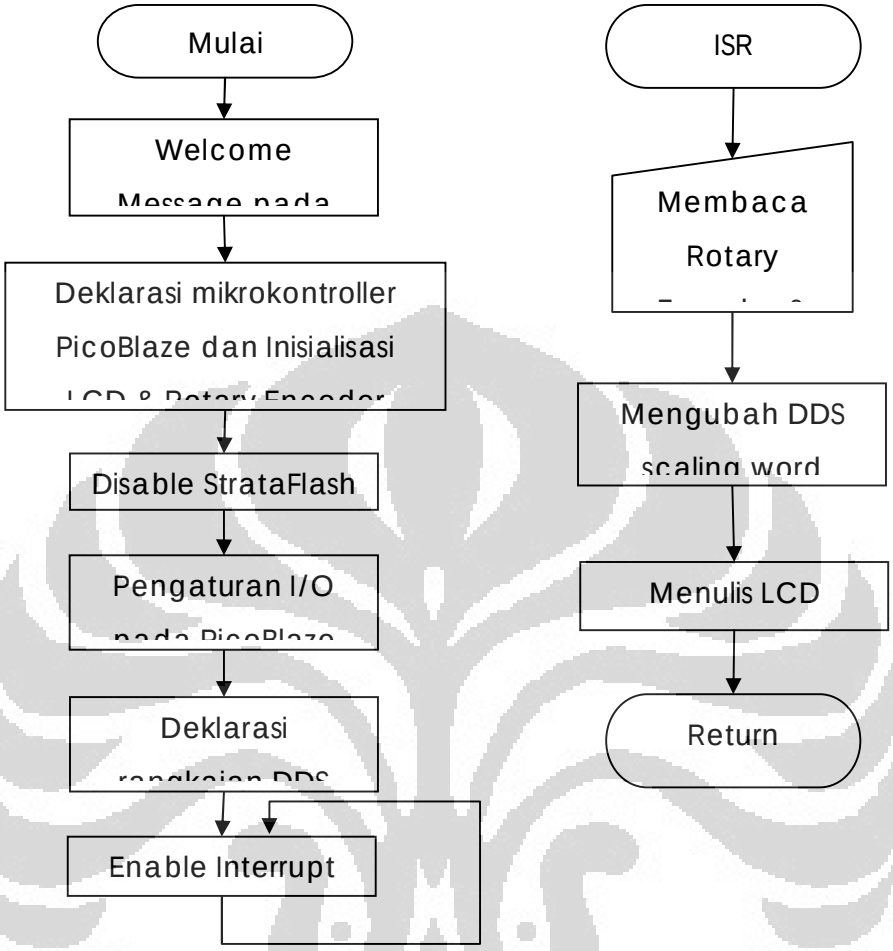
Gambar 3.8 Penjelasan Penggunaan Program

Sumber : Frequency Generator for Spartan-3E Starter Kit.

frequency_generator.vhd	Top level file dan merupakan deskripsi dari hardware. Terdapat pengaturan I/O yang diperlukan untuk mematikan fungsi StrataFlash memory yang dapat menginterferensi display LCD.
frequency_generator.ucf	File I/O constrain untuk Spartan-3E Starter Kit. Terdapat spesifikasi timing untuk kontroler 50MHz PicoBlaze untuk rangkaian DDS 200MHz.
kcpsm3.vhd	program untuk mikroprosesor PicoBlaze pada Spartan-3E Starter Kit.
fg_ctrl.vhd	program assembly untuk PicoBlaze (disimpan di blok memori)
fg_ctrl.psm	PicoBlaze program source assembler code

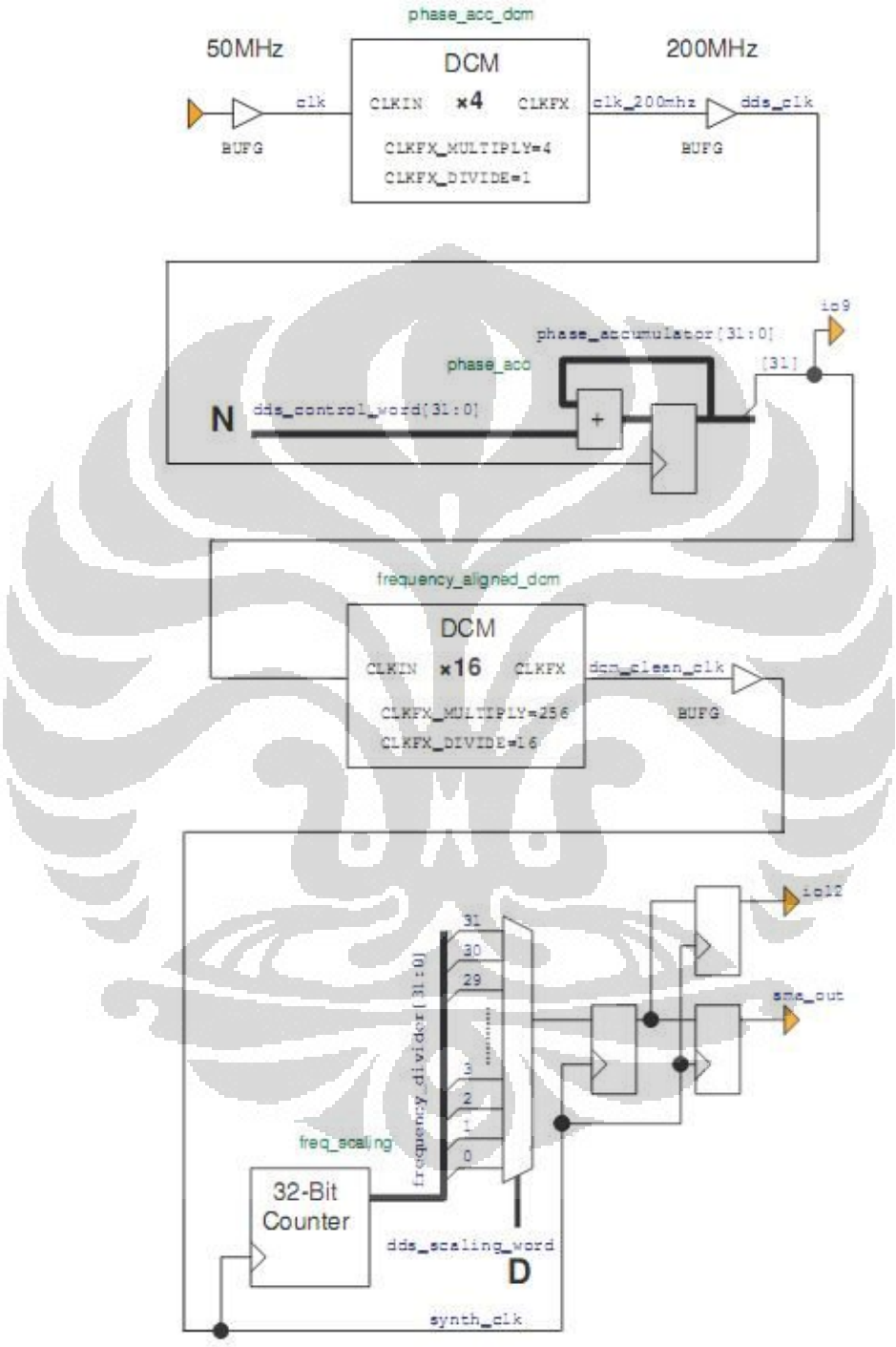
Gambar 3.9 File

3.4.2 Flowchart



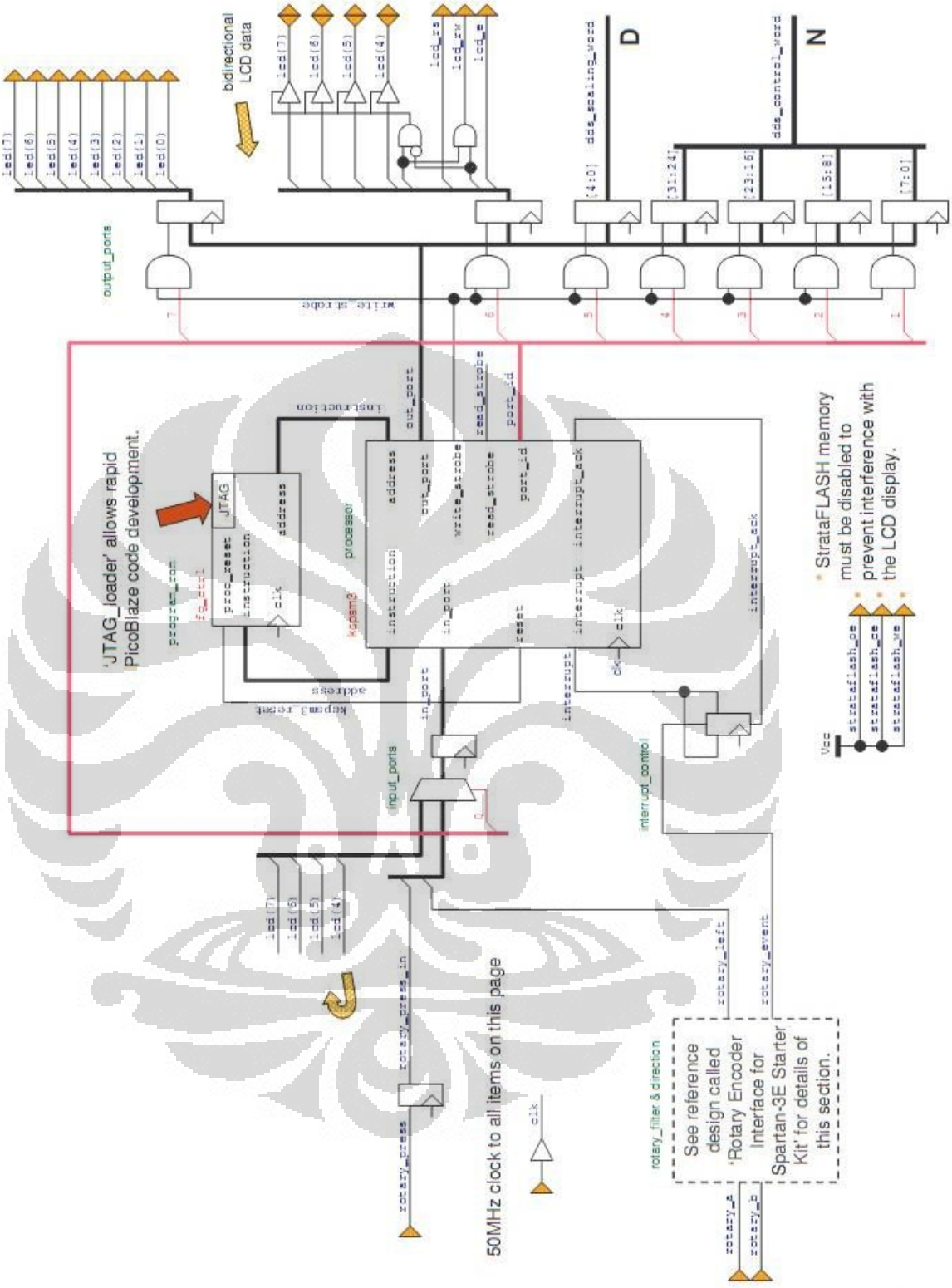
Gambar 3.10 Flowchart

3.4.2 Rangkaian DDS dan Picoblaze



Gambar 3.11 Rangkaian DDS

Sumber : Frequency Generator for Spartan-3E Starter Kit. (telah diolah kembali)



Gambar 3.12 Rangkaian PicoBlaze

Sumber : Frequency Generator for Spartan-3E Starter Kit.

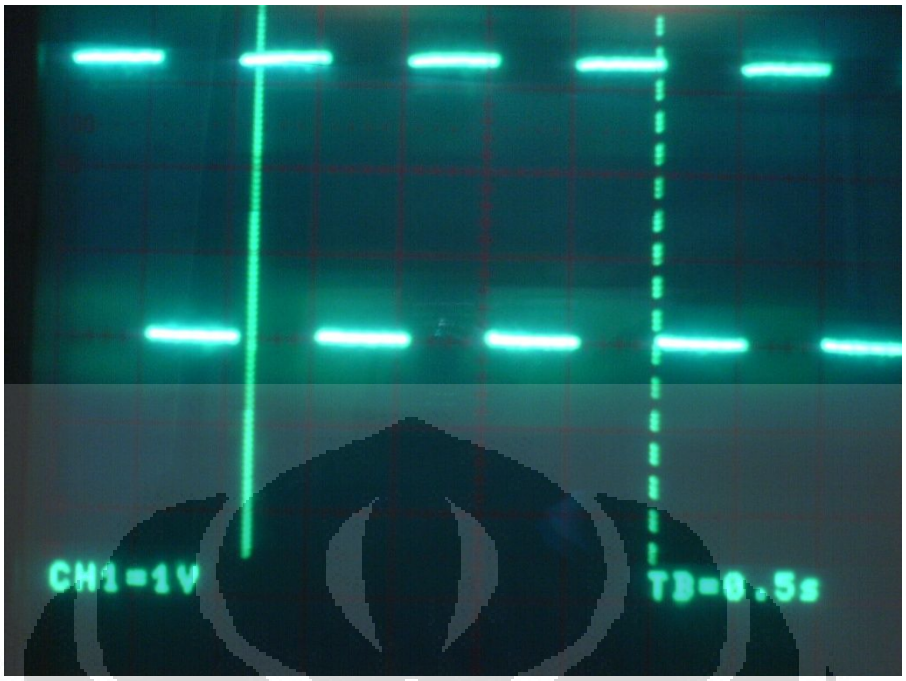
BAB IV
PENGUJIAN SISTEM DAN ANALISA

Pada bab ini akan dibahas tentang pengujian dan analisa sistem yang telah dikerjakan. Pengujian ini dilakukan untuk mengetahui kemampuan sistem apakah telah berfungsi seperti apa yang diharapkan dan menganalisa apabila terjadi kegagalan.

4.1 Pengujian Frekuensi

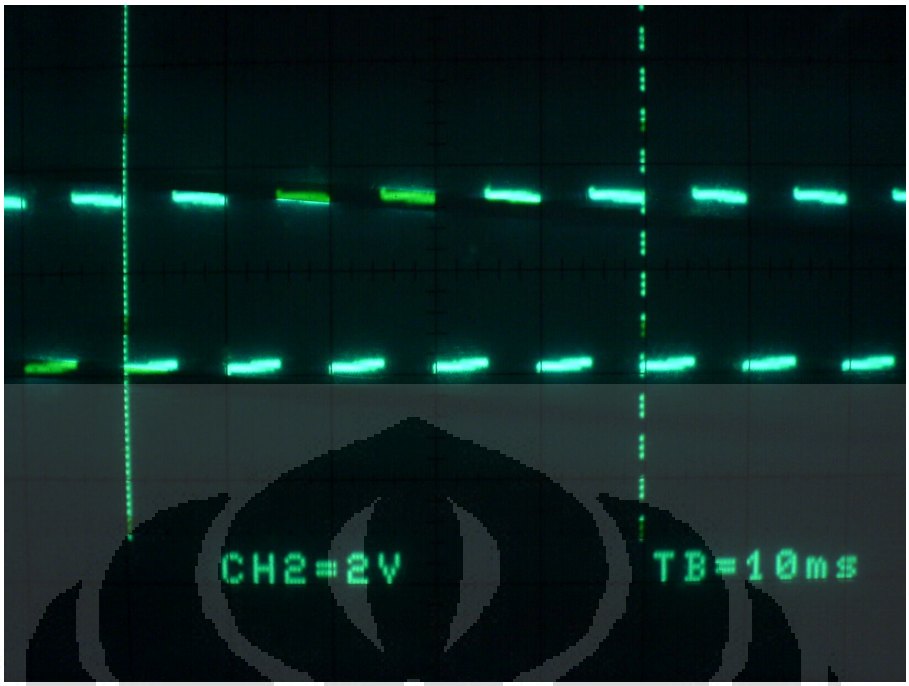
Dilakukan pengujian terhadap frekuensi yang keluar dari FPGA dengan dihubungkan ke osiloskop dengan mode AC.

Gambar 4.1 Frekuensi 1Hz

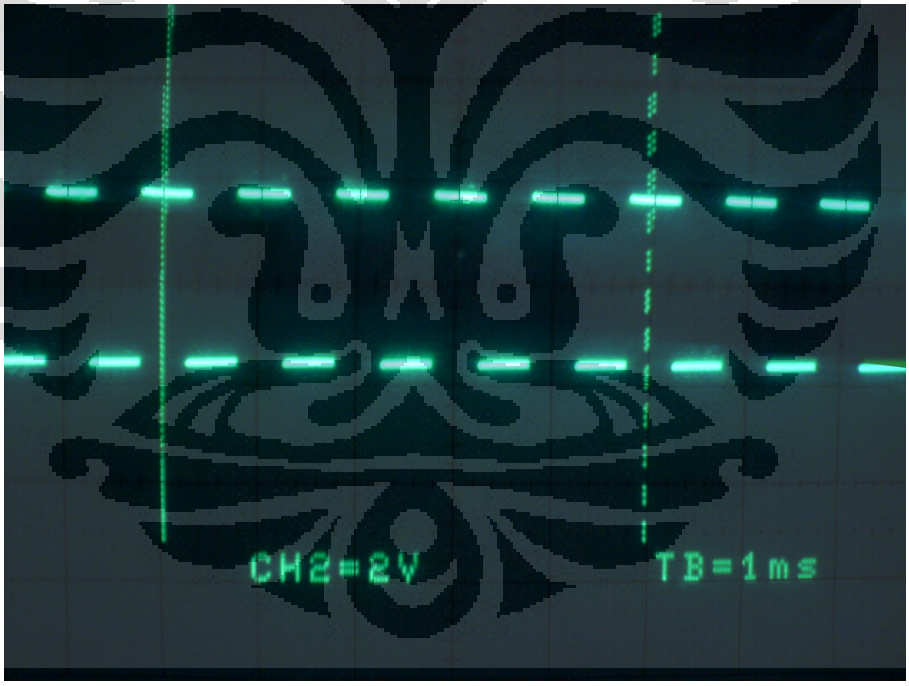


Gambar 4.2 Frekuensi 1Hz (mode DC)

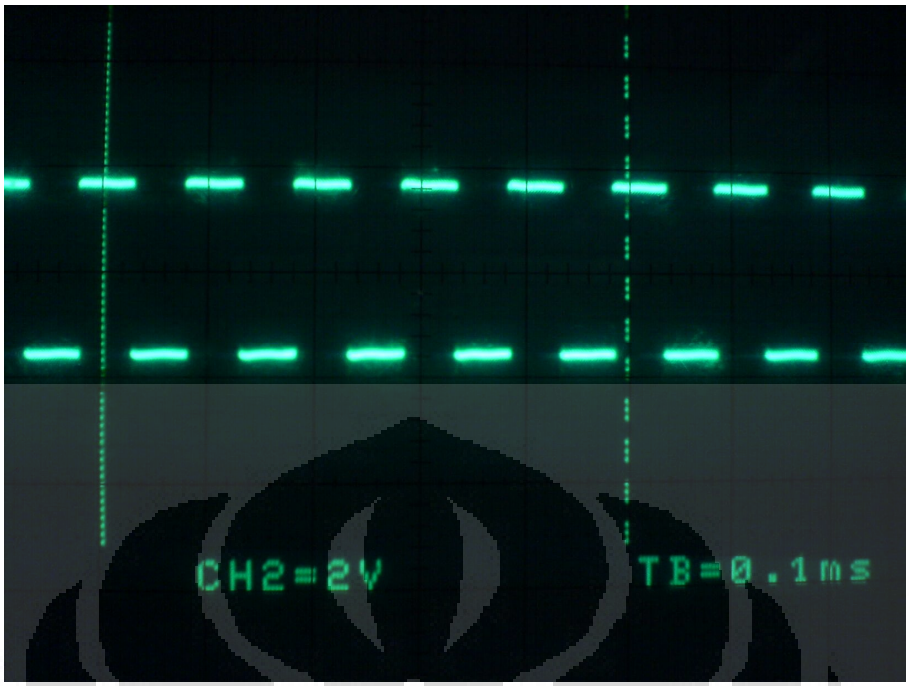
Gambar 4.3 Frekuensi 10Hz



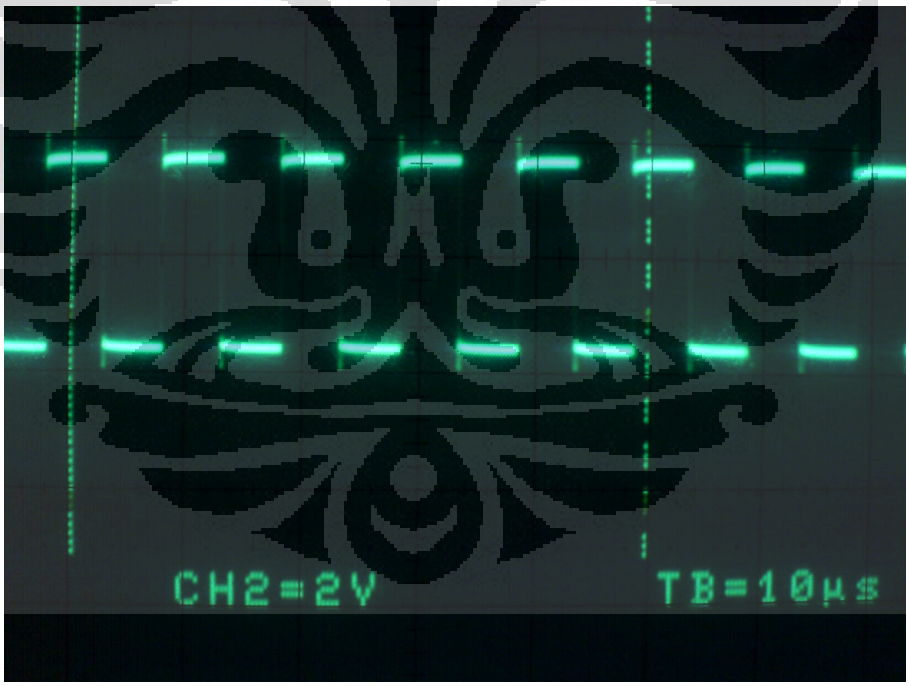
Gambar 4.4 Frekuensi 100Hz



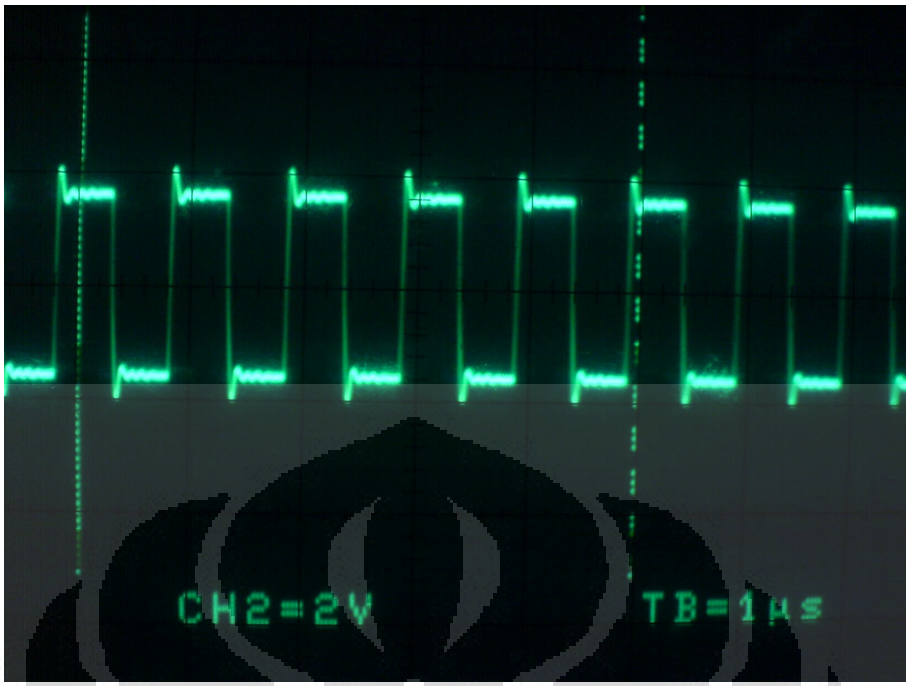
Gambar 4.5 Frekuensi 1KHz



Gambar 4.6 Frekuensi 10KHz

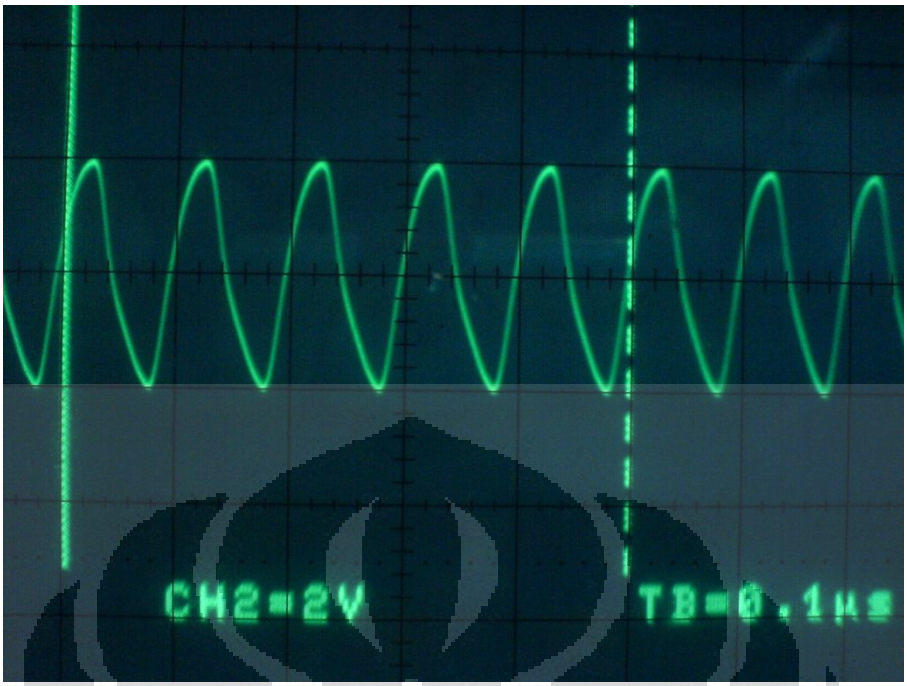


Gambar 4.7 Frekuensi 100KHz

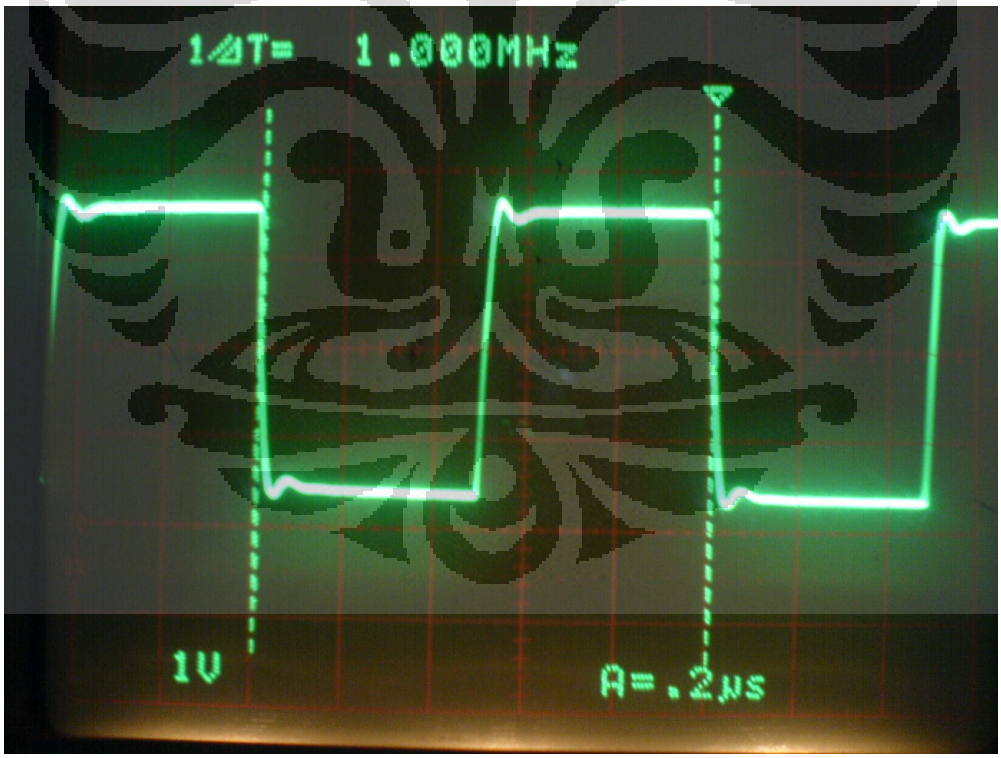


Gambar 4.8 Frekuensi 1MHz

Gambar 4.9 Frekuensi 10MHz



Gambar 4.10 Frekuensi 100MHz



Gambar 4.11 Frekuensi 1MHz Dengan Keterangan

4.2 Analisa

Pada pengujian untuk frekuensi rendah (1Hz – 10Hz), terlihat tidak dihasilkan gelombang kotak sempurna. Tetapi, setelah mengubah mode coupling pada oscilloscope, yaitu dari AC ke DC, dihasilkan gelombang kotak yang sempurna. Hal ini mungkin disebabkan karena frekuensi yang dekat ke frekuensi DC dan juga keterbatasan oscilloscope.

Pengujian untuk frekuensi tinggi (>10MHz) tidak dapat dilakukan karena keterbatasan oscilloscope. Pada oscilloscope hanya bisa mengukur dengan teliti sampai ~10MHz.

Pada gambar frekuensi 1MHz dengan keterangan, dapat dilihat, dihasilkan frekuensi kotak dengan frekuensi tepat 1MHz dengan amplitudo sekitar 3,3 V.

BAB V
KESIMPULAN DAN SARAN

5.1 Kesimpulan

- FPGA memiliki banyak aplikasi, salah satunya adalah function generator
- Dalam penelitian ini, dipakai hanya 2 chip yaitu crystal oscillator, FPGA, dengan 2 peripheral LCD dan rotary encoder.
- Untuk menghasilkan frekuensi yang presisi dapat digunakan metode DDS
- Bila diinginkan frekuensi yang fix, dapat dilakukan dengan membuat nilai N konstan dan dengan fixed counter divider
- Dengan mengganti oscillator yang dipakai (on-board) dengan oscillator lainnya, dapat menghasilkan rentang frekuensi yang berbeda
- Dapat dihasilkan frekuensi dengan rentang 1Hz – 100MHz, dengan amplitudo sekitar 3,3V (logika 1 / high). Karena keterbatasan alat uji, pada frekuensi tinggi (>10MHz) amplitudonya berkurang.

5.2 Saran

Untuk dapat mengukur frekuensi dengan presisi, khususnya untuk frekuensi tinggi, diperlukan alat-alat, seperti osiloskop, probe, kabel, dll, yang dapat digunakan untuk frekuensi tinggi.

Supaya gelombang frekuensi tidak hanya kotak, tetapi dapat menjadi sinus, segitiga, ataupun sawtooth, maka diperlukan pembelajaran terhadap ADC yang berada di Spartan-3E Starter Kit ini.

Lalu dapat juga digabungkan dengan program frequency counter, yang akan berguna untuk mengukur output frekuensi bila dihubungkan dengan suatu alat / chip yang akan diuji.

DAFTAR PUSTAKA

Chu, Pong P. (2008). *FPGA Prototyping by VHDL Examples Xilinx SpartanTM-3 Version*. New Jersey : John Wiley and Sons.

Grout, Ian. (2008) *Digital Systems Design With FPGAs and CPLDs*, Oxford : Newnes.

Dubey, Rahul. (2009) *Introduction to Embedded Systems Using Field-Programmable Gate Array*, London : Springer.

Xilinx, Inc. (18 April 2008). Spartan-3E FPGA Family : Complete Data Sheet. 15 Mei 2009. http://www.xilinx.com/support/documentation/data_sheets/ds312.pdf

Chapman, Ken. (18 Juli 2006). Frequency Generator for Spartan-3E Starter Kit. 15 Mei 2009. http://www.xilinx.com/products/boards/s3estarter/files/s3esk_frequency_generator.pdf

Wikimedia Foundation, Inc. (28 April 2009). 15 Mei 2009. Field Programmable Gate Array. http://en.wikipedia.org/wiki/Field-programmable_gate_array

Green Mountain Computing Systems. (1995). 15 Mei 2009. http://www.gmvhdl.com/tutorial.htm

FPGAforFun.com. (December 12 2008). 15 Mei 2009. http://www.fpga4fun.com/

```
--
-- Reference design - Frequency Generator for the Spartan-3E Starter Kit.
--
-- Ken Chapman - Xilinx Ltd - 12th July 2006
--
--
--  *** This design contains an evaluation test feature of the DCM.   ****
--  *** Before this design can be processed a special BITGEN option   ****
--  *** needs to be set. Please read the notes provided in the design ****
--  *** documentation or read the comments in this file for details   ****
--  *** of this special requirement.                                  ****
--
--
-- Provides a frequency generator with resolution of 1Hz up to a maximum
frequency of
-- approximately 100MHz (with internal 200MHz clocks). The controller is
deliberately supports
-- higher frequencies so that you can experiment with finding the limits of the
Spartan device
-- fitted to your board.
--
-- The 50MHz on board oscillator is used as the basic reference for a Direct
Digital Synthesis (DDS)
-- circuit which synthesizes frequencies in the range 6.25MHz to 12.5MHz. This
is then multiplied up
-- by a DCM using a special mode to reduce the cycle to cycle jitter content that is
present in the
-- synthesized waveform. Finally a counter and multiplexer are used to divide
down to the desired
-- output frequency.
--
```

```
-- PicoBlaze is used to provide a human interface in which the rotary knob and
LCD display are
-- used to set the frequency required. PicoBlaze also performs some very high
precision calculations
-- in order to generate the required control values for the DDS circuit.
--
-- LEDs are used to indicate status and what would a design be without an LED
flashing?
--
----------------------------------------------------------------------------
--
-- NOTICE:
--
-- Copyright Xilinx, Inc. 2006.   This code may be contain portions patented by
other
-- third parties.  By providing this core as one possible implementation of a
standard,
-- Xilinx is making no representation that the provided implementation of this
standard
-- is free from any claims of infringement by any third party.  Xilinx expressly
-- disclaims any warranty with respect to the adequacy of the implementation,
including
-- but not limited to any warranty or representation that the implementation is free
-- from claims of any third party.  Furthermore, Xilinx is providing this core as a
-- courtesy to you and suggests that you contact all third parties to obtain the
-- necessary rights to use this implementation.
--
----------------------------------------------------------------------------
--
-- Library declarations
--
-- Standard IEEE libraries
```

```
--
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
--
-- The Unisim Library is used to define Xilinx primitives. It is also used during
-- simulation. The source can be viewed at
%XILINX%\vhdl\src\unisims\unisim_VCOMP.vhd
--
library unisim;
use unisim.vcomponents.all;
--
--
----------------------------------------------------------------------------------
--
--
entity frequency_generator is
    Port (         sma_out : out std_logic;
                   simple : out std_logic_vector(12 downto 9);
                      led : out std_logic_vector(7 downto 0);
           strataflash_oe : out std_logic;
           strataflash_ce : out std_logic;
           strataflash_we : out std_logic;
                    lcd_d : inout std_logic_vector(7 downto 4);
                   lcd_rs : out std_logic;
                   lcd_rw : out std_logic;
                    lcd_e : out std_logic;
                rotary_a : in std_logic;
                rotary_b : in std_logic;
             rotary_press : in std_logic;
                      clk : in std_logic);
```

```
    end frequency_generator;
--
------------------------------------------------------------------------------
--
-- Start of test architecture
--
architecture Behavioral of frequency_generator is
--
------------------------------------------------------------------------------
--
-- declaration of KCPSM3
--
  component kcpsm3
    Port (      address : out std_logic_vector(9 downto 0);
            instruction : in std_logic_vector(17 downto 0);
                port_id : out std_logic_vector(7 downto 0);
           write_strobe : out std_logic;
               out_port : out std_logic_vector(7 downto 0);
           read_strobe : out std_logic;
                in_port : in std_logic_vector(7 downto 0);
              interrupt : in std_logic;
          interrupt_ack : out std_logic;
                  reset : in std_logic;
                    clk : in std_logic);
    end component;
--
-- declaration of program ROM
--
  component fg_ctrl
    Port (      address : in std_logic_vector(9 downto 0);
            instruction : out std_logic_vector(17 downto 0);
             proc_reset : out std_logic;                    --JTAG Loader version
```

```
                clk : in std_logic);
  end component;
--
----------------------------------------------------------------------------------
--
-- Signals used to connect KCPSM3 to program ROM and I/O logic
--
signal  address         : std_logic_vector(9 downto 0);
signal  instruction     : std_logic_vector(17 downto 0);
signal  port_id         : std_logic_vector(7 downto 0);
signal  out_port        : std_logic_vector(7 downto 0);
signal  in_port         : std_logic_vector(7 downto 0);
signal  write_strobe    : std_logic;
signal  read_strobe     : std_logic;
signal  interrupt       : std_logic :='0';
signal  interrupt_ack   : std_logic;
signal  kcpsm3_reset    : std_logic;
--
--
-- Signals for LCD operation
--
-- Tri-state output requires internal signals
-- 'lcd_drive' is used to differentiate between LCD and StrataFLASH
communications
-- which share the same data bits.
--
signal   lcd_rw_control : std_logic;
signal  lcd_output_data : std_logic_vector(7 downto 4);
signal       lcd_drive : std_logic;
--
--
-- Signals used to interface to rotary encoder
```

```
--
signal       rotary_a_in : std_logic;
signal       rotary_b_in : std_logic;
signal   rotary_press_in : std_logic;
signal         rotary_in : std_logic_vector(1 downto 0);
signal         rotary_q1 : std_logic;
signal         rotary_q2 : std_logic;
signal   delay_rotary_q1 : std_logic;
signal      rotary_event : std_logic;
signal       rotary_left : std_logic;
--
--
-- Signals used to form DDS circuit
--
signal        clk_200mhz : std_logic;
signal           dds_clk : std_logic;
signal  dds_control_word : std_logic_vector (31 downto 0);
signal  dds_scaling_word : std_logic_vector (4 downto 0);
--
signal phase_accumulator : std_logic_vector (31 downto 0);
signal   dcm_clean_clock : std_logic;
signal         synth_clk : std_logic;
--
signal frequency_divider : std_logic_vector (31 downto 0);
signal     frequency_out : std_logic;
signal     freq_out_pipe : std_logic;
--
--

------------------------------------------------------------------------------------------------
-----------------------------------------------------------------
--
```

```
-- Start of circuit description
--
begin
  --
  ------------------------------------------------------------------------------------------------
--------------------------------
  -- Disable unused components
  ------------------------------------------------------------------------------------------------
--------------------------------
  --
  --StrataFLASH must be disabled to prevent it conflicting with the LCD display
  --
  strataflash_oe <= '1';
  strataflash_ce <= '1';
  strataflash_we <= '1';
  --
  --
  ------------------------------------------------------------------------------------------------
--------------------------------
  -- KCPSM3 and the program memory
  ------------------------------------------------------------------------------------------------
--------------------------------
  --

  processor: kcpsm3
    port map(      address => address,
                instruction => instruction,
                    port_id => port_id,
               write_strobe => write_strobe,
                   out_port => out_port,
                read_strobe => read_strobe,
                    in_port => in_port,
```

```
              interrupt => interrupt,
            interrupt_ack => interrupt_ack,
                  reset => kcpsm3_reset,
                    clk => clk);

  program_rom: fg_ctrl
    port map(      address => address,
              instruction => instruction,
               proc_reset => kcpsm3_reset,                    --JTAG Loader version
                    clk => clk);

  --
  ------------------------------------------------------------------------------------------
--------------------------------
  -- Interrupt
  ------------------------------------------------------------------------------------------
--------------------------------
  --
  --
  -- Interrupt is used to detect rotation of the rotary encoder.
  -- It is anticipated that the processor will respond to interrupts at a far higher
  -- rate that the rotary control can be operated and hence events will not be
missed.
  --

  interrupt_control: process(clk)
  begin
    if clk'event and clk='1' then

      -- processor interrupt waits for an acknowledgement
      if interrupt_ack='1' then
        interrupt <= '0';
```

```
      elsif rotary_event='1' then
       interrupt <= '1';
      else
       interrupt <= interrupt;
     end if;

   end if;
 end process interrupt_control;

 --
 ------------------------------------------------------------------------------------------
-------------------------------
 -- KCPSM3 input ports
 ------------------------------------------------------------------------------------------
-------------------------------
 --
 --
 -- The inputs connect via a pipelined multiplexer
 --

 input_ports: process(clk)
 begin
   if clk'event and clk='1' then

     case port_id(0) is

       -- read rotary control signals at address 00 hex
       when '0' =>    in_port <=  "000000" & rotary_press_in & rotary_left ;

       -- read LCD data at address 01 hex
       when '1' =>    in_port <= lcd_d & "0000";
```

```
    -- Don't care used for all other addresses to ensure minimum logic
implementation
    when others =>    in_port <= "XXXXXXXX";

   end case;

  end if;

 end process input_ports;


 --
 ------------------------------------------------------------------------------------
-------------------------------
 -- KCPSM3 output ports
 ------------------------------------------------------------------------------------
-------------------------------
 --

 -- adding the output registers to the processor

 output_ports: process(clk)
 begin

  if clk'event and clk='1' then
   if write_strobe='1' then

    -- Write to LEDs at address 80 hex.

    if port_id(7)='1' then
     led <= out_port;
    end if;
```

```
-- LCD data output and controls at address 40 hex.

if port_id(6)='1' then
  lcd_output_data <= out_port(7 downto 4);
  lcd_drive <= out_port(3);
  lcd_rs <= out_port(2);
  lcd_rw_control <= out_port(1);
  lcd_e <= out_port(0);
end if;

-- Write DDS frequency scaling word at addresses 20 hex.

if port_id(5)='1' then
  dds_scaling_word <= out_port(4 downto 0);
end if;

-- Write 32-bit DDS control word at addresses 02, 04, 08 and 10 hex.

if port_id(4)='1' then
  dds_control_word(31 downto 24) <= out_port;
end if;

if port_id(3)='1' then
  dds_control_word(23 downto 16) <= out_port;
end if;

if port_id(2)='1' then
  dds_control_word(15 downto 8) <= out_port;
end if;

if port_id(1)='1' then
```

```
          dds_control_word(7 downto 0) <= out_port;
        end if;

      end if;

    end if;

  end process output_ports;

  --
  ------------------------------------------------------------------------------------------
--------------------------------
  -- LCD interface
  ------------------------------------------------------------------------------------------
--------------------------------
  --
  -- The 4-bit data port is bidirectional.
  -- lcd_rw is '1' for read and '0' for write
  -- lcd_drive is like a master enable signal which prevents either the
  -- FPGA outputs or the LCD display driving the data lines.
  --
  --Control of read and write signal
  lcd_rw <= lcd_rw_control and lcd_drive;

  --use read/write control to enable output buffers.
  lcd_d <= lcd_output_data when (lcd_rw_control='0' and lcd_drive='1') else
"ZZZZ";

  --
  ------------------------------------------------------------------------------------------
--------------------------------
  -- Interface to rotary encoder.
```

```
 -- Detection of movement and direction.
 ------------------------------------------------------------------------------------------------
--------------------------------
 --
 -- The rotary switch contacts are filtered using their offset (one-hot) style to
 -- clean them. Circuit concept by Peter Alfke.
 -- Note that the clock rate is fast compared with the switch rate.

 rotary_filter: process(clk)
 begin
  if clk'event and clk='1' then

    --Synchronise inputs to clock domain using flip-flops in input/output blocks.
    rotary_a_in <= rotary_a;
    rotary_b_in <= rotary_b;
    rotary_press_in <= rotary_press;

    --concatinate rotary input signals to form vector for case construct.
    rotary_in <= rotary_b_in & rotary_a_in;

    case rotary_in is

      when "00" => rotary_q1 <= '0';
                   rotary_q2 <= rotary_q2;

      when "01" => rotary_q1 <= rotary_q1;
                   rotary_q2 <= '0';

      when "10" => rotary_q1 <= rotary_q1;
                   rotary_q2 <= '1';

      when "11" => rotary_q1 <= '1';
```

```
                rotary_q2 <= rotary_q2;

      when others => rotary_q1 <= rotary_q1;
                rotary_q2 <= rotary_q2;
    end case;

  end if;
 end process rotary_filter;
 --
 -- The rising edges of 'rotary_q1' indicate that a rotation has occurred and the
 -- state of 'rotary_q2' at that time will indicate the direction.
 --
 direction: process(clk)
 begin
  if clk'event and clk='1' then

    delay_rotary_q1 <= rotary_q1;
    if rotary_q1='1' and delay_rotary_q1='0' then
     rotary_event <= '1';
     rotary_left <= rotary_q2;
    else
     rotary_event <= '0';
     rotary_left <= rotary_left;
    end if;

  end if;
 end process direction;
 --
 --
 --
 ------------------------------------------------------------------------------------------
---------------------------------
```

```
  --
  -- Direct Digital Synthesizer (DDS)
  --
  ------------------------------------------------------------------------------------------------
--------------------------------
  --
  -- The heart of the DDS is the phase accumulator. This accumulator is provided
with
  -- a control word by PicoBlaze which is accumulates every clock cycle such that
the
  -- most significant bit toggles at a required frequency. Note that this output will
have
  -- a cycle to cycle jitter equal to the rate at which the phase accumulator is
clocked.
  -- The only exception to this are the few cases where control word is such that
the
  -- output is a perfect square wave (equal number of cycles High and Low). In
order to
  -- minimise this cycle to cycle jitter, the phase accumulator is clocked as fast as
  -- possible. In this case the 50MHz clock on the board is multiplied by 4 to give
  -- a 200MHz reference resulting in 5ns of jitter.



  --
  -- Multiply 50MHz clock by 4 to form fast 200MHz clock for phase
accumulator.
  --

  phase_acc_dcm: DCM
  generic map(  CLK_FEEDBACK => "NONE",
                CLKFX_DIVIDE => 1,
              CLKFX_MULTIPLY => 4,
```

```
            CLKIN_PERIOD => 20.0)
port map (    CLKFX => clk_200mhz,
            CLKFB => '0',
            CLKIN => clk,
            DSSEN => '0',
            PSCLK => '0',
             PSEN => '0',
          PSINCDEC => '0',
             RST => '0');

--
-- Buffer 200MHz clock for use by phase accumulator.
--

buffer_dds_clk: BUFG
port map( I => clk_200mhz,
        O => dds_clk);


phase_acc: process(dds_clk)
begin
  if dds_clk'event and dds_clk='1' then

    phase_accumulator <= phase_accumulator + dds_control_word;

  end if;
end process phase_acc;



--
```

-- The most significant bit of the phase accumulator 'phase_accumulator(31)' is the
-- numerically synthesized frequency but will have cycle to cycle jitter of 5ns
-- due to the 200MHz clock.
--
-- Since 5ns of jitter is smaller in percentage terms when synthesizing lower frequencies,
-- the output from the phase accumulator is deliberately kept relatively low (<12.5MHz) and
-- then a second DCM used to multiply this to the higher rates (x16).
--
-- However, the DCM does not normally like the 5ns cycle to cycle jitter because it is
-- designed to maintain phase alignment. So in this case the DCM is used in a special
-- 'frequency aligned mode' in which the DCM output will reflect the average frequency of
-- input and deliberately ignore phase alignment. This will result in a cycle to cycle jitter
-- which is typically much less than 300ps which is an incredible improvement over 5ns.
--
-- Of course you an not get something for nothing! When using the frequency aligned mode, the
-- DCM output will vary slightly in frequency as it 'tracks' the average of the input. This is just
-- like a control loop used to position a platform. The feedback gain of such a control loop will
-- result in different responses. A high gain feedback will result in the smallest positional error
-- overall but the rapid movements can be unpleasant and damaging. In many ways a DCM which is

```
  -- locked to the phase of a waveform is doing exactly this. A low gain feedback
will provide
  -- smooth adjustments that can feel much more comfortable but the slower
reaction will allow greater
  -- errors to accumulate before being corrected each time. This is how the DCM is
behaving in
  -- in frequency aligned mode.
  --
  -- To use the DCM in frequency aligned mode starts with inserting the DCM
primitive in your design
  -- the same way as it would normally be used. Obviously we use the CLKFX
output and there is no
  -- point providing any feedback as phase alignment is being deliberately
deactivated.
  --
  -- In this design the DCM is required to multiply the clock by 16 as well as
provide the
  -- frequency aligned mode to reduce the cycle to cycle jitter.
  --

  frequency_aligned_dcm: DCM
  generic map(  CLK_FEEDBACK => "NONE",
                CLKFX_DIVIDE => 1,          -- CLKFX factors are shown but will be
overwritten
               CLKFX_MULTIPLY => 16,
                CLKIN_PERIOD => 80.0)
  port map (    CLKFX => dcm_clean_clock,
                CLKFB => '0',
                CLKIN => phase_accumulator(31),
                DSSEN => '0',
                PSCLK => '0',
                 PSEN => '0',
```

```
          PSINCDEC => '0',
              RST => '0');


  --
  -- Then to activate the frequency aligned mode a special option must be applied
to BITGEN when
  -- processing the design with the ISE tools.....
  --
  --  Inside Project Navigator go to the 'processes' window
  --    right click 'Generate Programming File'
  --      Properties
  --        General Options
  --          Then add the following text string into the 'Other Bitgen Command Line
Options' box
  --
  --      -g
cfg_dfs_s_x1y1:1111000011111111xxx111xxxxx1xxxxxxxxxx1xxxxxxxxxxxxxx
xxxxxxxxxxxxxxxx01000000
  --
  -- You will also notice that the UCF file supplied with this design has a
constraint which places
  -- this DCM into a known position (INST "frequency_aligned_dcm"
LOC=DCM_X1Y1;) and it is that
  -- position which is referenced by the 'x1y1' part of the text string.
  --
  -- There are a few other key parts to the text string for those that wish to
experiment further.
  -- To see these it helps to segment the line temporarily as follows...
  --
  --                    D-1      M-1                                        C-1
  --  -g cfg_dfs_s_x1y1: 11110000 11111111
xxx111xxxxx1xxxxxxxxxx1xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx 01000000
```

--

-- You can see that the line contains three 8-bit values. These directly set the CLKFX_DIVIDE,

-- CLKFX_MULTIPLY values along with something we will call 'CCount'. The multiply and divide values

-- are exactly the same as those normally set using the parameters on the DCM primitives but require

-- a little bit of effort to set this way!

--

-- For CLKFX_MULTIPLY = 256

-- M-1 = 255 = 11111111 binary. Then reverse the bit order and you still get 11111111.

--

-- For CLKFX_DIVIDE = 16

-- D-1 = 15 = 00001111 binary. Then reverse the bit order to get 11110000.

--

-- You will notice that the CLKFX factors have been set to 256/16 to multiply the clock by 16 rather

-- than the normal 16/1. This is because the averaging effect (damping) is related to the CLKFX_MULTIPLY

-- value. The larger CLKFX_MULTIPLY is, then the smoother the output frequency and lower the cycle to

-- cycle jitter. Normally you are restricted to a value of 32 or less because large values would be

-- detrimental to the ability to phase lock to a signal but here we need to achieve the opposite.

--

-- Which is where the 'CCount' value comes in. This is formed in the same way....

--

-- For CCount = 3

-- C-1 = 2 = 00000010 binary. Then reverse the bit order to get 01000000.

```
  --
  -- 'CCount' is related to how many cycles are received between updates to the
output. So again this
  -- is a form of damping control. In fact the damping effect is therefore related to
the value of
  -- CCOUNT x CLKFX_MULTIPLY. The larger this product the more stable the
output will be but the greater
  -- the accumulated error can be before it corrects itself.
  --
  -- In practice, the largest product 256 x256 = 65536 has such a damping effect
that if used in this application
  -- it takes several seconds per mega-Hertz for the output to change frequency as
new setting are applied. Large
  -- frequency changes would be very slow to achieve which is often inconvenient.
However, a large value of
  -- CCount would often be the best setting if using the generator as a clock source
that is not adjusted
  -- once set. Using lower values of CCount with large value of
CLKFX_MULTIPLY allows fairly rapid changes to
  -- the frequency to be made (when turning the knob) but still provides a
reasonably stable output frequency
  -- with low cycle to cycle jitter.
  --




  --
  -- Create new clock domain for the clean synthesized clock using a global buffer.
  --

  buffer_synth_clk: BUFG
  port map( I => dcm_clean_clock,
```

```
        O => synth_clk);


  --
  -- The synthesized clock covers a high frequency range 100 to 200MHz. The final output is
  -- achieved by dividing this by multiples of 2 using a simple counter and multiplexer structure.
  --

  freq_scaling: process(synth_clk)
  begin
    if synth_clk'event and synth_clk='1' then

     frequency_divider <= frequency_divider + 1;


     case dds_scaling_word is

       when "00000" =>     frequency_out <= frequency_divider(0);   -- 50MHz and above output
       when "00001" =>     frequency_out <= frequency_divider(1);   -- 25MHz to 50MHz output
       when "00010" =>     frequency_out <= frequency_divider(2);   -- 12.5MHz to 25MHz output
       when "00011" =>     frequency_out <= frequency_divider(3);   -- etc...
       when "00100" =>     frequency_out <= frequency_divider(4);
       when "00101" =>     frequency_out <= frequency_divider(5);
       when "00110" =>     frequency_out <= frequency_divider(6);
       when "00111" =>     frequency_out <= frequency_divider(7);
       when "01000" =>     frequency_out <= frequency_divider(8);
       when "01001" =>     frequency_out <= frequency_divider(9);
```

```
      when "01010" =>     frequency_out <= frequency_divider(10);
      when "01011" =>     frequency_out <= frequency_divider(11);
      when "01100" =>     frequency_out <= frequency_divider(12);
      when "01101" =>     frequency_out <= frequency_divider(13);
      when "01110" =>     frequency_out <= frequency_divider(14);
      when "01111" =>     frequency_out <= frequency_divider(15);
      when "10000" =>     frequency_out <= frequency_divider(16);
      when "10001" =>     frequency_out <= frequency_divider(17);
      when "10010" =>     frequency_out <= frequency_divider(18);
      when "10011" =>     frequency_out <= frequency_divider(19);
      when "10100" =>     frequency_out <= frequency_divider(20);
      when "10101" =>     frequency_out <= frequency_divider(21);
      when "10110" =>     frequency_out <= frequency_divider(22);
      when "10111" =>     frequency_out <= frequency_divider(23);
      when "11000" =>     frequency_out <= frequency_divider(24);
      when "11001" =>     frequency_out <= frequency_divider(25);
      when "11010" =>     frequency_out <= frequency_divider(26);  -- 1Hz
output
      when "11011" =>     frequency_out <= frequency_divider(27);
      when "11100" =>     frequency_out <= frequency_divider(28);
      when "11101" =>     frequency_out <= frequency_divider(29);
      when "11110" =>     frequency_out <= frequency_divider(30);
      when "11111" =>     frequency_out <= frequency_divider(31);

      -- Don't care used for all other to ensure minimum logic implementation
      when others =>    frequency_out <= 'X';

    end case;

    -- Pipeline output from multiplexer to maintain performance up to 200MHz

    freq_out_pipe <= frequency_out;
```

```
    sma_out <= freq_out_pipe;

  end if;
 end process freq_scaling;

 --
 -- Test points
 --

 simple(9) <= phase_accumulator(31);
 simple(10) <= '0';
 simple(11) <= '0';
 simple(12) <= freq_out_pipe;

 --
 --
 --
end Behavioral;

----------------------------------------------------------------------------------
------------------------------
--
-- END OF FILE frequency_generator.vhd
--
----------------------------------------------------------------------------------
------------------------------
```