ELEC 402 Assignment 1

Tyler Keeling 60697448

FSM Description

This FSM implements part of the sha-256 hash algorithm. It works as follows:

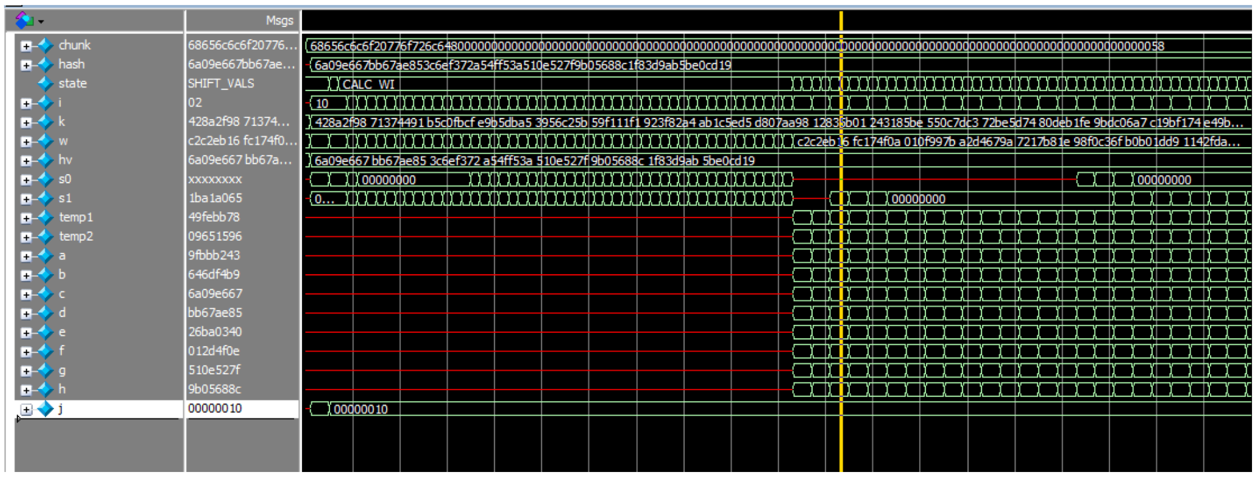
In order for a series of bits to be hashed by sha-256, it must be padded with a ‘1’ bit and also the length of the message to be hashed, in such a way that the entire “chunk” of data is a multiple of 512. Then, the hash is calculated, one 512-bit chunk at a time.

This FSM is the part responsible for handling the 512-bit chunks, so long as the data is already pre-processed. It uses valid/ready handshaking, meaning that the FSM will only start if the input “valid” flag is set high and the FSM is currently not processing another chunk (i.e., it is ready).

This implementation uses only 6 states and not the required 10 but I ask that you please be lenient with marking as this project is somewhat complex and it took a lot of time for the functionality to be correct 🙏. Thank you.

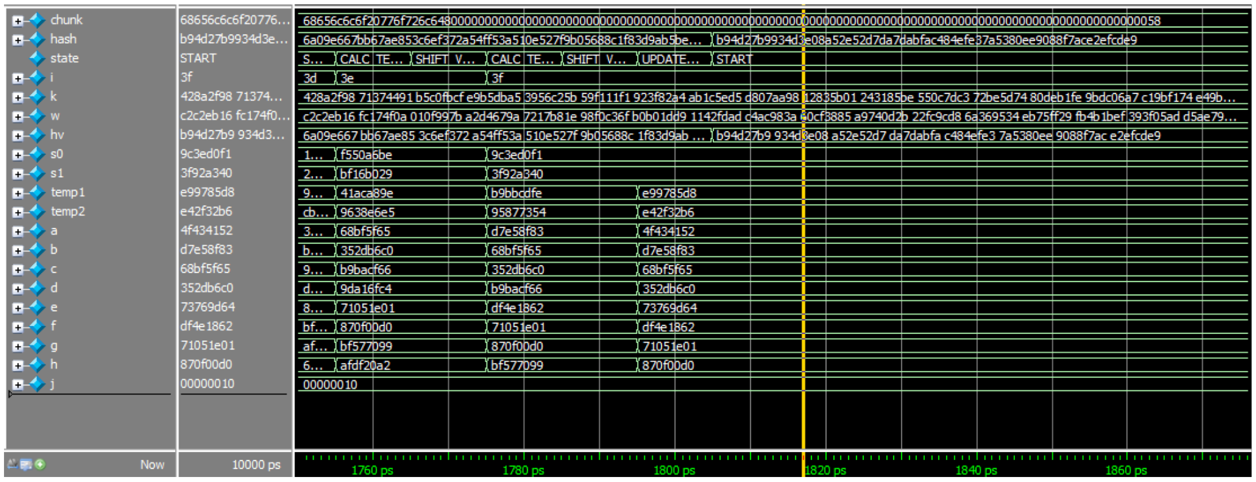
Simulation Waveform Results

Figure 1: See the state machine at the beginning of its operation. Notice how internal registers a-f are being initialized.



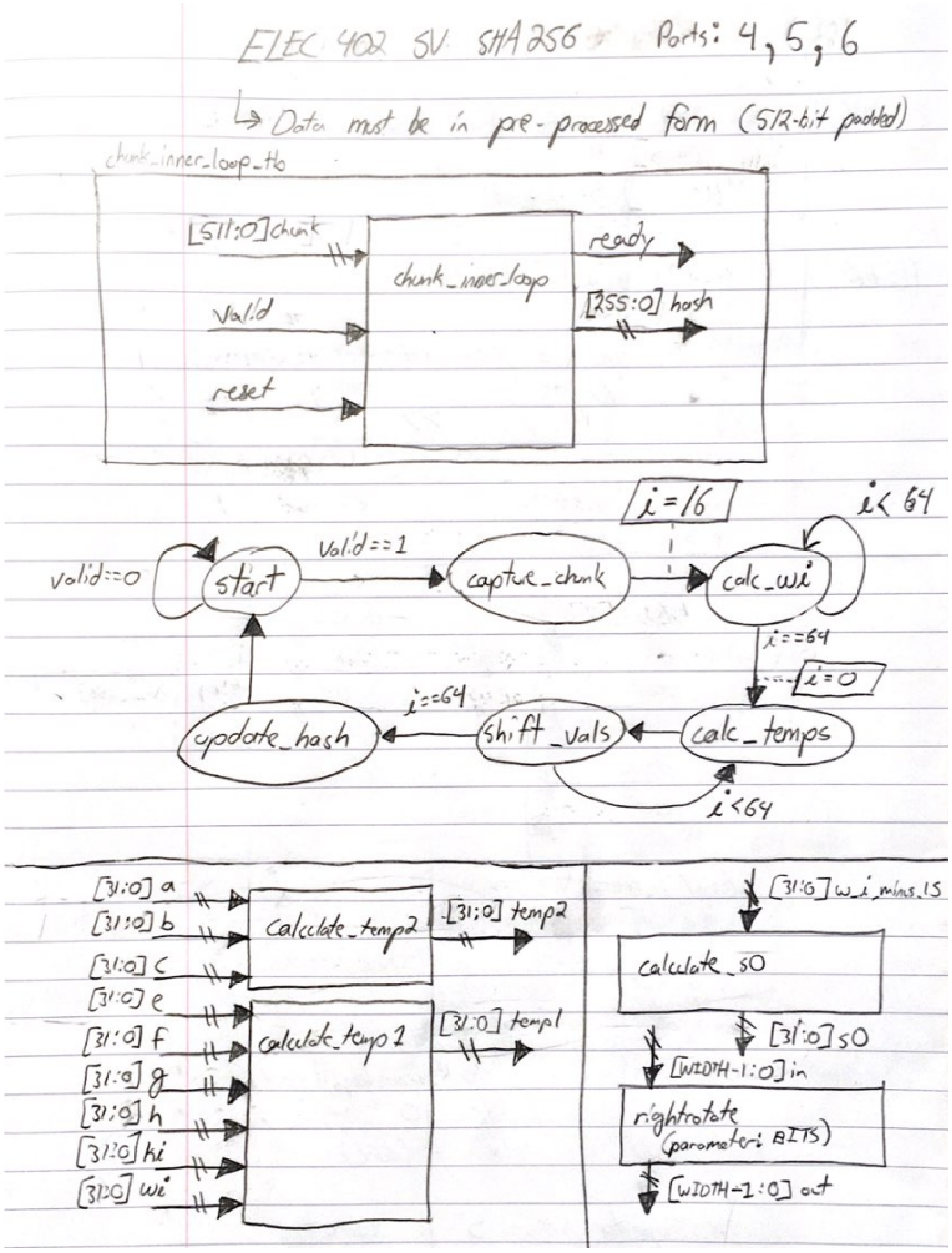
	Msgs
chunk	68656c6c6f20776...
hash	6a09e667bb67ae...
state	SHIFT_VALS
i	02
k	428a2f98 71374...
w	c2c2eb16 fc174f0...
hv	6a09e667 bb67a...
s0	xxxxxxxx
s1	1ba1a065
temp1	49febb78
temp2	09651596
a	9fbbb243
b	646df4b9
c	6a09e667
d	bb67ae85
e	26ba0340
f	012d4f0e
g	510e527f
h	9b05688c
j	00000010

Figure 2: The state machine is complete and the state is moved back to START (in case there is another chunk to be processed). Notice how the hash value for the input “hello world” is the same as one used in an online tool like this.



	Msgs
chunk	68656c6c6f20776...
hash	b94d27b9934d3e...
state	START
i	3f
k	428a2f98 71374...
w	c2c2eb16 fc174f0...
hv	b94d27b9 934d3...
s0	9c3ed0f1
s1	3f92a340
temp1	e99785d8
temp2	e42f32b6
a	4f434152
b	d7e58f83
c	68bf5f65
d	352db6c0
e	73769d64
f	df4e1862
g	71051e01
h	870f00d0
j	00000010
Now	10000 ps

Block Diagram of FSM, modules, and testbench connection:



Copy of Code

All .sv files are also attached to this submission

Chunk Inner Loop (main module)

```
module chunk_inner_loop(
    input logic [511:0]chunk, output logic [255:0]hash,
    input logic valid, reset, clk,
    output logic ready
);

enum {START, CAPTURE_CHUNK, CALC_WI, CALC_TEMP, SHIFT_VALS, UPDATE_HASH} state;

reg [5:0] i; // counts up to 63

// k and w are inverted
reg [31:0] k [63:0]; // 32-bit values with array depth 64
reg [31:0] w [63:0];
reg [31:0] hv [7:0];

wire [31:0] s0;
wire [31:0] s1;

wire [31:0] temp1;
wire [31:0] temp2;

reg [31:0] a;
reg [31:0] b;
reg [31:0] c;
reg [31:0] d;
reg [31:0] e;
reg [31:0] f;
reg [31:0] g;
reg [31:0] h;

calculate_temp1 ctemp1(
    .e(e),
    .f(f),
    .g(g),
    .h(h),
    .ki(k[63-i]),
    .wi(w[i]),
    .temp1(temp1)
);

calculate_temp2 ctemp2(
    .a(a),
    .b(b),
    .c(c),
    .temp2(temp2)
);

// For the right rotates
calculate_s0 d0(
    .w_i_minus_15(w[i-15]),
    .s0(s0)
);

calculate_s1 d1(
    .w_i_minus_2(w[i-2]),
    .s1(s1)
);

assign ready = (state == START);

assign hash = {
    hv[7],
    hv[6],
    hv[5],
    hv[4],
    hv[3],
    hv[2],
    hv[1],
    hv[0]
};

// initialize values of k, hv, when reset
always_ff @(posedge clk) begin
    if (reset) begin
        k <= {
            'h428a2f98, 'h71374491, 'hb5c0fbcf, 'he9b5dba5, 'h3956c25b, 'h59f111f1, 'h923f82a4, 0'hab1c5ed5,
            'hd807aa98, 'h12835b01, 'h243185be, 'h550c7dc3, 'h72be5d74, 'h80deb1fe, 'h9bdc06a7, 0'hc19bf174,
            'he49b69c1, 'hefbe4786, 'h0fc19dc6, 'h240ca1cc, 'h2de92c6f, 'h4a7484aa, 'h5cb0a9dc, 0'h76f988da,
            'h983e5152, 'ha831c66d, 'hb00327c8, 'hbf597fc7, 'hc6e00bf3, 'hd5a79147, 'h06ca6351, 0'h14292967,
            'h27b70a85, 'h2e1b2138, 'h4d2c6dfc, 'h53380d13, 'h650a7354, 'h766a0abb, 'h81c2c92e, 0'h92722c85,
            'ha2bfe8a1, 'ha81a664b, 'hc24b8b70, 'hc76c51a3, 'hd192e819, 'hd6990624, 'hf40e3585, 0'h106aa070,
            'h19a4c116, 'h1e376c08, 'h2748774c, 'h34b0bcb5, 'h391c0cb3, 'h4ed8aa4a, 'h5b9cca4f, 0'h682e6ff3,
            'h748f82ee, 'h78a5636f, 'h84c87814, 'h8cc70208, 'h90befffa, 'ha4506ceb, 'hbef9a3f7, 0'hc67178f2
        };

        hv <= {
            'h6a09e667,
            'hbb67ae85,
            'h3c6ef372,
            'ha54ff53a,
            'h510e527f,
            'h9b05688c,
            'h1f83d9ab,
            'h5be0cd19
        };
    end
end
```

```
integer j;

// FSM Traversal
always_ff @(posedge clk) begin
    if (reset) begin
        state <= START;

        for (j = 0; j < 64; j++) begin : zero_w
            w[j] <= 32'b0; // zero all elements of w in 1 clock cycle
        end

        i <= 16;
    end

    else case(state)
        START: if (valid) begin
            state <= CAPTURE_CHUNK;

            // populate the first 16 elements of w in 1 clock cycle
            for (j = 0; j < 16; j++) begin
                w[15-j] <= chunk[j*32+:32]; // endian might be wrong, watch out
            end
        end

        // This can be removed but my state machine needs more states
        CAPTURE_CHUNK: begin
            state <= CALC_WI;
        end

        CALC_WI: begin

            // s0 and s1 are determined combinationally in other module
            w[i] <= w[i-16] + s0 + w[i-7] + s1;

            if (i == 63) begin
                i <= 0;
                state <= CALC_TEMP;

                // set a to h for calc_temp operations
                a <= hv[7];
                b <= hv[6];
                c <= hv[5];
                d <= hv[4];
                e <= hv[3];
                f <= hv[2];
                g <= hv[1];
                h <= hv[0];
            end else begin
                i <= i + 1'b1;
            end
        end

        // temp1, temp2 are computed combinationally so this is less necessary
        CALC_TEMP: begin
            state <= SHIFT_VALS;
        end

        SHIFT_VALS: begin
            h <= g;
            g <= f;
            f <= e;
            e <= d + temp1;
            d <= c;
            c <= b;
            b <= a;
            a <= temp1+temp2;

            if (i == 63) begin
                state <= UPDATE_HASH;
            end else begin
                i <= i + 1'b1;
                state <= CALC_TEMP;
            end
        end

        UPDATE_HASH: begin
            hv[0] <= hv[0] + h;
            hv[1] <= hv[1] + g;
            hv[2] <= hv[2] + f;
            hv[3] <= hv[3] + e;
            hv[4] <= hv[4] + d;
            hv[5] <= hv[5] + c;
            hv[6] <= hv[6] + b;
            hv[7] <= hv[7] + a;

            state <= START;
        end
    endcase
end

endmodule
```

Calculate s0

```
module calculate_s0(
    input [31:0] w_i_minus_15,
    output [31:0] s0
);

wire [31:0] t1;
wire [31:0] t2;

rightrotate #(.BITS(7))  r0 (.in(w_i_minus_15), .out(t1));
rightrotate #(.BITS(18)) r1 (.in(w_i_minus_15), .out(t2));
```

```
assign s0 = t1 ^ t2 ^ (w_i_minus_15 >> 3);

endmodule
```

Calculate s1

```
module calculate_s1(
    input [31:0] w_i_minus_2,
    output [31:0] s1
);

wire [31:0] t1;
wire [31:0] t2;

rightrotate #(.BITS(17)) r0 (.in(w_i_minus_2), .out(t1));
rightrotate #(.BITS(19)) r1 (.in(w_i_minus_2), .out(t2));

assign s1 = t1 ^ t2 ^ (w_i_minus_2 >> 10);

Endmodule
```

Calculate temp1

```
module calculate_temp1(
    input logic [31:0] e, f, g, h, ki, wi,
    output logic [31:0] temp1
);

reg [31:0] s1, ch, t1, t2, t3;

rightrotate #(.BITS(6))  r0 (.in(e), .out(t1));
rightrotate #(.BITS(11)) r1 (.in(e), .out(t2));
rightrotate #(.BITS(25)) r2 (.in(e), .out(t3));

always_comb begin
    s1 = t1 ^ t2 ^ t3;
    ch = (e & f) ^ (~e & g);
    temp1 = h + s1 + ch + ki + wi;
end

endmodule
```

Calculate temp2

```
module calculate_temp2(
    input logic [31:0] a, b, c,
    output logic [31:0] temp2
);
```

```
logic [31:0] s0, maj, t1, t2, t3;

rightrotate #(.BITS(2))  r0 (.in(a), .out(t1));
rightrotate #(.BITS(13)) r1 (.in(a), .out(t2));
rightrotate #(.BITS(22)) r2 (.in(a), .out(t3));

always_comb begin
    s0 = t1 ^ t2 ^ t3;
    maj = (a & b) ^ (a & c) ^ (b & c);
    temp2 = s0 + maj;
end

endmodule
```

Right Rotate

```
module rightrotate
#(
    parameter WIDTH=32,
    parameter BITS=4
)
(
    input logic [WIDTH-1:0] in,
    output [WIDTH-1:0] out
);

assign out = (in >> BITS) | (in << (WIDTH-BITS));

endmodule
```