Assignment 1 - DNS Lookup System Verilog FSM

ELEC 402

Isabelle André – 12521589

1.0 Project Description

This project consisted in creating a custom finite state machine in system verilog and a testbench to verify its design. As freedom of topic for this project was given, an FSM depicting the process of a complete DNS lookup and Webpage Query was created. This state machine contains 13 states, including the multiple stages of recursive and iterative queries between different data servers. Depending on the web address queried from the client, an address may be mapped to its corresponding IP address and cached in the browser for future usage, decreasing the total execution time of the lookup process.

2.0 DNS Lookup and Webpage Query Background

When a client accesses a website, a Domain Name System (DNS) is used to locate the server at which the domain's website is located. This process allows the mapping from a web address such as www.google.com, to its corresponding IP address, 172.217.12.46 in the background.

2.1 DNS Servers

There are four relevant DNS servers to this process.

First, the **DNS Resolver** is designed to receive queries from client machines through various applications, such as web browsers. The recursor then makes requests as needed to other servers in order to resolve the client's initial query.

The **Root Nameserver** maps and returns the corresponding Top Level Domain server address to a web address.

The **Top Level Domain (TLD) Nameserver** hosts a portion of a host name, such as .com or .ca. The TLD Nameserver maps and returns the corresponding Authoritative Nameserver address to a web address.

Finally, the **Authoritative Nameserver** contains a record of every IP address within its domain. If the server has access to the requested record, it returns the corresponding IP address for the requested hostname.

Once a web address is accessed, the address of any of these servers may be cached by the OS, reducing the time of execution of this procedure. However in this project, we will not take OS caching into account and simply focus on a single address browser caching policy.

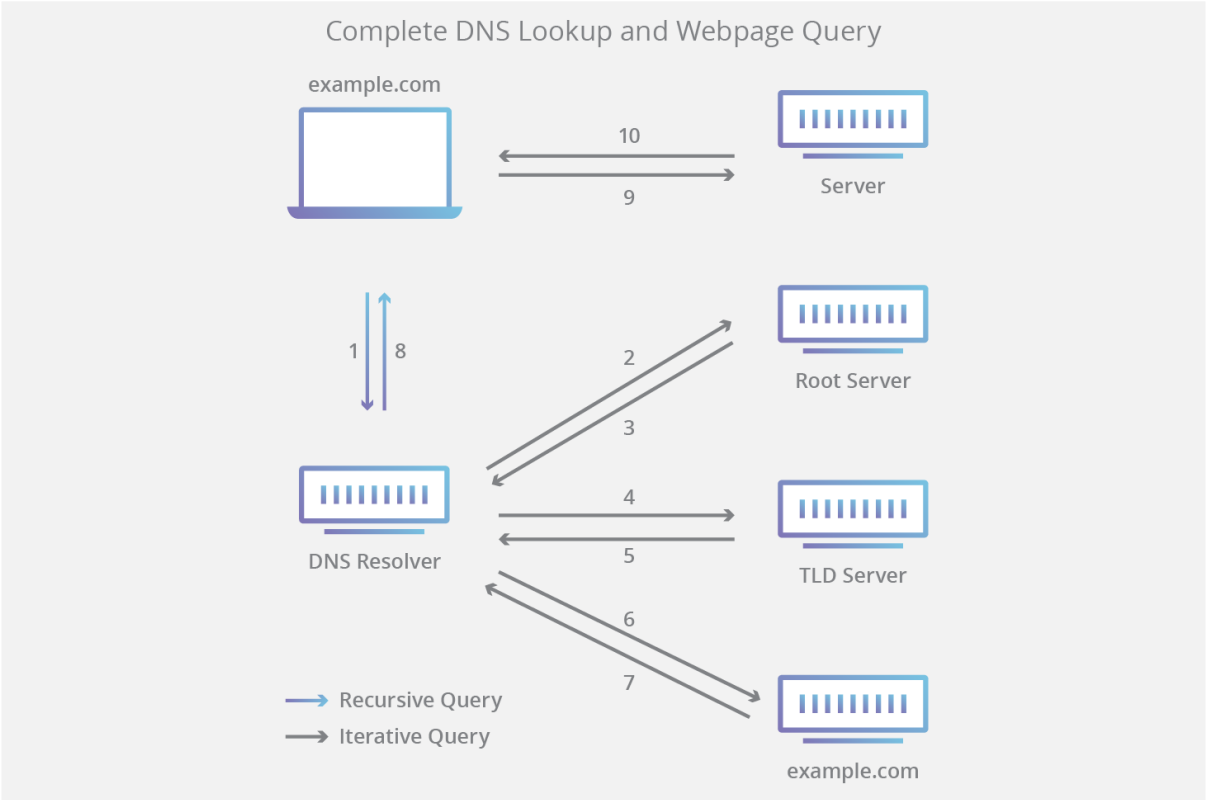
3.0 State Machine Description

The DNS Lookup process begins with the user entering a web address into a browser such as google.ca, received by a DNS recursive resolver. The resolver uses this address to query a DNS root nameserver, which in turn responds with the address of a TLD DNS server, such as .ca. Once the resolver then queries to the .ca TLD server, and receives the IP address of the domain's nameserver for google.ca. The resolver sends another query to the domain's nameserver, which returns the corresponding IP address for google.ca. Finally, the DNS server is able to return the corresponding IP address to the web browser for the requested domain.

Once the client browser receives a response, it makes an HTTP request to the IP address. The server at the IP address returns the webpage data and components to render the website.

The state machine includes a caching state in which the browser is able to cache the IP address of the latest website accessed. If the same website is accessed the next request, the FSM skips the DNS lookup process, and is able to directly make an HTTP request to the cached IP address. This is a very simplified approach to simulate the caching abilities of browsers to reduce request latency, and does not include OS caching or caching server addresses.

Figure 1 describes this process and the iterative nature of server requests from the DNS resolver. Note that this is not a complete state transition diagram, and simply a representation of the servers and client interactions.



Cloudflare, What is DNS?, 2022, https://www.cloudflare.com/learning/dns/what-is-dns/
Figure 1: Complete DNS Lookup and Webpage Query

3.1 States Description

- **IDLE:** Initial state. If a client request arrives, reset execution time counter
- **CLIENT START:** Start execution time counter and begin to process request
- **CLIENT RESOLVER REQ:** Browser cache is queried to see if web address matches a cached IP address. If so, the state machine skips to CLIENT SERVER REQ. If not, the client query is sent to the DNS Resolver for a DNS Lookup.
- **RESOLVER ROOT REQ:** Resolver request is sent to a Root Nameserver. A query bit is enabled to allow a mock TLD address to be processed and returned.
- **ROOT RES:** The mock TLD address is generated and returned.
- **RESOLVER TLD REQ:** The Resolver sends a request to the TLD server. A query bit is enabled to allow a mock Domain Nameserver address to be processed and returned.
- **TLD RES:** The mock Domain Nameserver address is generated and returned.
- **RESOLVER DOMAIN REQ:** The Resolver sends a request to the Domain Nameserver. A query bit is enabled to allow a mock web IP address to be processed and returned.
- **DOMAIN RES:** The mock web IP address is generated and returned.
- **RESOLVER RES:** The DNS Resolver responds to the client with the IP address of the domain requested. The IP address was found through DNS Lookup and resolved.
- **CLIENT SERVER REQ:** The client browser makes an HTTP request to the IP address. If the IP address was found through DNS Lookup, use this address, if not, use the cached address. A query bit is enabled to allow mock web data to be processed and returned.
- **SERVER RES:** The mock web data is generated and returned.
- **CACHING:** The last used IP address is cached in the browser. The execution time counter is terminated and outputted, and the client request has been resolved. The state machine returns to the IDLE state until the next client request arrives.

3.2 Inputs Description

- **clk:** Input clock signal.
- **rst:** Input reset signal.
- **client req:** Flag indicating an incoming client request.
- **web addr:** Mock client web address requesting corresponding IP address.

3.3 Outputs Description

- **webpage idx out:** Mock web page data output corresponding for website rendering.
- **tld addr out:** Mock Top Level Domain address.

- **domain ip out:** Mock Domain Nameserver IP address.
- **web ip out:** Mock web IP Address corresponding to initial client query.
- **exec time:** Total execution time for DNS Lookup and caching process, simulated by counting cycles.
- **ip resolved:** Mock web IP address was found during DNS Lookup.
- **client res:** Web data has been returned and rendered, state machine is complete.

3.4 Modules

- **DNSLookup:** Main state machine.
- **ExecCounter:** A simple up-counter to count mock execution time in cycles.
- **WebAddrToTLDAddr:** A simple shifting module to generate a mock TLD address returned to the Resolver. This simulates a Root Nameserver, returning the corresponding TLD address to a Web Address.
- **TLDAddrToDomainIP:** A simple XOR module to generate a mock Domain address returned to the Resolver. This simulates a TLD Server, returning the corresponding Domain address to a Web Address.
- **DomainIPToWebIP:** A module of combined operations to generate a mock IP address returned to the Resolver. This simulates a Domain Nameserver, returning the corresponding IP address to a Web Address.
- **WebIPToWebdata:** A 4 to 16 Decoder module generating mock web data upon an HTTP request to an IP address. This simulates the web data being returned and rendered in a browser. In a normal rendering process, multiple HTTP requests would be made until the website is fully rendered.

3.5 Assumptions

As this project allows for much flexibility in design, assumptions were made when designing this state machine. First, the initial client queried web address is represented by an 8 bit value chosen arbitrarily, and inputted to the state machine.

As it is not possible to accurately simulate the correct server and IP addresses for a mock web address represented by bits, modules operating on the mock web address were designed in order to simulate Root, TLD, and Domain server responses to the Resolver. These addresses were then outputted by the state machine. The operations applied to the addresses were chosen arbitrarily, such that results can be reproduced.

As previously mentioned in section 2.1, the state machine includes a caching mechanism. However we simply cache a single IP address mapped to the most recently queried web address, to simulate a single block browser caching policy.

Lastly, the total execution time of the DNS Lookup process is an output to the state machine, and simulated by counting the number of clock cycles from the client request to the client resolution.

4.0 State Transition Diagram

The DNS Lookup state machine described in Section 3.0 is shown by Figure 4.1. In this state transition diagram, inputs to the state machine as well as all outputted signals per individual state are displayed.

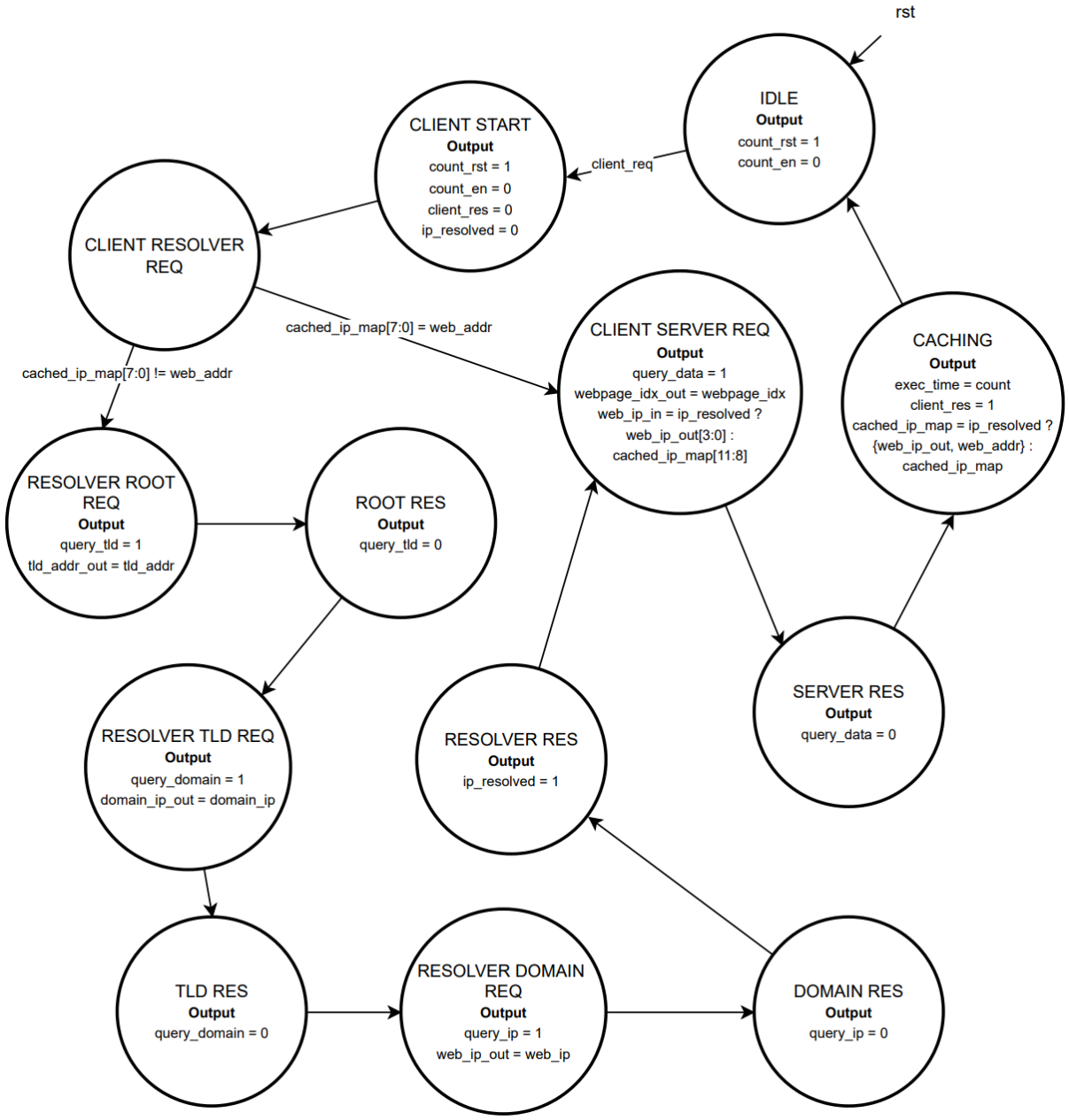


Figure 4.1: DNS Lookup State Transition Diagram

5.0 State Machine Block Diagram

A state machine and module block diagram depicting the state transitions are shown below in Figure 5.1 and Figure 5.2. The top level DNSLookup FSM module contains all modules.

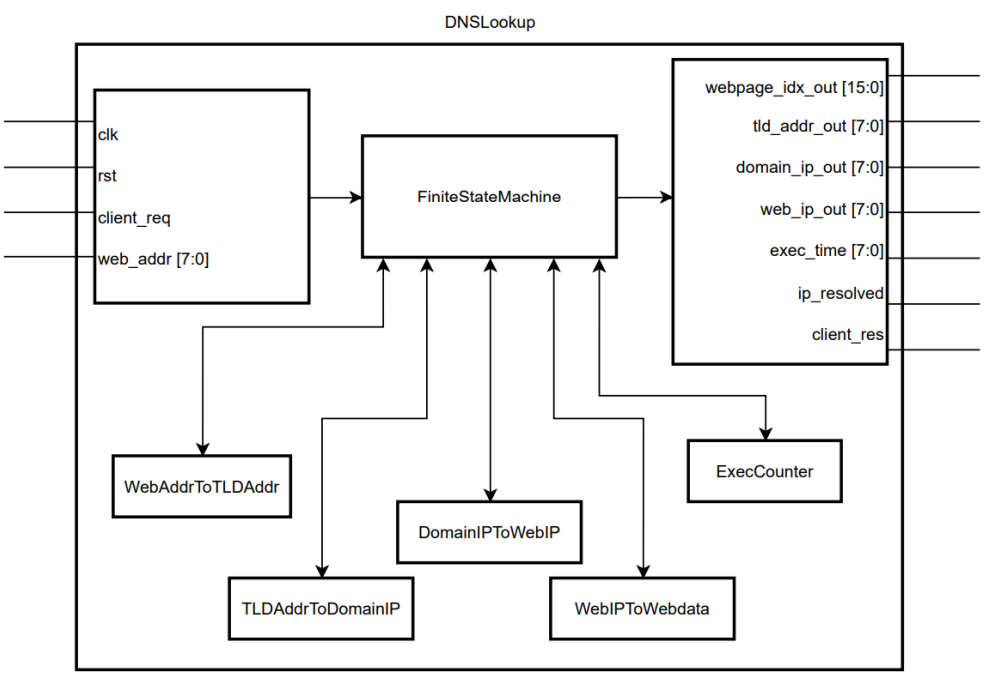


Figure 5.1: DNSLookup Top Level FSM Module

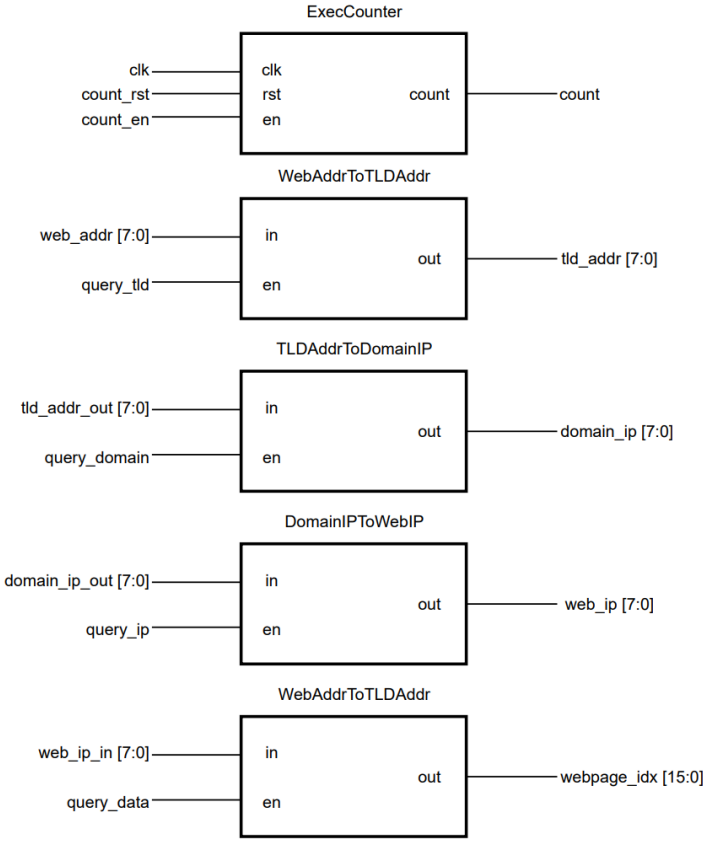


Figure 5.2: DNS Server Address Mapping and Counter Modules

Figure 5.3 describes the input and output connections of the DNSLookup top level module DUT and the testbench. This testbench stimulates a clock signal every 5 ps, a reset signal, a client request flag signal, and a mock web address chosen arbitrarily.
The objective of this testbench is not only to check the correctness of the output signals, but also to observe the state transitions when a client requests multiple addresses consecutively as the functioning of the browser caching mechanism can be verified.

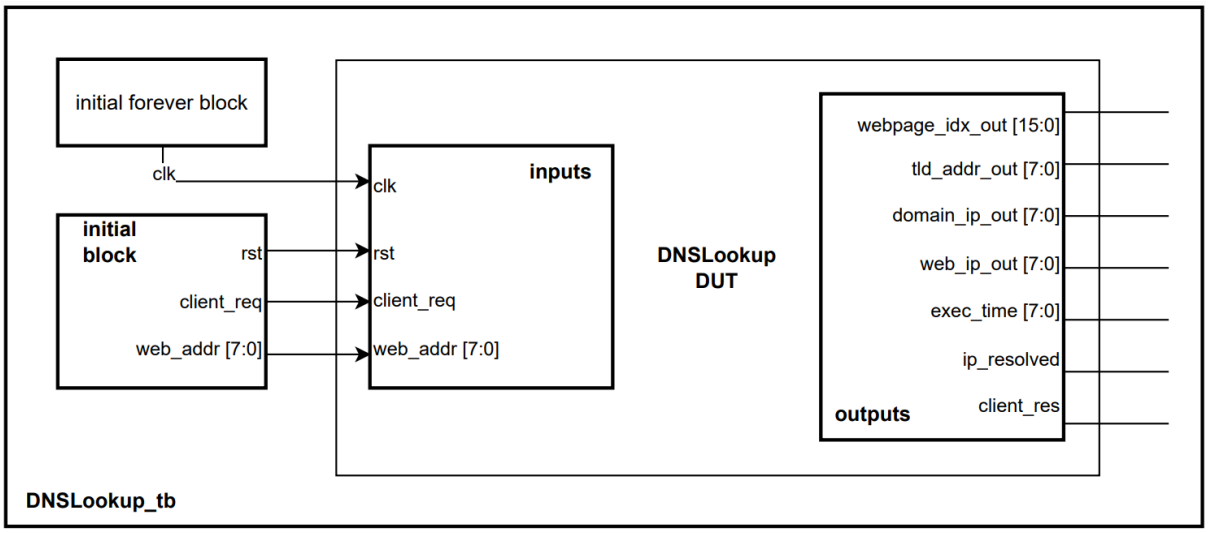


Figure 5.3: DNSlookup Testbench and Top Level Module Connections

6.0 Waveform and Testbench Simulations

A SystemVerilog Testbench was designed in order to test the FSM design and simulate various cases. In our test bench, three mock web addresses were simulated to test the caching and address mapping behaviors of the DNS Lookup state machine.

The first stimulated client query is an arbitrary Web Address (A) represented by 8 bits. In Figure 6.1 a), the query reaches the DNS Resolver, which queries the Root, TLD, and Domain servers one by one until the correct IP address is found in Figure 6.1 b). The IP Address is returned to the client, initiating an HTTP request, then cached for future use once webdata is returned and client query is resolved. The total execution time is counted to be 12.

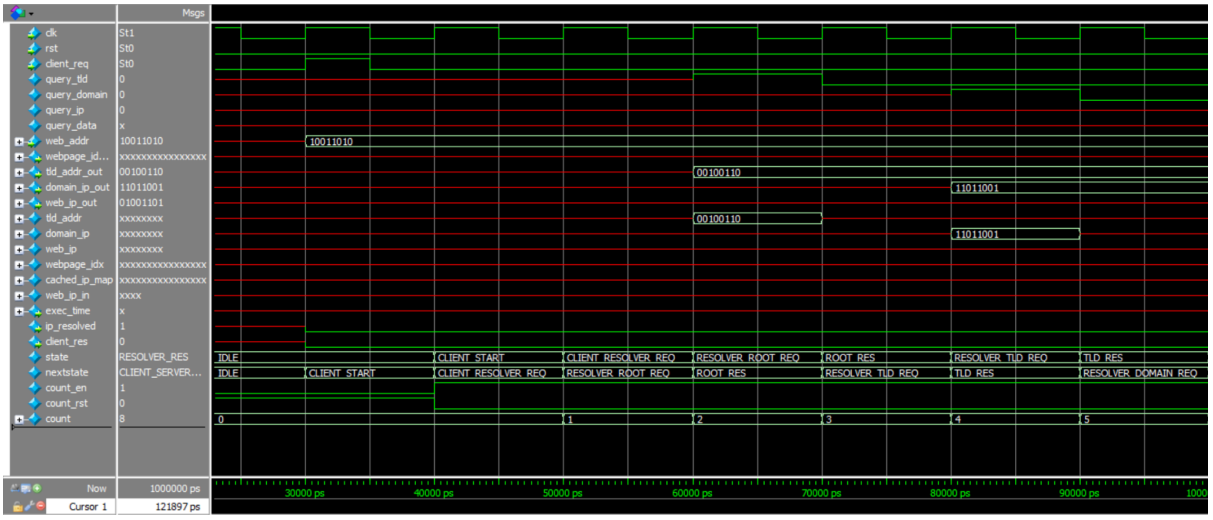


Figure 6.1 a): Web Address (A) is queried by client. Servers return corresponding addresses.

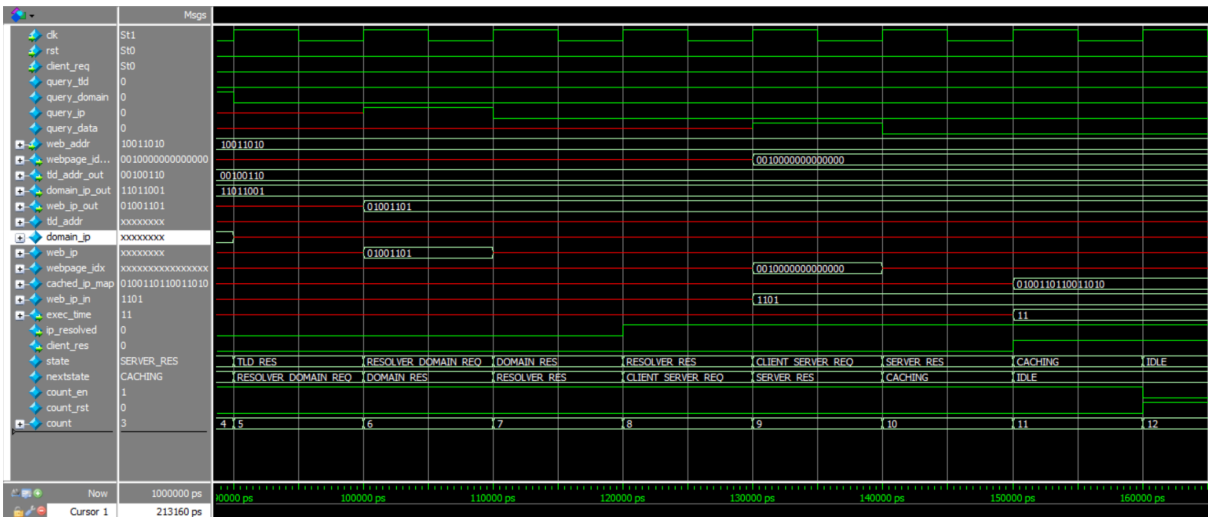


Figure 6.1 b): Corresponding IP Address to Web Address (A) is resolved, cached, and returned to client.

Once again, we use the same client web query to check that its IP address has been successfully cached. The query reaches the DNS Resolver, which maps the web address to a cached IP address as shown in Figure 6.2. The state machine directly moves to the HTTP request state, before caching again and resolving the client request as before. The total execution time is counted to be only 5 this time.

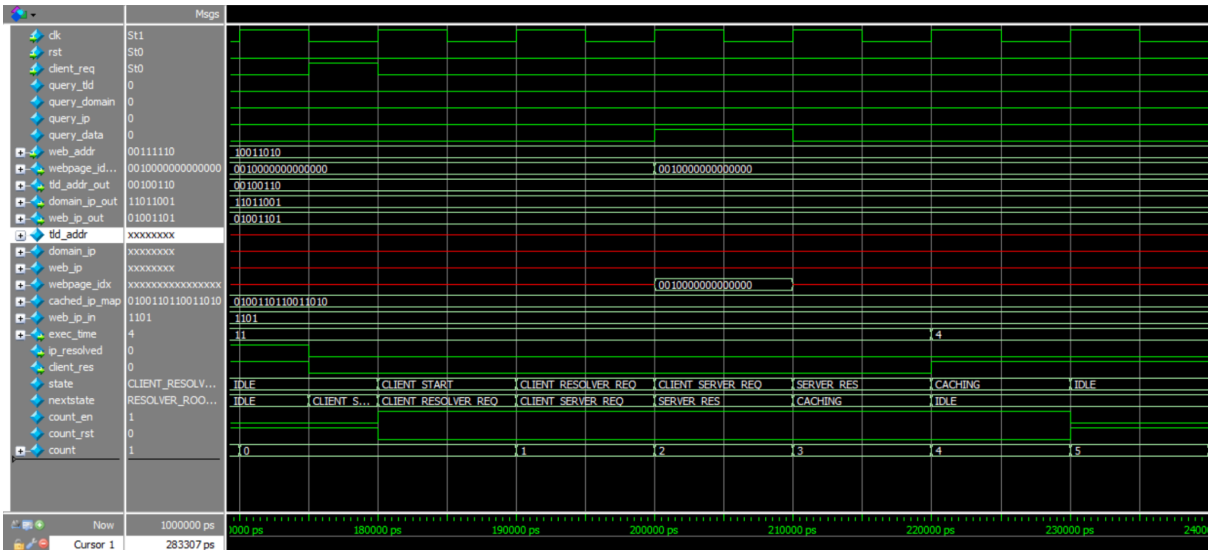


Figure 6.2: Web Address (A) is queried again by client. The client can directly query the server to render the page.

This next client query is a different arbitrary web address from the last query, represented by 8 bits. In Figure 6.2 a), the query reaches the DNS Resolver. As Web Address (B) is not cached by the browser. The Resolver proceeds to query the DNS server similarly to Figures 6.1 a) and b). A new IP address is cached, and the total execution time is 12.

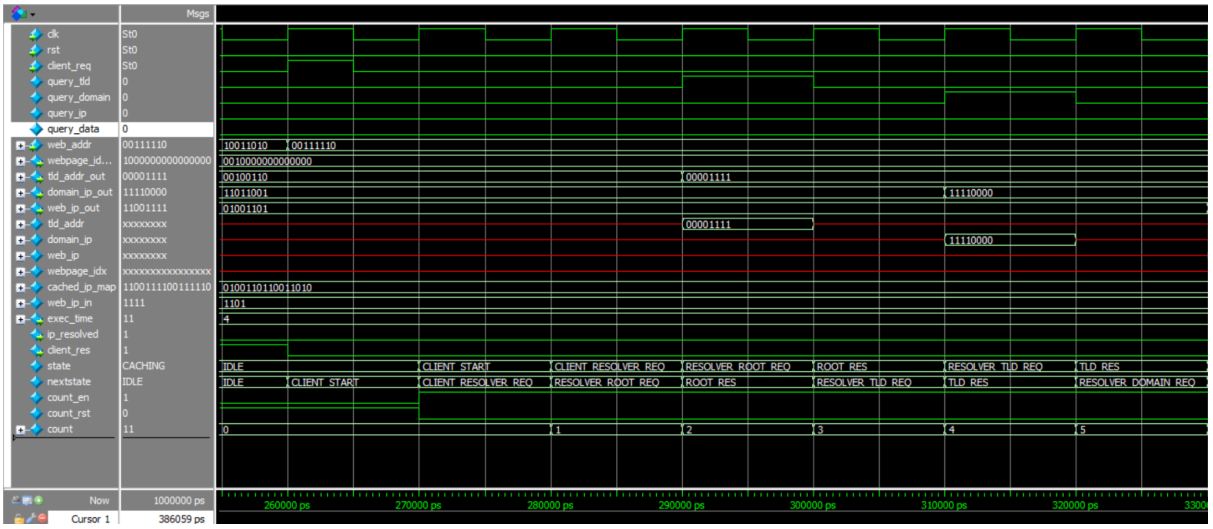


Figure 6.3 a): Web Address (B) is queried by client. Address is not cached, servers return corresponding addresses.

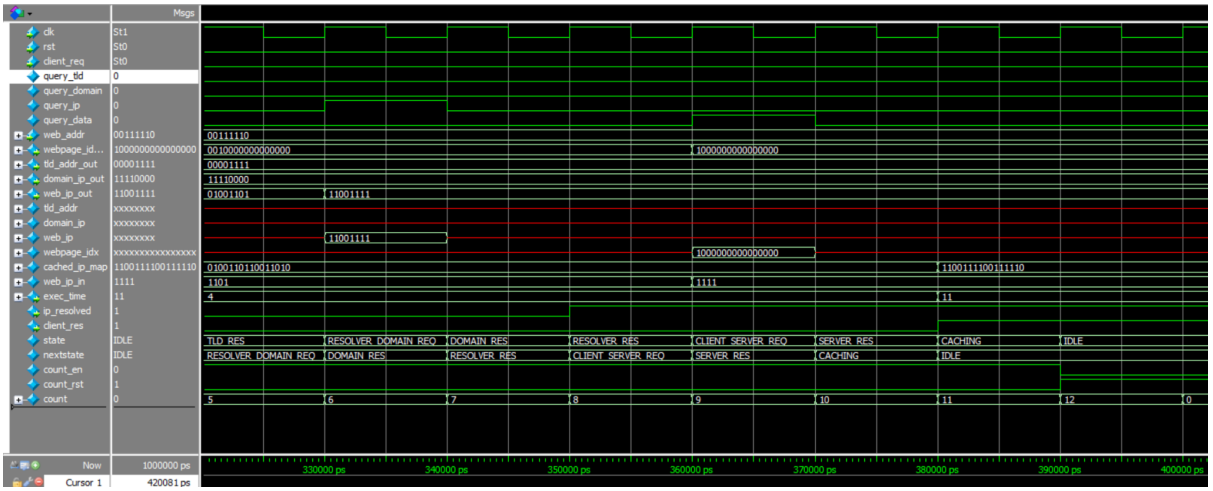


Figure 6.3 b): Corresponding IP Address to Web Address (B) is resolved, cached, and returned to client.

Appendix A: Main Code and Modules

DNSLookup.sv

```
/* Name:Isabelle Andre
* Student ID: 12521589
* Purpose: A State Machine Depicting a DNS server webpage rendering
*/

module DNSLookup (
	input logic clk,
	input logic rst,
	input logic client_req,
	input logic [7:0] web_addr,
	output logic [15:0] webpage_idx_out,
	output logic [7:0] tld_addr_out,
	output logic [7:0] domain_ip_out,
	output logic [7:0] web_ip_out,
	output logic [7:0] exec_time,
	output logic ip_resolved,
	output logic client_res
);

	typedef enum logic [4:0] {IDLE, CLIENT_START, CLIENT_RESOLVER_REQ,
RESOLVER_ROOT_REQ, ROOT_RES,
		RESOLVER_TLD_REQ, TLD_RES, RESOLVER_DOMAIN_REQ, DOMAIN_RES, RESOLVER_RES,
CLIENT_SERVER_REQ, SERVER_RES, CACHING} enumstate;
	enumstate state, nextstate;

	logic count_en, count_rst;
	logic [3:0] count, web_ip_in;
	logic [7:0] tld_addr, domain_ip, web_ip;
	logic [15:0] cached_ip_map, webpage_idx;
	logic query_tld, query_domain, query_ip, query_data;

	counter ExecCounter(
		.clk(clk),
		.rst(count_rst),
		.en(count_en),
		.count(count)
	);

	TLDAddr WebAddrToTLDAddr (
		.in(web_addr),
		.out(tld_addr),
		.en(query_tld)
	);

	DomainIP TLDAddrToDomainIP (
		.in(tld_addr_out),
		.out(domain_ip),
		.en(query_domain)
	);

	WebIP DomainIPToWebIP (
		.in(domain_ip_out),
		.out(web_ip),
		.en(query_ip)
	);

	decoder416 WebIPToWebdata (
		.in(web_ip_in),
		.out(webpage_idx),
		.en(query_data)
	);

	always_ff @(posedge clk) begin
		if (rst) begin
			state <= IDLE;
		end
		else
			state <= nextstate;
	end

	always_comb begin
```

```
		case (state)
			IDLE: begin
				{count_rst, count_en} <= 2'b10;

				if (client_req) begin
					{client_res, ip_resolved} <= 2'b00;
					nextstate <= CLIENT_START;
				end
			end
			CLIENT_START: begin
				{count_rst, count_en} <= 2'b01;
				nextstate <= CLIENT_RESOLVER_REQ;
			end
			CLIENT_RESOLVER_REQ: begin

				if (cached_ip_map[7:0] == web_addr) begin
					nextstate <= CLIENT_SERVER_REQ;
				end
				else
					nextstate <= RESOLVER_ROOT_REQ;
			end
			RESOLVER_ROOT_REQ: begin
				query_tld <= 1'b1;
				tld_addr_out <= tld_addr;
				nextstate <= ROOT_RES;
			end
			ROOT_RES: begin
				query_tld <= 1'b0;
				nextstate <= RESOLVER_TLD_REQ;
			end
			RESOLVER_TLD_REQ: begin
				query_domain <= 1'b1;
				domain_ip_out <= domain_ip;
				nextstate <= TLD_RES;
			end
			TLD_RES: begin
				query_domain <= 1'b0;
				nextstate <= RESOLVER_DOMAIN_REQ;
			end
			RESOLVER_DOMAIN_REQ: begin
				query_ip <= 1'b1;
				web_ip_out <= web_ip;
				nextstate <= DOMAIN_RES;
			end
			DOMAIN_RES: begin
				query_ip <= 1'b0;
				nextstate <= RESOLVER_RES;
			end
			RESOLVER_RES: begin
				ip_resolved <= 1'b1;
				nextstate <= CLIENT_SERVER_REQ;
			end
			CLIENT_SERVER_REQ: begin
				query_data <= 1'b1;
				web_ip_in <= ip_resolved ? web_ip_out[3:0] :
cached_ip_map[11:8];
				webpage_idx_out <= webpage_idx;
				nextstate <= SERVER_RES;
			end
			SERVER_RES: begin
				query_data <= 1'b0;
				nextstate <= CACHING;
			end
			CACHING: begin
				cached_ip_map <= ip_resolved ? {web_ip_out, web_addr} :
cached_ip_map;

				exec_time <= count;
				client_res <= 1'b1;
				nextstate <= IDLE;
			end
			default: begin
				nextstate <= IDLE;
			end
		endcase
	end
endmodule
```

TLDAddr.sv

```
/* Name:Isabelle Andre
* Student ID: 12521589
* Purpose: A module simulating the process of querying for a TLD address
*/
module TLDAddr (
        input logic [7:0] in,
    output logic [7:0] out,
    input logic en
);

        assign out = en ? in>>2 : 8'bx;
endmodule
```

DomainIP.sv

```
/* Name:Isabelle Andre
* Student ID: 12521589
* Purpose: A module simulating the process of querying for a DomainIP
*/
module DomainIP (
        input logic [7:0] in,
    output logic [7:0] out,
    input logic en
);

        assign out = en ? in^8'b1111_1111 : 8'bx;
endmodule
```

WebIP.sv

```
/* Name:Isabelle Andre
* Student ID: 12521589
* Purpose: A module simulating the process of querying for a TLD address
*/
module WebIP (
        input logic [7:0] in,
    output logic [7:0] out,
    input logic en
);
        assign out = en ? {in[7:4]<<2, (in[3:0]>>2)^4'b1111} : 8'bx;
endmodule
```

counter.sv

```
/* Name:Isabelle Andre
* Student ID: 12521589
* Purpose: A simple up-counter
*/
module counter (
        input logic clk,
        input logic rst,
        input logic en,
        output logic[3:0] count
);

always_ff @(posedge clk) begin
        if (rst) begin
                count <= 0;
        end
```

```
        else if (en) begin
                count <= count + 1;
        end
end
endmodule
```

decoder416.sv

```
/* Name:Isabelle Andre
* Student ID: 12521589
* Purpose: A 4 to 16 decoder module to simulate web data to ip address mapping
*/

module decoder416 (
   input logic [3:0] in,
   output logic [15:0] out,
   input logic en
);
   parameter tmp = 16'b0000_0000_0000_0001;

   always_comb begin
      if (en) begin
         out = (in == 4'b0000) ? tmp   :
                     (in == 4'b0001) ? tmp<<1:
                     (in == 4'b0010) ? tmp<<2:
                     (in == 4'b0011) ? tmp<<3:
                     (in == 4'b0100) ? tmp<<4:
                     (in == 4'b0101) ? tmp<<5:
                     (in == 4'b0110) ? tmp<<6:
                     (in == 4'b0111) ? tmp<<7:
                     (in == 4'b1000) ? tmp<<8:
                     (in == 4'b1001) ? tmp<<9:
                     (in == 4'b1010) ? tmp<<10:
                     (in == 4'b1011) ? tmp<<11:
                     (in == 4'b1100) ? tmp<<12:
                     (in == 4'b1101) ? tmp<<13:
                     (in == 4'b1110) ? tmp<<14:
                     (in == 4'b1111) ? tmp<<15: 16'bx;
      end
      else
         out = 16'bx;
   end

endmodule
```

Appendix B: Testbench

```
// Specify simulation timescape
// Format: unit step / resolution
`timescale 1ns/1ps

module DNSLookup_tb();
        logic clk, rst, client_req;
        logic [7:0] web_addr, tld_addr, domain_ip, web_ip, exec_time;
        logic [15:0] webpage_idx;
        logic ip_resolved, client_res;

        // Instantiate the module to test
        DNSLookup DUT (
                .clk(clk),
                .rst(rst),
                .client_req(client_req),
```

```
                .web_addr(web_addr),
                .webpage_idx_out(webpage_idx),
                .tld_addr_out(tld_addr),
                .domain_ip_out(domain_ip),
                .web_ip_out(web_ip),
                .exec_time(exec_time),
                .ip_resolved(ip_resolved),
                .client_res(client_res)
        );

        // Initialize signals
        initial begin
                {clk, rst} = 2'b01;
                client_req = 1'b0;

        forever
        begin
                clk = ~clk;
                #5;
        end
        end

        initial begin
                #15;
                rst = 1'b0;
                #15;

                // query first web addr
                client_req = 1'b1;
                web_addr = 8'b1001_1010;
                #5;
                client_req = 1'b0;
                #140;

                // test that first web addr is cached by querying again
                client_req = 1'b1;
                web_addr = 8'b1001_1010;
                #5;
                client_req = 1'b0;
                #80

                // query second web addr, check that new address is cached
                client_req = 1'b1;
                web_addr = 8'b0011_1110;
                #5;
                client_req = 1'b0;
                #140;
        end
endmodule
```