Design of Chip Security - Spring 2024

Lab02-FSM Report

**Table of Contents**

1. **Design Architecture and Performance1**
2. Algorithm 1
3. Implementation 5
4. Block diagram 5
5. SHA2 module structure 12
6. FSM 12
7. Performance 23
8. **Testbench24**
9. **Coverage Report26**

Name: 王品然

Student ID: 113063572

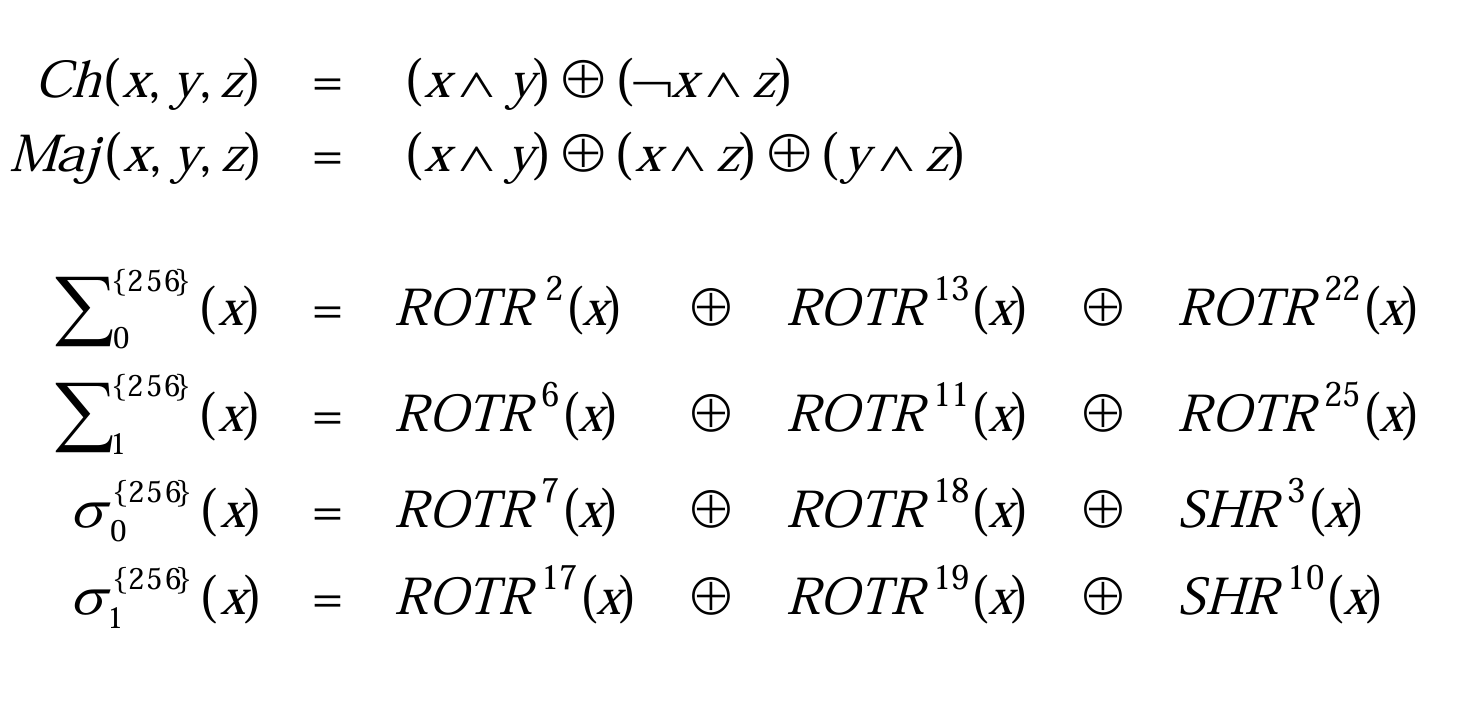
1. **Design Architecture and Performance**
2. Algorithm

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

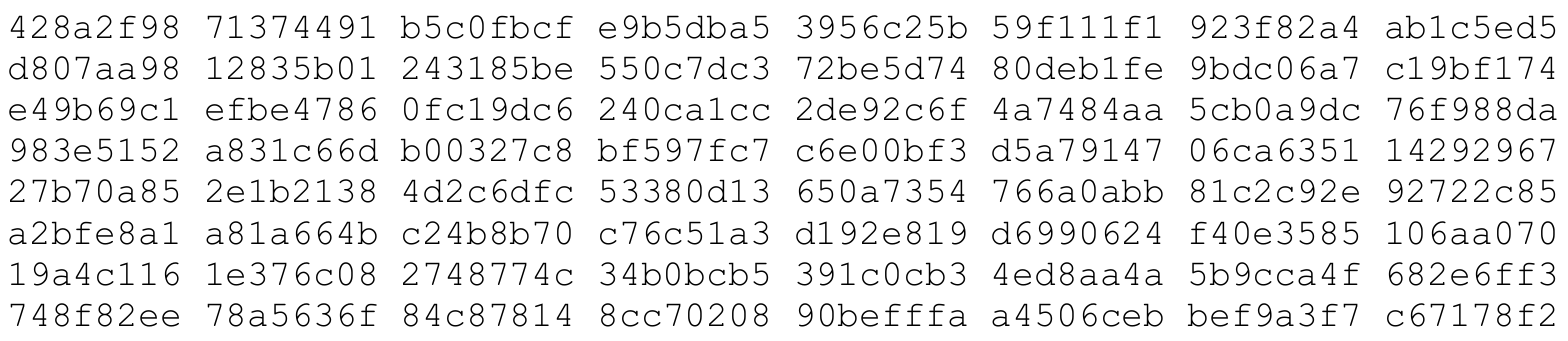
To begin with, I will introduce the functions and constants required for the algorithm.

1. Functions:



For simplicity, the functions , , , and will be denoted as , , , and , respectively, throughout this report.

1. Constants:



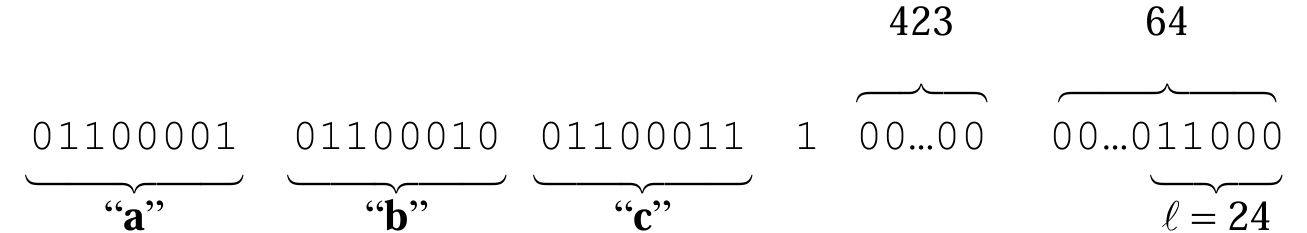
This is a sequence of 64 constant 32-bit words, , ,…,. For simplicity, I will refer to them as , ,…, in this report.

1. Preprocessing

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

1. 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 𝓀 zero bits, where 𝓀 is the smallest, non-negative solution to the equation . Then append the 64-bit block that is equal to the number ℓ expressed using a binary representation. An example is shown below:



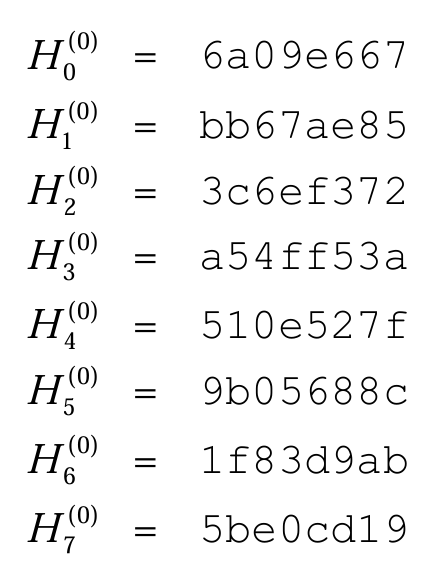
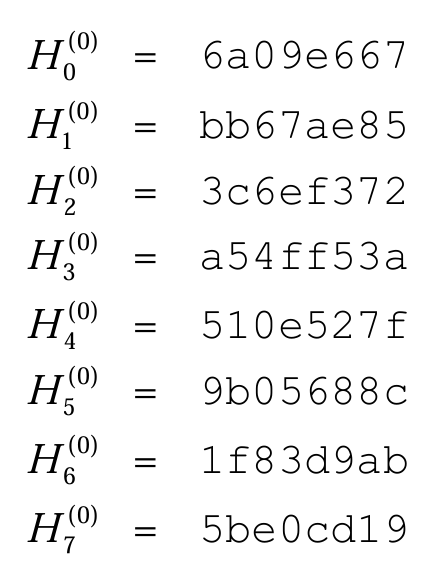
The length of the padded message should now be a multiple of 512 bits.

1. Parsing the message:

The padded message is divided into N 512-bit blocks, , ,…, . 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 , the next 32 bits are , and so on, up to .

1. Setting the initial hash value:

For SHA-256, the initial hash value, , shall consist of the following eight 32-bit words, in hex:

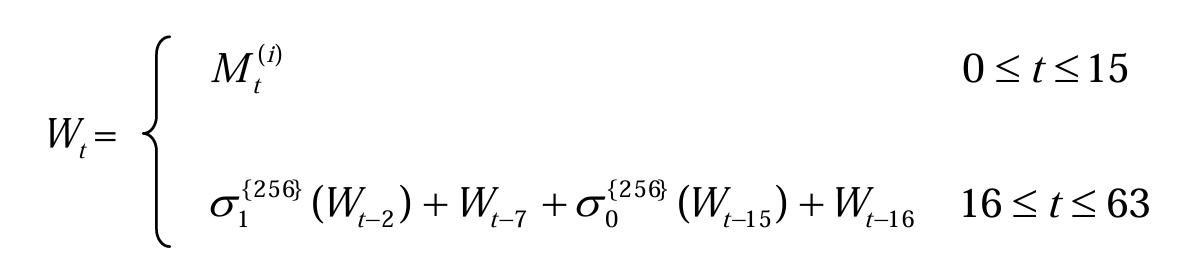
 

1. Computation

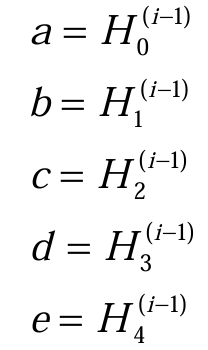
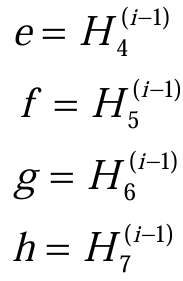
Each message block, , ,…, , is processed in order, using the following steps:

**For *i = 1 to N*, repeat:**

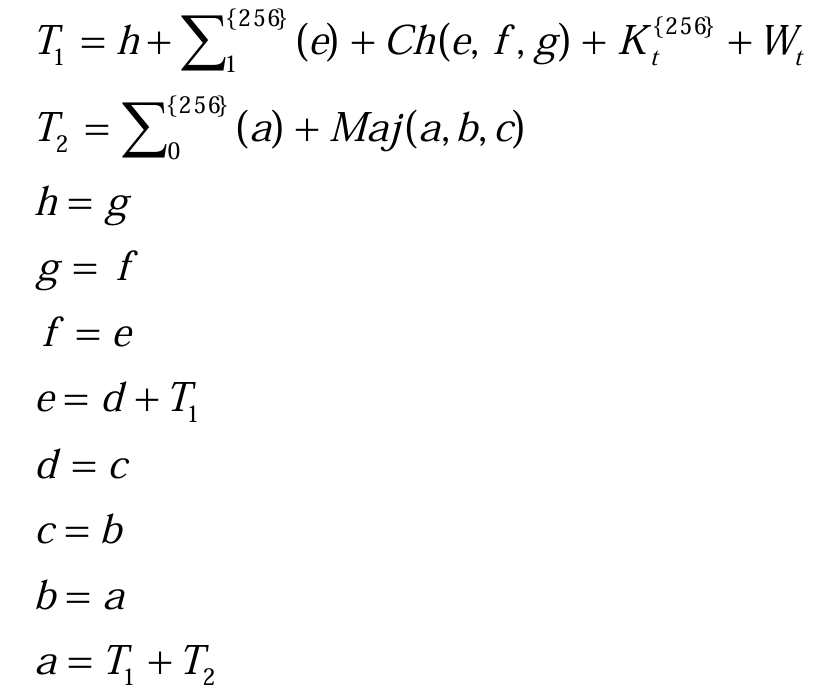
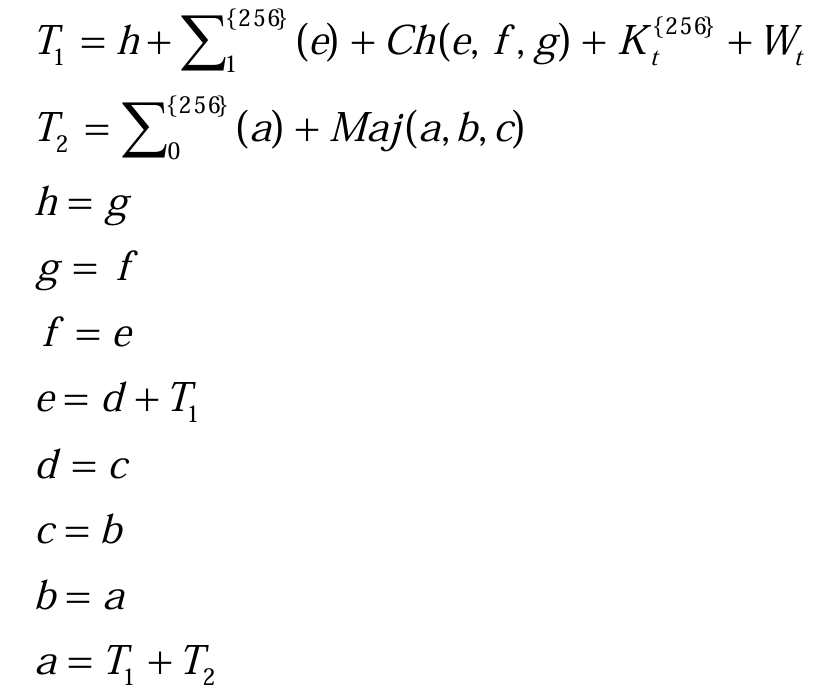
1. Prepare the message schedule {}:



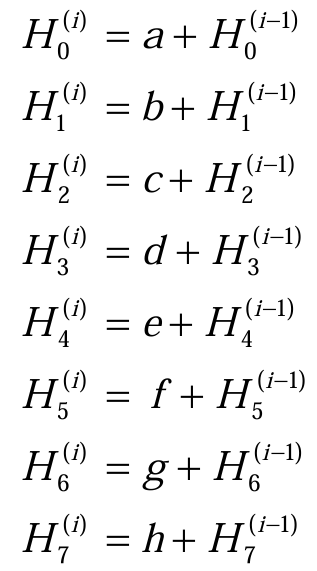
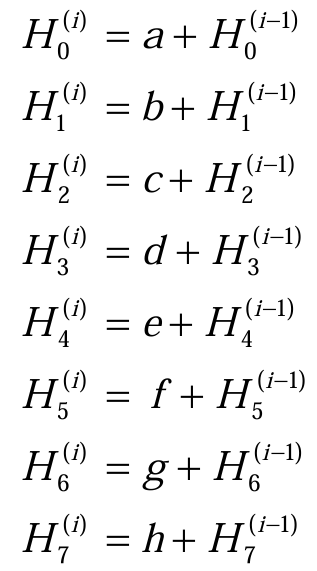
1. Initialize the eight working variables, **a**, **b**, **c**, **d**, **e**, **f**, **g**, and **h**, with the hash value:

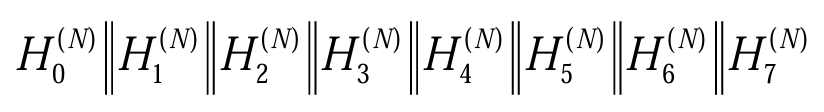
1. For *t = 0 to 63* , repeat:

1. Compute the intermediate hash value :

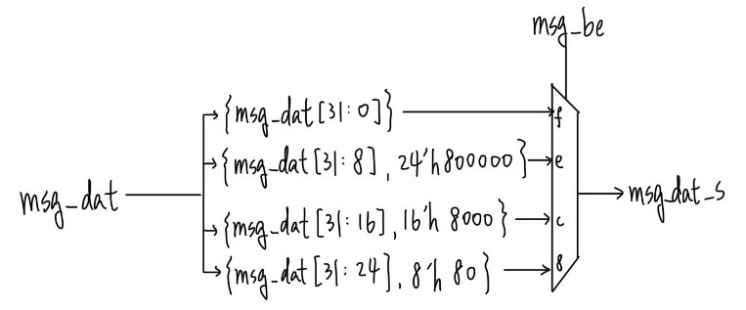
 

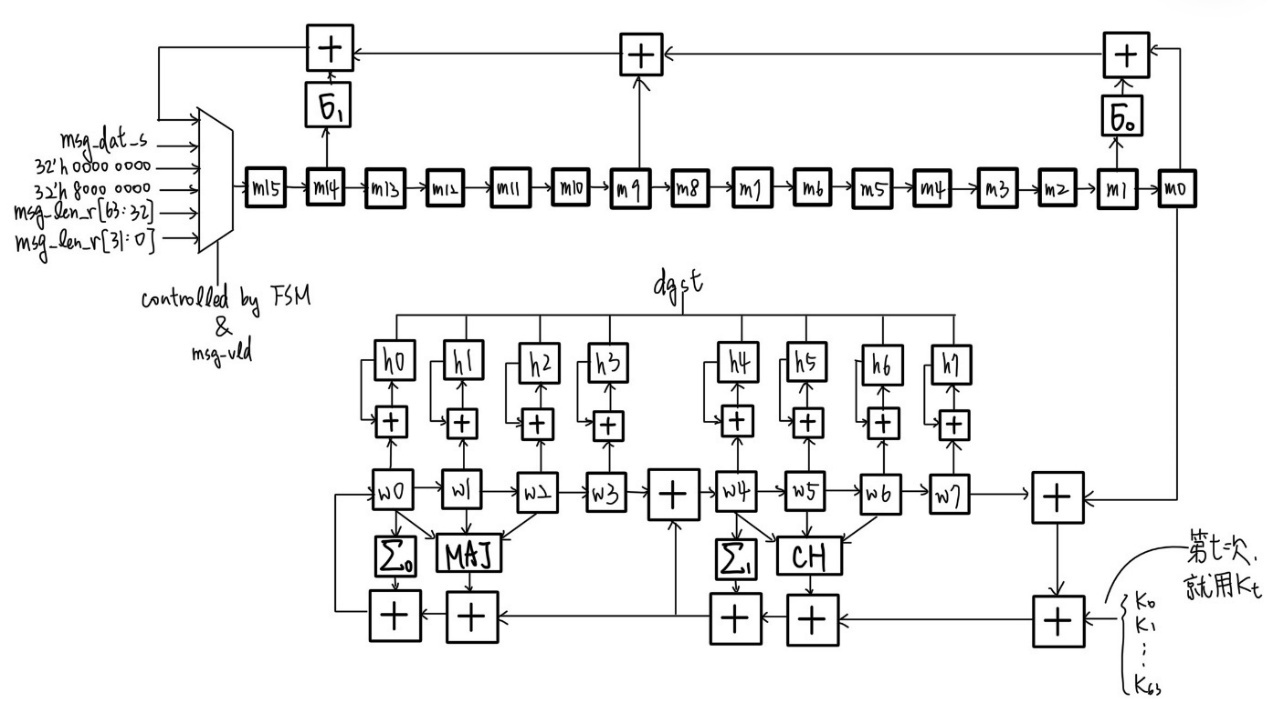
After repeating steps (a) through (d) for all N message blocks, the resulting 256-bit message digest of M is obtained by concatenating the eight 32-bit intermediate hash values from the iteration as follow:



1. Implementation
2. Block diagram

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





**Notations:**

1. m[0~15]:

Sixteen 32-bit registers for the message schedule to .

1. h[0~7]:

Eight 32-bit registers for the intermediate hash value to .

1. w[0~7]:

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

1. msg\_len\_r:

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

1. CH, MAJ, , , , and :

Functions defined in the algorithm.

1. , ,…, :

Constants required in the algorithm.

1. **+**:

Modular addition (modulo )

1. :

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

1. msg\_dat\_s:

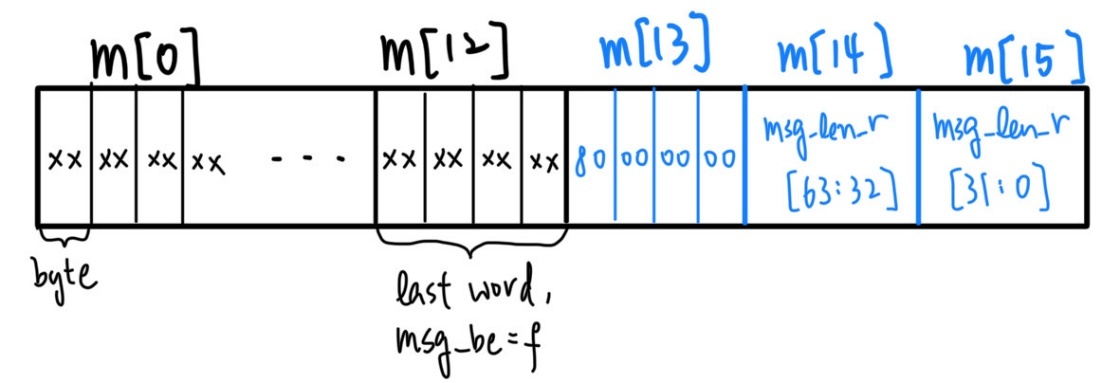
The masked version of msg\_dat, generated based on the value of msg\_be.

**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：

1. 收到last word時，m[0:15]已收到少於13個(含)words，且
2. msg\_be = f

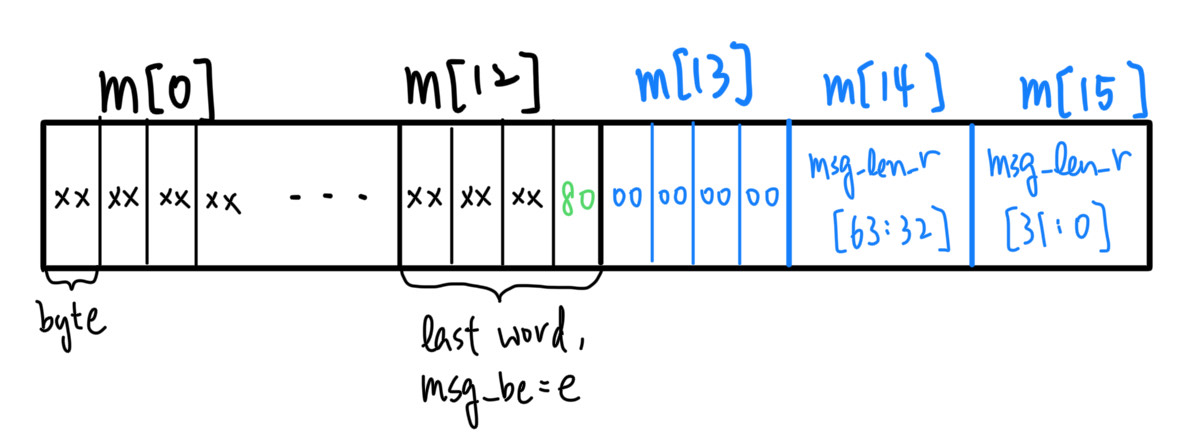
因為在masking的步驟中，msg\_be = f的情況沒有append “1”上去，因此在下一步要將32’h8000\_0000塞入m[15]，接著再塞入32’h0000\_0000，直到這個message block已存有14個word為止，而剩下最後兩個padding word分別要填入msg\_len\_r[63:32]以及msg\_len\_r[31:0]。接下來再開始message schedule和work variables的iteration，並計算最終的hash value。操作結果的message block如附圖：



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。

1. msg\_be = 8, c, or e

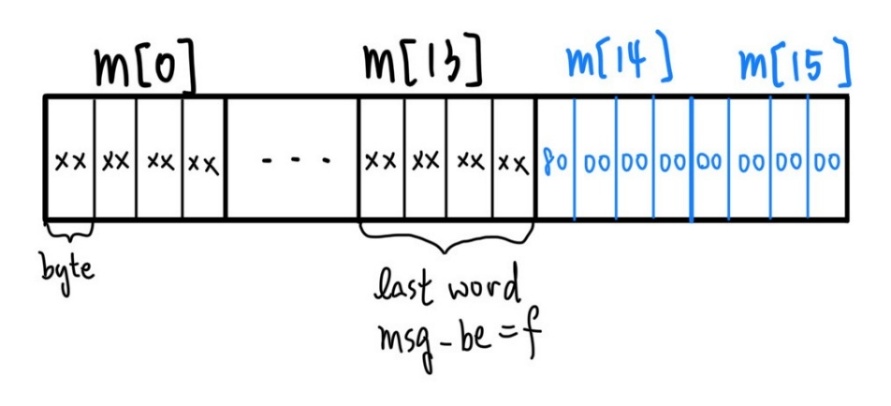
因為在masking的步驟中，msg\_be = 8, c, or e的情況已經append “1”上去，因此下一步只要塞入32’h0000\_0000到m[15]，直到這個message block已存有14個word為止，而剩下最後兩個padding word分別要填入msg\_len\_r[63:32]以及msg\_len\_r[31:0]。接下來再開始message schedule和work variables的iteration，並計算最終的hash value。操作結果的message block如附圖：

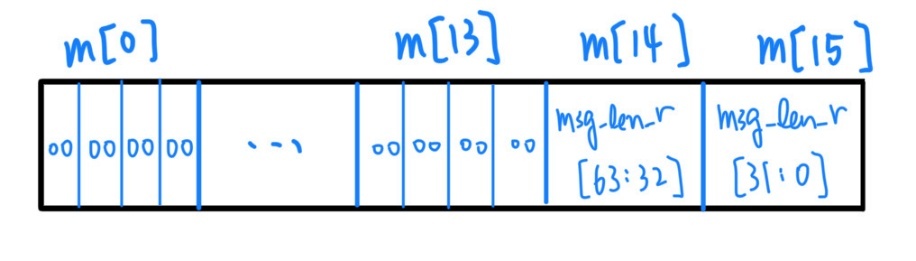


其中綠筆表示非原始msg\_dat的data，而是在masking時加上去的。

1. 收到last word時，m[0:15]已收到14個words，且
2. msg\_be = f

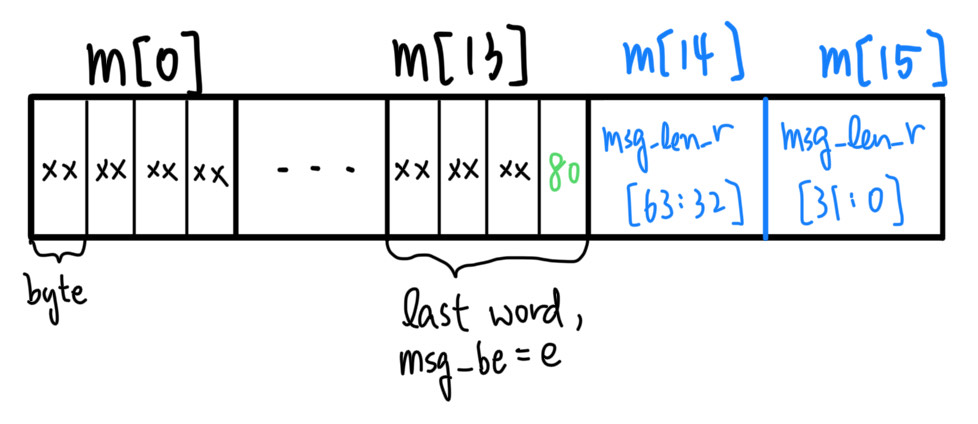
與(a) 1.的情況相似，但已經無法在塞入32’h8000\_0000後，繼續放入msg\_len\_r[63:0]，因為已經只剩一個word的空間了，因此接著再塞入一個32’h0000\_0000，完成這個message block，並且開始message schedule和work variables的疊代與intermediate hash value的計算。之後再來padding最後一組message block進去m[0:15]中，此時的m[0:13]皆為0，而m[14:15]存放msg\_len\_r[63:0]。接下來再開始message schedule和work variables的iteration，並計算最終的hash value。結果分別如附圖的兩個message block：





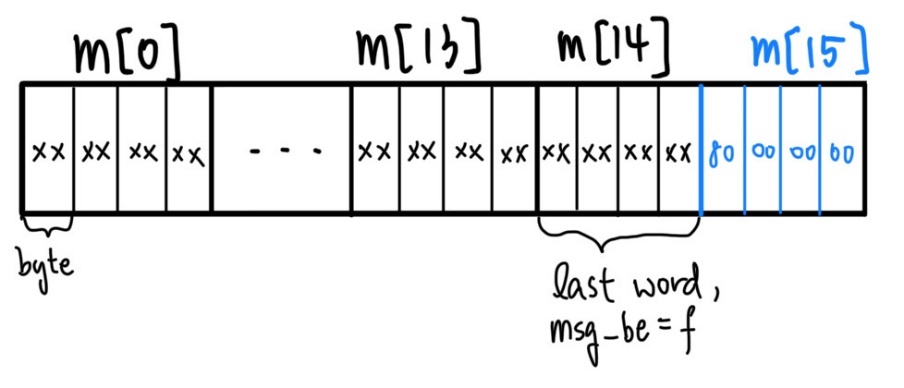
1. msg\_be = 8, c, or e

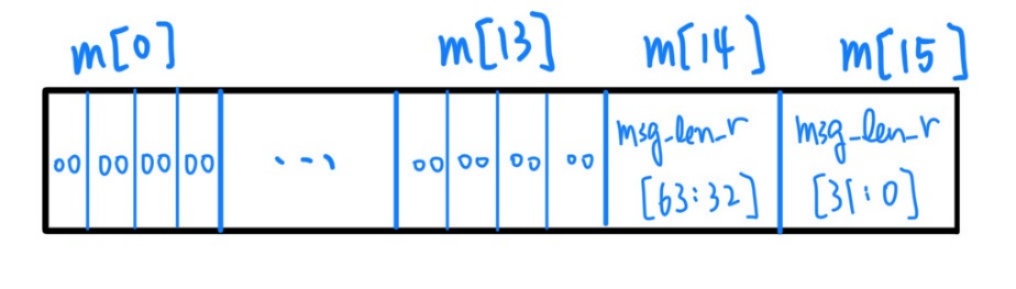
與(a) 2.的情況極為相似，只是此時不用在補0上去，因為只剩最後兩個word的空間，直接塞入msg\_len\_r[63:0]即可開始message schedule和work variables的iteration，並計算最終的hash value。最終message block如附圖：



1. 收到last word時，m[0:15]已收到15個words，且
2. msg\_be = f

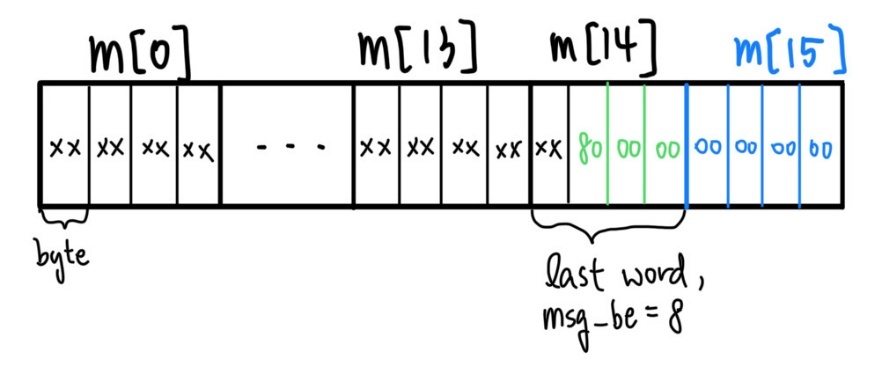
與(b) 1.時的情況極為相似，不再贅述。結果如附圖的兩個message block

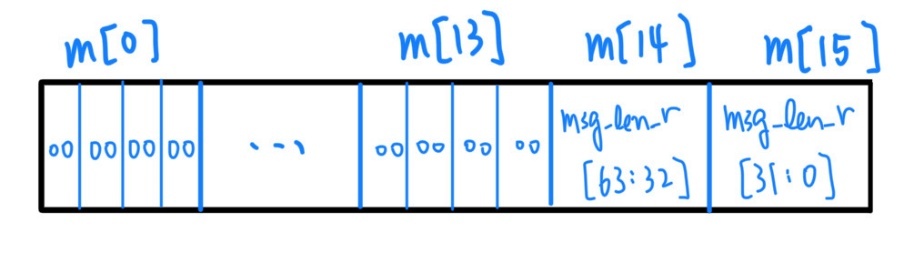




1. msg\_be = 8, c, or e

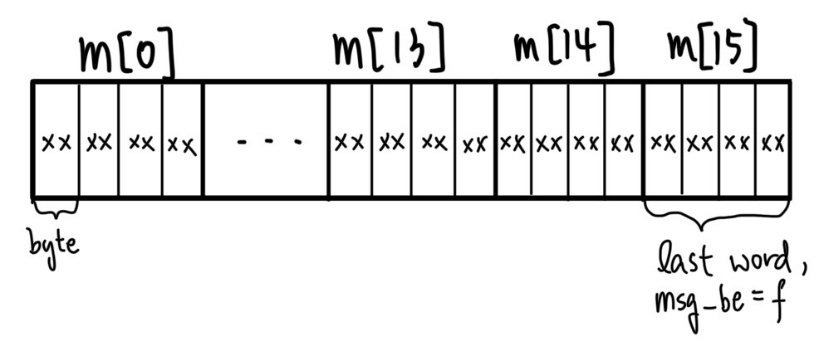
此時的message block已存有15個word，因此不足以填入msg\_len\_r[63:0]，所以先填入一個32’h0000\_0000完成這個message block，並開始message schedule和work variables的疊代與intermediate hash value的計算。之後再來padding最後一組message block進去m[0:15]中，此時的m[13:0]皆為0，而m[14:15]存放msg\_len\_r[63:0]。接下來再開始message schedule和work variables的iteration，並計算最終的hash value。結果分別如附圖的兩個message block：

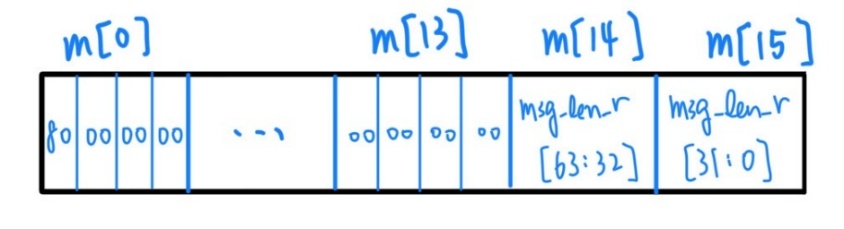




1. 收到last word時，m[0:15]剛好收到第16個word，且
2. msg\_be = f

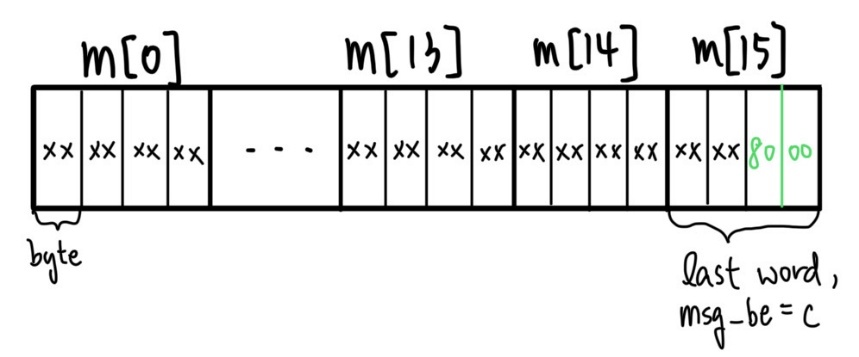
此時的message block已有16個word，因此直接開始message schedule和work variables的疊代與intermediate hash value的計算。之後再來padding最後一組message block進去m[0:15]中，因為尚未append “1”，所以此時m[0]填入32’h8000\_0000而m[13:1]皆為0，m[14:15]存放msg\_len\_r[63:0]。接下來再開始message schedule和work variables的iteration，並計算最終的hash value。結果分別如附圖的兩個message block：

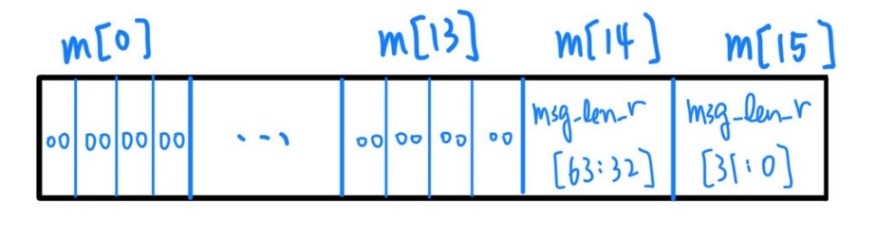




1. msg\_be = 8, c, or e

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。

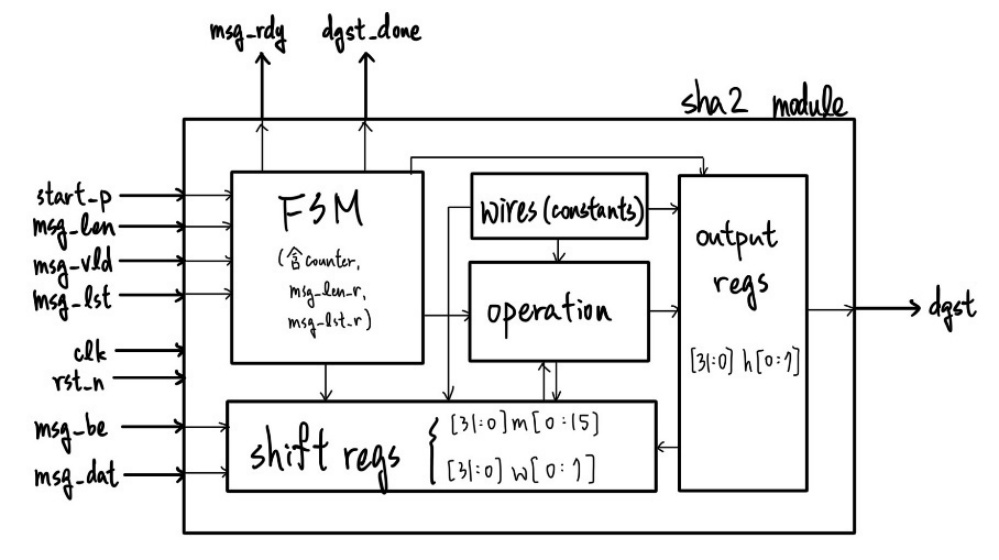




最後還有一種極端狀況沒有考慮到：當msg\_len=0時，不會有任何message進來，因此需要直接填入512’b10…0到message block。以上便是對於block diagram的詳細解釋，可以看到目前還缺乏flow control以及控制I/O protocol的部分，將在接下來module structure和FSM中繼續完善。

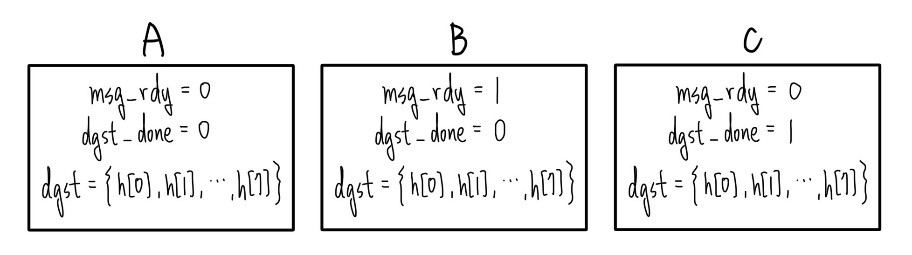
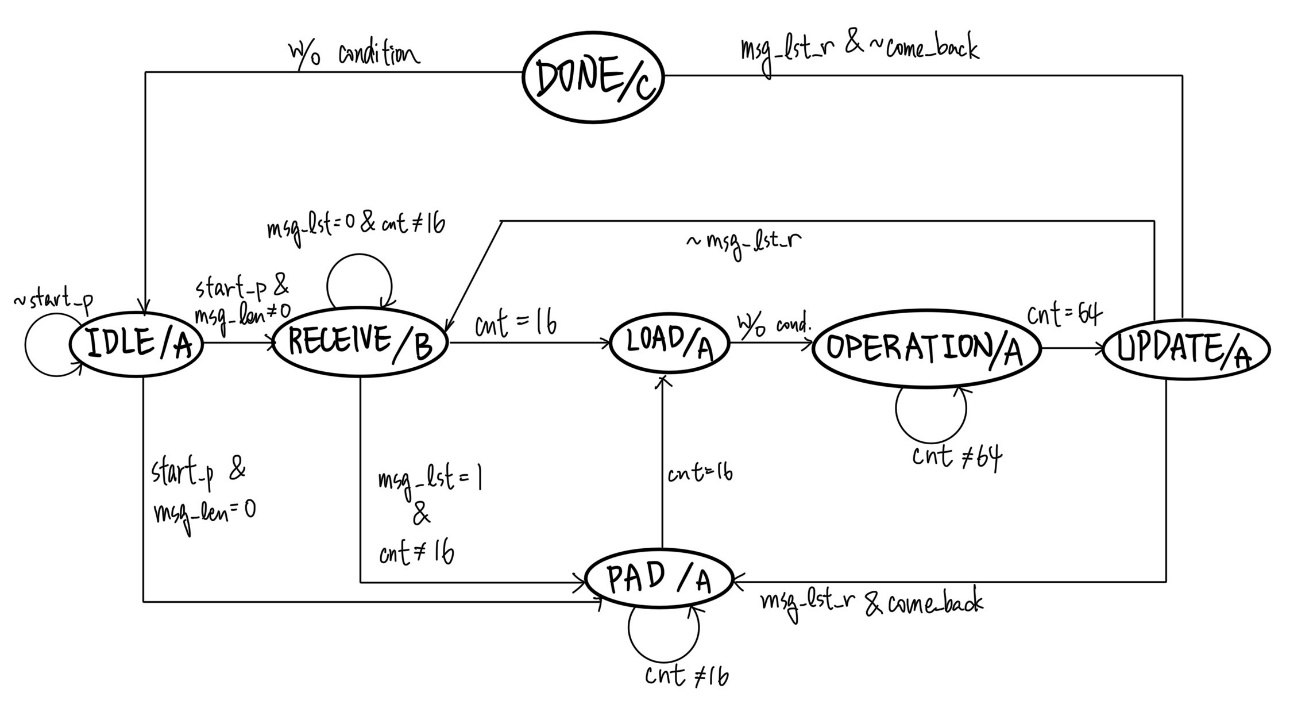
1. sha2 module structure

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



1. Control logic (FSM)

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



**Notation:**

1. 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.

1. cnt:

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

1. 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。

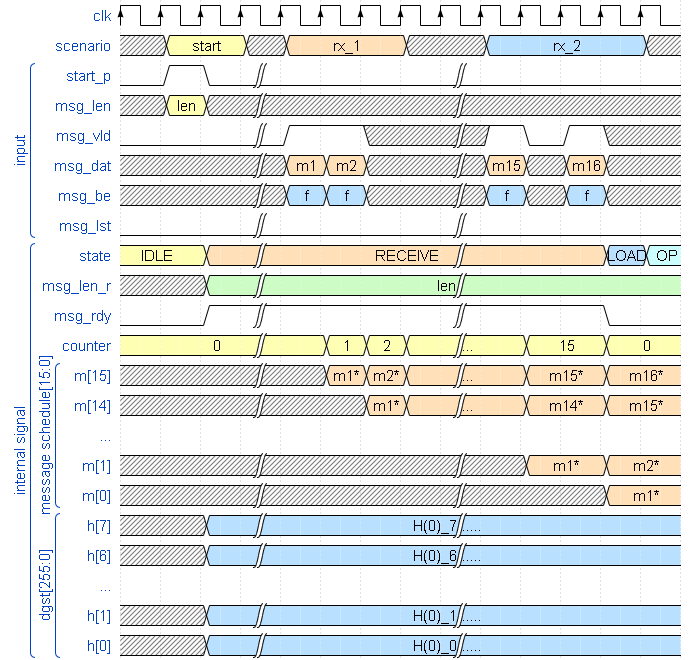
LOAD：將現在h[0:7]暫存的initial or intermediate hash value 存入work variables w[0:7]。

OPERATION：做message schedule產生~，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)

1. 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:

1. reset 放開後
2. 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 to (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。

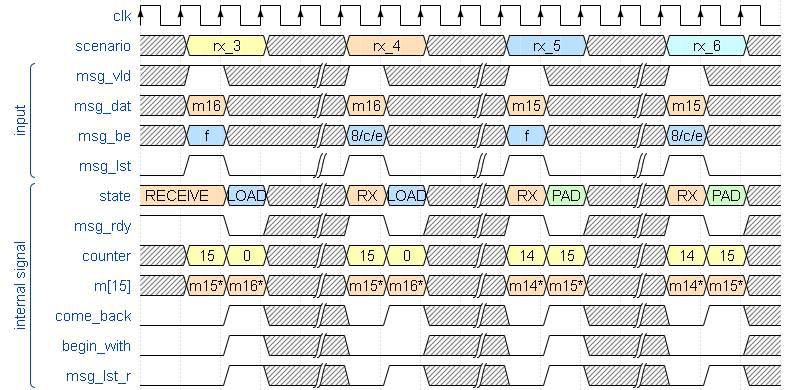
進入RECEIVE stage，design 會拉高msg\_rdy持續等待msg\_vld，細部行為可以區分成以下10個 scenario (rx\_1~10) 進行討論：

1. 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] ()
3. counter+1
4. 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。

1. 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。

1. rx\_4

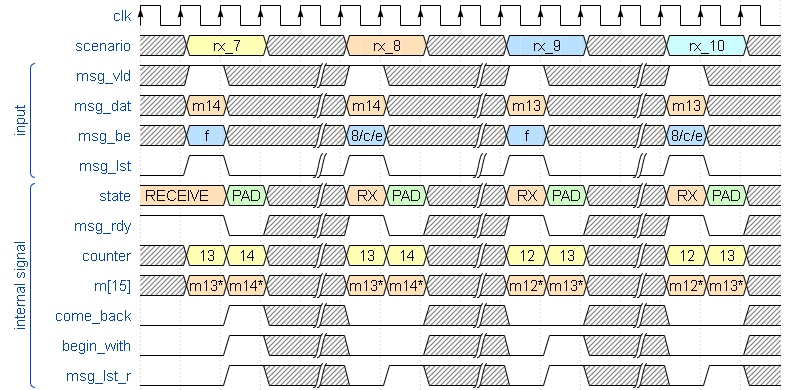
rx\_4代表last word = m16且msg\_be=8/c/e，這種情況下，將come\_back和begin\_with分別存入1和0，並依據cnt=15判斷next state是LOAD。

1. 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。

1. 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。



1. 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。

1. rx\_8

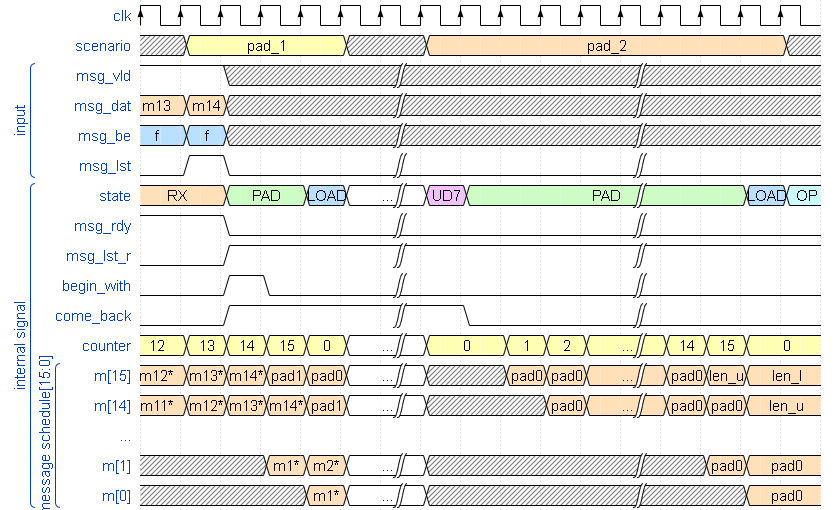
rx\_8代表last word = m14且msg\_be=8/c/e，這種情況下，將come\_back和begin\_with都存入0，並依據msg\_lst=1 & cnt≠15判斷next state是PAD。

1. 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。

1. 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。



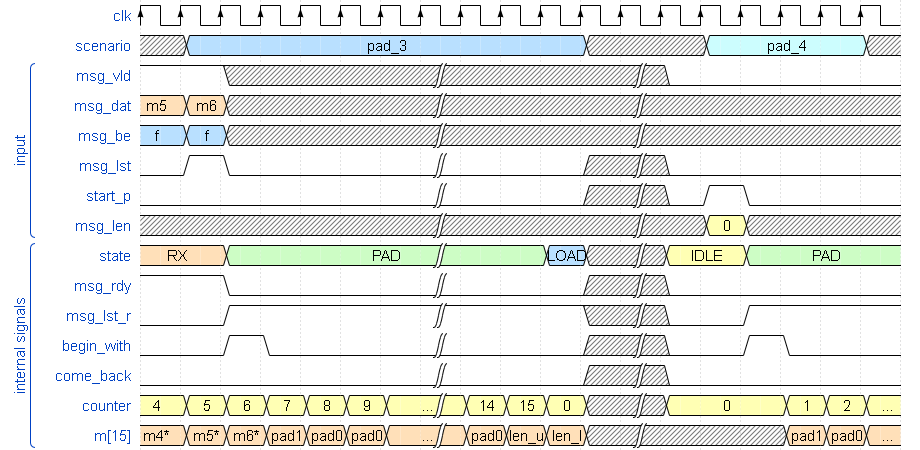
首先介紹PAD state時，每個clock rising edge會：

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] ()
6. Counter + 1
7. 當cnt = 15則清空cnt並轉換到LOAD
8. 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。

1. 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。

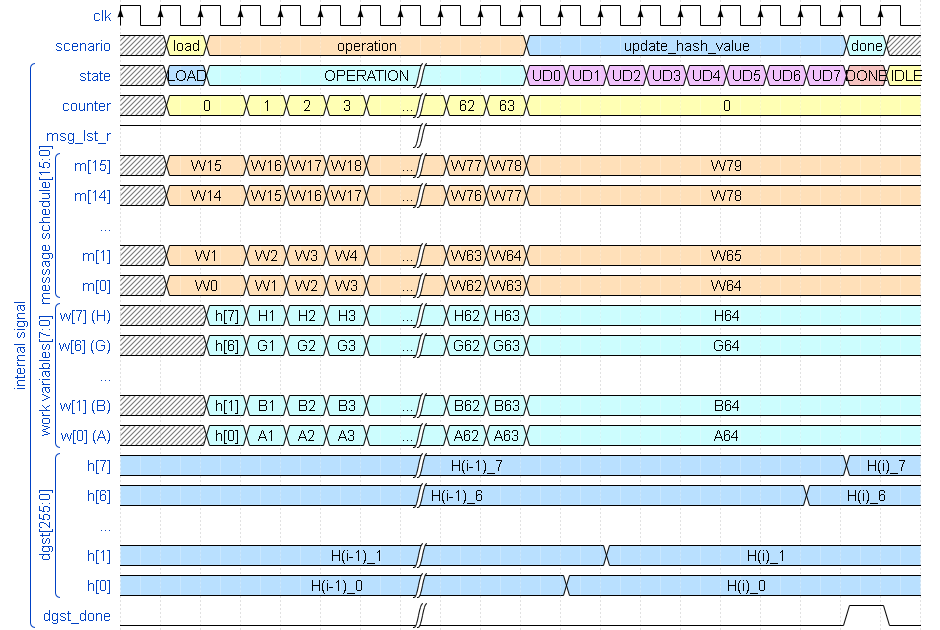


1. pad\_3

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。

1. 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。



1. load

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

1. operation

做message schedule產生~，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。

1. 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] ()，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。

1. 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 cycles

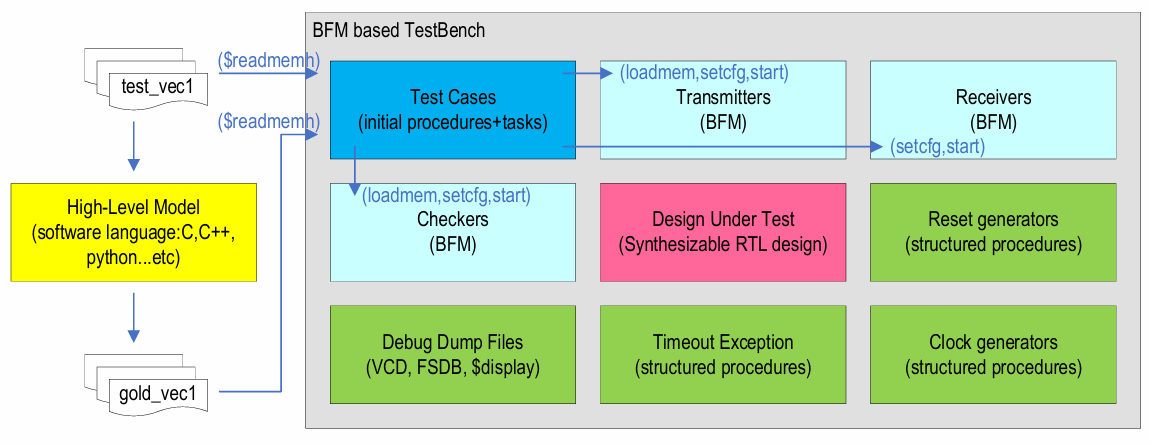
1. N=16n+1~13 or (16n+14且last word msg\_be=8/c/e)

從第一筆input到第一筆output需要 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)。

1. **Testbench**

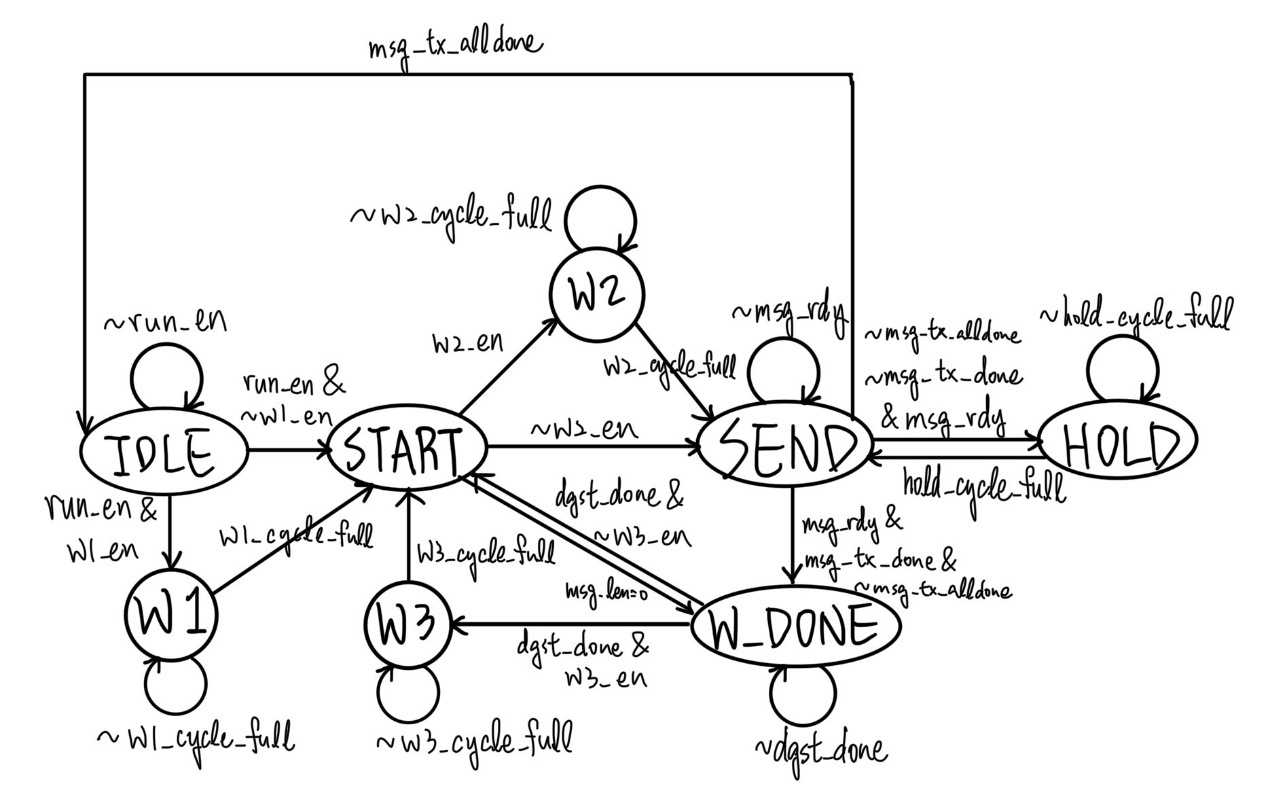
這次的testbench大多參考lecture 5的講義以及所附的例子，使用的是BFM的模式，架構如講義中的圖：



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。

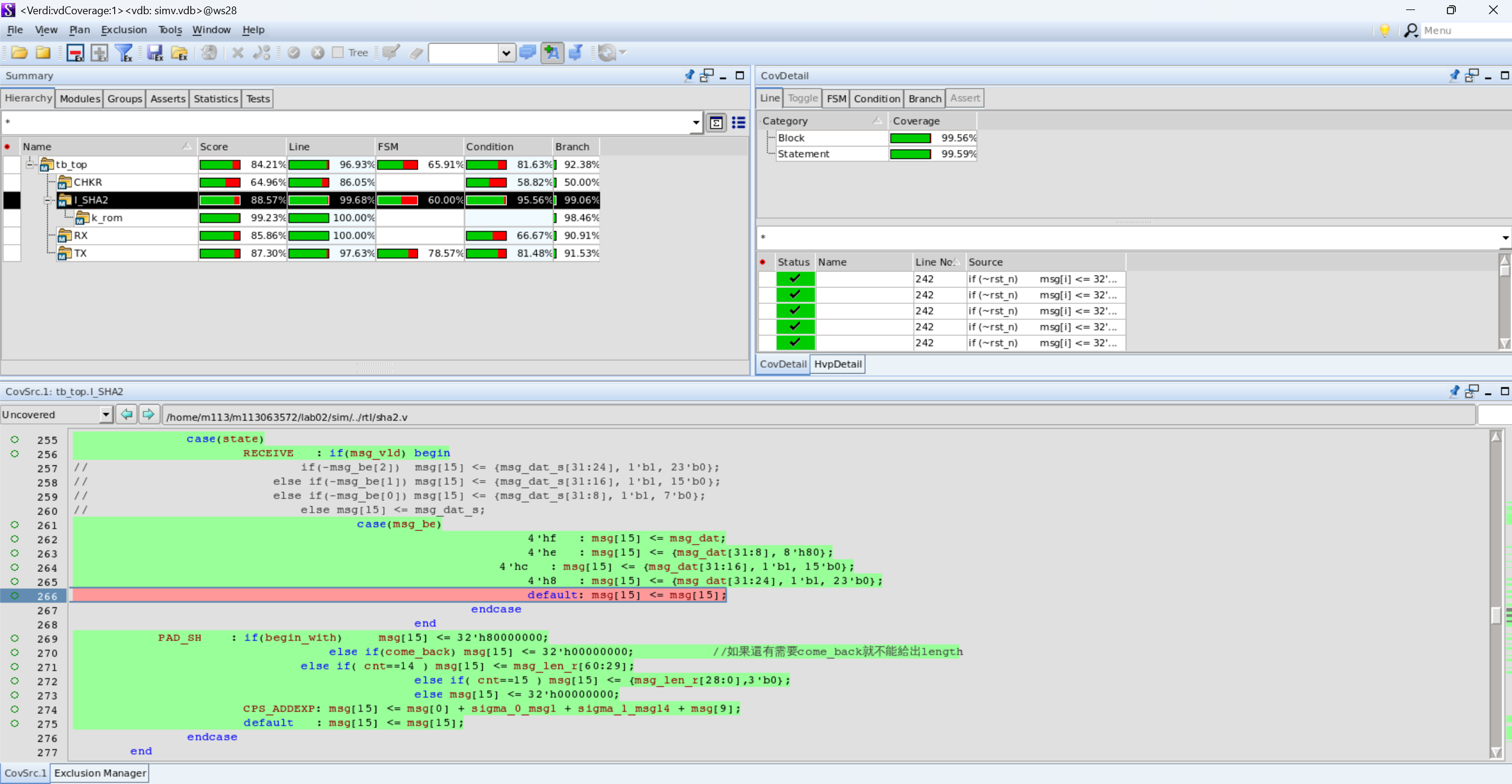
這次的receiver其實可以與checker直接合併，因為output protocol中receive端只能無條件接收DUT的output，但我還是保留了receiver的框架，並且在receiver中拉了一條dgst\_vld的訊號，當dgst\_done=1就拉起它，直到start\_p=1再拉下它，其存在目的是為了要驗證dgst在dgst\_done拉起後直到start\_p放下前是否都維持著正確值。因此checker檢查的不只有dgst\_done=1時的dgst是否為正確值，還有dgst\_vld=1時的dgst是否為正確值。

1. **Coverage report**

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

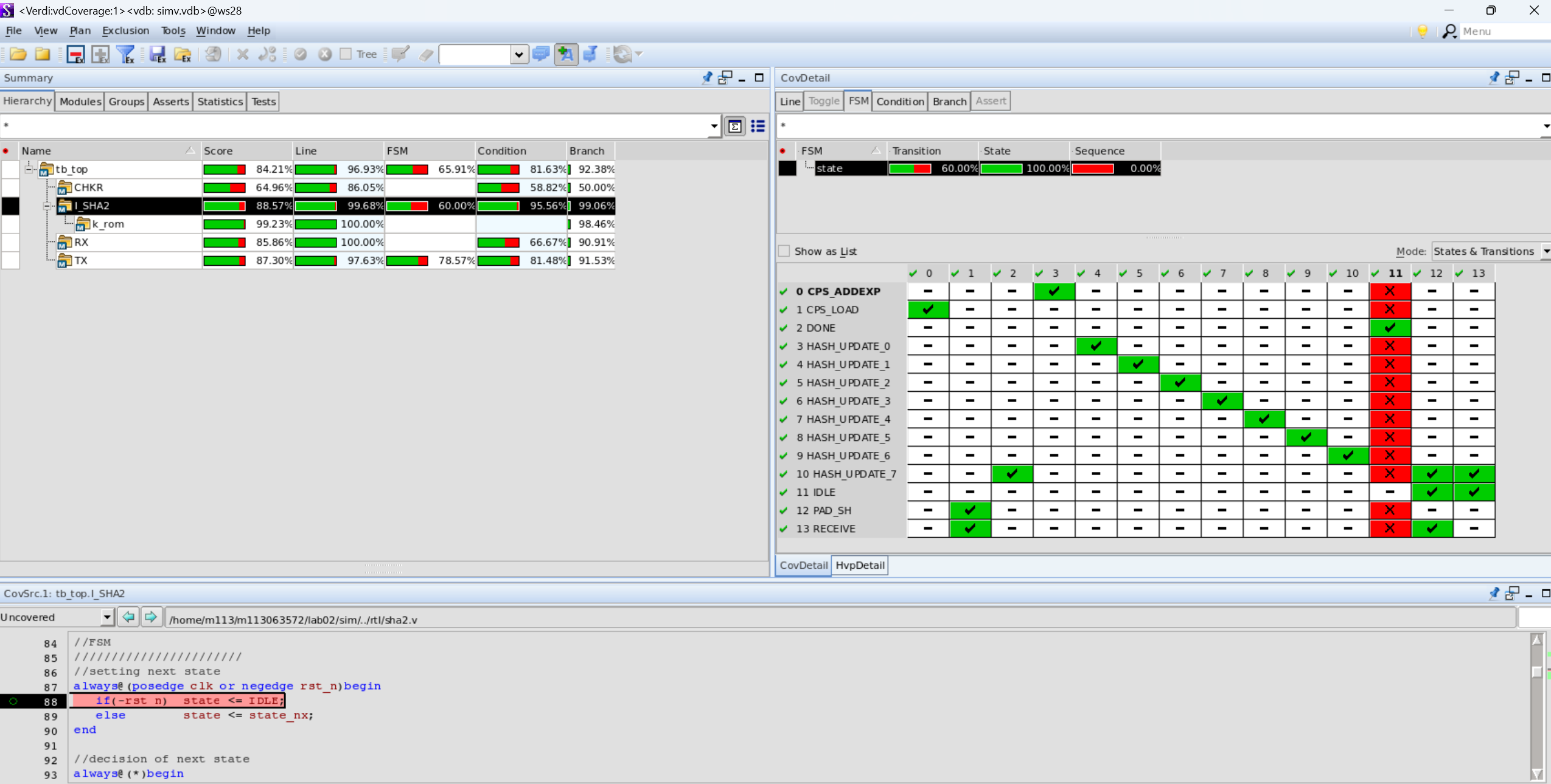
Line coverage:

In the case selection of msg\_be, default case did not occur, because TB provided valid msg\_be when msg\_vld asserts.



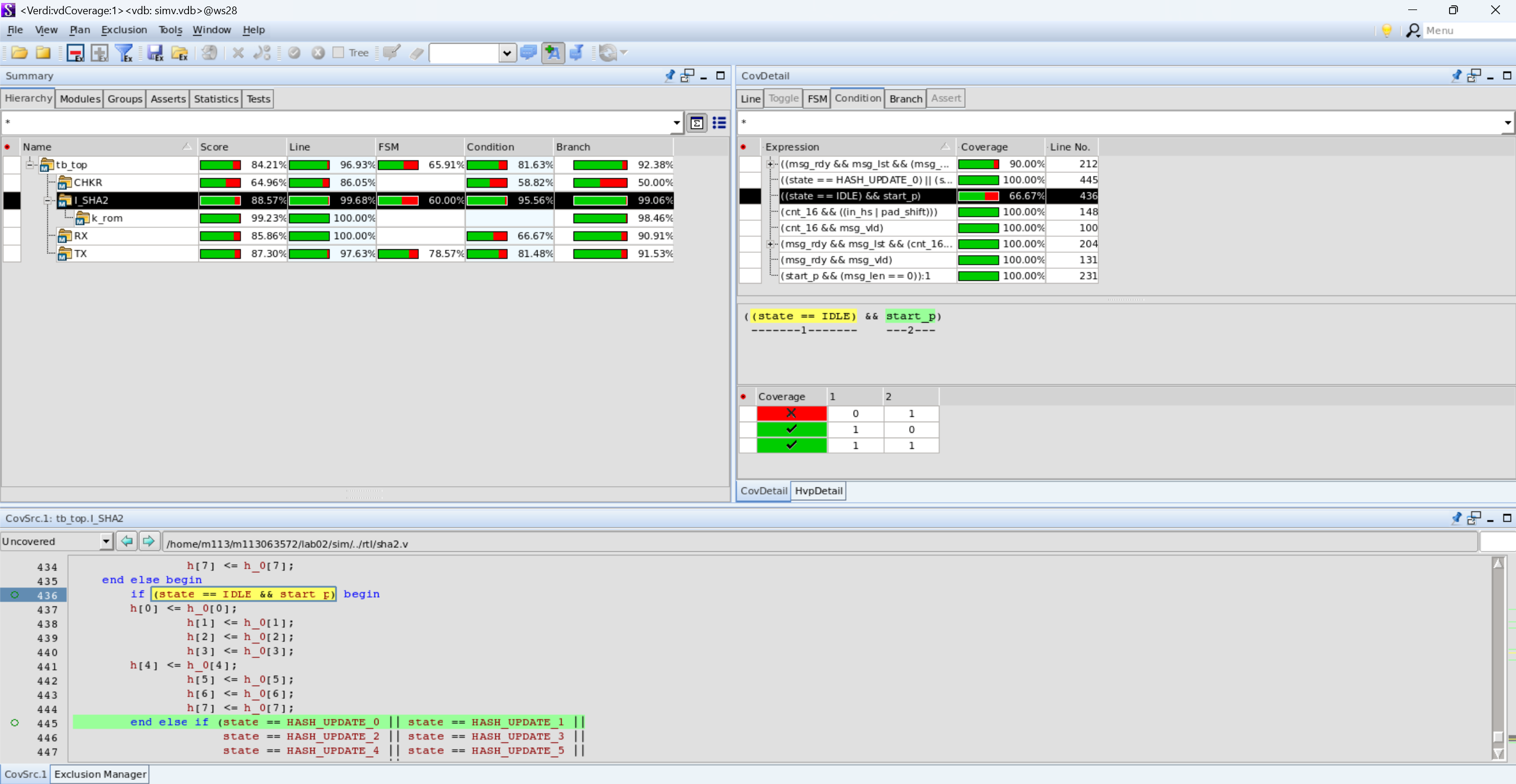
FSM coverage:

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.

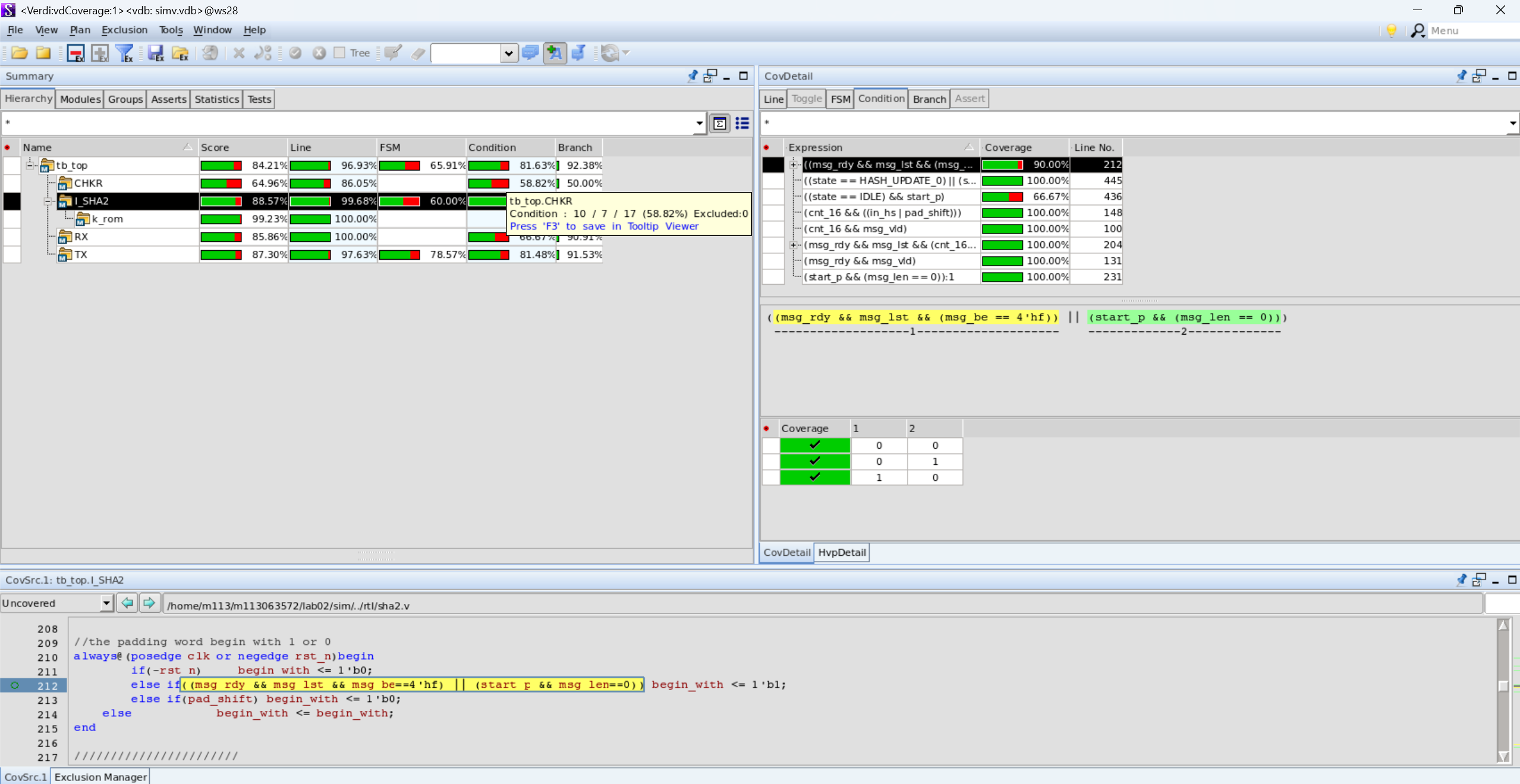


Condition coverage:

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)

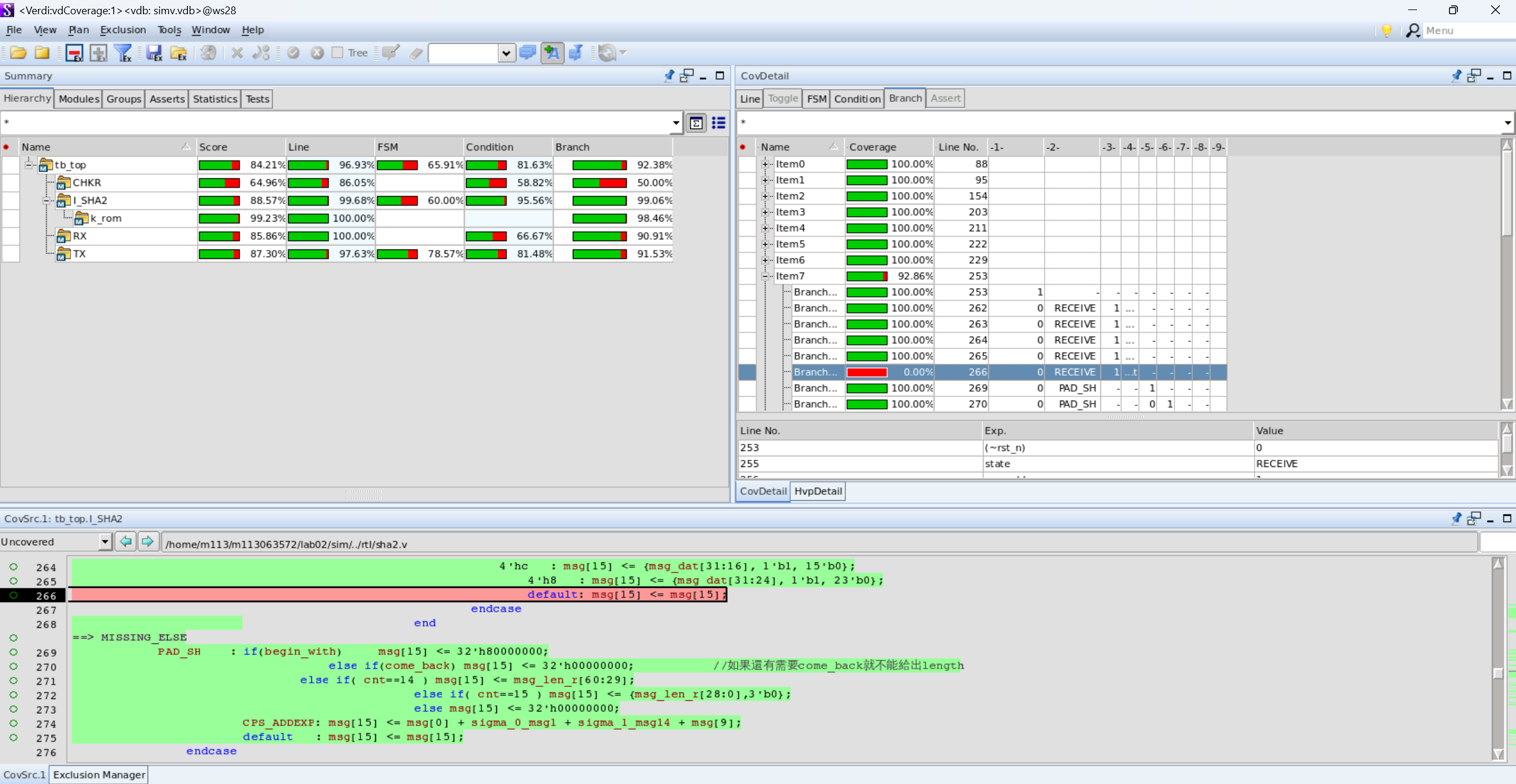


The two conditions that causes begin\_with to assert is one hot.



Branch coverage:

The same thing as described in the Line coverage section.



k\_rom is a submodule that stores the constants to . Since the address range is guaranteed to be within 0 to 63, no default value is required.

