Design of Chip Security - Spring 2024

Lab02-FSM Report

Table of Contents

Name: 王品然

Student ID: 113063572

A. Design Architecture and Performance

1. Algorithm

This project implements the complete SHA-2 256-bit algorithm, including the padding stage. The following procedures are based on the specifications defined in **FIPS 180-4**.

(1) Functions and constants

To begin with, I will introduce the functions and constants required for the algorithm.

(a) Functions:

$$
\begin{aligned}
Ch(x, y, z) &= (x \wedge y) \oplus (\neg x \wedge z) \\
Maj(x, y, z) &= (x \wedge y) \oplus (x \wedge z) \oplus (y \wedge z)
\end{aligned}
$$

$$
\begin{aligned}
\sum_0^{\{256\}}(x) &= ROTR^2(x) \oplus ROTR^{13}(x) \oplus ROTR^{22}(x) \\
\sum_1^{\{256\}}(x) &= ROTR^6(x) \oplus ROTR^{11}(x) \oplus ROTR^{25}(x) \\
\sigma_0^{\{256\}}(x) &= ROTR^7(x) \oplus ROTR^{18}(x) \oplus SHR^3(x) \\
\sigma_1^{\{256\}}(x) &= ROTR^{17}(x) \oplus ROTR^{19}(x) \oplus SHR^{10}(x)
\end{aligned}
$$

For simplicity, the functions $\Sigma_0^{\{256\}}(x)$, $\Sigma_1^{\{256\}}(x)$, $\sigma_0^{\{256\}}(x)$, and $\sigma_1^{\{256\}}(x)$ will be denoted as $\Sigma_0(x)$, $\Sigma_1(x)$, $\sigma_0(x)$, and $\sigma_1(x)$, respectively, throughout this report.

(b) Constants:

```
428a2f98 71374491 b5c0fbcf e9b5dba5 3956c25b 59f111f1 923f82a4 ab1c5ed5
d807aa98 12835b01 243185be 550c7dc3 72be5d74 80deb1fe 9bdc06a7 c19bf174
e49b69c1 efbe4786 0fc19dc6 240ca1cc 2de92c6f 4a7484aa 5cb0a9dc 76f988da
983e5152 a831c66d b00327c8 bf597fc7 c6e00bf3 d5a79147 06ca6351 14292967
27b70a85 2e1b2138 4d2c6dfc 53380d13 650a7354 766a0abb 81c2c92e 92722c85
a2bfe8a1 a81a664b c24b8b70 c76c51a3 d192e819 d6990624 f40e3585 106aa070
19a4c116 1e376c08 2748774c 34b0bcb5 391c0cb3 4ed8aa4a 5b9cca4f 682e6ff3
748f82ee 78a5636f 84c87814 8cc70208 90befffa a4506ceb bef9a3f7 c67178f2
```

This is a sequence of 64 constant 32-bit words, $K_0^{\{256\}}$, $K_1^{\{256\}}$,…,$K_{63}^{\{256\}}$. For simplicity, I will refer to them as K_0, K_1,…, K_{63} in this report.

(2) Preprocessing

Before applying SHA, we must first perform the following preprocessing steps: padding the message, parsing the message and setting the initial hash value.

(a) Padding the message:

The purpose of this padding is to ensure that the padded message is a multiple of 512 bits. Suppose that the length of the message, M, is ℓ bits. Append a single "1" bit to the end of the message, followed by $\mathcal{k}$ zero bits, where $\mathcal{k}$ is the smallest, non-negative solution to the equation $\ell + 1 + \mathcal{k} \equiv 448 \ mod \ 512$. Then append the 64-bit block that is equal to the number ℓ expressed using a binary representation. An example is shown below:

$$\underbrace{01100001}_{\text{“a”}}\ \underbrace{01100010}_{\text{“b”}}\ \underbrace{01100011}_{\text{“c”}}\ 1\ \overbrace{00\ldots00}^{423}\ \overbrace{00\ldots011000}^{64}\ (\ell = 24)$$

The length of the padded message should now be a multiple of 512 bits.

(b) Parsing the message:

The padded message is divided into N 512-bit blocks, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$. Since the 512 bits of the input block may be expressed as sixteen 32-bit words, the first 32 bits of message block i are denoted $M_0^{(i)}$, the next 32 bits are $M_1^{(i)}$, and so on, up to $M_{15}^{(i)}$.

(c) Setting the initial hash value:

For SHA-256, the initial hash value, $H^{(0)}$, shall consist of the following eight 32-bit words, in hex:

$$H_0^{(0)} = 6a09e667 \qquad H_4^{(0)} = 510e527f$$
$$H_1^{(0)} = bb67ae85 \qquad H_5^{(0)} = 9b05688c$$
$$H_2^{(0)} = 3c6ef372 \qquad H_6^{(0)} = 1f83d9ab$$
$$H_3^{(0)} = a54ff53a \qquad H_7^{(0)} = 5be0cd19$$

(3) Computation

Each message block, $M^{(1)}$, $M^{(2)}$,…, $M^{(N)}$, is processed in order, using the following steps:

For *i = 1 to N*, repeat:

(a) Prepare the message schedule $\{W_t\}$:

$$W_t = \begin{cases} M_t^{(i)} & 0 \le t \le 15 \\ \sigma_1^{\{256\}}(W_{t-2}) + W_{t-7} + \sigma_0^{\{256\}}(W_{t-15}) + W_{t-16} & 16 \le t \le 63 \end{cases}$$

(b) Initialize the eight working variables, **a**, **b**, **c**, **d**, **e**, **f**, **g**, and **h**, with the $(i-1)^{st}$ hash value:

$$\begin{aligned} a &= H_0^{(i-1)} & e &= H_4^{(i-1)} \\ b &= H_1^{(i-1)} & f &= H_5^{(i-1)} \\ c &= H_2^{(i-1)} & g &= H_6^{(i-1)} \\ d &= H_3^{(i-1)} & h &= H_7^{(i-1)} \end{aligned}$$

(c) For *t = 0 to 63* , repeat:

$$\begin{aligned} T_1 &= h + \sum_1^{\{256\}}(e) + Ch(e, f, g) + K_t^{\{256\}} + W_t & e &= d + T_1 \\ T_2 &= \sum_0^{\{256\}}(a) + Maj(a, b, c) & d &= c \\ h &= g & c &= b \\ g &= f & b &= a \\ f &= e & a &= T_1 + T_2 \end{aligned}$$

(d) Compute the i^{th} intermediate hash value $H^{(i)}$:

$$\begin{aligned} H_0^{(i)} &= a + H_0^{(i-1)} & H_4^{(i)} &= e + H_4^{(i-1)} \\ H_1^{(i)} &= b + H_1^{(i-1)} & H_5^{(i)} &= f + H_5^{(i-1)} \\ H_2^{(i)} &= c + H_2^{(i-1)} & H_6^{(i)} &= g + H_6^{(i-1)} \\ H_3^{(i)} &= d + H_3^{(i-1)} & H_7^{(i)} &= h + H_7^{(i-1)} \end{aligned}$$

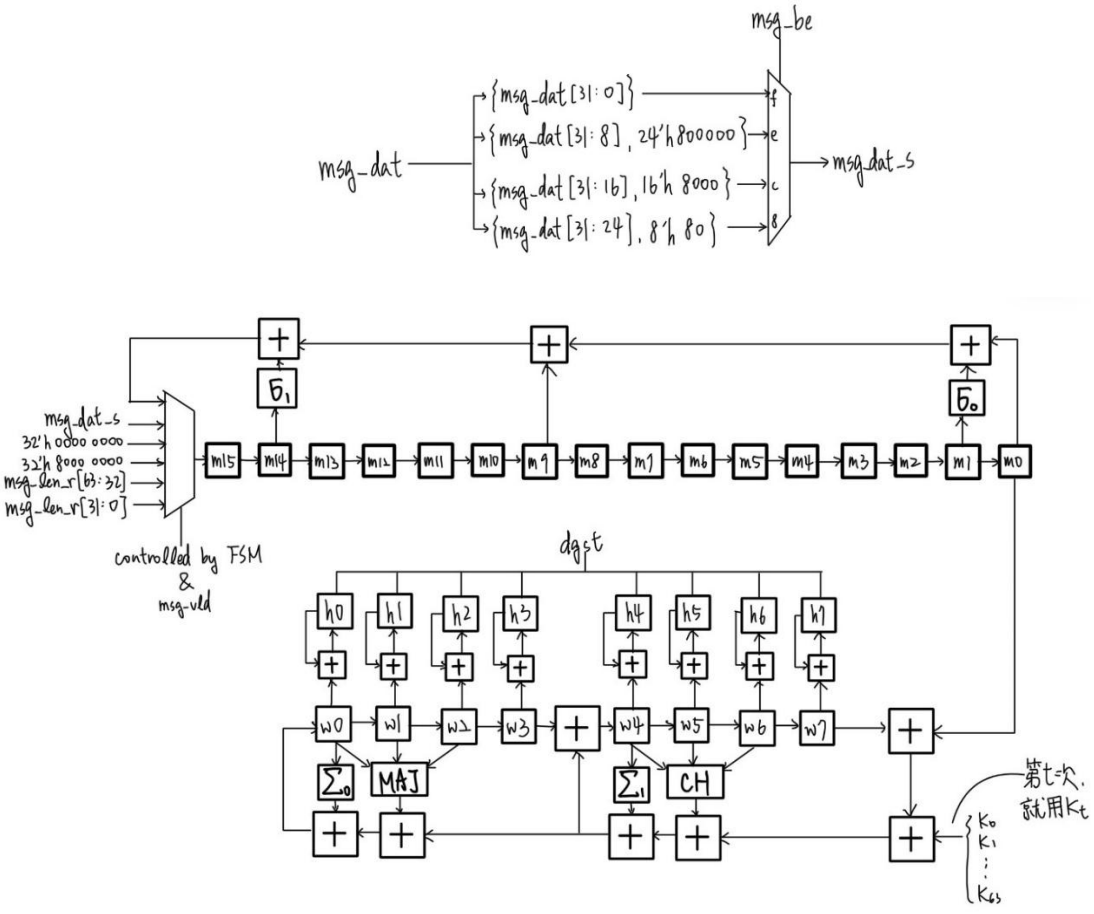
After repeating steps (a) through (d) for all N message blocks, the resulting 256-bit message digest $H^{(N)}$ of M is obtained by concatenating the eight 32-bit intermediate hash values from the N^{th} iteration as follow:

$$H_0^{(N)} \| H_1^{(N)} \| H_2^{(N)} \| H_3^{(N)} \| H_4^{(N)} \| H_5^{(N)} \| H_6^{(N)} \| H_7^{(N)}$$

2. Implementation

(1) Block diagram

Based on the algorithm described above, the overall architecture is illustrated in the following block diagram.



Notations:

(a) m[0~15]:

Sixteen 32-bit registers for the message schedule W_0 to W_{63}.

(b) h[0~7]:

Eight 32-bit registers for the intermediate hash value $H_0^{(i)}$ to $H_7^{(i)}$.

(c) w[0~7]:

Eight 32-bit registers for the work variables **a** to **h**.

(d) msg_len_r:

A 64-bit register for storing the length of every message.

(e) CH, MAJ, Σ_0, Σ_1, σ_0, and σ_1:

Functions defined in the algorithm.

(f) K_0, K_1,…, K_{63}:

Constants required in the algorithm.

(g) +:

Modular addition (modulo 2^{32})

(h) ⟶:

32-bit wide wire; the direction of the arrow indicates the data flow.

(i) msg_dat_s:

The masked version of msg_dat, generated based on the value of msg_be.

Explanation:

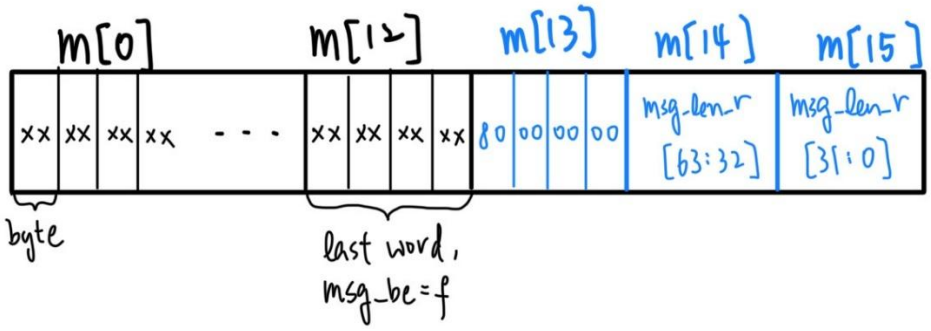
首先在 msg_dat 進來時先做 masking 的動作產生 msg_dat_s，使得 msg_dat_s 只留下 msg_dat 中 enable 的 byte，並且對於 msg_be 是 8, c, or e(同時代表這一定是 last word)的情況，做 padding 步驟中 append a single "1" bit 的動作，上述動作在 verilog 中我使用 case 配上 concatenation 實作。由 m[15]接收 msg_dat_s，並且使 m[0:15]這 16 個 register 開始 shift，直到收滿 16 個 words(需要 counter)或是收到了 last word，若是收滿了 16 個 words 且還未收到 last word，則直接開始 message schedule (參考 1. (3) (a))

和 work variables 的 iteration (參考 1. (3) (c)) 共 64 次(需要 counter，可與前面的 counter 合用)，並將疊代出來的 work variables 加上上一個 message block 產生的 intermediate hash value，得到這個 message block 的 intermediate hash value，接下來再次開始接收 msg_dat。若是收到了 last word，則有下列幾種可能：

(a) 收到 last word 時，m[0:15]已收到少於 13 個(含)words，且

1. msg_be = f

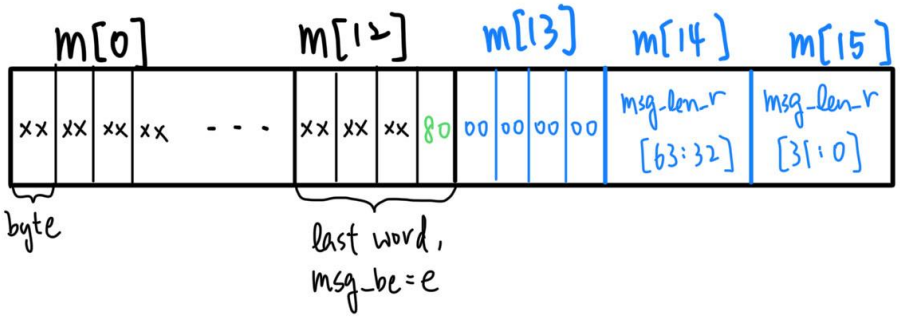
因為在 masking 的步驟中，msg_be = f 的情況沒有 append "1"上去，因此在下一步要將 32'h8000_0000 塞入 m[15]，接著再塞入 32'h0000_0000，直到這個 message block 已存有 14 個 word 為止，而剩下最後兩個 padding word 分別要填入 msg_len_r[63:32]以及 msg_len_r[31:0]。接下來再開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。操作結果的 message block 如附圖：



其中黑筆的部分是 input msg_dat 的 data，x 代表 don't care，數字皆為 hexadecimal，藍筆部分是額外需要填入的部分。

2. msg_be = 8, c, or e

因為在 masking 的步驟中，msg_be = 8, c, or e 的情況已經 append "1"上去，因此下一步只要塞入 32'h0000_0000 到 m[15]，直到這個 message block 已存有 14 個 word 為止，而剩下最後兩個 padding word 分別要填入 msg_len_r[63:32]以及 msg_len_r[31:0]。接下來再開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。操作結果的 message block 如附圖：

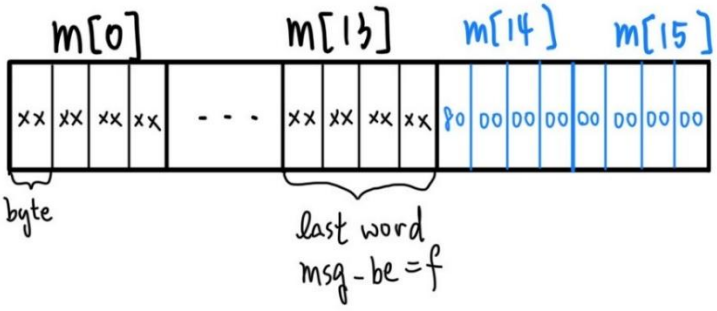


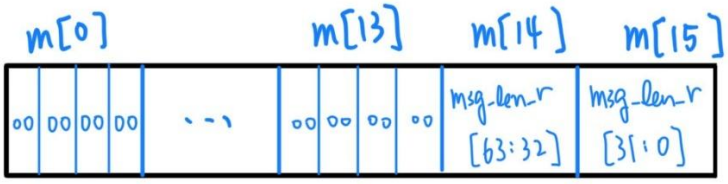
其中綠筆表示非原始 msg_dat 的 data，而是在 masking 時加上去的。

(b) 收到 last word 時，m[0:15]已收到 14 個 words，且

1. msg_be = f

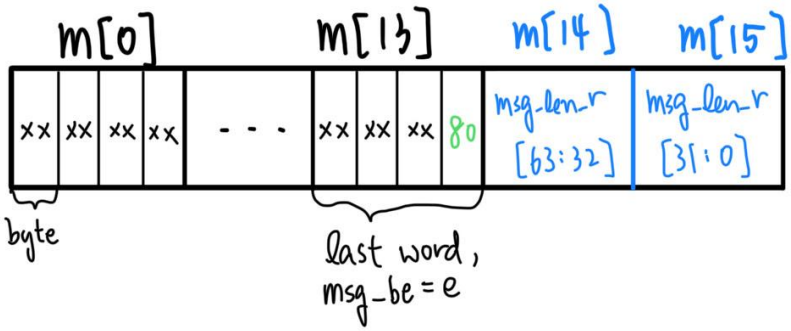
與(a) 1.的情況相似，但已經無法在塞入 32'h8000_0000 後，繼續放入 msg_len_r[63:0]，因為已經只剩一個 word 的空間了，因此接著再塞入一個 32'h0000_0000，完成這個 message block，並且開始 message schedule 和 work variables 的疊代與 intermediate hash value 的計算。之後再來 padding 最後一組 message block 進去 m[0:15]中，此時的 m[0:13]皆為 0，而 m[14:15]存放 msg_len_r[63:0]。接下來再開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。結果分別如附圖的兩個 message block：





2. msg_be = 8, c, or e

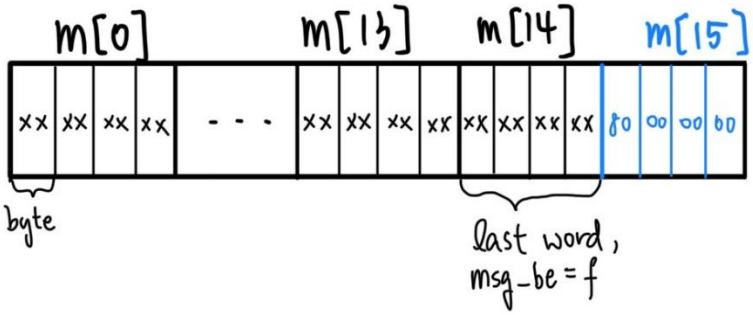
與(a) 2.的情況極為相似，只是此時不用在補 0 上去，因為只剩最後兩個 word 的空間，直接塞入 msg_len_r[63:0]即可開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。最終 message block 如附圖：

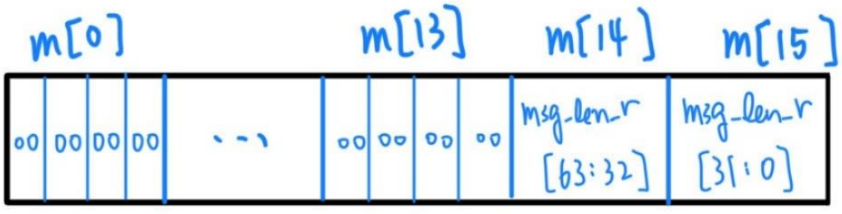


(c) 收到 last word 時，m[0:15]已收到 15 個 words，且

1. msg_be = f

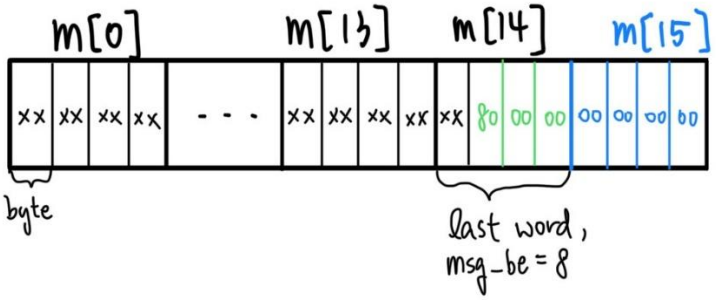
與(b) 1.時的情況極為相似，不再贅述。結果如附圖的兩個 message block

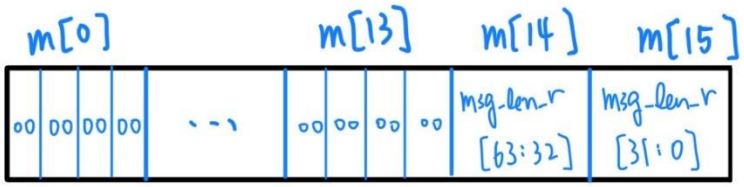




2. msg_be = 8, c, or e

此時的 message block 已存有 15 個 word，因此不足以填入 msg_len_r[63:0]，所以先填入一個 32'h0000_0000 完成這個 message block，並開始 message schedule 和 work variables 的疊代與 intermediate hash value 的計算。之後再來 padding 最後一組 message block 進去 m[0:15]中，此時的 m[13:0]皆為 0，而 m[14:15]存放 msg_len_r[63:0]。接下來再開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。結果分別如附圖的兩個 message block：

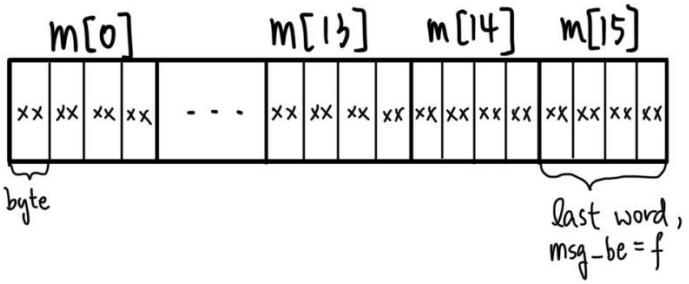


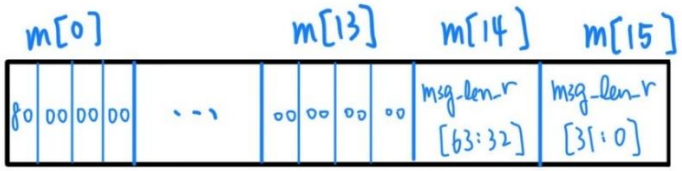


(d) 收到 last word 時，m[0:15]剛好收到第 16 個 word，且

1. msg_be = f

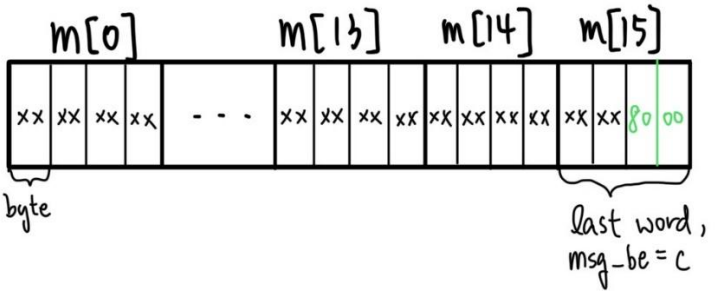
此時的 message block 已有 16 個 word，因此直接開始 message schedule 和 work variables 的疊代與 intermediate hash value 的計算。之後再來 padding 最後一組 message block 進去 m[0:15]中，因為尚未 append "1"，所以此時 m[0]填入 32'h8000_0000 而 m[13:1]皆為 0，m[14:15]存放 msg_len_r[63:0]。接下來再開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。結果分別如附圖的兩個 message block：

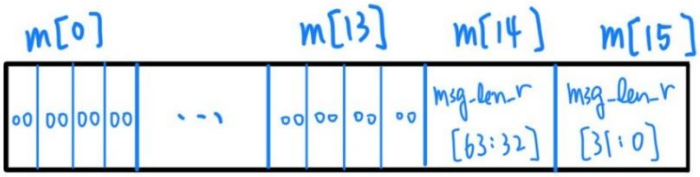




2. msg_be = 8, c, or e

此時的 message block 已有 16 個 word，因此直接開始 message schedule 和 work variables 的疊代與 intermediate hash value 的計算。之後再來 padding 最後一組 message block 進去 m[0:15]中，其中 m[13:0]皆為 0，而 m[14:15]存放 msg_len_r[63:0]。接下來再開始 message schedule 和 work variables 的 iteration，並計算最終的 hash value。

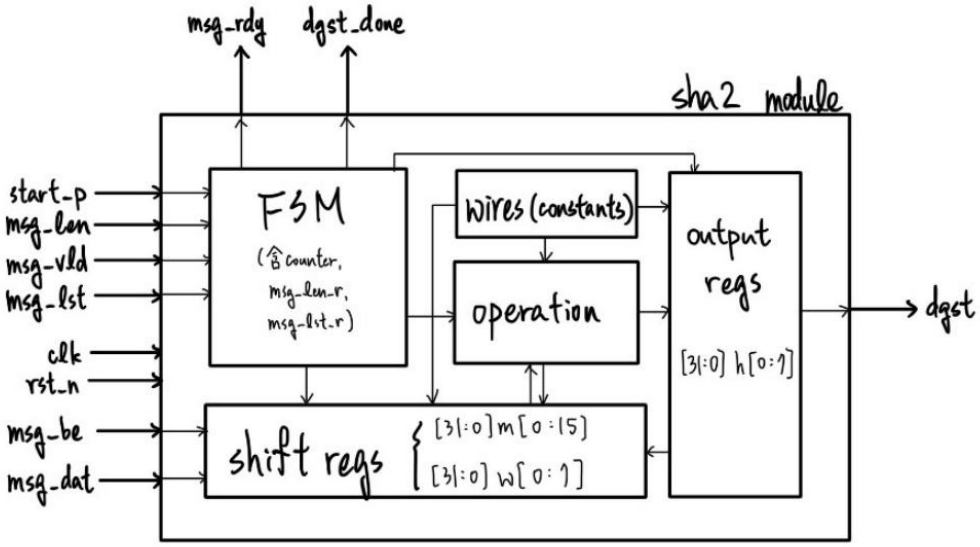




最後還有一種極端狀況沒有考慮到：當 msg_len=0 時，不會有任何 message 進來，因此需要直接填入 512'b10…0 到 message block。以上便是對於 block diagram 的詳細解釋，可以看到目前還缺乏 flow control 以及控制 I/O protocol 的部分，將在接下來 module structure 和 FSM 中繼續完善。

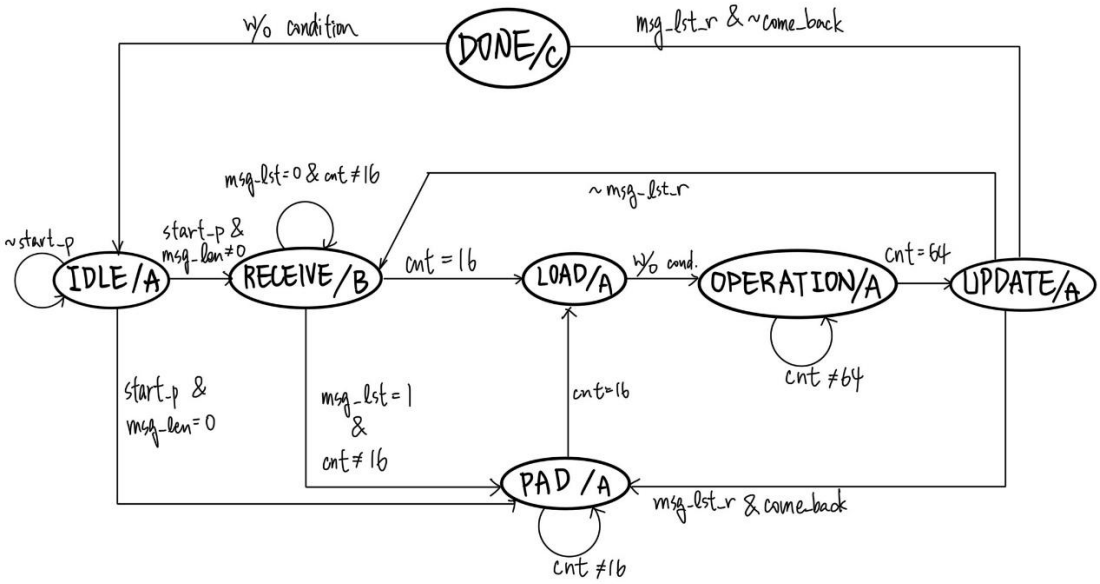
(2) sha2 module structure

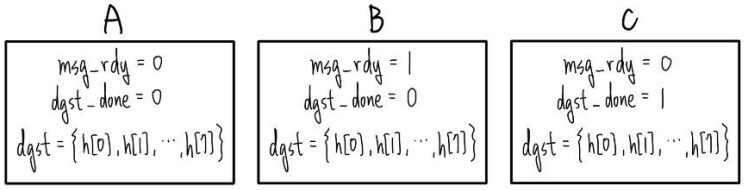
Based on the block diagram, I have constructed a high-level structural overview of the SHA-2 module:



(3) Control logic (FSM)

Based on the block diagram and overall structure, I have designed the following FSM to serve as the control unit.





Notation:

(a) msg_lst_r:

A 1-bit register used to store the value of msg_lst, indicating whether the last word of the message is received.

(b) cnt:

A 6-bit register used to indicate the number of received words, padded words, or iteration counts during the hashing process.

(c) come_back:

A 1-bit register that determines whether the next state after UPDATE should return to PAD or transition to DONE, depending on whether padding is required after the last message word has been received.

Explanation:

這個 FSM 的起始狀態在 IDLE，state transition 還有 output 如上圖所示，首先簡介每個 state 在做的事：

IDLE：等待 start_p 來臨，初始化 h[0:7]，並依據 msg_len 判斷接下來是要接收 msg_dat 或是直接開始 padding

RECEIVE：拉高 msg_rdy，接收 msg_dat，每接收滿 16 個 word(也就是收滿一個 message block)，就進到 LOAD，若收到最後一個 word 卻沒滿 16 個 word，就要去 padding。

PAD：依據不同情況 ，填入不同的 data 到 message block 中，直到填滿並進到 LOAD。

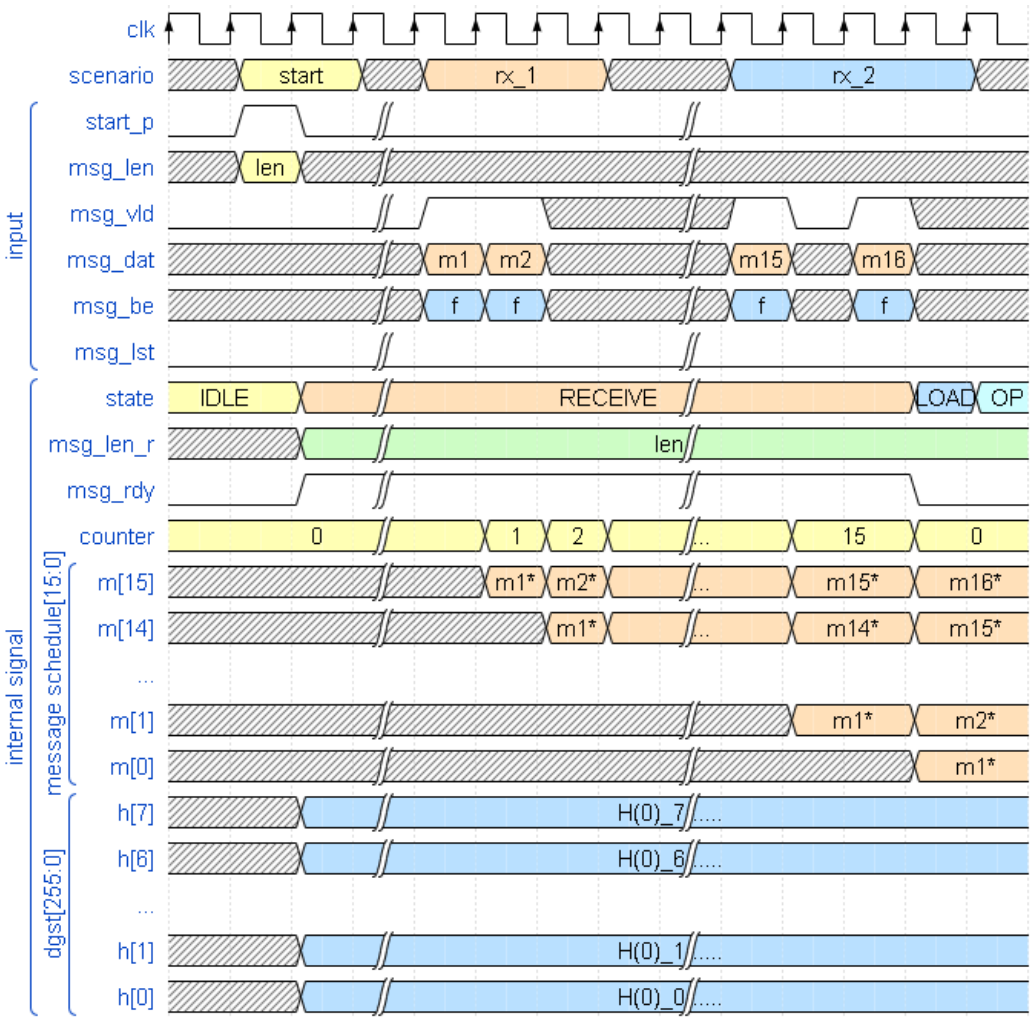
LOAD：將現在 h[0:7]暫存的 initial or intermediate hash value 存入 work variables w[0:7]。

OPERATION：做 message schedule 產生W_0~W_{63}，並不斷更新 work variables w[0:7]。

UPDATE：將本次 message block 經過疊代運算後得到的 w[0:7]與上一次的 intermediate (or initial) hash value (存在 h[0:7])相加，並存入 h[0:7]。如果 last word 還沒來，則回到 RECEIVE；如果 last word 來了，但還需要再 padding 一次則回到 PAD；如果 last word 來了，且也不需要再回到 PAD，則到 DONE。

DONE：拉高 dgst_done，表示完成 message 的 digest。緊接著回到 IDLE。

詳細運作過程我將分成數個 scenario 並搭配 waveform 做說明：



(以下標題為圖中的 scenario name)

(a) start

一個新的 message 要準備進到 design 裡面計算 SHA. 可能進入到這個 stage 的狀況是:

a) reset 放開後
b) 前一個 message 的 digest 已算完，output 出去了

這兩個其況下，FSM 都會在 IDLE，這個 state 是用來等待 start_p 的到來，並在 start_p=1 時

1. 給予 h[0:7]演算法所規定的初始值$H_7^{(0)}$ to $H_0^{(0)}$(即為 waveform 中的

H(0)_7 to H(0)_0)

2. 接收 msg_len 到 msg_len_r 中暫存
3. 根據 msg_len 判斷 next state，若 msg_len≠0，則 next state 為 RECEIVE；若 msg_len=0，在之後的 pad_4 scenario 會介紹。

進入 RECEIVE stage，design 會拉高 msg_rdy 持續等待 msg_vld，細部行為可以區分成以下 10 個 scenario (rx_1~10) 進行討論：

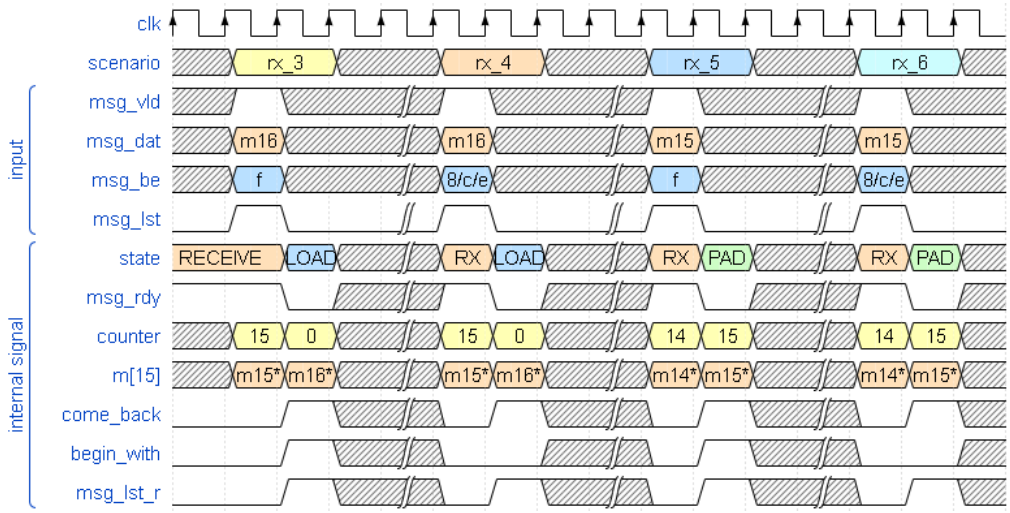
(b) rx_1

msg_vld 連續拉高，cnt 還沒數到 16。

1. 接收 msg_dat，並做 block diagram 介紹的 masking 後成為 msg_dat_s 並填入 m[15]暫存，waveform 中的 m1 代表傳進來的是本次 message block 的第一個 word，m1*代表 m1 經過 masking 後的 data。
2. 將 m[i+1]的 data 填入 m[i] ($0 \leq i \leq 14$)
3. counter+1

(c) rx_2

有別於 rx_1 連續收兩個 message，rx_2 展示了 msg_vld 是不連續時的情況，以及當確定會收滿 16 個 word 時 (cnt=15 且 input handshake 成功時)，則判斷 next state 為 LOAD，並清空 cnt。



Notation:

cnt:

RECEIVE state 依據 cnt=15 判斷 next_state 為 LOAD，如果 msg_lst 拉高且 cnt!=15，next_stage 為 PAD。

come_back:

一個 flag 的功能，提醒當 intermediate hash value 算出來時(UPDATE state 在做的事)，還不是最終的 digest，還需要回來 PAD 來給出最後一個 padding 的 message block。當 PAD 時看到 come_back 是 1 的話，還不用填入 msg_len_r[63:0]。

begin_with:

一個 flag 的功能，提醒 PAD 時要先給一個 32'b8000_0000，補上 masking 時沒有 append 的"1" bit。

msg_lst_r:

代表著這個 message 的 last word 接收過了沒，是作為 UPDATE 時判斷 next state 的條件之一。msg_lst 來臨時，msg_lst 會被存入 msg_lst_r。

rx_3 到 rx_10 在探討的是當 msg_lst 來臨時的各種不同可能狀況，分類如同 block diagram 的 explanation 中的分法。

(d) rx_3

last word = m16 且 msg_be=f，這種情況下，將 come_back 和 begin_with 都存入 1，並依據 cnt=15 判斷 next state 是 LOAD。

(e) rx_4

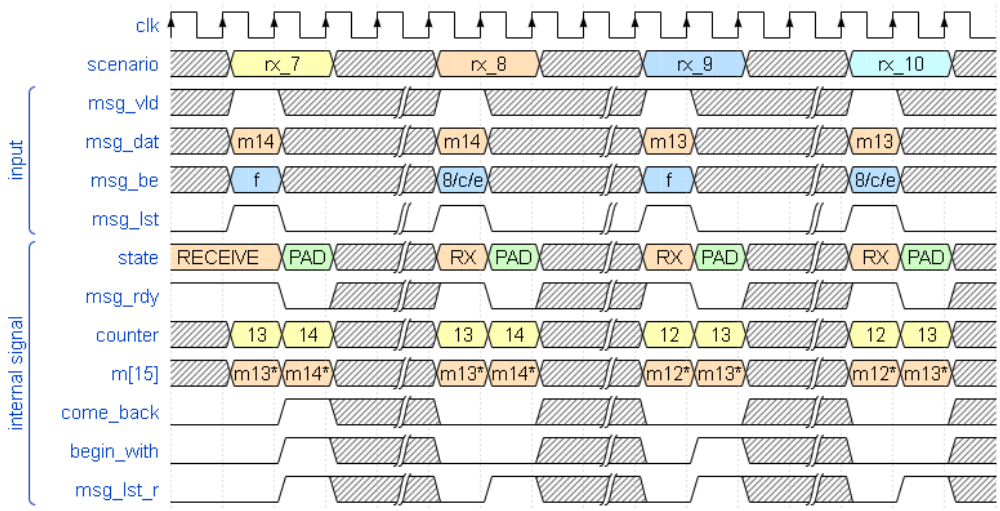
rx_4 代表 last word = m16 且 msg_be=8/c/e，這種情況下，將 come_back 和 begin_with 分別存入 1 和 0，並依據 cnt=15 判斷 next state 是 LOAD。

(f) rx_5

rx_5 代表 last word = m15 且 msg_be=f，這種情況下，將 come_back 和 begin_with 都存入 1，並依據 msg_lst=1 & cnt≠15 判斷 next state 是 PAD。

(g) rx_6

rx_6 代表 last word = m15 且 msg_be=8/c/e，這種情況下，將 come_back 和 begin_with 分別存入 1 和 0，並依據 msg_lst=1 & cnt≠15 判斷 next state 是 PAD。



(h) rx_7

rx_7 代表 last word = m14 且 msg_be=f，這種情況下，將 come_back 和 begin_with 都存入 1，並依據 msg_lst=1 & cnt≠15 判斷 next state 是 PAD。

(i) rx_8

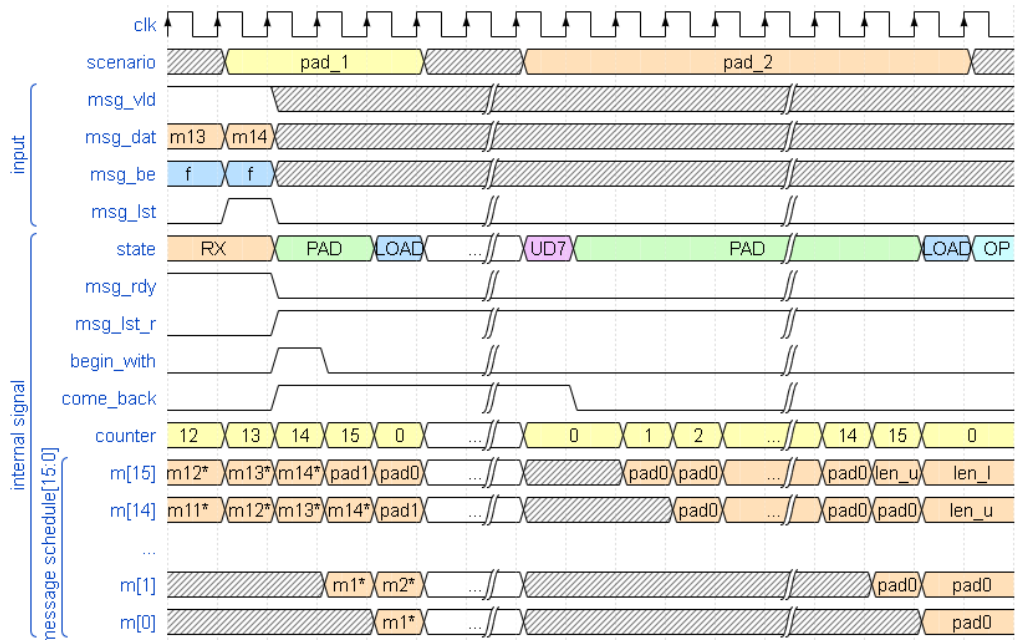
rx_8 代表 last word = m14 且 msg_be=8/c/e，這種情況下，將 come_back 和 begin_with 都存入 0，並依據 msg_lst=1 & cnt≠15 判斷 next state 是 PAD。

(j) rx_9

rx_9 代表 last word = m13(或還不到 13)且 msg_be=f，這種情況下，將 come_back 和 begin_with 分別存入 0 和 1，並依據 msg_lst=1 & cnt≠15 判斷 next state 是 PAD。

(k) rx_10

rx_10 代表 last word = m13(或還不到 13)且 msg_be=8/c/e，這種情況下，將 come_back 和 begin_with 都存入 0，並依據 msg_lst=1 & cnt≠15 判斷 next state 是 PAD。



首先介紹 PAD state 時，每個 clock rising edge 會：

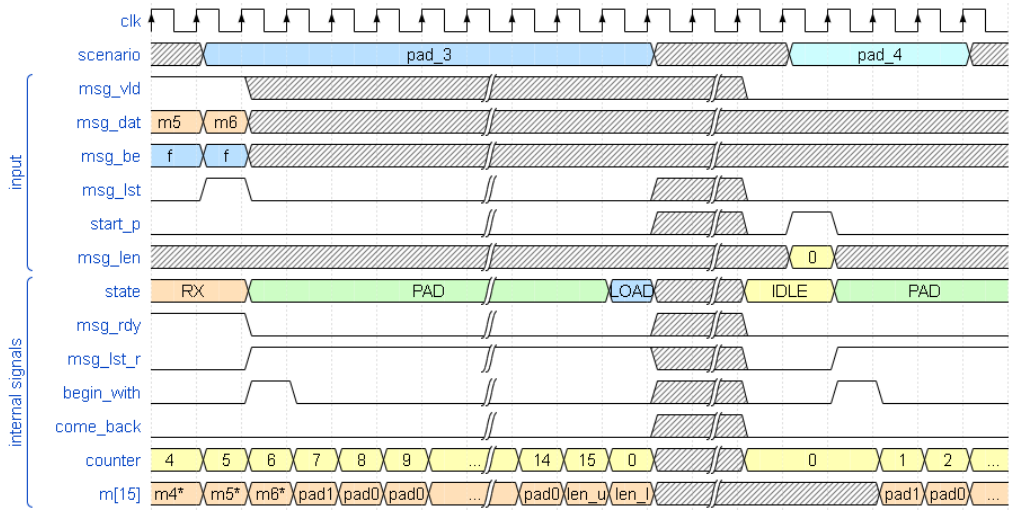
1. 如果 come_back = 0，且 cnt = 14 時，對 m[15]填入 msg_len_r[63:32]
2. 如果 come_back = 0，且 cnt = 15 時，對 m[15]填入 msg_len_r[31:0]
3. 如果 come_back = 1 或是 cnt ≠ 14 or 15，則根據 begin_with=0 or 1，對 m[15]填入 32'h0000_0000 or 32'h8000_0000
4. 將 0 填入 begin_with
5. 將 m[i+1]的 data 填入 m[i] ($0 \leq i \leq 14$)
6. Counter + 1
7. 當 cnt = 15 則清空 cnt 並轉換到 LOAD

(l) pad_1

pad_1 是承接 rx_7 的情況(come_back=1 且 begin_with=1)，在 PAD state 時看到 come_back=1 且 begin_with=1 會對 m[15]填入 32'h8000_0000 (即為 waveform 中的"pad1")，並且因為在 PAD 時 begin_with 會被填入 0，因此 bit "1"只會被 append 一次，接著因為 come_back=1 且 begin_with=0，32'h0000_0000(即為 waveform 中的"pad0")會接著被填入，直到 cnt =15，清空 cnt 並轉換到 LOAD。

(m) pad_2

pad_2 是說明當 UPDATE 時因為 come_back=1 而還要回來 PAD 的情況，注意當要離開 UPDATE 時要將 come_back 拉回 0。pad_2 一開始看到 come_back=0，begin_with=0 且 cnt ≠ 14 or 15，因此會對 m[15]填入 32'h0000_0000，直到 cnt =14 時，填入 msg_len_r[63:32] (即為 waveform 中的"len_u")；cnt = 15 時，填入 msg_len_r[31:0] (即為 waveform 中的"len_l")，並清空 cnt 進入 LOAD。



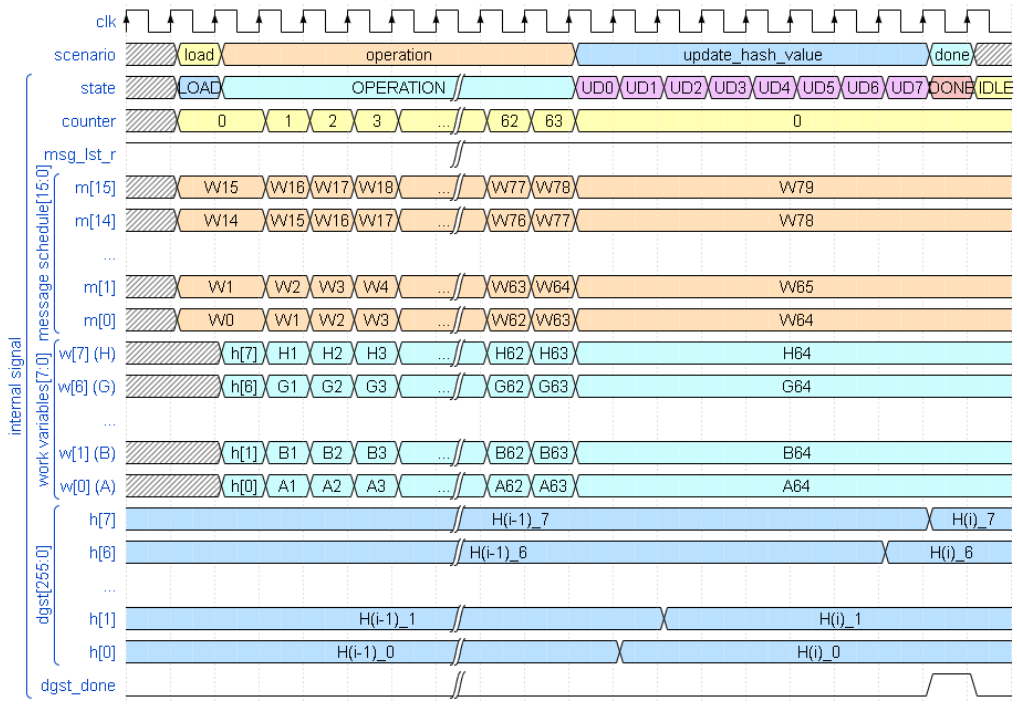
(n) pad_3

pad_3 是承接 rx_9 的情況(come_back=0 且 begin_with=1)，在 PAD state 時看到 come_back=0，begin_with=1 且 cnt ≠ 14 or 15 會對 m[15]填入 32'h8000_0000 (即為 waveform 中的"pad1")，並且因為在 PAD 時 begin_with 會被填入 0，因此 bit "1"只會被 append 一次，接著因為 come_back=0，begin_with=0 且 cnt ≠ 14 or 15，32'h0000_0000(即為 waveform 中的"pad0")會接著被填入，直到 cnt =14 時，填入 msg_len_r[63:32] (即為 waveform 中的"len_u")；cnt = 15 時，填入 msg_len_r[31:0] (即為 waveform 中的"len_l")，並清空 cnt 進入 LOAD。

(o) pad_4

這個 scenario 在說明特殊狀況，當 IDLE 收到 start_p=1 時的 msg_len=0，則因為部會進到 RECEIVE，因此原本是在 RECEIVE 收到 last word 時所更新的 3 個 register：come_back, begin_with, and msg_lst_r 就需要在這裡完成。

需要拉起 msg_lst_r 以向後面的 state 表示該要做最後一個 message block 該做的事，並且分別設定 come_back=1 和 begin_with=1。接下來就會跟 pad_3 的情況一模一樣，不再贅述。



(p) load

LOAD 時會將現在 h[0:7]暫存的 initial or intermediate hash value $H_7^{(i-1)}$ to $H_0^{(i-1)}$ (waveform 中的 H(i-1)_7 to H(i-1)_0) 存入 work variables w[0:7]。

(q) operation

做 message schedule 產生W_0~W_{63}，分別是 waveform 中的 m[0]依序產生的 W0~W63，並以此不斷更新 work variables w[0:7]，waveform 中以 A2 代表做了 2 次 iteration 的 work variable, **a**，其他代號依此類推。以 cnt 來記錄做了幾次疊代，因此當 cnt =63 時，進入下一個 state。

(r) update_hash_value

這邊要注意的是原本我的 FSM 架構是一個 UPDATE state 去做 h[j] = h[j] + w[j] ($0 \leq j \leq 7$)，但這樣會需要八個 32-bit 加法器，因此將此狀態拆成 8 個狀態分別是 UD0 ~ UD7，將 next state 的決定交給 UD7，每個狀態更新一

個 h[j]，這樣就能共用加法器，雖然會多一些控制邏輯和 state 的數量，但整體面積會下降。沒有更新 FSM 的架構圖是因為如果畫上去會變得超級擁擠，可讀性下降很多，因此用文字以及 waveform 描述。

(s) done

拉高 dgst_done，表示完成 message 的 digest。接著回到 IDLE 等待下個 start_p 的到來。

3. Performance

定義 Throughput 是從第一筆 input 到第一筆 output 需要多少 cycles？

假設是一個 16 個 word 的 message，則需要花 16 cycles 接收 words 進來**(假設 msg_vld 是連續的)**，並開始 LOAD (1 cycle), OPERATION (64 cycles), UPDATE (UD0 ~ UD7, 8 cycles)，接著再回到 PAD 花 16 cycles 填入 16 個 words，再開始 LOAD (1 cycle), OPERATION (64 cycles), UPDATE (UD0 ~ UD7, 8 cycles)，因此完成 output 需要 16+(1+64+8)+16+(1+64+8)=178 個 cycles。推廣到 N 個 word 的 message 的話，分成以下 2 種情況 (n 為非負整數)：

(1) N=16n or 16n+15 or (16n+14 且 last word msg_be=f)

從第一筆 input 到第一筆 output 需要 $89 \times (n+1)$ cycles

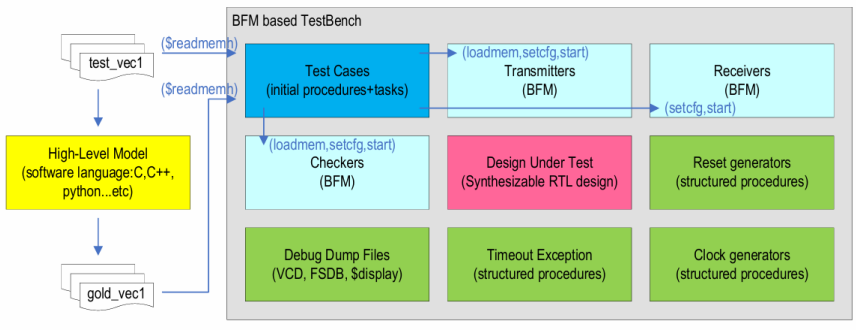
(2) N=16n+1~13 or (16n+14 且 last word msg_be=8/c/e)

從第一筆 input 到第一筆 output 需要 $89n$ cycles

其實就是分成，在 last word 進來後的這個 message block 算完後，還需不需要回到 PAD 在算一輪，一個 message block 需要 89 cycles (包含 receiving, padding, iteration, and update)。

B. Testbench

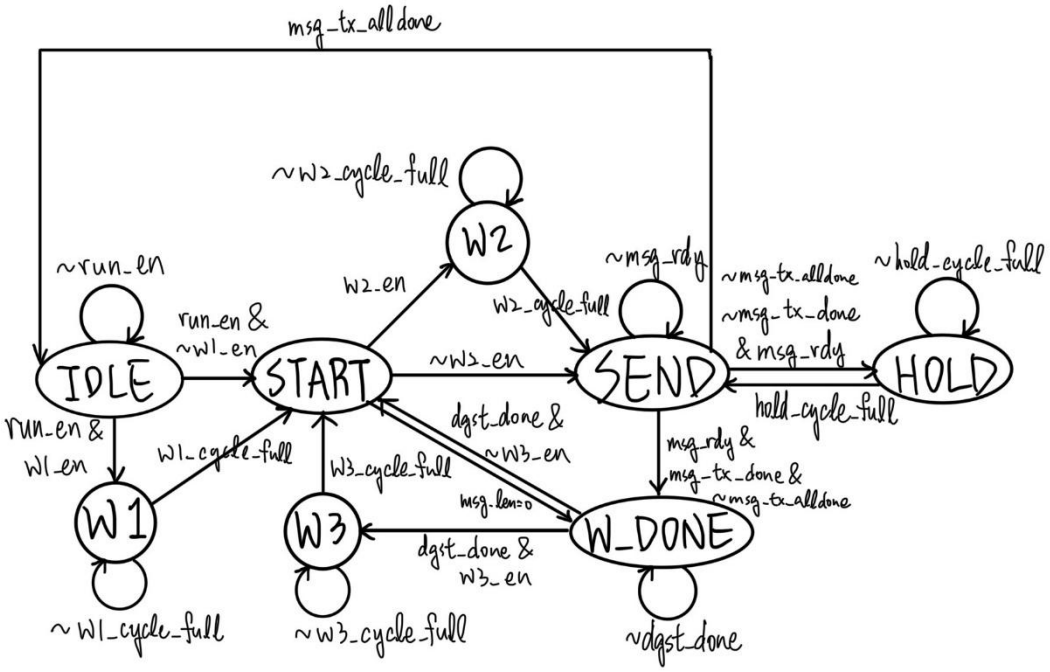
這次的 testbench 大多參考 lecture 5 的講義以及所附的例子，使用的是 BFM 的模式，架構如講義中的圖：



High level model 的部分是使用 python 寫 SHA-256 的 algorithm 並且輸入不同長度的 test vector 與使其生成每個 test vector 所對應之 golden vector 以及 message length，並且還有每個 word 所對應的 msg_be 和 msg_lst，並輸出五份.dat 檔包含 test vector, golden vector, msg_len, msg_be, msg_lst，在 test case 中讀取，並將 golden vector 存入 checker 的 memory 中，再將剩下的 data 都存進 transmitter 的 memory 中。為求打滿 coverage，輸入的 test vector 要包含設計時所設想過的每種情形以及所能想到的極端狀況，包含：

1. msg_len < 14 words, msg_be of last word is f
2. msg_len < 14 words, msg_be of last word is 8/c/e
3. msg_len = 14 words, msg_be of last word is f
4. msg_len = 14 words, msg_be of last word is 8/c/e
5. msg_len = 15 words, msg_be of last word is f
6. msg_len = 15 words, msg_be of last word is 8/c/e
7. msg_len = 16 words, msg_be of last word is f
8. msg_len = 16 words, msg_be of last word is 8/c/e
9. msg_len > 16 words, msg_be of last word is f
10. msg_len > 16 words, msg_be of last word is8b/c/e
11. msg_len = 0 word
12. msg_len = 1 word, msg_be of last word is 8/c/e/f
13. msg_len >> 16 word, msg_be of last word is 8/c/e/f

接下來是 transmitter 的設計，以下是其內部的 FSM：



初始狀態在 IDLE，當 run_en 打近來則可能前往 W1 等待或是直接前往 START。START 時 output 出 start_p=1，並可能前往 W2 等待，或是直接前往 SEND 拉高 msg_vld 並等待 msg_rdy 發送 msg_dat，當 msg_rdy=1 可能會前往 HOLD 拉低 msg_vld 等待(在 test case 中設定)，當送完一個 message 時，進入 W_DONE，這個 state 是在等待 dgst_done 的訊號到來(等 DUT 算完)，當 dgst_done，可能進入 W3 等待或是直接進入 START 開始下一個的 message 的傳送。若送完最後一個 message，SEND 就不會再前往 W_DONE，而是回到 IDLE 宣告 transmit 已經結束(tx_busy=0)並等待模擬結束。

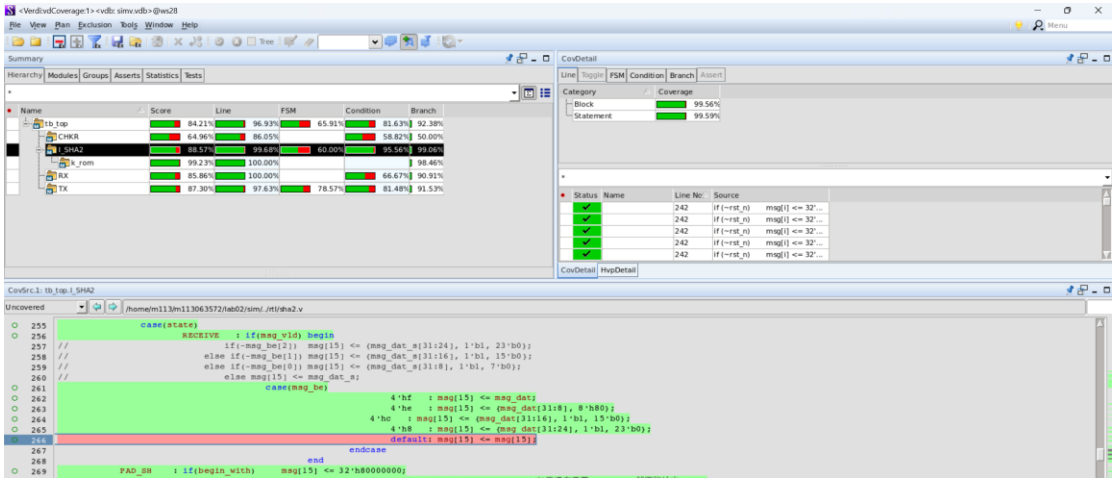
這次的 receiver 其實可以與 checker 直接合併，因為 output protocol 中 receive 端只能無條件接收 DUT 的 output，但我還是保留了 receiver 的框架，並且在 receiver 中拉了一條 dgst_vld 的訊號，當 dgst_done=1 就拉起它，直到 start_p=1 再拉下它，其存在目的是為了要驗證 dgst 在 dgst_done 拉起後直到 start_p 放下前是否都維持著正確值。因此 checker 檢查的不只有 dgst_done=1 時的 dgst 是否為正確值，還有 dgst_vld=1 時的 dgst 是否為正確值。

C. Coverage report

All coverage is as expected. The uncovered cases in line, FSM, condition, and branch are explained below.

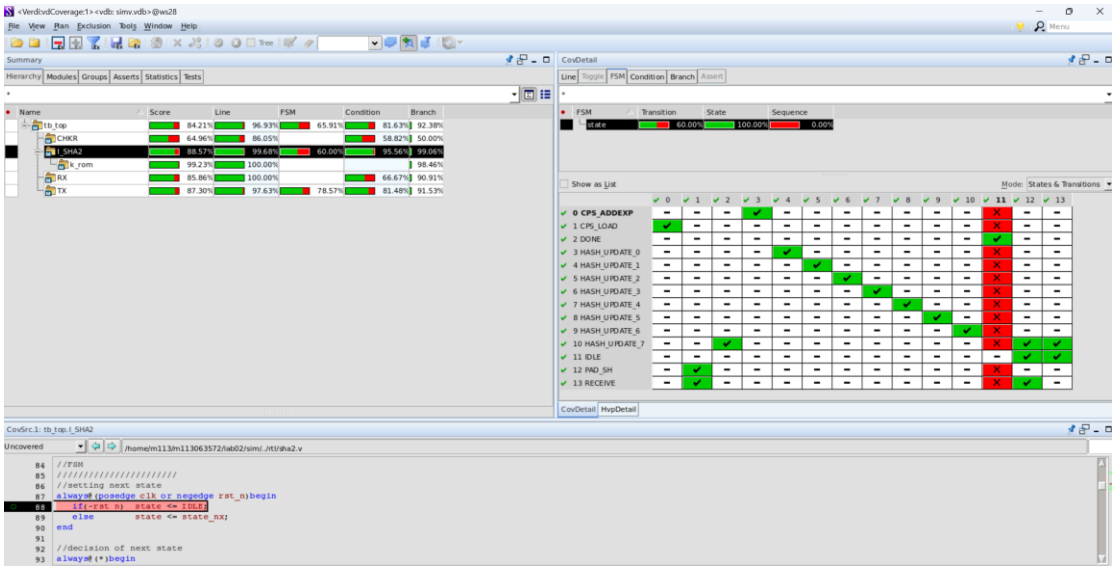
Line coverage:

In the case selection of msg_be, default case did not occur, because TB provided valid msg_be when msg_vld asserts.



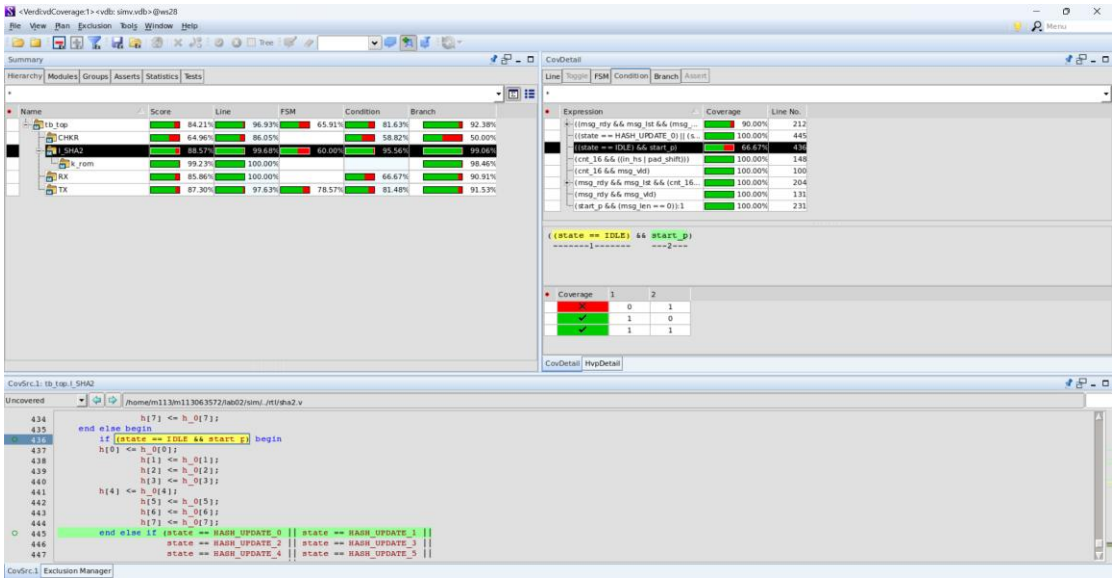
FSM coverage:

rst_n will only assert at the beginning, so there will not be any other state jumping to IDLE at the reset. As a result, only DONE jumping to IDLE have occurred as expected in the FSM structure.

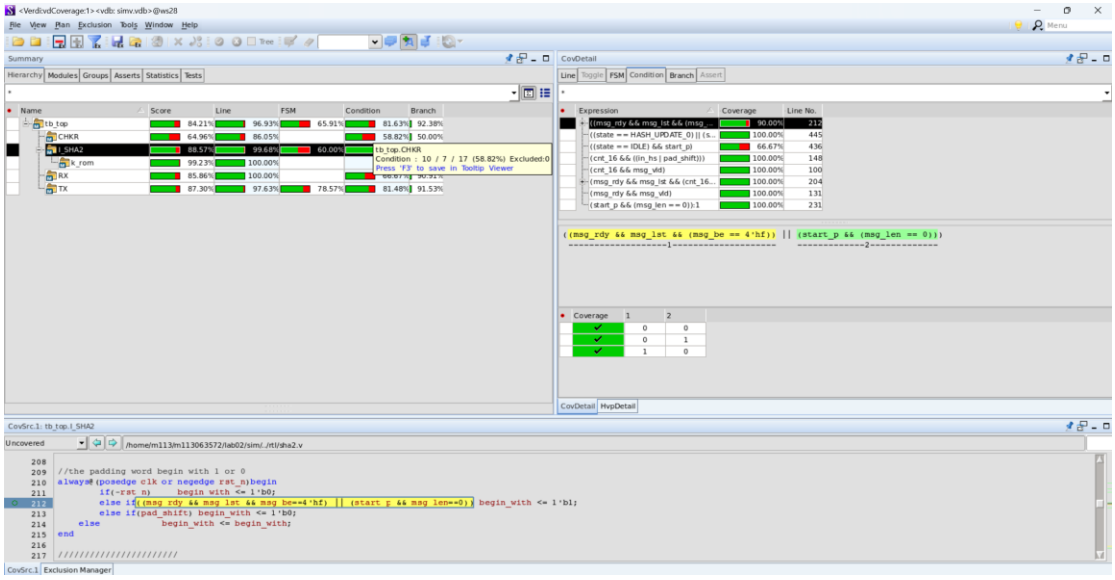


Condition coverage:

When start_p asserts, FSM must be in IDLE case. (代表其實 state==IDLE 是可以省略的，但如果省略，面積會上升，所以我才沒有省略)

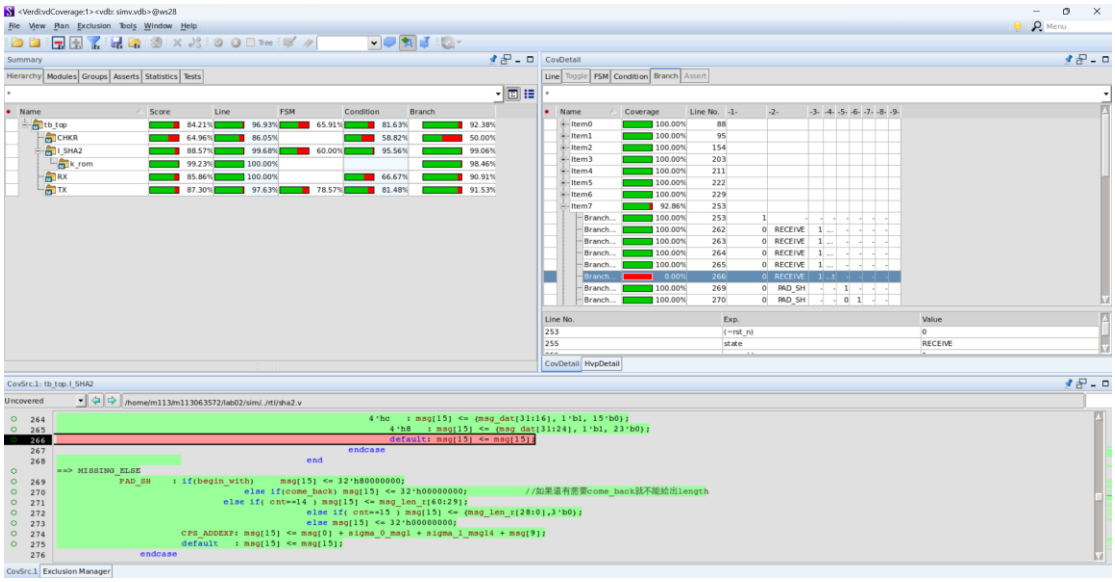


The two conditions that causes begin_with to assert is one hot.



Branch coverage:

The same thing as described in the Line coverage section.



k_rom is a submodule that stores the constants K_0 to K_{63}. Since the address range is guaranteed to be within 0 to 63, no default value is required.