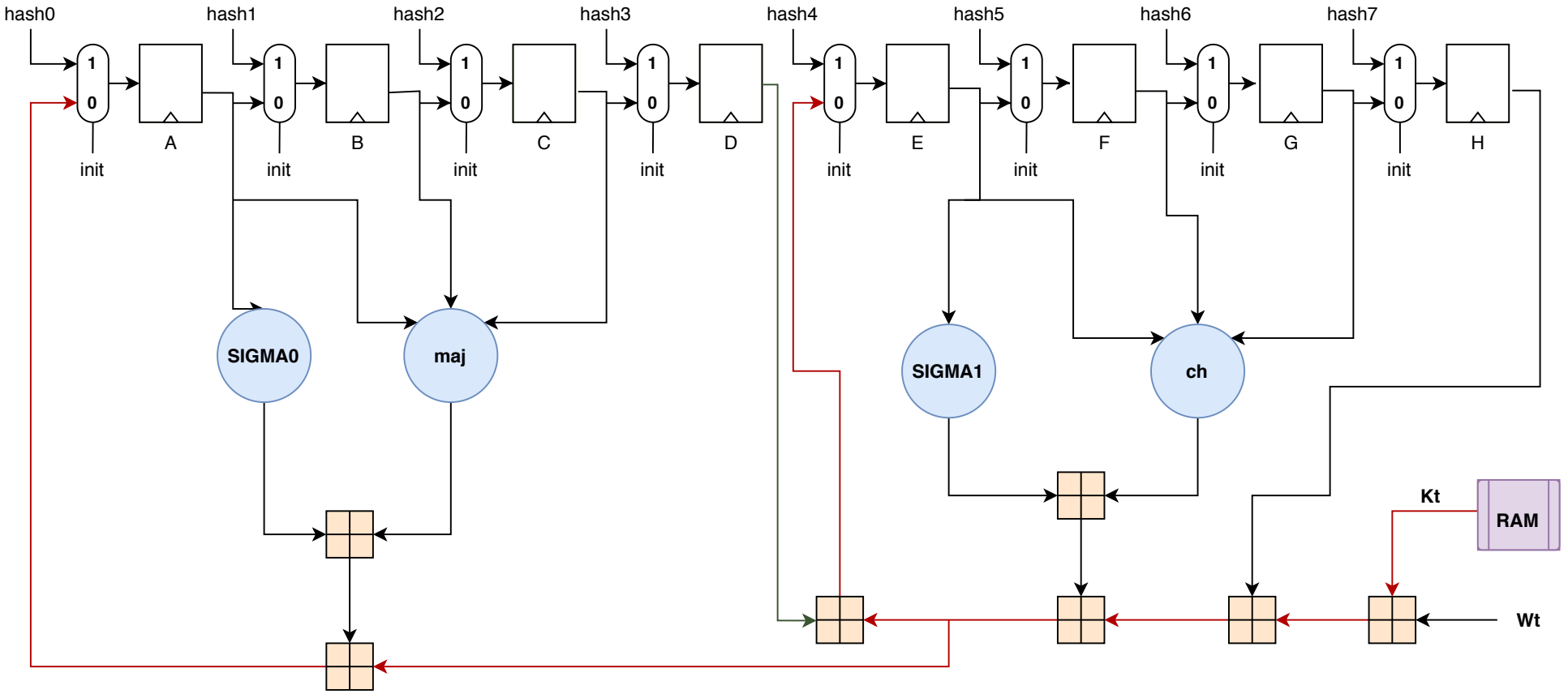
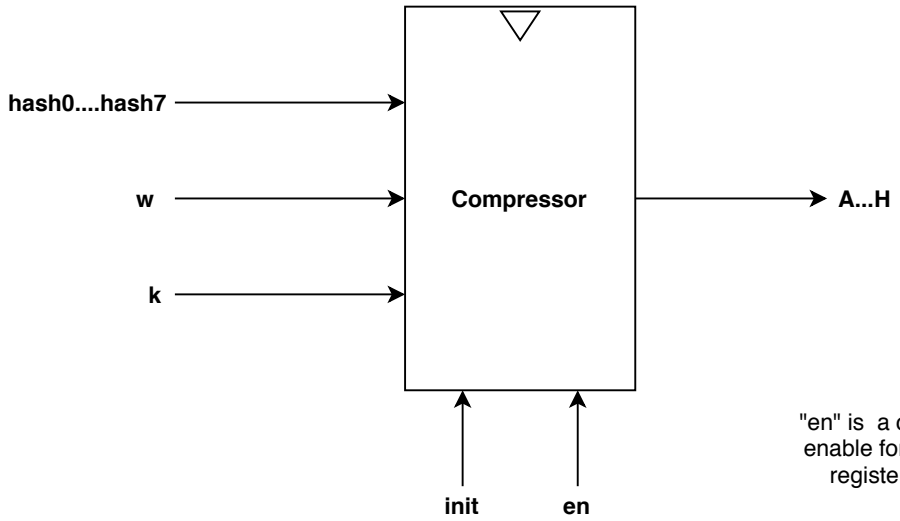
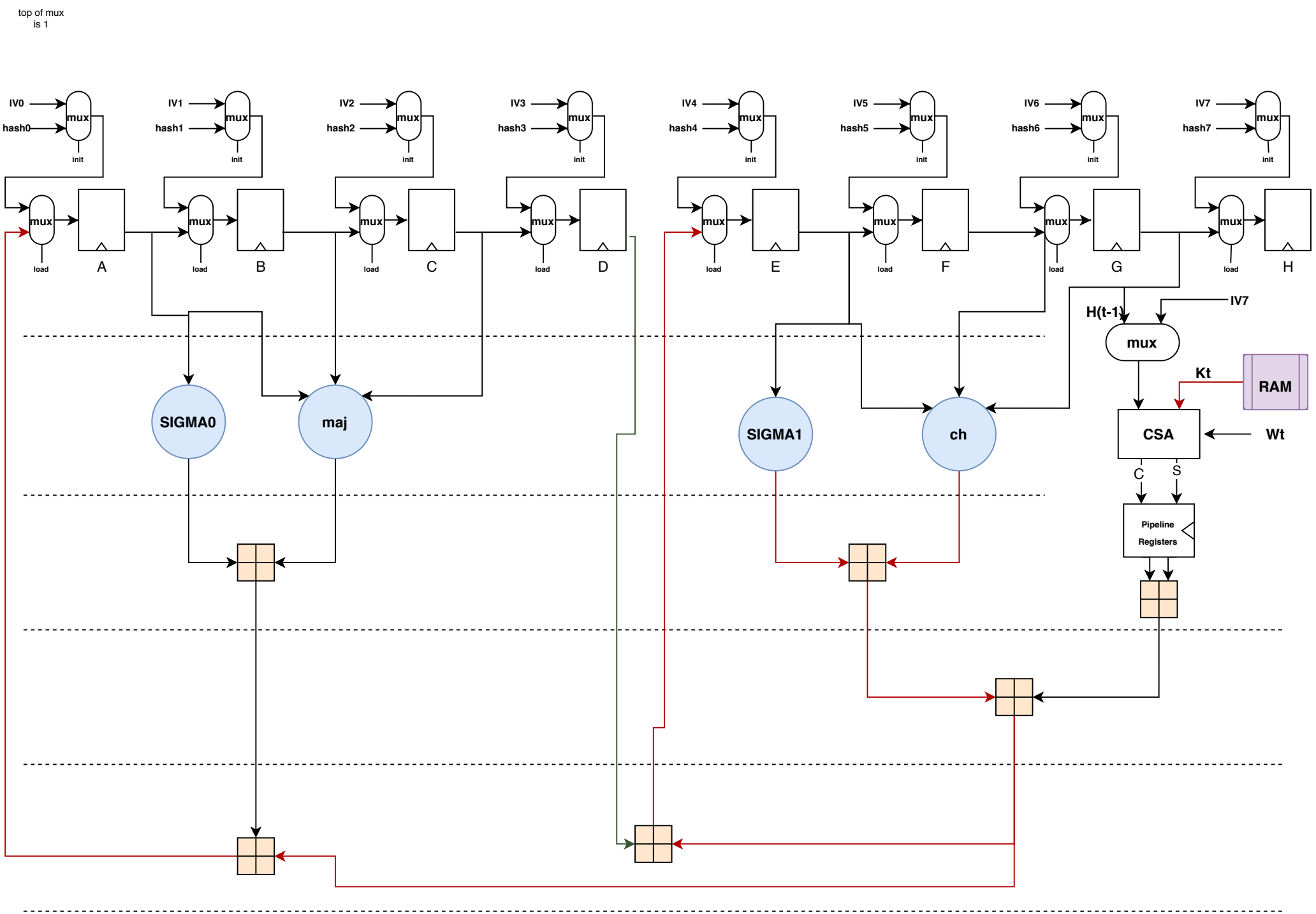


The critical path is (4) adders and (1) RAM access

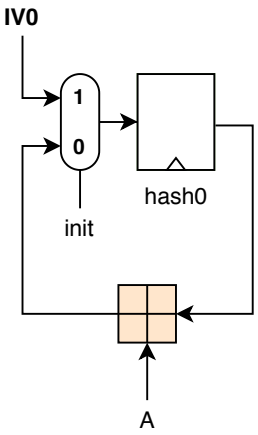
Compressor

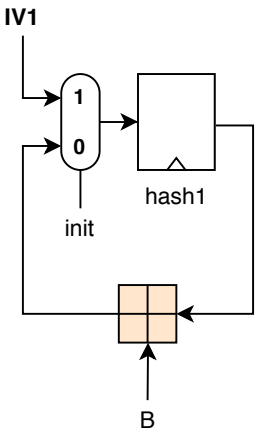


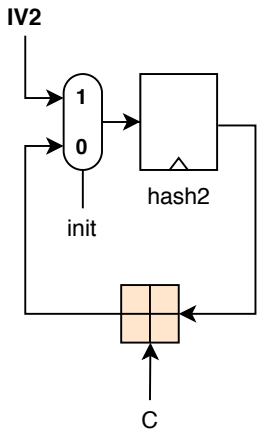
"en" is a clock enable for the registers

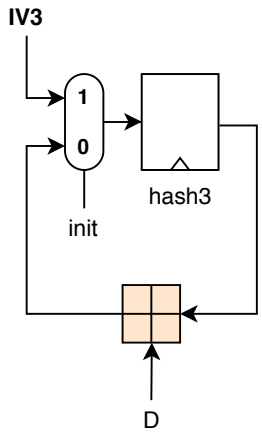


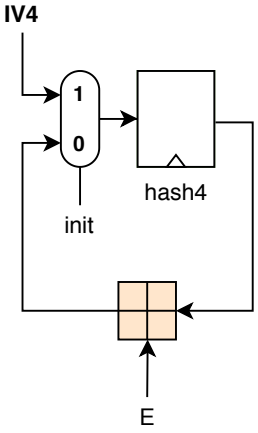
H + Kt + Wt is can be calculated early since the next state of H is G, so a pre-calculation of that sum can be performed. **(1) adder is saved in the critical path** by doing this. This adds additional controls. A mux will need to be added for H(t-1) for when the working registers are first loaded with the initial vector (for any block). The address in RAM will need to properly adjusted.

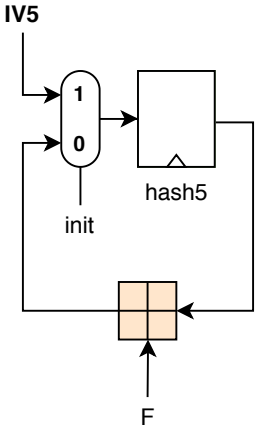


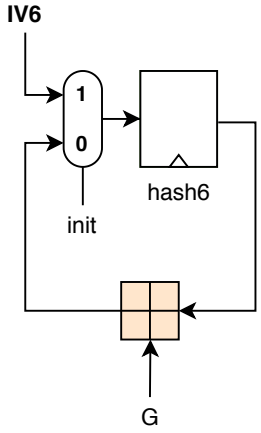


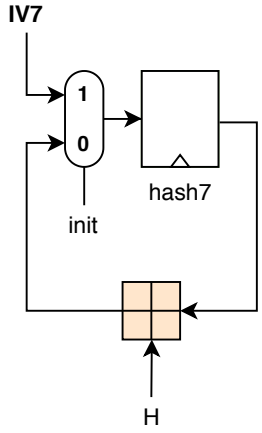












Initial Values

IV0: 0x6A09E667

IV1: 0xBB67AE85

IV2: 0x3C6EF372

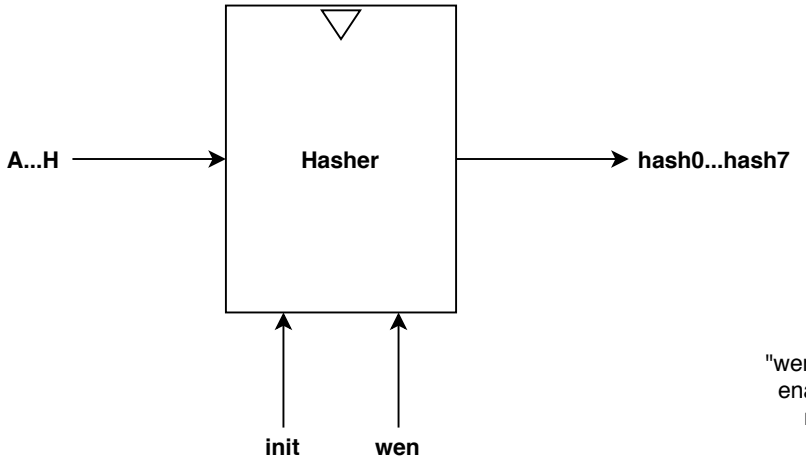
IV3: 0xA54FF53A

IV4: 0x510E527F

IV5: 0x9B05688C

IV6: 0x1F83D9AB

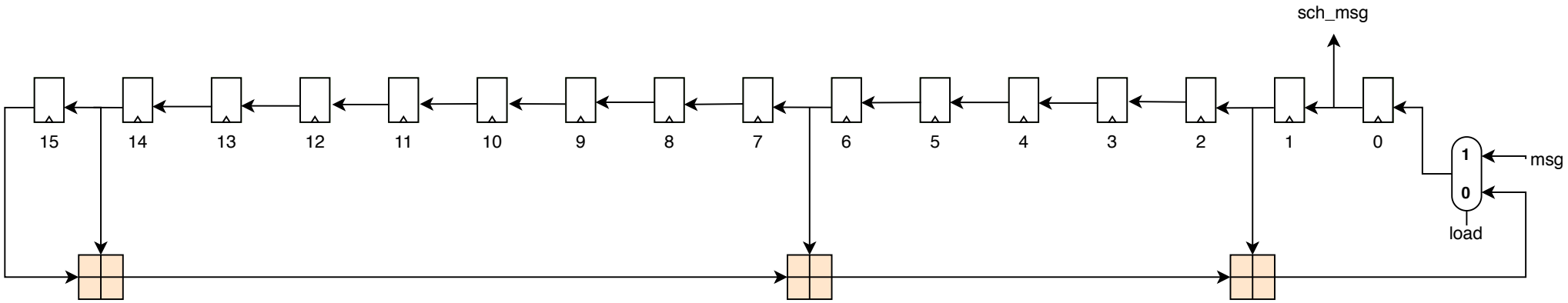
IV7: 0x5BE0CD19

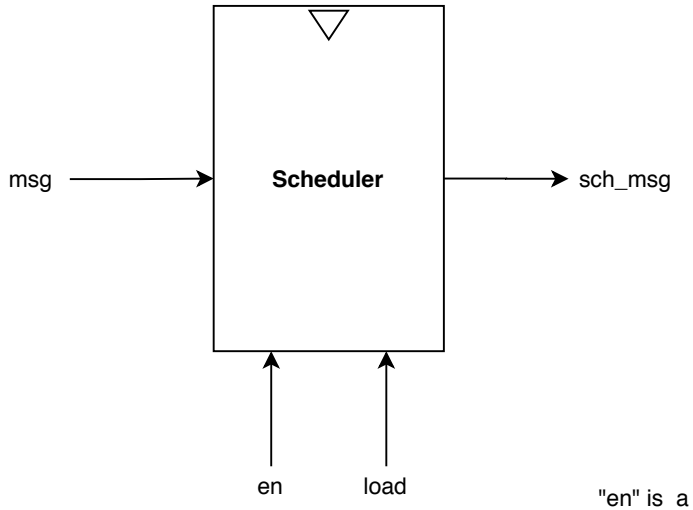


Hasher

"wen" is a clock enable for the registers

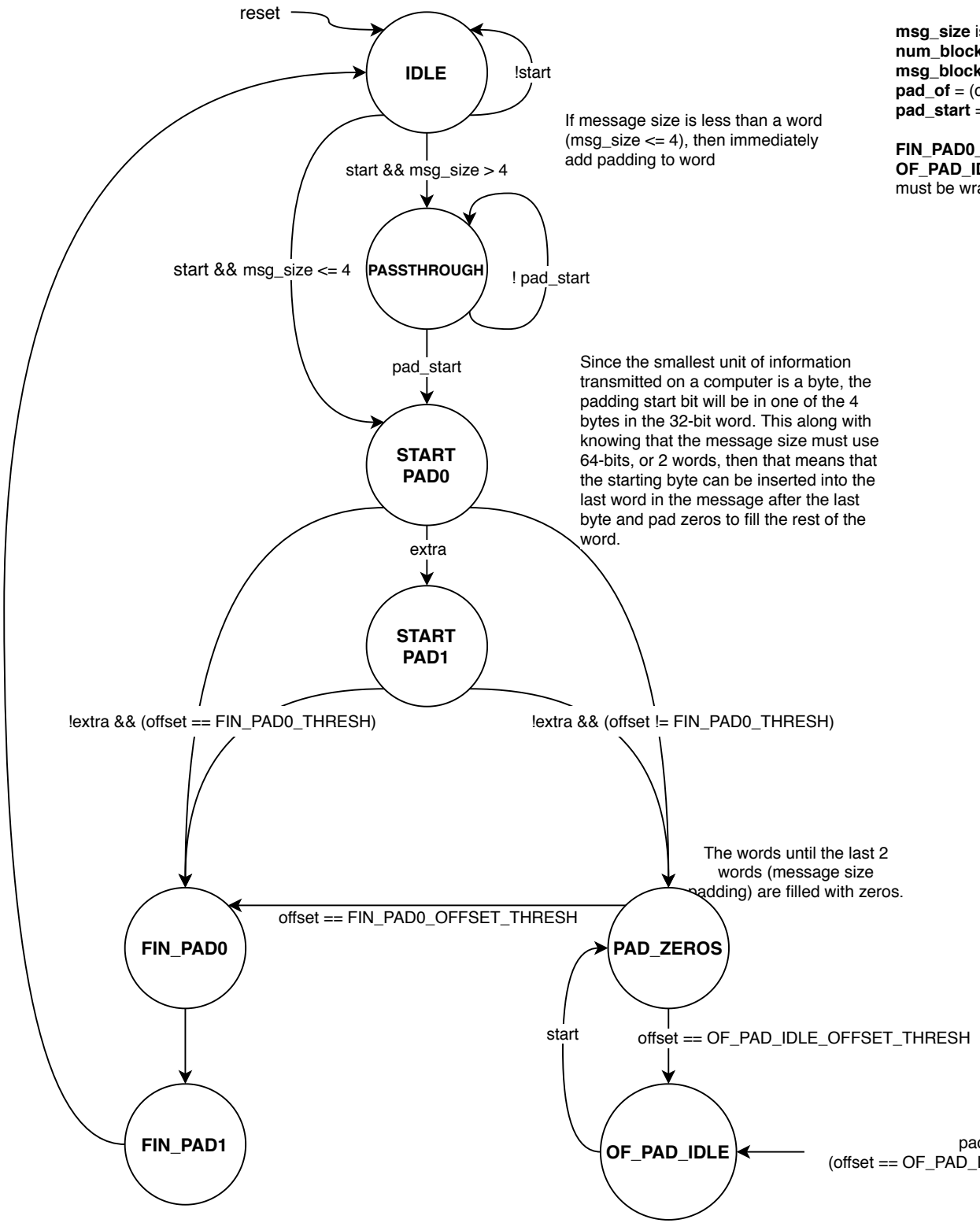
Scheduler





"en" is a clock enable for the registers

Scheduler design citation:
{INSERT}



If message size is less than a word (msg_size <= 4), then immediately add padding to word

Since the smallest unit of information transmitted on a computer is a byte, the padding start bit will be in one of the 4 bytes in the 32-bit word. This along with knowing that the message size must use 64-bits, or 2 words, then that means that the starting byte can be inserted into the last word in the message after the last byte and pad zeros to fill the rest of the word.

The words until the last 2 words (message size padding) are filled with zeros.

msg_size is the number of bytes in total to be hashed
num_blocks = msg_size[63:6]
msg_block_size = msg_size[5:0]
pad_of = (cur_block == num_blocks - 1) && (msg_block_size > 55)
pad_start = {cur_block, offset} = (msg_size - 1) >> 2

FIN_PAD0_OFFSET_THRESH = 13 // last word before msg_size padding
OF_PAD_IDLE_OFFSET_THRESH = 15 // last word in msg reached, so the padding must be wrapped onto an extra msg block

State Descriptions

PASSTHROUGH: Copy input to output (no padding added)

START_PAD0: First padding start state. Adds start byte, 0x80, and zeros to the last word in the message. If the last byte of the message is in the LS byte of the word (the entire word is full of message bytes), then the **extra** signal will go high to add the start byte and zeros on the next word with no message bytes.

START_PAD1: Adds the start byte and zeros to output word (0x80000000).

PAD_ZEROS: Each output word is zero

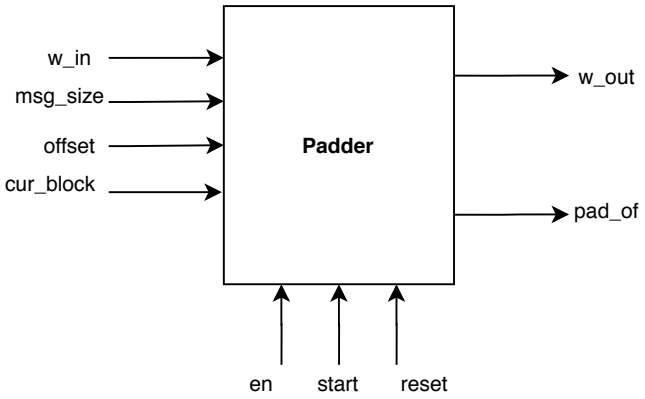
OF_PAD_IDLE: An idle state that is reached if the padding is started but cannot fit in the current message block and must be finished on one more block (OverFlow padding). The idle state waits for the start signal, like the IDLE state, to start the next message block (all padding - zeros and message size).

FIN_PAD0: The output word is the upper 32 bits of the message size.

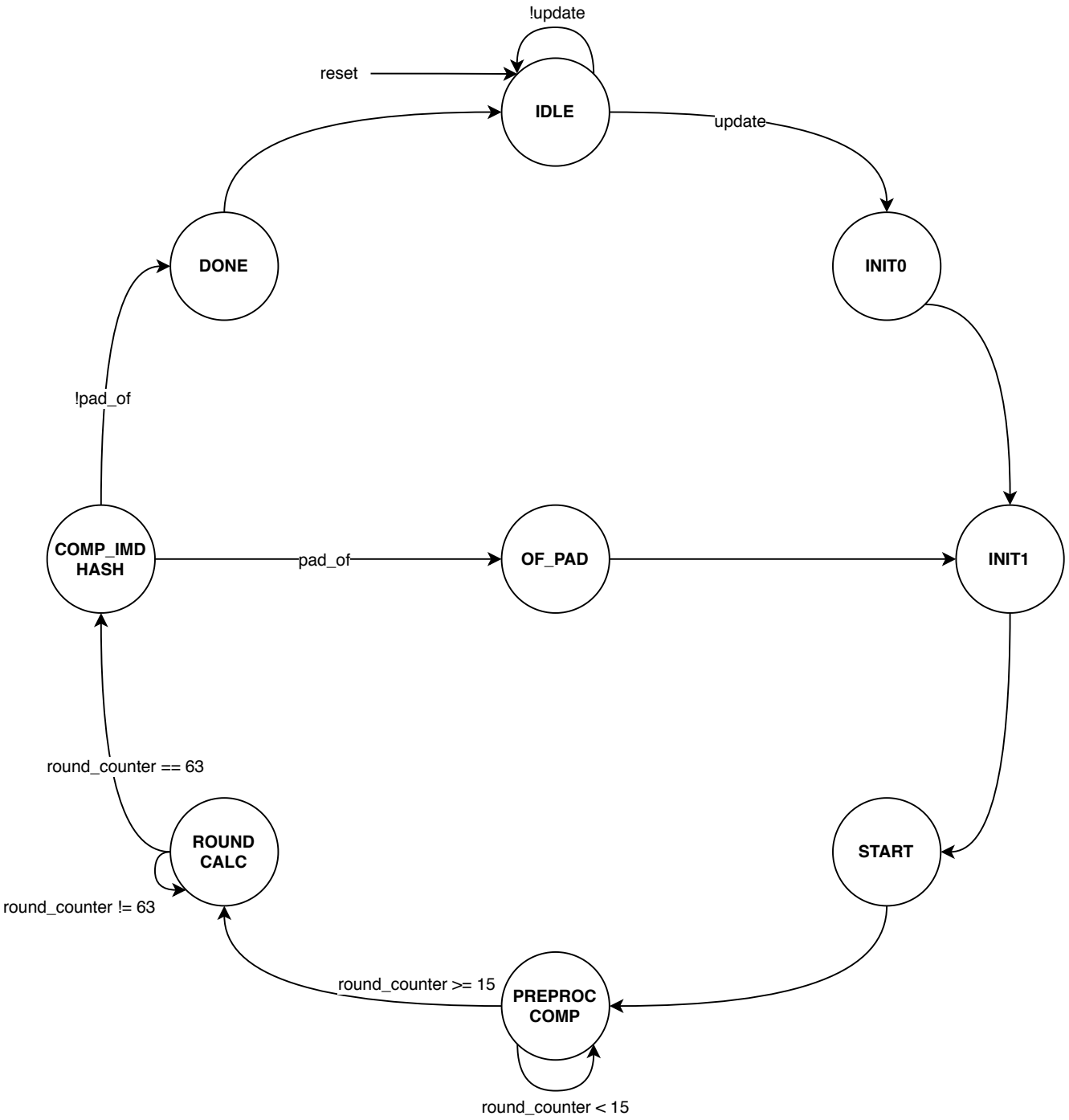
FIN_PAD1: The output word is the lower 32 bits of the message size.

Padder

```
w_out = 0;
extra = 0;
//pad_of = 0;

case (state)
    STATE_IDLE: begin end // nothing

    STATE_OF_PAD_IDLE: begin end

    STATE_PASSTHROUGH:
        w_out = w_in_reg;

    STATE_START_PAD0: begin
        case( msg_size_minus1[1:0] )
            2'b00:
                w_out = {w_in_reg[31:24], 8'h80, 16'd0};

            2'b01:
                w_out = {w_in_reg[31:16], 8'h80, 8'd0};

            2'b10:
                w_out = {w_in_reg[31:8], 8'h80};

            2'b11: begin
                w_out = w_in_reg;
                extra = 1;
            end
        endcase
    end

    STATE_START_PAD1:
        w_out = 32'h80000000;

    STATE_PAD_ZEROS:
        w_out = 0;

    STATE_FIN_PAD0:
        w_out = msg_size_bits[63:32];

    STATE_FIN_PAD1:
        w_out = msg_size_bits[31:0];
endcase
```



SHA256_Update



State Descriptions

Note: Keep in mind that at times the padder, scheduler, and compressor are all working at the same time as a pipeline. After the first 16 words, the scheduler generates the remaining 48 words for the hash computation (i.e. the padder is essentially off and "removed" from the pipeline for the rest of the message block).

INIT0: Since the working registers A...H need to be initialized with either the initial vectors (IVs) defined in the SHA256 specification or the hash values of the previous hash and the compressor module loads its initial hash values from the hasher module, if it is the first block in the message, the hasher needs to be first initialized with the IVs before the compressor module can initialize its working registers.

INIT1: Loads the compressor with initial hash values from the hasher.

START: Starts the padder's FSM, enables the padder, scheduler, compressor, and round counter, and signals the scheduler to load input values into its circular buffer.

PREPROC_COMP: Almost identical to the START state except that the start_pad signal to start the padder's FSM is not high. This state is for loading the scheduler with the first 16 words for pre-processing.

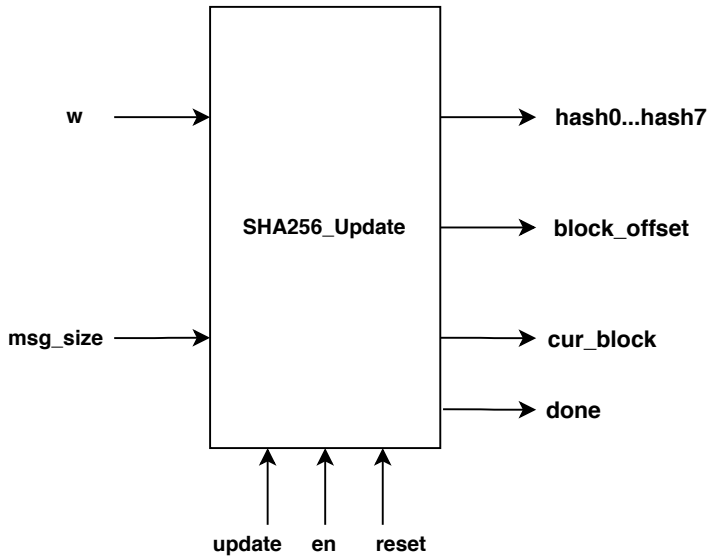
ROUND_CAL: Stops loading words from the padder into the scheduler's circular buffer - all registers in the scheduler are now filled with a padded message block and those 16 words are used in the pre-processing function described in the SHA256 spec to create 64 words.

COMP_IMD_HASH: This state is entered when the compressor has finished its 64 rounds and now the intermediate hash is calculated in the hasher.

OF_PAD: This state is entered when the padder emits an overflow padder signal high indicating that another message block is needed. The sha256_update module will automatically calculate the hash of the overflowed padding block. Increments the current block counter.

DONE: Sets done signal high and increments the current block counter.

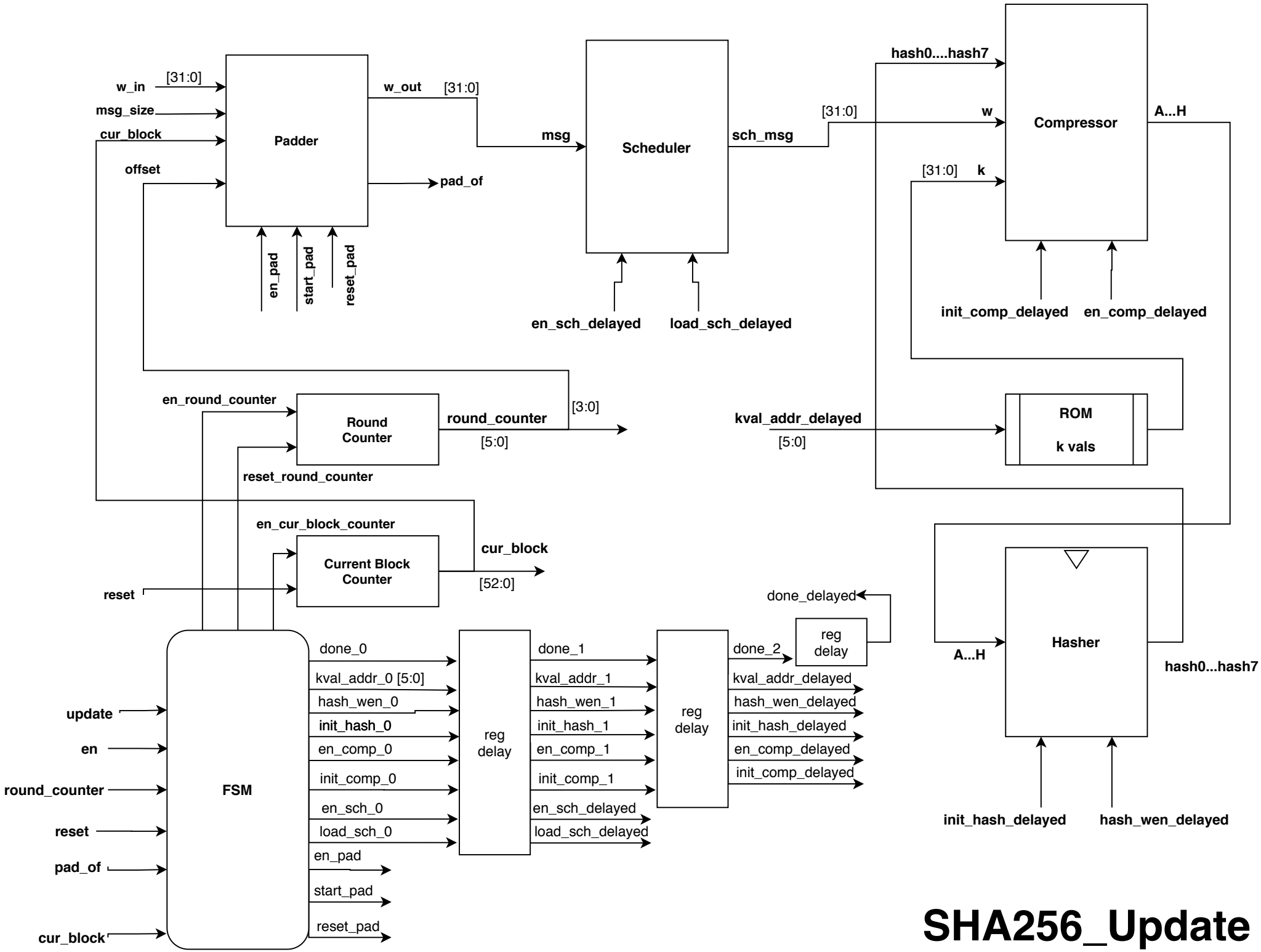
```
case (state)
    STATE_IDLE: begin
        reset_round_counter = 1;
        reset_pad = 1;
    end

    STATE_INIT0: begin
        // load inital hash values if the first block
        if(first_block) begin
            hash_wen = 1;
            init_hash = 1;
        end
    end

    STATE_INIT1: begin
        // load compressor with inital block hash values
        init_comp = 1;
        en_comp = 1;
    end

    STATE_START: begin
        en_pad = 1;
        start_pad = 1;
        en_sch = 1;
        load_sch = 1;
        en_comp = 1;
        en_round_counter = 1;
    end

    STATE_PREPROC_COMP: begin
        en_pad = 1;
        en_sch = 1;
        load_sch = 1;
        en_comp = 1;
        en_round_counter = 1;
    end

    STATE_ROUND_CALC: begin
        en_pad = 1;
        en_sch = 1;
        en_comp = 1;
        en_round_counter = 1;
    end

    STATE_OF_PAD: begin
        cur_block_wen = 1;
    end

    STATE_COMP_IMD_HASH: begin
        hash_wen = 1;
    end

    STATE_DONE: begin
        done_control = 1;
        cur_block_wen = 1;
    end
endcase
```



The sha256_update module requests a word from a message buffer. The sha256_update module keeps track of the current state of the hash (which block of the message and the offset from the current block). The offset is which word of a message block the update module is needing to be hashed. The word address requested from the buffer is as following:

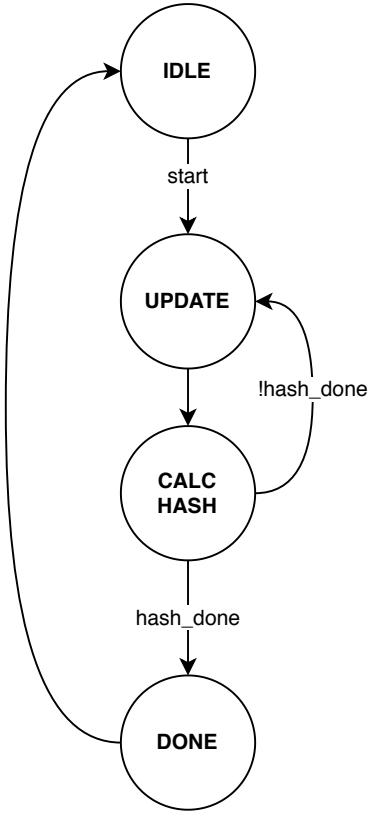
address = {cur_block, offset}

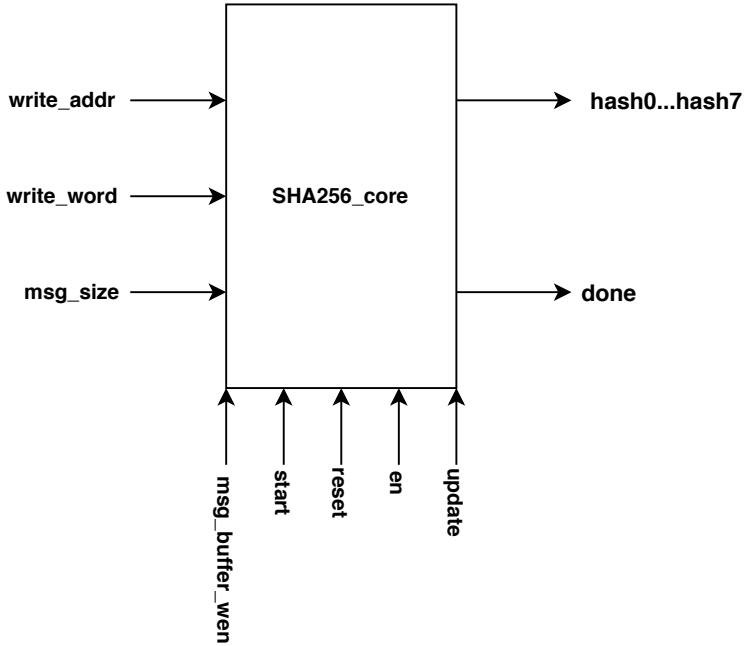
The message buffer is chosen to be word-aligned to minimize memory fetches for bytes when doing byte manipulation for the padding. The padder FSM was designed to handle words and only adds a single cycle delay in calculating the padded word.



SHA256_Update

SHA256_Core






```
case(state)
    STATE_IDLE: begin end

    STATE_CALC_HASH: begin end

    STATE_UPDATE:
        sha256_update = 1;

    STATE_DONE:
        done = 1;
endcase
```

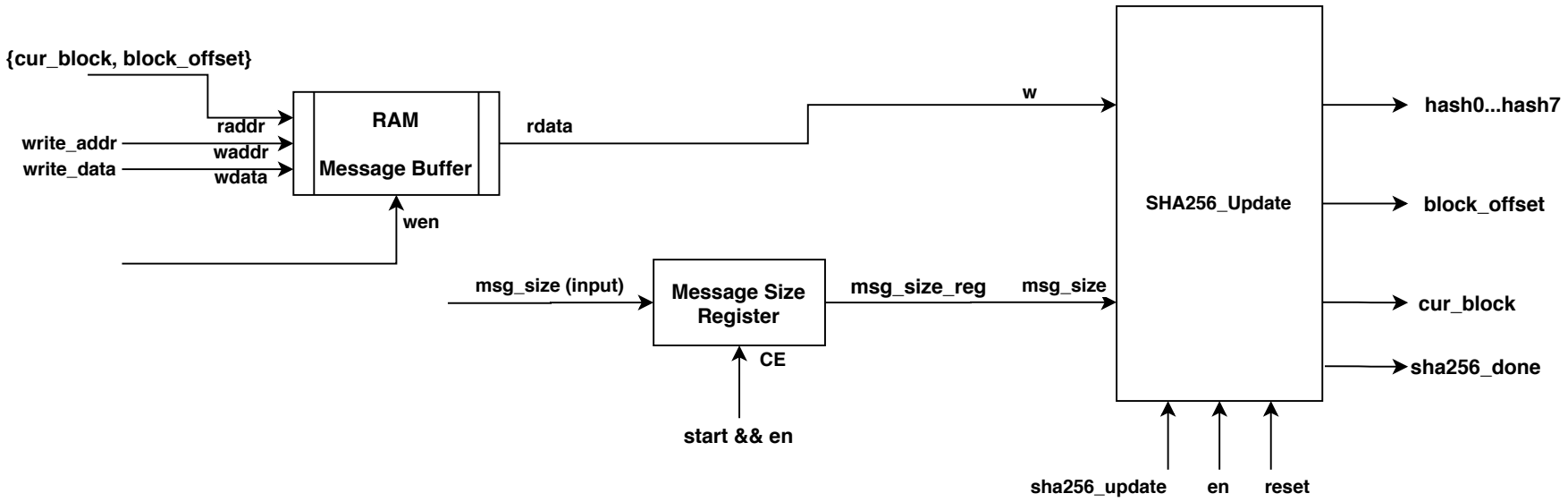
State Descriptions

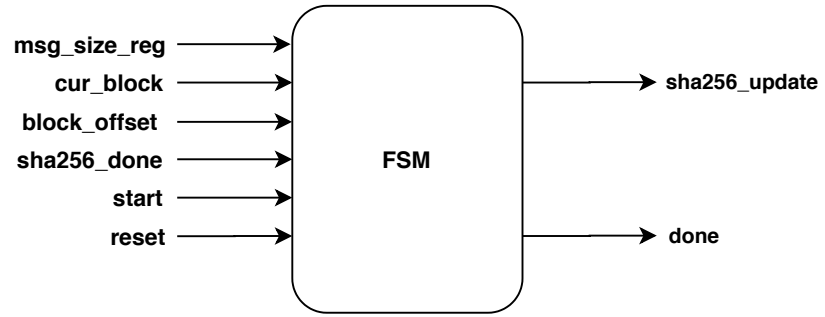
UPDATE: Sends signal to update module to start the intermediate hash.

CALC_HASH: Waits for the update module to finish with the intermediate hash, then either starts another intermediate hash if there are more message blocks to be hashed or marks the end of the hash.

DONE: Completion of the hash

SHA256_Core


{cur_block, block_offset}
raddr
write_addr
waddr
write_data
wdata
RAM
Message Buffer
wen
rdata
w
SHA256_Update
hash0...hash7
block_offset
cur_block
sha256_done
msg_size (input)
Message Size Register
msg_size_reg
msg_size
CE
start && en
sha256_update
en
reset


msg_size_reg
cur_block
block_offset
sha256_done
start
reset
FSM
sha256_update
done