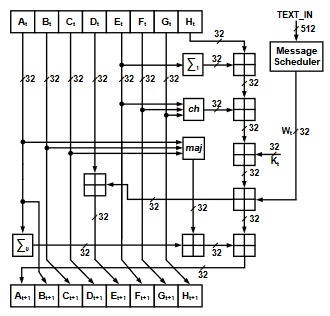
SHA256 Project

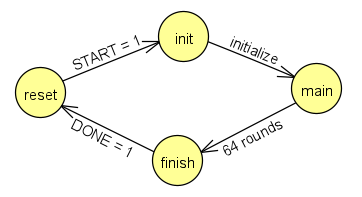
디지털설계및실험

2021110704 김다진

# Datapath

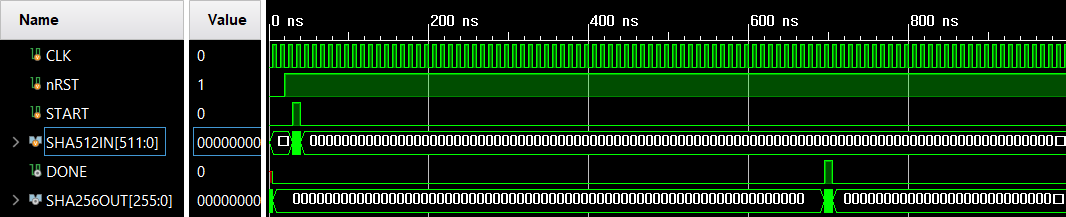


# FSM



# Waveforms

## Behavioral simulation



# Source description

## C

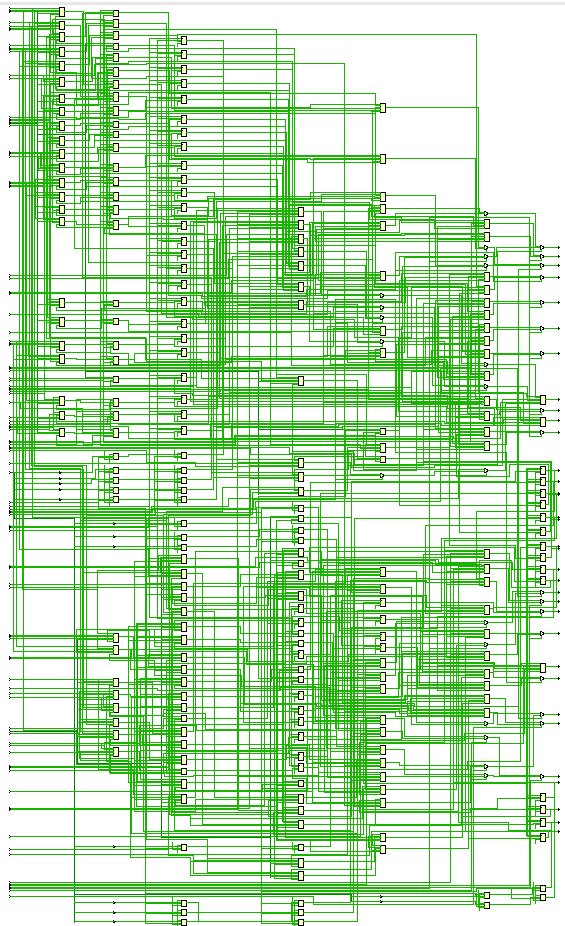
|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <string.h>  #define DEBUG 1  #define BYTE2BIT 8  #define WORD2BYTE 4  #define N\_ROUND 64  #define MESSAGE\_SIZE 512 / BYTE2BIT  #define HASH\_SIZE 256 / BYTE2BIT / sizeof(int)  #define ROTR(x, n) (((x) >> (n)) | ((x) << (sizeof(int) \* BYTE2BIT - (n))))  // SHA-256 Functions  #define CH(x, y, z) (((x) & (y)) ^ (~(x) & (z)))  #define MAJ(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))  #define BSIG0(x) (ROTR((x), 2) ^ ROTR((x), 13) ^ ROTR((x), 22))  #define BSIG1(x) (ROTR((x), 6) ^ ROTR((x), 11) ^ ROTR((x), 25))  #define SSIG0(x) (ROTR((x), 7) ^ ROTR((x), 18) ^ ((x) >> 3))  #define SSIG1(x) (ROTR((x), 17) ^ ROTR((x), 19) ^ ((x) >> 10))  // SHA-256 Constants  const unsigned int K[N\_ROUND] = {  0x428a2f98, 0x71374491, 0xb5c0fbcf, 0xe9b5dba5, 0x3956c25b,  0x59f111f1, 0x923f82a4, 0xab1c5ed5, 0xd807aa98, 0x12835b01,  0x243185be, 0x550c7dc3, 0x72be5d74, 0x80deb1fe, 0x9bdc06a7,  0xc19bf174, 0xe49b69c1, 0xefbe4786, 0x0fc19dc6, 0x240ca1cc,  0x2de92c6f, 0x4a7484aa, 0x5cb0a9dc, 0x76f988da, 0x983e5152,  0xa831c66d, 0xb00327c8, 0xbf597fc7, 0xc6e00bf3, 0xd5a79147,  0x06ca6351, 0x14292967, 0x27b70a85, 0x2e1b2138, 0x4d2c6dfc,  0x53380d13, 0x650a7354, 0x766a0abb, 0x81c2c92e, 0x92722c85,  0xa2bfe8a1, 0xa81a664b, 0xc24b8b70, 0xc76c51a3, 0xd192e819,  0xd6990624, 0xf40e3585, 0x106aa070, 0x19a4c116, 0x1e376c08,  0x2748774c, 0x34b0bcb5, 0x391c0cb3, 0x4ed8aa4a, 0x5b9cca4f,  0x682e6ff3, 0x748f82ee, 0x78a5636f, 0x84c87814, 0x8cc70208,  0x90befffa, 0xa4506ceb, 0xbef9a3f7, 0xc67178f2  };  #if DEBUG  int count = 0;  #endif  void parse(unsigned char text[MESSAGE\_SIZE], unsigned char text\_parse[MESSAGE\_SIZE]);  void process(unsigned char text\_parse[MESSAGE\_SIZE], unsigned int hash[HASH\_SIZE]);  void print(unsigned int hash[HASH\_SIZE]);  int main() {  unsigned char text[MESSAGE\_SIZE], text\_parse[MESSAGE\_SIZE] = {};  // SHA-256 Initialization  unsigned int hash[HASH\_SIZE] = {  0x6A09E667, 0xBB67AE85, 0x3C6EF372, 0xA54FF53A, 0x510E527F, 0x9B05688C, 0x1F83D9AB, 0x5BE0CD19  };  int i, j;  printf("input text: ");  scanf("%s", text);  parse(text, text\_parse);  #if DEBUG  printf("input 512 bits: ");  for (i = 0; i < MESSAGE\_SIZE; i++) {  printf("%02x", text\_parse[i]);  }  printf("\n\n");  #endif  process(text\_parse, hash);  printf("\nresult: ");  print(hash);  return 0;  }  // SHA-256 Message Pading and Parsing  void parse(unsigned char text[MESSAGE\_SIZE], unsigned char text\_parse[MESSAGE\_SIZE]) {  int len, i;  len = strlen(text);  for (i = 0; i < len; i++) {  text\_parse[i] = text[i];  }  text\_parse[i] = 1 << 7;  text\_parse[MESSAGE\_SIZE - 1] = len \* BYTE2BIT;  }  /\*  \* SHA-256 Processing  \* 1. prepare the message schedule W  \* 2. initialize the working variables  \* 3. perform the main hash computation  \* 4. compute the hash value  \*/  void process(unsigned char text\_parse[MESSAGE\_SIZE], unsigned int hash[HASH\_SIZE]) {  unsigned int W[N\_ROUND] = {}, a, b, c, d, e, f, g, h, T1, T2;  int i, j;  // 1. prepare the message schedule W  for (i = 0; i < 16; i++) {  for (j = 0; j < WORD2BYTE; j++) {  W[i] |= text\_parse[WORD2BYTE \* i + j] << BYTE2BIT \* (WORD2BYTE - (j + 1));  }  }  for (; i < N\_ROUND; i++) {  W[i] = SSIG1(W[i - 2]) + W[i - 7] + SSIG0(W[i - 15]) + W[i - 16];  }  // 2. initialize the working variables  a = hash[0];  b = hash[1];  c = hash[2];  d = hash[3];  e = hash[4];  f = hash[5];  g = hash[6];  h = hash[7];  // 3. perform the main hash computation  for (i = 0; i < N\_ROUND; i++) {  T1 = h + BSIG1(e) + CH(e, f, g) + K[i] + W[i];  T2 = BSIG0(a) + MAJ(a, b, c);  h = g;  g = f;  f = e;  e = d + T1;  d = c;  c = b;  b = a;  a = T1 + T2;  #if DEBUG  printf("Round: %d --> A: %08x, B: %08x, C: %08x, D: %08x, E: %08x, F: %08x, G: %08x, H: %08x, T1: %08x, T2: %08x\n", ++count, a, b, c, d, e, f, g, h, T1, T2);  #endif  }  // 4. compute the hash value  hash[0] = a + hash[0];  hash[1] = b + hash[1];  hash[2] = c + hash[2];  hash[3] = d + hash[3];  hash[4] = e + hash[4];  hash[5] = f + hash[5];  hash[6] = g + hash[6];  hash[7] = h + hash[7];  }  // Print hash value  void print(unsigned int hash[HASH\_SIZE]) {  int i;  for (i = 0; i < HASH\_SIZE; i++) {  printf("%08x", hash[i]);  }  } |

## Verilog

|  |
| --- |
| `define OUT\_SIZE 256  `define IN\_SIZE 512  `define WORD\_SIZE 32  `define N\_ROUND 64  // SHA-256 Functions  `define ROTR(x, n) (((x) >> (n)) | ((x) << (`WORD\_SIZE - (n))))  `define CH(x, y, z) (((x) & (y)) ^ (~(x) & (z)))  `define MAJ(x, y, z) (((x) & (y)) ^ ((x) & (z)) ^ ((y) & (z)))  `define BSIG0(x) (`ROTR((x), 2) ^ `ROTR((x), 13) ^ `ROTR((x), 22))  `define BSIG1(x) (`ROTR((x), 6) ^ `ROTR((x), 11) ^ `ROTR((x), 25))  `define SSIG0(x) (`ROTR((x), 7) ^ `ROTR((x), 18) ^ ((x) >> 3))  `define SSIG1(x) (`ROTR((x), 17) ^ `ROTR((x), 19) ^ ((x) >> 10))  module sha256(DONE, SHA256OUT, CLK, nRST, START, SHA512IN);  output reg DONE;  output reg [`OUT\_SIZE - 1:0] SHA256OUT;  input CLK, nRST, START;  input [`IN\_SIZE - 1:0] SHA512IN;  parameter reset = 2'd0,  init = 2'd1,  main = 2'd2,  finish = 2'd3;  reg [`IN\_SIZE - 1:0] data;  reg [`OUT\_SIZE - 1:0] hash;  reg [`WORD\_SIZE - 1:0] a, b, c, d, e, f, g, h, T1, T2, K[0:`N\_ROUND - 1], W[0:`N\_ROUND - 1];  reg [1:0] state, next\_state;  reg [6:0] round\_counter;  initial begin  // SHA-256 Constants  K[0] = 32'h428a2f98; K[1] = 32'h71374491; K[2] = 32'hb5c0fbcf;  K[3] = 32'he9b5dba5; K[4] = 32'h3956c25b; K[5] = 32'h59f111f1;  K[6] = 32'h923f82a4; K[7] = 32'hab1c5ed5; K[8] = 32'hd807aa98;  K[9] = 32'h12835b01; K[10] = 32'h243185be; K[11] = 32'h550c7dc3;  K[12] = 32'h72be5d74; K[13] = 32'h80deb1fe; K[14] = 32'h9bdc06a7;  K[15] = 32'hc19bf174; K[16] = 32'he49b69c1; K[17] = 32'hefbe4786;  K[18] = 32'h0fc19dc6; K[19] = 32'h240ca1cc; K[20] = 32'h2de92c6f;  K[21] = 32'h4a7484aa; K[22] = 32'h5cb0a9dc; K[23] = 32'h76f988da;  K[24] = 32'h983e5152; K[25] = 32'ha831c66d; K[26] = 32'hb00327c8;  K[27] = 32'hbf597fc7; K[28] = 32'hc6e00bf3; K[29] = 32'hd5a79147;  K[30] = 32'h06ca6351; K[31] = 32'h14292967; K[32] = 32'h27b70a85;  K[33] = 32'h2e1b2138; K[34] = 32'h4d2c6dfc; K[35] = 32'h53380d13;  K[36] = 32'h650a7354; K[37] = 32'h766a0abb; K[38] = 32'h81c2c92e;  K[39] = 32'h92722c85; K[40] = 32'ha2bfe8a1; K[41] = 32'ha81a664b;  K[42] = 32'hc24b8b70; K[43] = 32'hc76c51a3; K[44] = 32'hd192e819;  K[45] = 32'hd6990624; K[46] = 32'hf40e3585; K[47] = 32'h106aa070;  K[48] = 32'h19a4c116; K[49] = 32'h1e376c08; K[50] = 32'h2748774c;  K[51] = 32'h34b0bcb5; K[52] = 32'h391c0cb3; K[53] = 32'h4ed8aa4a;  K[54] = 32'h5b9cca4f; K[55] = 32'h682e6ff3; K[56] = 32'h748f82ee;  K[57] = 32'h78a5636f; K[58] = 32'h84c87814; K[59] = 32'h8cc70208;  K[60] = 32'h90befffa; K[61] = 32'ha4506ceb; K[62] = 32'hbef9a3f7;  K[63] = 32'hc67178f2;  // SHA-256 Initialization  hash = {  32'h6A09E667, 32'hBB67AE85, 32'h3C6EF372, 32'hA54FF53A, 32'h510E527F, 32'h9B05688C, 32'h1F83D9AB, 32'h5BE0CD19  };  next\_state = 2'd0;  round\_counter = 5'd0;  end  always @(START) begin  if (START) begin  data = SHA512IN;  end  end  always @(DONE) begin  if (DONE) begin  SHA256OUT = hash;  end else begin  SHA256OUT = 256'd0;  end  end  always @(state or START or DONE) begin  case (state)  reset: begin  if (START) begin  next\_state = init;  end  end  init: begin  next\_state = main;  end  main: begin  next\_state = finish;  end  finish: begin  if (DONE) begin  next\_state = reset;  end  end  endcase  end  always @(posedge CLK) begin  if (nRST) begin  if (state == main) begin  if (round\_counter == `N\_ROUND) begin  state <= next\_state;  end  end else begin  state <= next\_state;  end  end else begin  state <= reset;  end  end  always @(state) begin  case (state)  // Prepare the message schedule W[0] ~ W[15]  // and initialize the working variables  init: begin  W[0] = data[16 \* `WORD\_SIZE - 1:15 \* `WORD\_SIZE];  W[1] = data[15 \* `WORD\_SIZE - 1:14 \* `WORD\_SIZE];  W[2] = data[14 \* `WORD\_SIZE - 1:13 \* `WORD\_SIZE];  W[3] = data[13 \* `WORD\_SIZE - 1:12 \* `WORD\_SIZE];  W[4] = data[12 \* `WORD\_SIZE - 1:11 \* `WORD\_SIZE];  W[5] = data[11 \* `WORD\_SIZE - 1:10 \* `WORD\_SIZE];  W[6] = data[10 \* `WORD\_SIZE - 1:9 \* `WORD\_SIZE];  W[7] = data[9 \* `WORD\_SIZE - 1:8 \* `WORD\_SIZE];  W[8] = data[8 \* `WORD\_SIZE - 1:7 \* `WORD\_SIZE];  W[9] = data[7 \* `WORD\_SIZE - 1:6 \* `WORD\_SIZE];  W[10] = data[6 \* `WORD\_SIZE - 1:5 \* `WORD\_SIZE];  W[11] = data[5 \* `WORD\_SIZE - 1:4 \* `WORD\_SIZE];  W[12] = data[4 \* `WORD\_SIZE - 1:3 \* `WORD\_SIZE];  W[13] = data[3 \* `WORD\_SIZE - 1:2 \* `WORD\_SIZE];  W[14] = data[2 \* `WORD\_SIZE - 1:1 \* `WORD\_SIZE];  W[15] = data[1 \* `WORD\_SIZE - 1:0 \* `WORD\_SIZE];  a = hash[`WORD\_SIZE \* 8 - 1:`WORD\_SIZE \* 7];  b = hash[`WORD\_SIZE \* 7 - 1:`WORD\_SIZE \* 6];  c = hash[`WORD\_SIZE \* 6 - 1:`WORD\_SIZE \* 5];  d = hash[`WORD\_SIZE \* 5 - 1:`WORD\_SIZE \* 4];  e = hash[`WORD\_SIZE \* 4 - 1:`WORD\_SIZE \* 3];  f = hash[`WORD\_SIZE \* 3 - 1:`WORD\_SIZE \* 2];  g = hash[`WORD\_SIZE \* 2 - 1:`WORD\_SIZE \* 1];  h = hash[`WORD\_SIZE \* 1 - 1:`WORD\_SIZE \* 0];  end  // Compute the hash value  finish: begin  hash[`WORD\_SIZE \* 8 - 1:`WORD\_SIZE \* 7] = a + hash[`WORD\_SIZE \* 8 - 1:`WORD\_SIZE \* 7];  hash[`WORD\_SIZE \* 7 - 1:`WORD\_SIZE \* 6] = b + hash[`WORD\_SIZE \* 7 - 1:`WORD\_SIZE \* 6];  hash[`WORD\_SIZE \* 6 - 1:`WORD\_SIZE \* 5] = c + hash[`WORD\_SIZE \* 6 - 1:`WORD\_SIZE \* 5];  hash[`WORD\_SIZE \* 5 - 1:`WORD\_SIZE \* 4] = d + hash[`WORD\_SIZE \* 5 - 1:`WORD\_SIZE \* 4];  hash[`WORD\_SIZE \* 4 - 1:`WORD\_SIZE \* 3] = e + hash[`WORD\_SIZE \* 4 - 1:`WORD\_SIZE \* 3];  hash[`WORD\_SIZE \* 3 - 1:`WORD\_SIZE \* 2] = f + hash[`WORD\_SIZE \* 3 - 1:`WORD\_SIZE \* 2];  hash[`WORD\_SIZE \* 2 - 1:`WORD\_SIZE \* 1] = g + hash[`WORD\_SIZE \* 2 - 1:`WORD\_SIZE \* 1];  hash[`WORD\_SIZE \* 1 - 1:`WORD\_SIZE \* 0] = h + hash[`WORD\_SIZE \* 1 - 1:`WORD\_SIZE \* 0];  DONE = 1'b1;  end  default: begin  DONE = 1'b0;  end  endcase  end  always @(posedge CLK) begin  if (state == main) begin  // Prepare the message schedule W[16] ~ W[63]  if (round\_counter >= 16) begin  W[round\_counter] = `SSIG1(W[round\_counter - 2]) + W[round\_counter - 7] + `SSIG0(W[round\_counter - 15]) + W[round\_counter - 16];  end  // Perform the main hash computation  T1 = h + `BSIG1(e) + `CH(e, f, g) + K[round\_counter] + W[round\_counter];  T2 = `BSIG0(a) + `MAJ(a, b, c);  h = g;  g = f;  f = e;  e = d + T1;  d = c;  c = b;  b = a;  a = T1 + T2;  round\_counter = round\_counter + 1;  end  end  endmodule |

# Synthesis

## Schematic



## Report timing

텍스트이(가) 표시된 사진

자동 생성된 설명

## Report power

텍스트이(가) 표시된 사진

자동 생성된 설명