SEMESTA TEKNIKA
Vol.24, No.2, 120-128, November 2021
DOI : https://doi.org/10.18196/st.v24i2.12863

Aplikasi Perancang Abstraksi Verilog Mesin Keadaan Terbatas Otomatis

(Automatic Finite State Machine to Verilog Abstraction Application)

FAIRUZ AZMI

ABSTRAK

Saat ini, hampir semua perangkat elektronik menggunakan prosesor di dalamnya. Dalam sebuah prosesor, terdapat bagian *control unit* yang berfungsi mengatur operasi dari komponen-komponen di dalam prosesor. *Control unit* merupakan sebuah mesin keadaan terbatas atau disebut *finite state machine* (FSM). Rangkaian FSM dapat disintesis secara manual ataupun secara otomatis menggunakan bahasa abstraksi Verilog. Dalam penelitian ini, dibuat sebuah aplikasi yang dapat membantu pengguna merancang FSM dan selanjutnya menyimpannya dalam format Verilog. Aplikasi yang dibuat secara fungsional dapat berjalan dengan kesesuaian 100% dan mampu untuk membuat rancangan Verilog untuk FSM dengan berbagai model dan teknik pengkodean *state*. Simulasi modul Verilog yang dihasilkan juga sesuai dengan spesifikasi rangkaian FSM yang dirancang.

Kata kunci: *Control unit*, FSM, Verilog.

ABSTRACT

Today, almost all electronic devices use processors. Inside a processor, there is a control unit that controls the operation of the components in the processor. The control unit also called the finite state machine (FSM). FSM circuits can be synthesized manually by hand or automatically using the Verilog abstraction language. In this research, an application is made that can help users design FSM and then export it to Verilog format. Functionally, the applications is 100% complied with the requirements and are able to create Verilog designs for FSM with various FSM models and state encoding techniques. The resulting Verilog module simulation also conforms to the designed FSM circuit specifications.

***Keywords**: Control unit, FSM, Verilog.*

PENDAHULUAN

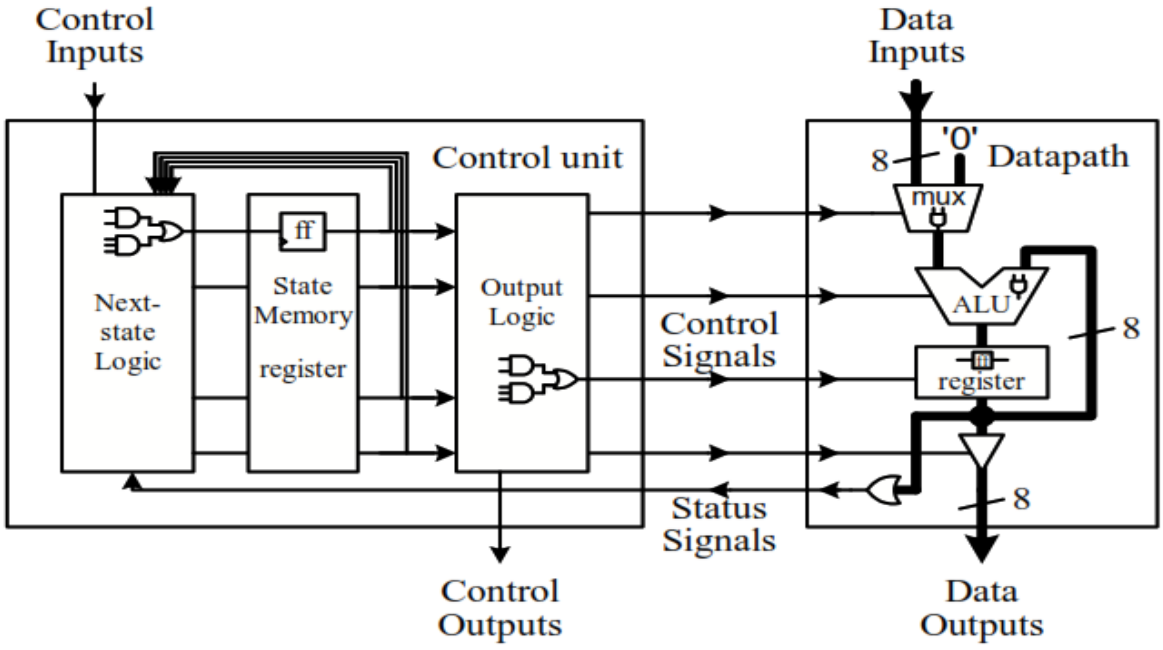
Saat ini, hampir semua perangkat elektronik menggunakan prosesor di dalamnya, baik itu komputer, jam tangan pintar (*smartwatch*), bahkan berbagai perangkat elektronik rumah tangga seperti lemari es dan pendingin ruangan juga menggunakan prosesor (Martindale, 2021). Prosesor atau sering disebut *central processing unit* (CPU) berfungsi untuk memproses dan mengeksekusi instuksi-instruksi dan berperan sebagai otak dari perangkat elektronik yang membutuhkan proses komputasi.

Rancangan sebuah prosesor dapat dibagi ke dalam dua bagian, yaitu bagian *datapath* dan bagian *control unit* (Hwang, 2005; Ledin, 2020), sebagaimana ditunjukkan pada Gambar 1. *Datapath* berfungsi untuk menangani segala operasi yang berhubungan dengan data, termasuk di dalamnya unit fungsional seperti *arithmetic-logic unit* (ALU), *register*, dan *bus* untuk mentransmisikan data antar komponen di dalam *datapath*. Sedangkan *control unit* atau disebut *controller* berfungsi untuk mengatur seluruh operasi di *datapath* dengan memberikan sinyal kendali yang sesuai di waktu yang tepat. Untuk setiap satuan waktu, *control unit* berada dalam suatu kondisi (*state*), yang secara fisik merupakan isi dari sebuah *state memory*. *Control unit* beroperasi dengan

cara bertransisi dari sebuah *state* ke *state* lainnya, yang berubah setiap siklus *clock*. Berdasarkan perilaku tersebut, maka *control unit* juga sering disebut sebagai sebuah mesin keadaan terbatas atau *finite state machine* (FSM).

FSM merupakan nama lain dari sebuah rangkaian logika sekuensial. Nama FSM lebih sering digunakan dalam literatur teknis (Brown & Vranesic, 2014). Rangkaian sekuensial adalah rangkaian yang keluarannya bergantung kepada kondisi keluaran di waktu lampau, selain dipengaruhi juga oleh nilai masukan saat ini sebagaimana di rangkaian logika kombinasional. Dalam banyak kasus, rangkaian sekuensial dikendalikan menggunakan sinyal *clock* sebagai komponen pewaktu, sehingga rangkaian sekuensial lebih sering merujuk kepada rangkaian sekuensial sinkron, selain rangkaian sekuensial asinkron yang bekerja tanpa ada pengendali sinyal pewaktu.

Sebuah FSM direalisasikan menggunakan gabungan rangkaian kombinasional dan flip-flop (Brown & Vranesic, 2014). Untuk merancang sebuah FSM, dibutuhkan beberapa tahapan, diantaranya membuat diagram transisi keadaan (*state diagram*), membuat tabel transisi (*state table*), kemudian memilih tipe flip-flop dan merealisasikan rangkaian (Bucaro, 2019). Dalam proses perancangan perangkat keras digital, perancang *integrated circuit* (IC) membutuhkan umpan balik yang cepat dari hasil rancangan yang dibuatnya untuk menentukan kesesuaian rancangan dibandingkan dengan spesifikasi sistem yang telah ditentukan (Weste & Harris, 2011). Menerjemahkan diagram blok dan FSM menjadi rangkaian skematik akan menghabiskan banyak waktu dan dapat meningkatkan peluang terjadinya *error* dalam proses verifikasi. Oleh karena itu, diperlukan suatu metode untuk merancang di level atas dan segera mengetahui apakah perlu dilakukan perancangan ulang berdasarkan hasil verifikasi rancangan *top module*. Untuk merancang abstraksi rangkaian, digunakan bahasa yang dapat mendeskripsikan sebuah sistem digital, yaitu *hardware description language* (HDL). Awalnya, HDL hanya digunakan untuk keperluan dokumentasi dan simulasi, tetapi saat ini HDL juga dapat digunakan untuk mensintesis rangkaian digital berdasarkan abstraksi yang telah dibuat, sehingga seluruh proses perancangan dapat dilakukan secara otomatis. Salah satu bahasa HDL yang umum digunakan saat ini adalah Verilog, bahasa yang distandarkan sebagai IEEE standard 1364-1995 (Thomas & Moorby, 2013). Standar Verilog terbaru saat ini adalah IEEE standard 1364-2005 (IEEE Computer Society, 2006).



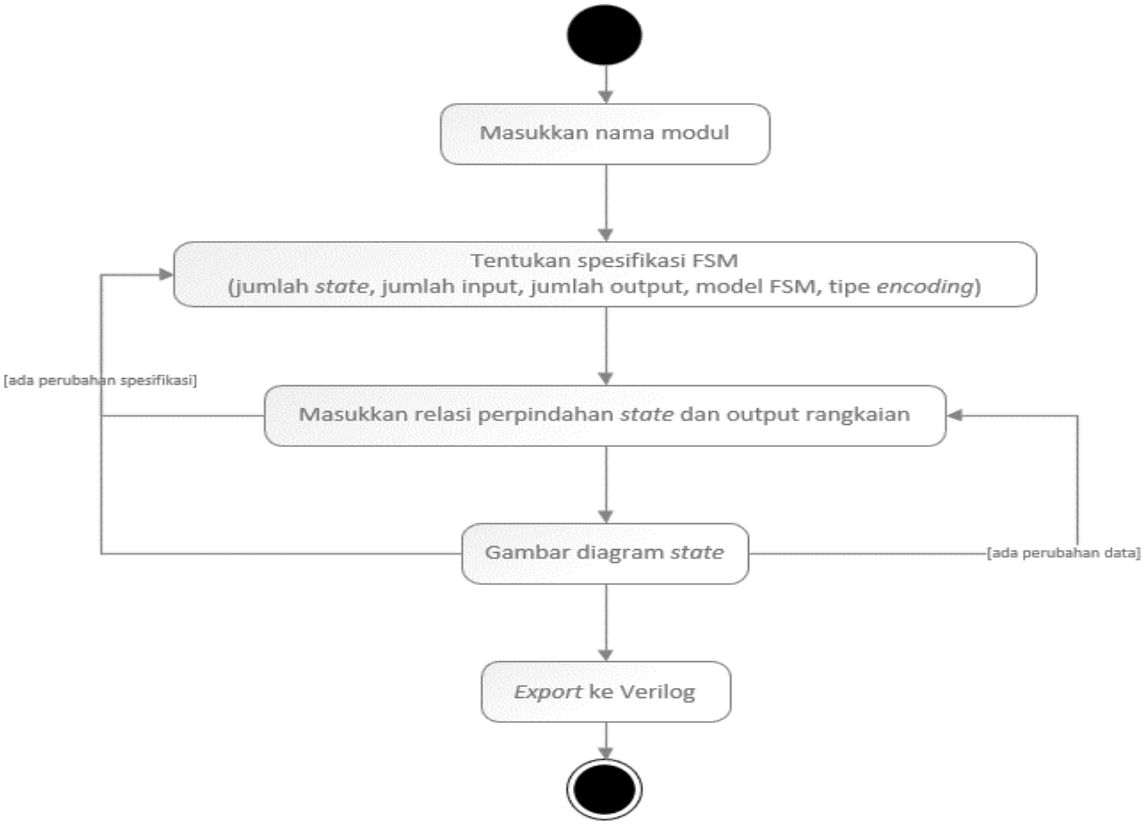
GAMBAR 1. Skema Sebuah Prosesor
(Hwang, 2005)

Dalam penelitian ini, dibuat sebuah aplikasi *desktop* yang dapat membantu perancang IC untuk membuat modul FSM secara visual menggunakan tabel dan ditampilkan dalam bentuk diagram. Selanjutnya, program secara otomatis akan menerjemahkan deskripsi Verilog dari diagram yang telah dirancang untuk selanjutnya dapat disimulasikan di aplikasi *HDL simulator*, dan lebih lanjut modul tersebut dapat disintesis untuk diimplementasikan baik di *field programmable gate array* (FPGA) maupun sebagai sebuah *application specific integrated circuit* (ASIC).

METODE PENELITIAN

Aplikasi FSM Designer dibangun menggunakan bahasa pemrograman C# di atas *platform* .NET Framework 4.7.2. Bahasa C# merupakan bahasa pemrograman yang sederhana, modern, serbaguna, serta berorientasi objek (Hejlsberg et al., 2002). Aplikasi yang digunakan dalam pembuatan aplikasi ini adalah Microsoft Visual Studio 2019. Aplikasi dibuat dalam bentuk aplikasi *Windows Form* dan berjalan di sistem operasi Windows 10.

Aplikasi FSM Designer memiliki tiga komponen utama yang ditampilkan dalam satu jendela aplikasi. Komponen utama aplikasi meliputi tabel data untuk mengisi *state table*, komponen untuk menggambar diagram di jendela aplikasi, dan komponen penerjemah tabel data menjadi bahasa Verilog. Komponen tabel data memanfaatkan *library* standar dari bahasa C# dan ditampilkan di komponen DataGridView. Untuk menggambar diagram, digunakan *library Microsoft Automatic Graph Layout* (MSAGL), *library* untuk .NET Framework yang dikembangkan di Microsoft untuk membuat dan menampilkan graf (Nachmanson et al., 2021). Untuk menerjemahkan tabel data menjadi bahasa Verilog, dibuat sebuah *class* yang berfungsi untuk membangkitkan sintaks bahasa verilog berdasarkan kondisi-kondisi yang telah diterapkan di tabel data dan di halaman utama aplikasi FSM Designer. Secara umum, aktivitas dalam aplikasi FSM Designer dapat dilihat pada Gambar 2.



GAMBAR 2. *Activity Diagram* Aplikasi FSM Designer

Kondisi terkini dari sebuah FSM disimpan di dalam sebuah *state memory*. *State memory* secara fisik adalah kumpulan flip-flop yang banyaknya sebanding dengan panjang kode *state*. Untuk mengkodekan *state*, terdapat tiga cara yang umum digunakan, yaitu *binary encoding*, *gray code encoding*, dan *one-hot encoding* (La Meres, 2017). Pemilihan tipe pengkodean *state* dapat mempengaruhi panjang ukuran *state memory*. Pengkodean dengan *binary encoding* dan *gray code encoding* memerlukan panjang kode *state* yang sama, sebagaimana ditunjukkan pada persamaan (1). Kode *state* untuk *binary encoding* merupakan nilai biner sesuai persamaan (2), sedangkan kode untuk *gray code encoding* ditunjukkan pada persamaan (3). Penggunaan *gray code encoding* bermanfaat untuk meminimalkan perubahan nilai flip-flop dalam *state memory* jika urutan perpindahan *state* dalam FSM bersifat linier. Teknik pengkodean *state* yang juga umum digunakan dalam perancangan FSM adalah *one-hot encoding*. Metode ini menerapkan kode dari *state* dengan membuat seluruh nilai flip-flop dalam *state memory* dalam kondisi logika '0', kecuali satu buah flip-flop, yang dikatakan sebagai "hot" (Brown & Vranesic, 2014). Berbeda dengan metode *binary* dan *gray code encoding* yang menitikberatkan efisiensi pada jumlah flip-flop yang digunakan di rangkaian, metode *one-hot encoding* membuat rangkaian logika *next-state* menjadi lebih sederhana (La Meres, 2017), sehingga membutuhkan waktu yang lebih cepat dibanding teknik pengkodean lainnya. Untuk mengodekan *state* menggunakan metode *one-hot encoding* dibutuhkan flip-flop sebanyak jumlah *state*, dan kode *state* ditunjukkan di persamaan (4). Tabel 1 menunjukkan contoh pengkodean *state*.

$$L = \lceil {}^{2}\log nS \rceil \quad (1)$$

$$S_i = bin(i, L) \quad , 0 \le i < nS \quad (2)$$

$$S_i = bin(i \oplus \lfloor i/2 \rfloor, L) \quad , 0 \le i < nS \quad (3)$$

$$S_i = bin(2^i, nS) \quad , 0 \le i < nS \quad (4)$$

dengan L = panjang kode *state*, nS = jumlah *state* dalam FSM, Si = kode *state* di urutan i dalam FSM.

Secara umum, rangkaian FSM terdiri dari dua rangkaian kombinasional dan sebuah *state memory* yang berupa flip-flop (Donzellini et al., 2019). Rangkaian kombinasional dalam FSM berfungsi untuk menentukan nilai masukan dari *state memory* untuk mengubah *state* yang disebut *next-state logic*. Selain itu, juga terdapat rangkaian kombinasional untuk menentukan nilai keluaran dari FSM, yang disebut *output logic*. FSM bekerja dengan mengubah isi *state memory* yang disebabkan oleh *next-state logic* yang sensitif terhadap input rangkaian. Selanjutnya, melalui rangkaian *output logic* yang sensitif terhadap *state memory*, output rangkaian akan dihasilkan.Terdapat dua model FSM, yaitu *Moore* dan *Mealy*. Keluaran FSM *Moore* hanya bergantung dari isi *state memory*, sedangkan FSM *Mealy*, selain sensitif terhadap *state memory*, juga ada pengaruh langsung dari masukan FSM untuk mengubah keluaran. Dalam proses sintesis, perbedaan model *Moore* dengan *Mealy* dapat dilihat pada bagian *output logic*. Masukan *output logic* untuk FSM *Moore* hanya berasal dari keluaran *state memory*, sedangkan masukan *output logic* untuk FSM *Mealy* berasal dari keluaran *state memory* dan juga masukan dari rangkaian FSM. Untuk spesifikasi yang sama, FSM *Mealy* memiliki lebih sedikit *state* dibanding FSM *Moore*, tetapi dengan *output logic* yang lebih kompleks (Brown & Vranesic, 2014).

TABEL 1. Contoh Pengkodean untuk Enam Buah *State*

State	**Kode *State***		
	Binary	***Gray Code***	***One-hot***
A (i=0)	000	000	000001
B (i=1)	001	001	000010
C (i=2)	010	011	000100
D (i=3)	011	010	001000
E (i=4)	100	110	010000
F (i=5)	101	111	100000

Setelah menentukan parameter spesifikasi sistem, selanjutnya adalah menerjemahkan rancangan yang telah dibuat ke dalam abstraksi Verilog. Referensi lengkap mengenai sintaks dan struktur bahasa Verilog dapat dilihat di standar IEEE 1364-2005 oleh IEEE Computer Society (2006). Sedangkan panduan cepat mengenai penggunaan Verilog dapat dilihat di tulisan Sutherland (2001). Templat dari kode Verilog FSM dapat dilihat di *pseudocode* berikut ini:

```
module nama_modul
(
    input wire CLK, RST,
    input wire inputs,
    output reg outputs
);

localparam [L-1:0]
    // Pengkodean state

reg[L-1:0] c_state, n_state;

always @(c_state or inputs)
begin
    n_state = c_state;
    case(c_state)
        // Next-state logic
    endcase
end

always @(posedge CLK)
begin
    if(RST == 1'b1)
        c_state <= S0;
    else
        c_state <= n_state;
end

always @(posedge CLK)
begin
    if(RST == 1'b1)
        outputs <= 0;
    else
        begin
            case(c_state)
                // Output logic
            endcase
        end
end
endmodule
```

Abstraksi Verilog disimpan dalam sebuah file dengan ekstensi ".v". File disimpan dengan nama yang sama dengan nama modul yang dibuat. Hal ini untuk mengakomodasi beberapa aplikasi HDL *simulator* dan *synthesizer* yang mengharuskan penamaan file yang sama dengan nama modul yang dibuat.

HASIL DAN PEMBAHASAN

Aplikasi FSM Designer dapat berjalan dengan baik di komputer dengan sistem operasi Windows 10 atau minimal Windows 7 yang terpasang .NET Framework 4.7.2. Tampilan aplikasi FSM Designer dapat dilihat di Gambar 3. Untuk menguji kesesuaian kinerja aplikasi dengan spesifikasi yang dirancang, dilakukan pengujian fungsional, yang meliputi pengujian ketepatan aplikasi dalam berinteraksi dengan pengguna. Selanjutnya, untuk menguji ketepatan hasil penerjemahan rangkaian ke bahasa Verilog, dilakukan uji simulasi modul Verilog yang dihasilkan menggunakan aplikasi Modelsim.

1. *Pengujian fungsional aplikasi*

Tabel 2 menunjukkan hasil pengujian fungsional dari aplikasi FSM Designer. Berdasarkan Tabel 2, dapat dilihat bahwa seluruh fungsi utama dari aplikasi FSM Designer berjalan sesuai dengan rancangan. Kesalahan-kesalahan yang terjadi ditangani secara langsung dengan menampilkan pesan kesalahan kepada pengguna, sehingga pengguna dapat langsung memperbaiki tanpa harus memaksa aplikasi untuk dihentikan.

2. *Simulasi file Verilog*

File Verilog yang dihasilkan diuji kesesuaiannya dengan aplikasi HDL *simulator*. Terdapat banyak aplikasi yang dapat mensimulasikan Verilog. Dalam penelitian ini, digunakan aplikasi Modelsim.

File Verilog yang dihasilkan beragam sesuai dengan kondisi yang telah ditetapkan di dalam Aplikasi FSM Designer. Kondisi-kondisi tersebut meliputi jumlah *state*, model FSM, jumlah *input*, jumlah *output*, jenis pengkodean *state*, serta penggunaan masukan untuk keperluan me-*reset* FSM. Berbagai kondisi tersebut disimulasikan dan menunjukkan bahwa 100% hasil simulasi sesuai dengan rancangan. Hasil pengujian simulasi Verilog dapat dilihat pada Tabel 3.

3. *Analisis hasil sintesis*

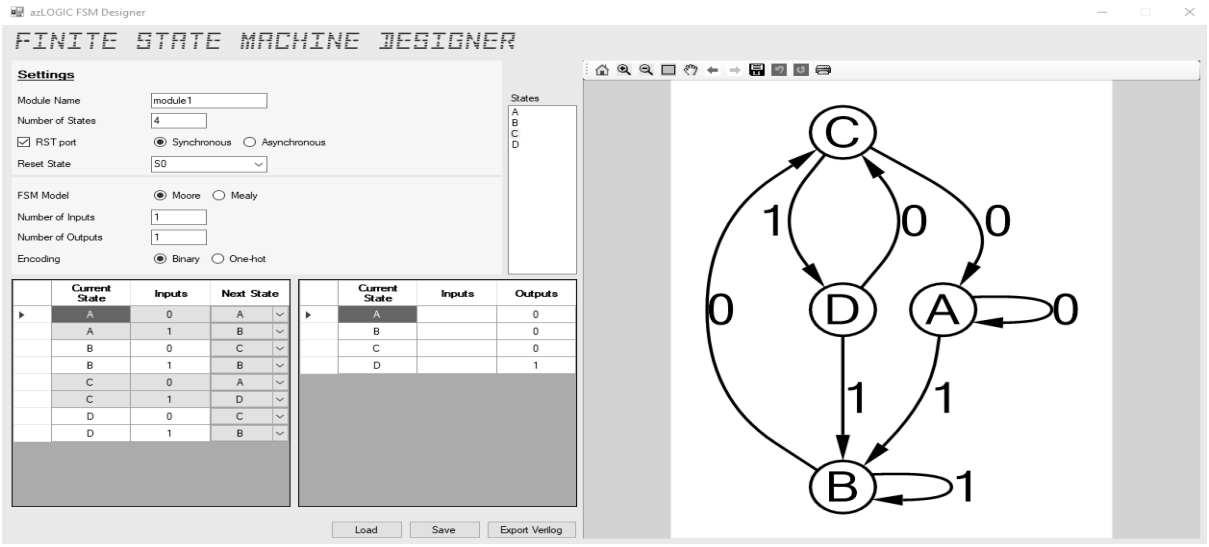
File Verilog yang dihasilkan disintesis menggunakan aplikasi Intel Quartus Prime. Sebagai contoh, dilakukan analisis untuk membuat file Verilog dari diagram transisi sebuah FSM yang ditunjukkan pada Gambar 4.

Diagram tersebut menggambarkan transisi keadaan untuk rangkaian detektor sekuensial

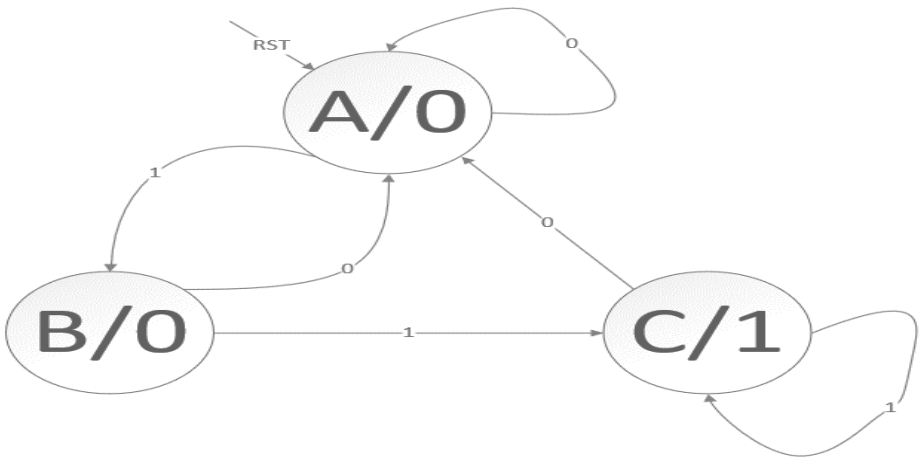
yang akan menghasilkan keluaran logika ’1’ jika terdeteksi urutan masukan logika ’1’ minimal dua kali berturut-turut. Dengan proses manual, rangkaian FSM yang terbentuk berdasarkan diagram transisi tersebut menggunakan metode *binary encoding* dapat dilihat pada Gambar 5.

TABEL 2. Hasil Pengujian Fungsional Aplikasi FSM Designer

No.	Parameter uji	Hasil yang diharapkan	Hasil pengujian	Keterangan
1	Aturan penamaan modul	Nama modul hanya boleh terdiri dari alfabet [A..Z, a..z], numerik [0..9], dan garis bawah (_), serta tidak boleh diawali oleh numerik.	Saat diberikan masukan nama modul selain alfanumerik dan garis bawah, atau jika dimulai dengan numerik, maka akan mengeluarkan pesan dan meminta pengguna untuk mengganti nama modul.	Sesuai
2	Memperbarui tabel *state*	Diagram *state* akan menyesuaikan isi tabel *state*.	Diagram *state* sesuai dengan isi tabel *state*.	Sesuai
3	Klik ”Export to Verilog”	Aplikasi menulis file Verilog sesuai spesifikasi yang dipilih dan juga isi tabel *state*.	File Verilog ditulis denga nama sesuai nama modul dan isi sesuai dengan isi tabel *state* dan spesifikasi yang ditetapkan.	Sesuai



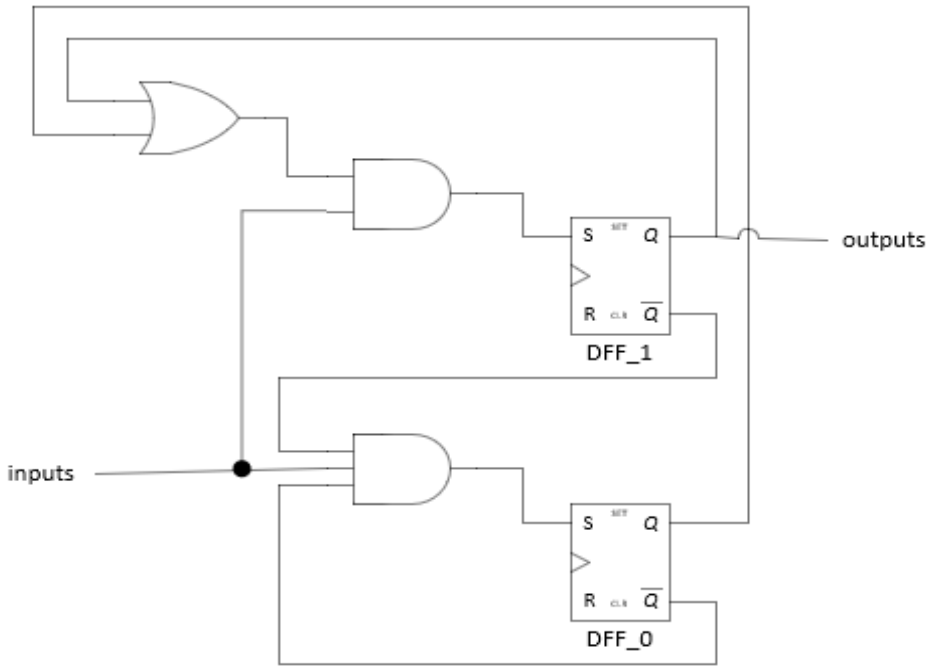
GAMBAR 3. Tampilan Aplikasi FSM Designer



GAMBAR 4. Diagram Transisi Keadaan Untuk Mendeteksi Urutan Input ”11”

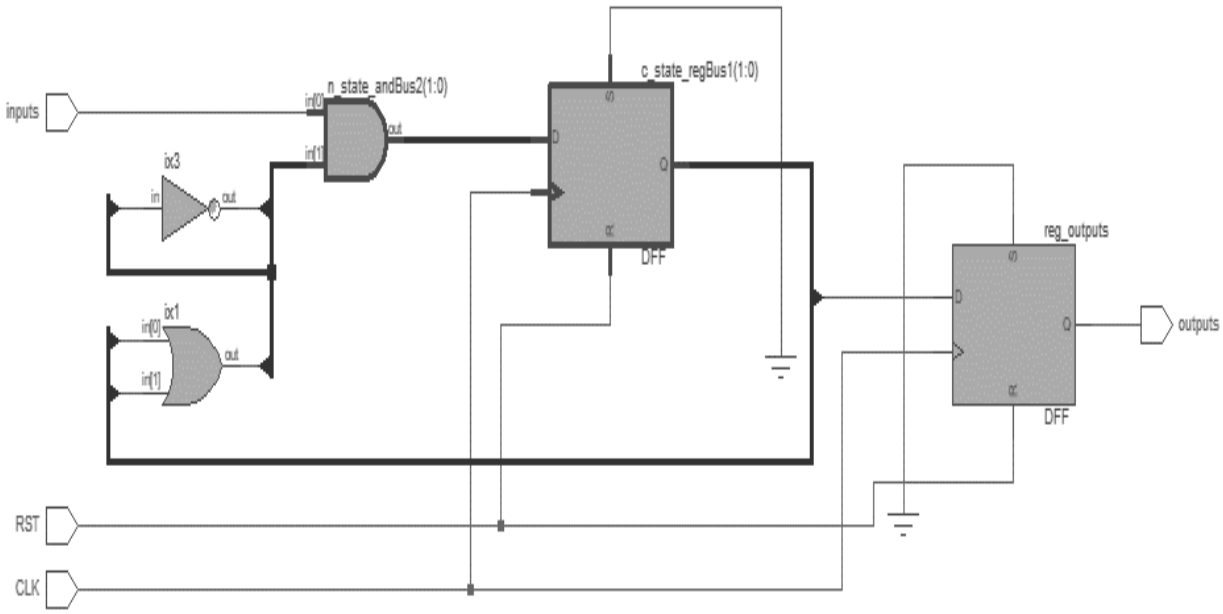
TABEL 3. Hasil Pengujian Simulasi File Verilog

No.	Spesifikasi FSM	Keterangan
1	Tanpa *reset*	
	a. Model *Moore*	
	i. *Binary encoding*	Sesuai
	ii. *Gray code encoding*	Sesuai
	iii. *One-hot encoding*	Sesuai
	b. Model *Mealy*	
	i. *Binary encoding*	Sesuai
	ii. *Gray code encoding*	Sesuai
	iii. *One-hot encoding*	Sesuai
2	Dengan *synchronous reset*	
	a. Model *Moore*	
	i. *Binary encoding*	Sesuai
	ii. *Gray code encoding*	Sesuai
	iii. *One-hot encoding*	Sesuai
	b. Model *Mealy*	
	i. *Binary encoding*	Sesuai
	ii. *Gray code encoding*	Sesuai
	iii. *One-hot encoding*	Sesuai
3	Dengan *asynchronous reset*	
	a. Model *Moore*	
	i. *Binary encoding*	Sesuai
	ii. *Gray code encoding*	Sesuai
	iii. *One-hot encoding*	Sesuai
	b. Model *Mealy*	
	i. *Binary encoding*	Sesuai
	ii. *Gray code encoding*	Sesuai
	iii. *One-hot encoding*	Sesuai

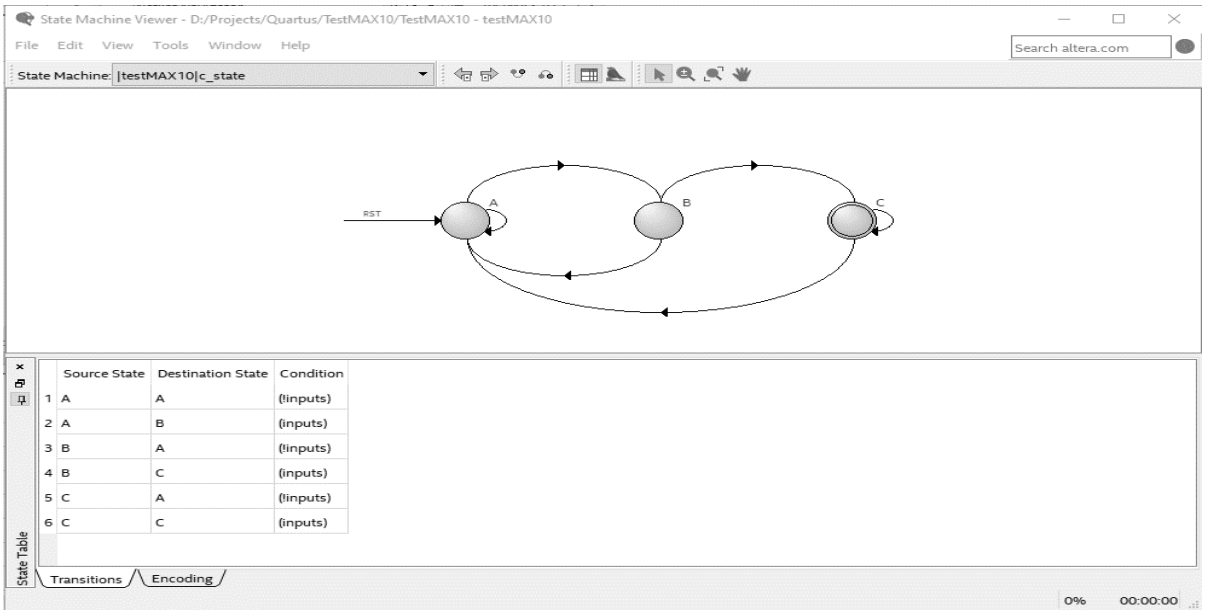


GAMBAR 5. Hasil Sintesis Manual Rangkaian FSM untuk Mendeteksi Urutan Input "11"

Berdasarkan hasil sintesis manual, diperoleh rangkaian dengan penggunaan dua buah D flip-flop, satu gerbang AND 2-input, satu gerbang AND 3-input, dan satu gerbang OR 2-input. Selanjutnya hasil sintesis file Verilog yang dihasilkan dapat dilihat pada Gambar 6. Berdasarkan hasil sintesis file verilog dengan pilihan metode *binary encoding*, diperoleh rangkaian dengan dua buah D flip-flop, satu gerbang AND 2-input, satu gerbang OR 2-input, dan satu gerbang NOT. Terdapat sedikit perbedaan dikarenakan adanya proses optimasi oleh aplikasi Intel Quartus Prime dan juga penyesuaian dengan teknologi perangkat keras yang digunakan. Akan tetapi, saat direkayasa balik akan menghasilkan diagram transisi yang sama sebagaimana ditunjukkan pada Gambar 7.



GAMBAR 6. Hasil Sintesis Rangkaian FSM Untuk Mendeteksi Urutan Input "11" Menggunakan Aplikasi Intel Quartus Prime



GAMBAR 7. Hasil Rekayasa Balik FSM Dari Rangkaian Hasil Sintesis Intel Quartus Prime

KESIMPULAN

Sesuai dengan hasil pengujian fungsional, aplikasi FSM Designer berjalan dengan baik dengan kesesuaian 100% dengan rancangan yang ditetapkan. File Verilog yang dihasilkan oleh aplikasi juga sesuai dengan spesifikasi rangkaian FSM yang ditetapkan, sehingga aplikasi FSM Designer dapat digunakan untuk membantu otomasi proses perancangan FSM.

DAFTAR PUSTAKA

Brown, S., & Vranesic, Z. (2014). *Fundamentals of Digital Logic with Verilog Design* (3 ed.). McGraw-Hill.

Bucaro, S. (2019). *Basic Digital Logic Design: Use Boolean Algebra, Karnaugh Mapping, or an Easy Free Open-Source Logic Gate Simulator*.

Donzellini, G., Oneto, L., Ponta, D., & Anguita, D. (2019). *Introduction to Digital Systems Design*. Springer.

Hejlsberg, A., Wiltamuth, S., & Golde, P. (2002). *Standard ECMA-334: C# Language Specification*.

Hwang, E. O. (2005). *Digital logic and Microprocessor Design with VHDL*. La sierra University, Riverside.

IEEE Computer Society. (2006). *IEEE Standard Verilog Hardware Description Language* (Nomor IEEE Std 1364™-2005). IEEE.

La Meres, B. J. (2017). Introduction to Logic Circuits & Logic Design with Verilog. In *Introduction to Logic Circuits & Logic Design with VHDL*. Springer. https://doi.org/10.1007/978-3-319-53883-9

Ledin, J. (2020). *Modern Computer Architecture and Organization Learn x86, ARM, and RISC-V architectures and the design of smartphones, PCs, and cloud servers*.

Martindale, J. (2021). *What Is a CPU? Here's Everything You Need to Know | Digital Trends*. Digital Trends. Retrieved from https://www.digitaltrends.com/computing/what-is-a-cpu/

Nachmanson, L., Pupyrev, S., Dwyer, T., Hart, T., & Prutkin, R. (2021). *Microsoft Automatic Graph Layout: A set of tools for graph layout and viewing*. Microsoft Research. Retrieved from https://github.com/Microsoft/automatic-graph-layout.

Sutherland, S. (2001). *Verilog-2001 Quick Reference Guide*. Sutherland HDL.

Thomas, D. E., & Moorby, P. R. (2013). *The Verilog® Hardware Description Language* (3 ed.). Springer.

Weste, N. H. E., & Harris, D. M. (2011). *CMOS VLSI Design : A Circuits and Systems Perspective* (4 ed.). Addison - Wesley.

PENULIS:

Fairuz Azmi

Program Studi Teknik Komputer, Fakultas Teknik Elektro, Universitas Telkom, Jl. Telekomunikasi Terusan Buah Batu, Bandung.

Email: worldliner@telkomuniversity.ac.id