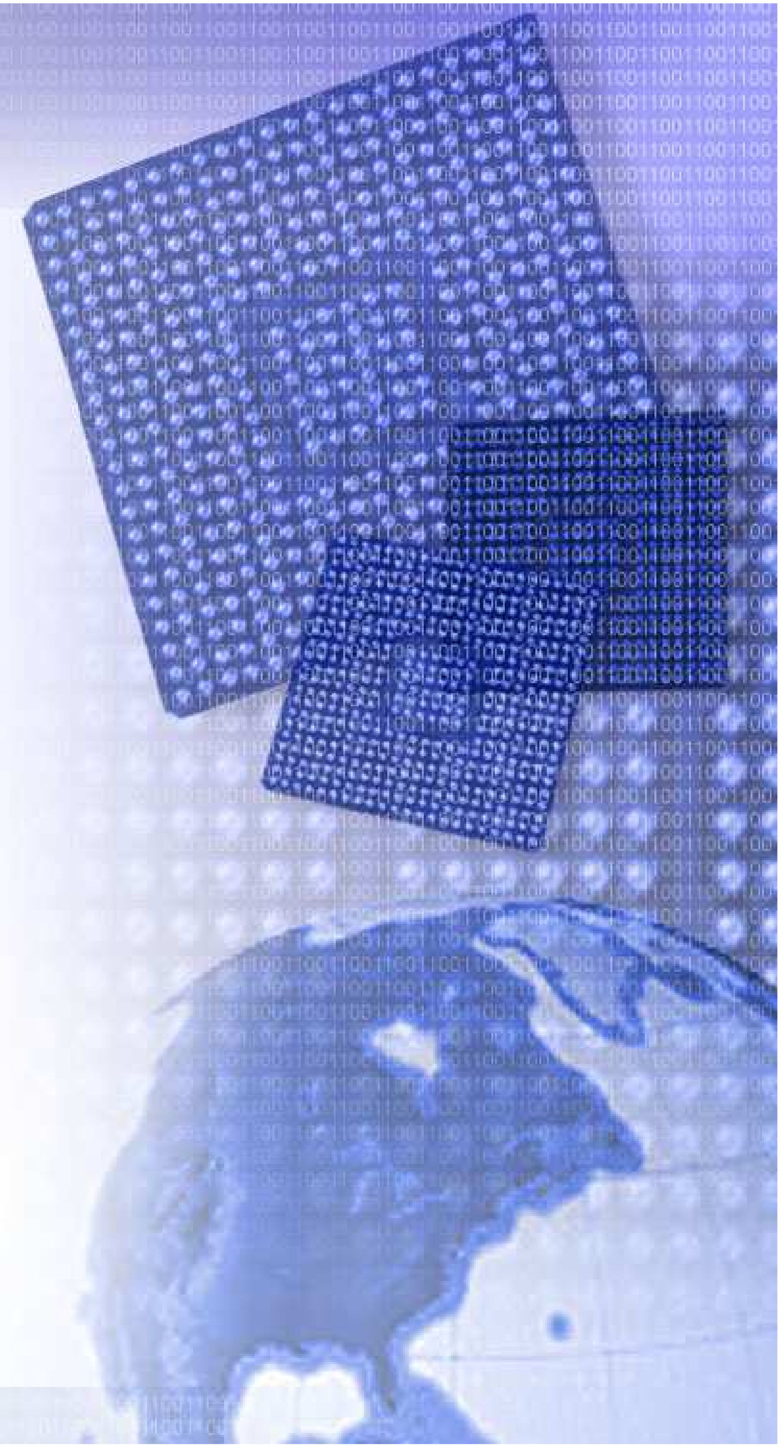
SHA-1 Algorithm for use with DS2432

Ken Chapman
Xilinx Ltd
April 2006

Rev.1



Limitations

Limited Warranty and Disclaimer. These designs are provided to you “as is”. Xilinx and its licensors make and you receive no warranties or conditions, express, implied, statutory or otherwise, and Xilinx specifically disclaims any implied warranties of merchantability, non-infringement, or fitness for a particular purpose. Xilinx does not warrant that the functions contained in these designs will meet your requirements, or that the operation of these designs will be uninterrupted or error free, or that defects in the Designs will be corrected. Furthermore, Xilinx does not warrant or make any representations regarding use or the results of the use of the designs in terms of correctness, accuracy, reliability, or otherwise.

Limitation of Liability. In no event will Xilinx or its licensors be liable for any loss of data, lost profits, cost or procurement of substitute goods or services, or for any special, incidental, consequential, or indirect damages arising from the use or operation of the designs or accompanying documentation, however caused and on any theory of liability. This limitation will apply even if Xilinx has been advised of the possibility of such damage. This limitation shall apply not-withstanding the failure of the essential purpose of any limited remedies herein.

This design module is **not** supported by general Xilinx Technical support as an official Xilinx Product.
Please refer any issues initially to the provider of the module.

Any problems or items felt of value in the continued improvement of KCPSM3 or this reference design would be gratefully received by the author.

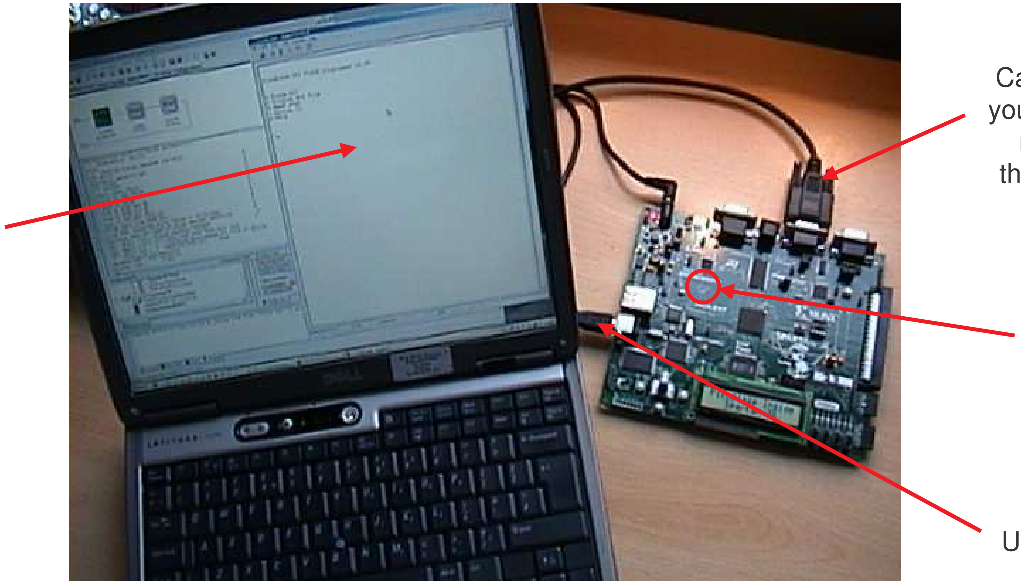
Ken Chapman
Senior Staff Engineer – Spartan Applications Specialist
email: chapman@xilinx.com

The author would also be pleased to hear from anyone using KCPSM3 or the UART macros with information about your application and how these macros have been useful.

Design Overview

This design builds on a previous reference design called “PicoBlaze DS2432 Communicator” and allows you to further investigate the Dallas Semiconductor DS2432 device which is a 1k-Bit Protected EEPROM with internal SHA-1 Engine. In this reference design, the focus is on those aspects of the device using the 64-bit secret and the SHA-1 algorithm.

This design also employs PicoBlaze to implement the SHA-1 algorithm as well as provide the 1-wire communication protocol and provide a simple user interface on your PC via the RS232 serial port (use HyperTerminal or similar). This design occupies less than 5% of the XC3S500E device. It is hoped that the design may be useful to anyone interested in using the DS2432 or other 1-wire devices in their own designs. PicoBlaze can easily be reprogrammed in this design using the JTAG_loader supplied with PicoBlaze.



HyperTerminal
(or similar)

RS232 Serial Cable.
Cable connects J9 on the board to
your PC serial port. For this you will
need a male to female straight
through cable (critically pin2-pin2,
pin3-pin3 and pin5-pin5).

Dallas Semiconductor
DS2432
1k-Bit Protected 1-Wire
EEPROM with SHA-1 Engine

USB cable for downloading the design or
changing the PicoBlaze program.

Hint – It is recommended that you obtain a copy of the DS2432 data sheet. Ideally print this document to refer to whist using this design and reading this description. It is particularly useful to have the flow charts available and the details concerning the SHA-1 algorithm and tables.

Hint – XAPP780 provides a design which can be used to provide copy protection for your own designs by exploiting the special properties of the DS2432.

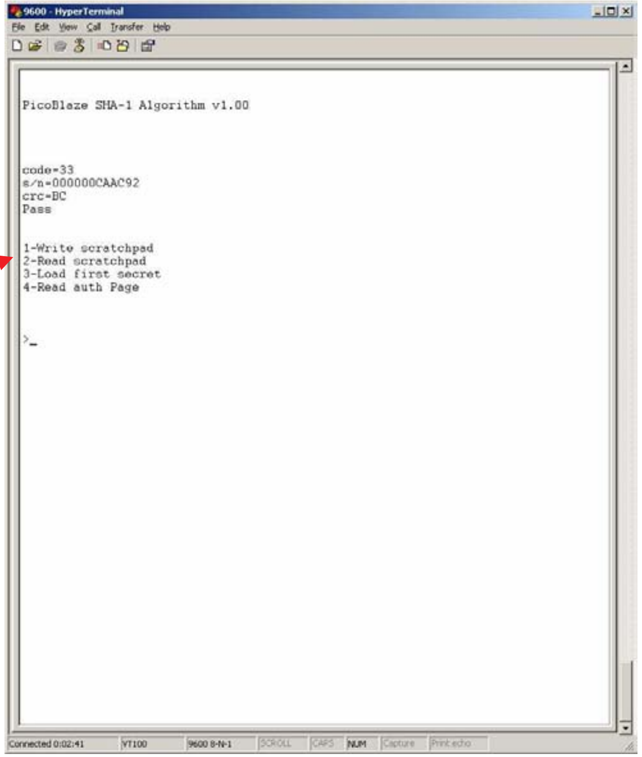
Using the Reference Design

This document is really in two sections. The first covers how to use the design 'AS IS' and in the process allow you to further study the DS2432 device with particular focus on the SHA-1 algorithm and the 64-bit secret. The second section covers particular aspects of the actual design implementation. These again focus on the SHA-1 algorithm requirements and are intended to help you follow the detailed comments to be read in the code supplied.

Configuring the Spartan-3E 'The Quick Way'!

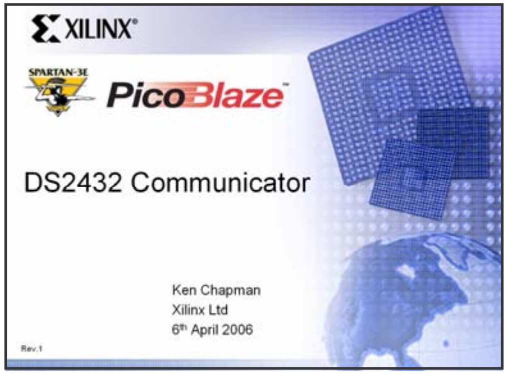
Unzip all the files provided into a directory.
Connect a suitable serial cable (see previous page).
Start a HyperTerminal (or similar) session using 9600 baud, 1 stop and no parity (see following pages).
Check you have the USB cable connected and the board is turned on.
Double click on the file **'install_PicoBlaze_DS2432_SHA1_Algorithm.bat'**.
This should open a DOS window and run iMPACT in batch mode to configure the Spartan device.

Your terminal session should indicate the design is working with a version number and simple menu.



```
PicoBlaze SHA-1 Algorithm v1.00



code=33
s/n=000000CAAC92
crc=BC
Pass


1-Write scratchpad
2-Read scratchpad
3-Load first secret
4-Read auth Page


>_
```

Alternatively use iMPACT manually to configure the XC3S500E device on the Spartan-3E Starter Kit via the USB cable with the BIT file provided.



XILINX
SPARTAN-3E
PicoBlaze
DS2432 Communicator
Ken Chapman
Xilinx Ltd
6th April 2006
Rev.1

Hint 1 – If you have not reviewed and tried the previous DS2432 reference design it is highly recommended that you do so now. The 'communicator' reference design will enable you to become familiar with the DS2432 device.

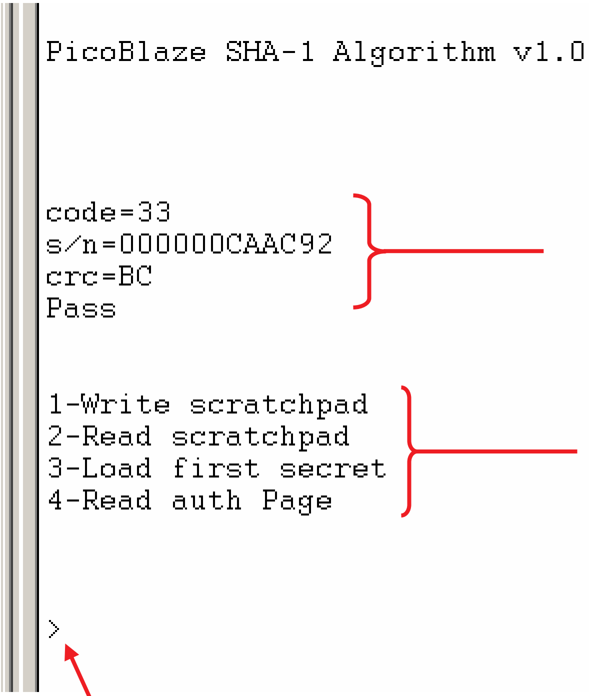
Hint 2 – The 'communicator' reference design also documents in detail how to set up HyperTerminal such that it will work as shown in the following pages.

Terminal settings required (see Hint 2)
9600 baud
1 stop bit
No parity
Append line feeds to line ends.
Do NOT echo locally typed characters.

Initial Response and Menu

After configuring the Spartan-3E with the design your terminal should display a simple message, some information about the DS2432 on your board and a menu with four options. These options correspond with four of the memory and SHA Functions offered by the DS2432 device.

From the 'communicator' reference design you will recall that these commands are only accessible after a master reset has been issued followed by one of the ROM commands. In this design, PicoBlaze automatically issues a master reset and checks for the corresponding 'presence pulse'. If no presence pulse is received it will keep issuing master reset pulses until it does. Then PicoBlaze will automatically follow on with a Read ROM command.



```
PicoBlaze SHA-1 Algorithm v1.00



code=33
s/n=000000CAAC92
crc=BC
Pass


1-Write scratchpad
2-Read scratchpad
3-Load first secret
4-Read auth Page



>
```

Message confirms design is loaded.
If nothing is displayed after this message, it means that the DS2432 is not responding to the master reset on your board.

This is the response to the read ROM command. It should show the family code of the DS2432 as '33' hex followed by the unique 48-bit serial number (or registration number) of the device fitted to your board. This number is used later in the SHA-1 algorithm. The 8-bit CRC is also tested by PicoBlaze and a 'Pass' or 'Fail' status be given.

The main purpose of this reference design is to exercise and experiment with the SHA-1 algorithm. Option '4' will execute the Read Authenticated Page command for memory page 0 which will result in the DS2432 generating a 160-bit message authentication code (MAC). This option will also invoke PicoBlaze to execute the SHA-1 algorithm such that the MAC received from the DS2432 can be compared.

Since the MAC can only match if the 64-bit secret programmed in the DS2432 is the same as that used by PicoBlaze, it is necessary to program the secret. Options '1', '2' and '3' help you to program a 64-bit secret into the DS2432 memory as well as define a 3-byte 'challenge' which is also fundamental to the SHA-1 algorithm.

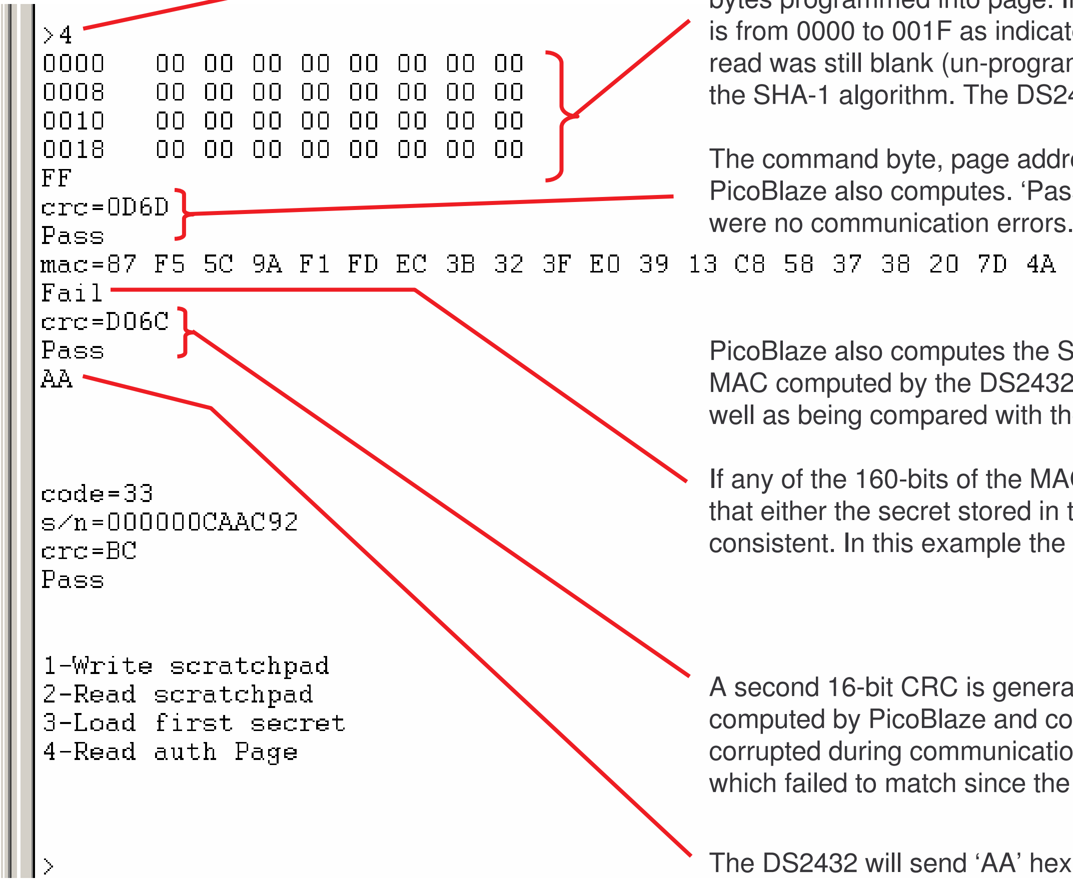
It is hoped that each option has been implemented in a such a way as to allow you to experiment and prove that the SHA-1 algorithm will only result in matching MAC's for identical secrets and challenges. However, this does mean that you should follow the next pages carefully to understand what each option does and does not do.

Enter your selection (1 to 4) to proceed.

SHA-1 Failure!

The PicoBlaze program has been assembled with the 64-bit secret code ‘01 23 45 67 89 AB CD EF’ (hex). Obviously this can be changed if you modify the program. Although this is rather silly choice for a ‘secret code’, it is highly unlikely that your DS2432 is programmed with the same secret unless you have already programmed it the same following the example shown in the ‘communicator’ design. So if you execute the read authenticated page command (menu option 4) the MAC received from the DS2432 will fail to match that computed by PicoBlaze as shown in this example below.

Option ‘4’



```
>4
0000    00 00 00 00 00 00 00 00
0008    00 00 00 00 00 00 00 00
0010    00 00 00 00 00 00 00 00
0018    00 00 00 00 00 00 00 00
FF
crc=0D6D
Pass
mac=87 F5 5C 9A F1 FD EC 3B 32 3F E0 39 13 C8 58 37 38 20 7D 4A
Fail
crc=D06C
Pass
AA


code=33
s/n=000000CAAC92
crc=BC
Pass


1-Write scratchpad
2-Read scratchpad
3-Load first secret
4-Read auth Page



>
```

The DS2432 responds to the Read Authenticated Page command by sending back the 32 bytes programmed into page. In this design, page 0 is always specified so the address range is from 0000 to 001F as indicated on the left of the screen. In this example the memory page read was still blank (un-programmed). The 32-bytes provide 50% of the information used by the SHA-1 algorithm. The DS2432 sends a single byte ‘FF’ after the data.

The command byte, page address and page data are used to form a 16-but CRC value which PicoBlaze also computes. ‘Pass’ indicates that the CRC value matched and shows that there were no communication errors. Good communications are vital if a MAC is ever to match.

PicoBlaze also computes the SHA-1 algorithm at the same time as the DS242. The 160bit MAC computed by the DS2432 is then read back as 20 bytes. These 20 bytes are displayed as well as being compared with the equivalent bytes of the PicoBlaze computed MAC.

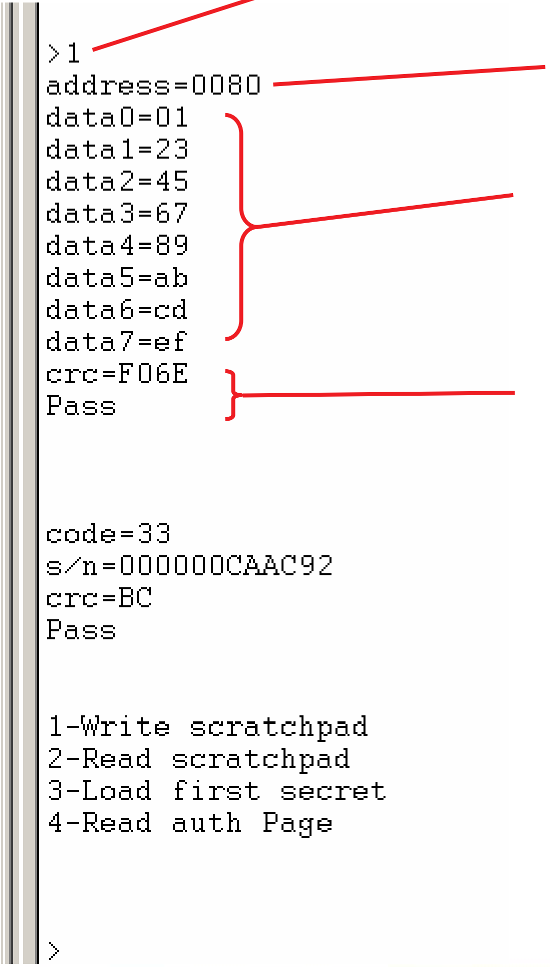
If any of the 160-bits of the MAC do not match, then a ‘Fail’ message is displayed. This indicates that either the secret stored in the DS2432 is not correct and/or the 3 byte challenge is not consistent. In this example the secret was definitely incorrect.

A second 16-bit CRC is generated based on the 20-bytes describing the MAC. This CRC is also computed by PicoBlaze and compared with the value received. This will indicate if the MAC was corrupted during communication. The ‘Pass’ in this example tells us that it really was the MAC which failed to match since the communication was reliable.

The DS2432 will send ‘AA’ hex until it is reset. PicoBlaze reads just one of these ‘AA’ bytes before repeating the master reset and read ROM commands.

Writing a Secret – part 1

To write your 64-bit secret into the DS2432 is a two stage process. First you need to put the secret into the DS2432 scratch pad memory using a Write Scratchpad command, and secondly, to copy the contents of the scratchpad into the EEPROM array using the Load First Secret. The next 3 pages show the programming of the 64-bit secret code '01 23 45 67 89 AB CD EF' required to match the reference design as supplied. A real application would definitely use a different and more obscure secret!



```
>1
address=0080
data0=01
data1=23
data2=45
data3=67
data4=89
data5=ab
data6=cd
data7=ef
crc=F06E
Pass


code=33
s/n=000000CAAC92
crc=BC
Pass


1-Write scratchpad
2-Read scratchpad
3-Load first secret
4-Read auth Page



>
```

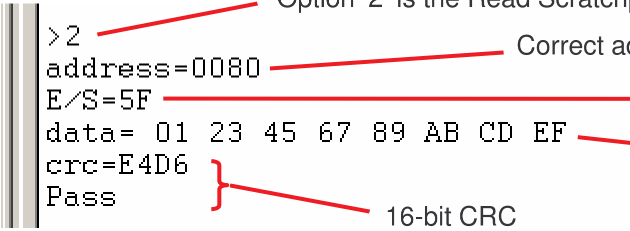
Option '1' is the Write Scratchpad command.

When prompted for the address you should enter 0080 which is the memory location for the secret. If you use any other address then the DS2432 will not allow your secret to be stored during the Write First Secret command.

You are then prompted to enter the 8 bytes defining the 64-bit secret. 'data0' is the least significant byte and will eventually be stored at address 0080. 'data7' is the most significant byte and will eventually be stored at address 0087. Enter each byte value as prompted. If you enter an illegal hexadecimal character you will be prompted to enter that byte again. If you make a mistake entering the secret, don't worry, jut repeat the write scratchpad command again.

The 16-bit CRC confirms that communication with the DS2432 is reliable.

Hint – It is a good idea to confirm your write of the scratchpad memory using the Read Scratchpad command (option 2). This lets you confirm that you entered the address and the data for the secret correctly before you actually copy it into the EEPROM array. This is particularly important since you can not directly read back and verify the secret (for obvious reasons!).



```
>2
address=0080
E/S=5F
data= 01 23 45 67 89 AB CD EF
crc=E4D6
Pass
```

Option '2' is the Read Scratchpad command.

Correct address of secret.

Note E/S register reports '5F'

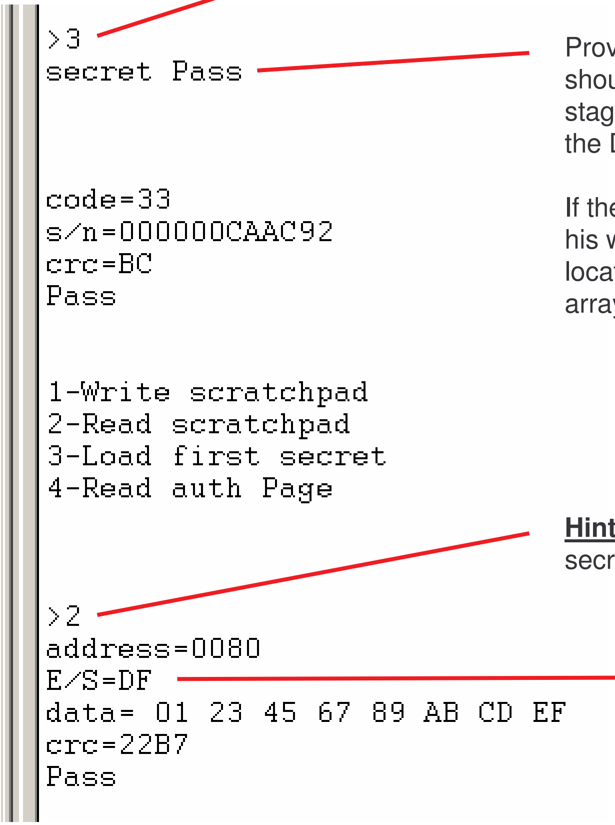
Correct data for secret.

16-bit CRC

Writing a Secret – part 2

When you are happy that the secret is correctly defined, you can move on to actually writing the secret into the memory array of the DS2432.

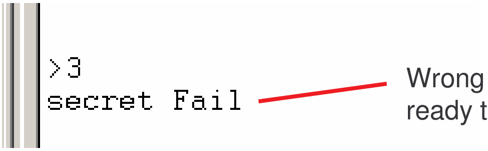
Option ‘3’ is the Write First Secret command.



```
>3
secret Pass



code=33
s/n=000000CAAC92
crc=BC
Pass


1-Write scratchpad
2-Read scratchpad
3-Load first secret
4-Read auth Page


>2
address=0080
E/S=DF
data= 01 23 45 67 89 AB CD EF
crc=22B7
Pass
```

Providing the address specified was correct and the E/S register of the DS2432 was ‘5F’ then the secret should successfully transfer into the non-volatile memory at the protected (un-readable) locations. At this stage, the only indication of success is if the DS2432 responds with ‘AA’ hex when read. PicoBlaze reads the DS2432 and reports ‘secret Pass’ if this is the case.

If the write to the array was not successful the DS2432 responds with ‘FF’ hex and PicoBlaze will indicate his with a ‘secret Fail’ message. This will happen if you did not specify the correct address (0080) for the location of the secret or the E/S register was not value ‘5F’ indicating that the DS2432 was ready to write the array in the first place.



```
>3
secret Fail
```

Wrong address or DS2432 not ready to write secret (E/S≠5F)

Hint – Use the Read Scratchpad command (option 2) to again confirm that you wrote the correct secret and see form the E/S register that the write was sucessful.

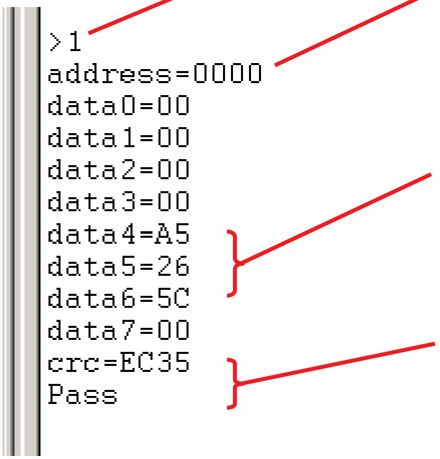
Note that the E/S register changes from ‘5F’ to ‘DF’ indicating a successful write of the non-volatile array.

Set the Challenge Bytes

The SHA-1 algorithm computes the MAC based on an input of 16 words of 32-bits. That is a total of 64 bytes which are made up as follows:-

32-bytes are the contents of the page being authenticated and provided at the start of the read authenticated page command.
8-bytes are the secret which no one can read directly and should normally be a complete secret unlike in this design!
6 bytes are the unique serial number of your DS2432 device which is known from the read ROM command.
15-bytes are constants (at least within the confines of the read authenticated page 0 command)
3-bytes are a challenge which we are about to set with a Write Scratchpad.

The 3-byte challenge is set using the scratchpad memory of the DS2432. This is easy to achieve with the Write Scratchpad command. Since the 32-bytes of page memory are not easy to change and all the other bytes are essentially fixed for the given device, the 3-byte challenge is the only quick way to cause different MACs to be generated by the SHA-1 algorithm. In practice, the 3-byte challenge allows 16,777,216 to be generated and this is useful in preventing 'copy-cat' or 'playback' attacks on security.



```
>1
address=0000
data0=00
data1=00
data2=00
data3=00
data4=A5
data5=26
data6=5C
data7=00
crc=EC35
Pass
```

Use option '1' to execute the Write Scratchpad command.

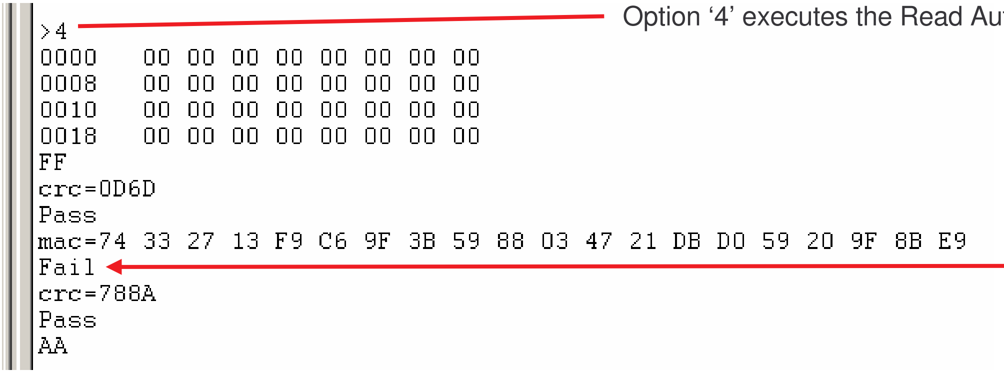
Any address can be used below because the data is never going to be copied into the memory array of the DS2432. However, the DS2432 will reject addresses above 0090 hex.

The challenge bytes are those in locations 4, 5 and 6 of scratchpad memory. All other bytes will be ignored by the SHA-1 algorithm.

The 16-bit CRC indicates if the communication was reliable.

Making SHA-1 Pass!

Assuming that the correct secret is now stored in the DS2432 and a 'challenge' has been set in the scratchpad memory, we can now expect the SHA-1 algorithms implemented in the DS2432 and PicoBlaze to compute the same MAC. This is where you may be disappointed to discover that the report is still of failure on your first attempt!



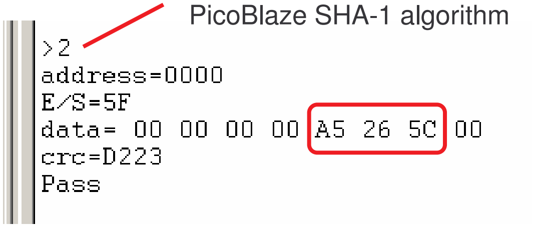
```
>4
0000   00 00 00 00 00 00 00 00
0008   00 00 00 00 00 00 00 00
0010   00 00 00 00 00 00 00 00
0018   00 00 00 00 00 00 00 00
FF
crc=0D6D
Pass
mac=74 33 27 13 F9 C6 9F 3B 59 88 03 47 21 DB D0 59 20 9F 8B E9
Fail
crc=788A
Pass
AA
```

Option '4' executes the Read Authenticated Page command

The MAC from the DS2432 (as displayed) has still failed to match with the PicoBlaze

The reason for this failure is purely down to the experimental properties of the PicoBlaze reference design. Indeed, this is a deliberate mechanism to help you prove that the challenge byes have an effect on the MAC. For PicoBlaze to generate the same MAC, it must also use the same 3-byte challenge. However, when you use the Write Scratchpad command (option '1') you only tell the DS2432 what the challenge is and the PicoBlaze SHA-1 algorithm continues to use an older value. In order to give the PicoBlaze SHA-1 algorithm the same challenge, you must execute a Read Scratchpad command which will cause the contents of the DS2432 scratchpad to be copied and used by PicoBlaze.

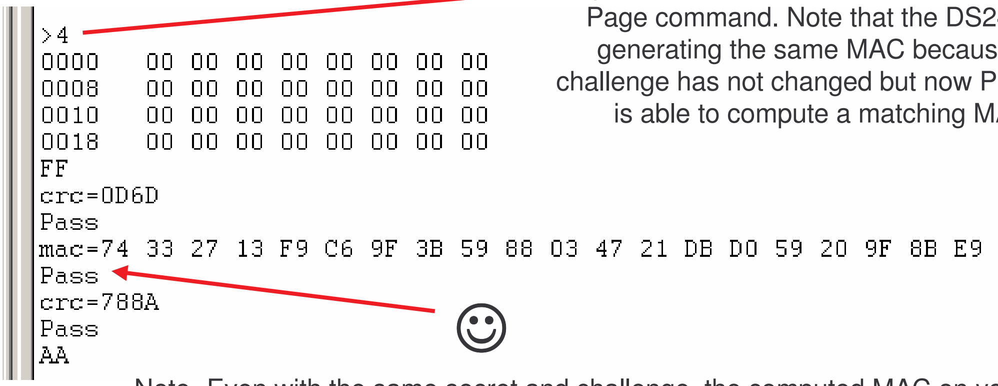
Option '2' reads the challenge and gives it to PicoBlaze SHA-1 algorithm



```
>2
address=0000
E/S=5F
data= 00 00 00 00 A5 26 5C 00
crc=D223
Pass
```

Hint – Read Scratchpad is always a good idea to confirm that the challenge is correctly set and both parties are working with the same information.

Option '4' executes the Read Authenticated Page command. Note that the DS2432 is generating the same MAC because the challenge has not changed but now PicoBlaze is able to compute a matching MAC.



```
>4
0000   00 00 00 00 00 00 00 00
0008   00 00 00 00 00 00 00 00
0010   00 00 00 00 00 00 00 00
0018   00 00 00 00 00 00 00 00
FF
crc=0D6D
Pass
mac=74 33 27 13 F9 C6 9F 3B 59 88 03 47 21 DB D0 59 20 9F 8B E9
Pass
crc=788A
Pass
AA
```

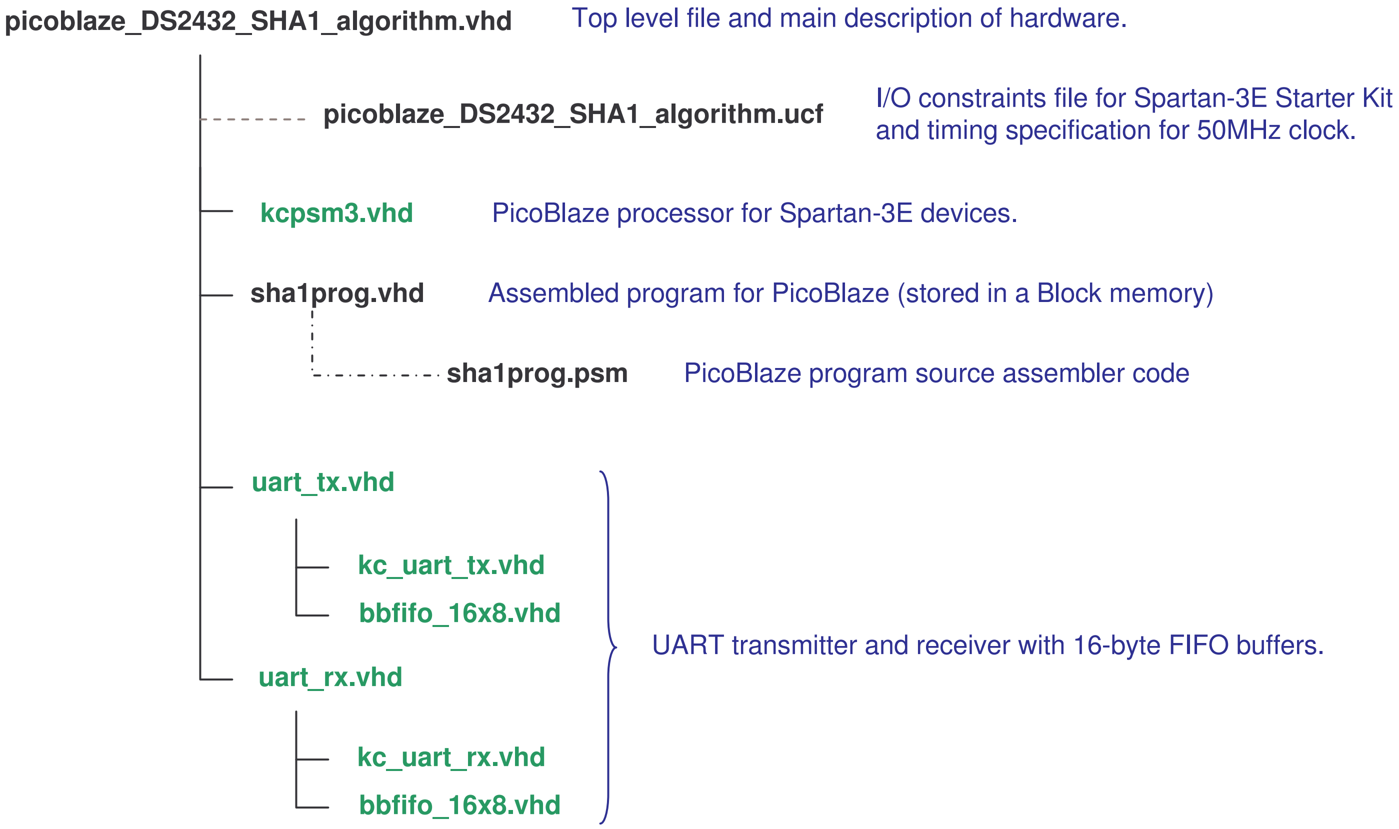
☺

Note- Even with the same secret and challenge, the computed MAC on your board will be different because of the unique serial number of each DS2432.

Design Files

For those interested in the actual design implementation, the following pages provide some details and an introduction to the source files provided. This description may be expanded in future to form a more complete reference design. As well as these notes, the VHDL and PicoBlaze PSM files contain many comments and descriptions describing the functionality. You are again recommended to study the ‘PicoBlaze DS2432 Communicator’ reference design before proceeding with this document which focuses specifically on the implementation of the SH1-algorithm.

The source files provided for the reference design are…..



picoblaze_DS2432_SHA1_algorithm.vhd Top level file and main description of hardware.

picoblaze_DS2432_SHA1_algorithm.ucf I/O constraints file for Spartan-3E Starter Kit and timing specification for 50MHz clock.

kcpsm3.vhd PicoBlaze processor for Spartan-3E devices.

sha1prog.vhd Assembled program for PicoBlaze (stored in a Block memory)

sha1prog.psm PicoBlaze program source assembler code

uart_tx.vhd

kc_uart_tx.vhd

bbfifo_16x8.vhd

uart_rx.vhd

kc_uart_rx.vhd

bbfifo_16x8.vhd

UART transmitter and receiver with 16-byte FIFO buffers.

Note: Files shown in **green** are not included with the reference design as they are all provided with PicoBlaze download. Please visit the PicoBlaze Web site for your free copy of PicoBlaze, assembler and documentation.

www.xilinx.com/picoblaze

PicoBlaze Design Size

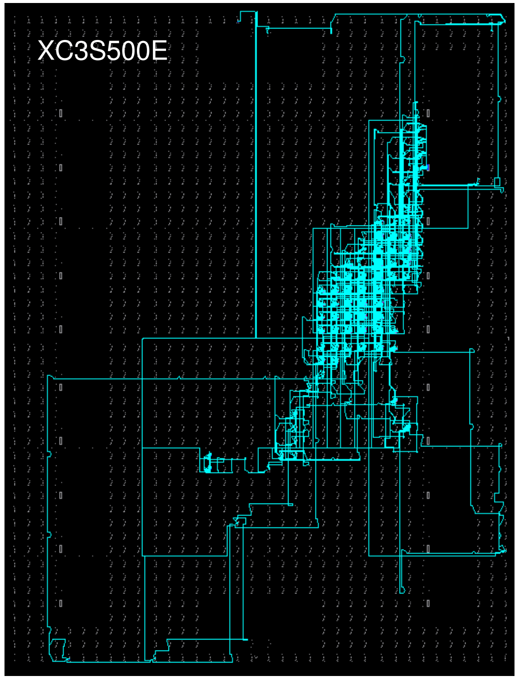
The images and statistics on this page show that the design occupies just 176 slices and 1 BRAM. This is only 3.8% of the slices and 5% of the BRAMs available in an XC3S500E device and would still be less than 19% of the slices in the smallest XC3S100E device.

MAP report

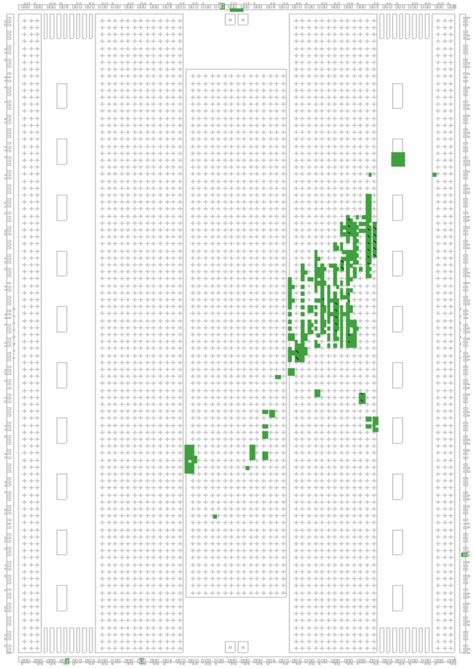
```
Number of occupied Slices:      176 out of    4,656    3%
Number of Block RAMs:             1 out of       20    5%

Total equivalent gate count for design:  80,878
```

PicoBlaze, the UART macros and 'W_t' buffer make extensive use of the distributed memory features of the Spartan-3E device leading to very high design efficiency. If this design was replicated to fill the XC3S500E device, it would represent the equivalent of over 1.5 million gates. Not bad for a device even marketing claims to be 500 thousand gates ☺

FPGA Editor view



Floorplanner view



Demands of the SHA-1 Algorithm

This functionality of this reference design is implemented almost entirely by a single PicoBlaze processor. In order to communicate with the DS2432 device it implements the 1-wire protocol and navigates the operational state machine associated with the commands. In many ways, these functions are ideal for PicoBlaze because they are very much ‘control’ and ‘state machine’ in nature. Another observation about these functions is that they only require a small number of variables to be handled for which the 8-bit registers are nearly always sufficient.

In looking at the SHA-1 algorithm we see a different kind of demand being placed on PicoBlaze. This is much more a ‘data processing’ application and begins to push the capability of what is really a controller. Fortunately there is no real demand to perform the SHA-1 algorithm at high speed and the algorithm is small enough to fit comfortably in the sit in the 1024 instructions of program space. The solution actually requires 219 instructions to describe the actual SHA-1 algorithm and this executes in approximately 2.9ms (at 50MHz) which is similar to the time taken by the DS2432.

However, there is one particular demand placed on PicoBlaze by the algorithm and that is the need to deal with more variables and data than it can hold internally. It is therefore necessary to add external memory via the input/output ports which the PicoBlaze program can then access as required. Of course, this ‘external’ memory is still inside the same Spartan-3E device.

Variables and Constants

The SHA-1 algorithm fundamentally a pair of processes with variations which must be executed 80 times (t=0 to t=79) to generate the Message Authentication Code (MAC). So the first variable required is the one which counts these iterations and this is within the range of a single register (‘sE’ is used).

One process involves five variables called ‘A’, ‘B’, ‘C’, ‘D’ and ‘E’ which are initialised with constants and eventually describe the MAC. These variables are 32-bits and require 4 bytes. That means that these variable alone require 20 bytes to store and this exceeds the 16 registers available in PicoBlaze. However, it is possible to store these variables in the 64-bytes of internal scratchpad memory and then use sets of 4 registers as temporary storage whilst manipulating them. There is also a requirement for four other constants referred to as ‘Kt’, but as with the initialisation of the variables these can be described within instructions and need no specific storage space.

The other process is used to generate a variable called ‘Wt’. This is also 32-bits and can be treated as a temporary value stored in registers. Whilst that is not an issue, the challenge is that each ‘Wt’ is computed from a pool of 16 variables which are again all 32-bits. That demands another 64-bytes of storage space which just isn’t available inside PicoBlaze. Because this is the critical demand and defines the hardware of the design, the following pages will focus on this aspect first.

W_t Buffer holds ‘M’ words

The SHA-1 algorithm takes as input 16 words of 32-bits. These words are called ‘M0’ through to ‘M15’ and are initialised in three different ways by the DS2432 depending on the command being executed (see DS2432 data sheet for details). This design implements Read Authenticated Page command, but obviously the ‘M’ words could be set for the other commands and this has no effect on the actual implementation of the SHA-1 algorithm which then reads them to produce a specific MAC.

Generic Table

	[31:24]	[23:16]	[15:8]	[7:0]
M0	M0_byte3	M0_byte2	M0_byte1	M0_byte0
M1	M1_byte3	M1_byte2	M1_byte1	M1_byte0
M2	M2_byte3	M2_byte2	M2_byte1	M2_byte0
M3	M3_byte3	M3_byte2	M3_byte1	M3_byte0
M4	M4_byte3	M4_byte2	M4_byte1	M4_byte0
M5	M5_byte3	M5_byte2	M5_byte1	M5_byte0
M6	M6_byte3	M6_byte2	M6_byte1	M6_byte0
M7	M7_byte3	M7_byte2	M7_byte1	M7_byte0
M8	M8_byte3	M8_byte2	M8_byte1	M8_byte0
M9	M9_byte3	M9_byte2	M9_byte1	M9_byte0
M10	M10_byte3	M10_byte2	M10_byte1	M10_byte0
M11	M11_byte3	M11_byte2	M11_byte1	M11_byte0
M12	M12_byte3	M12_byte2	M12_byte1	M12_byte0
M13	M13_byte3	M13_byte2	M13_byte1	M13_byte0
M14	M14_byte3	M14_byte2	M14_byte1	M14_byte0
M15	M15_byte3	M15_byte2	M15_byte1	M15_byte0

For the Read Authenticated Page 0 command the ‘M’ words are as follows…

	[31:24]	[23:16]	[15:8]	[7:0]
M0	secret0	secret1	secret2	secret3
M1	page_data0	page_data1	page_data2	page_data3
M2	page_data4	page_data5	page_data6	page_data7
M3	page_data8	page_data9	page_data10	page_data11
M4	page_data12	page_data13	page_data14	page_data15
M5	page_data16	page_data17	page_data18	page_data19
M6	page_data20	page_data21	page_data22	page_data23
M7	page_data24	page_data25	page_data26	page_data27
M8	page_data28	page_data29	page_data30	page_data31
M9	FF hex	FF hex	FF hex	FF hex
M10	40 hex	33 hex	Serial_No 0	Serial_No 1
M11	Serial_No 2	Serial_No 3	Serial_No 4	Serial_No 5
M12	secret4	secret5	secret6	secret7
M13	scratchpad4	scratchpad5	scratchpad6	80 hex
M14	00 hex	00 hex	00 hex	00 hex
M15	00 hex	00 hex	00 hex	B8 hex

What the table does show is that 64-byes of storage are required so that these ‘M’ words can be set up before calling the SHA-1 algorithm itself. During the first 16 iterations of the SHA-1 algorithm (t=0 to t=15) the temporary variable called ‘Wt’ is a simple assignment of each ‘M’ word.

For iterations t=0 to t=15

$$W_t = M_t$$

(e.g. $W_8 = M_8$)

This is obviously very easy to implement as it only requires the appropriate 4-bytes value to be read from the initial table of values. If this were the only use of this 64-byte storage then it would be possible to avoid it and create each ‘M’ word dynamically as part of the SHA-1 algorithm. However, the remaining 64 iterations require something more significant than this simple value assignment.

W_t Buffer holds ‘W’ words

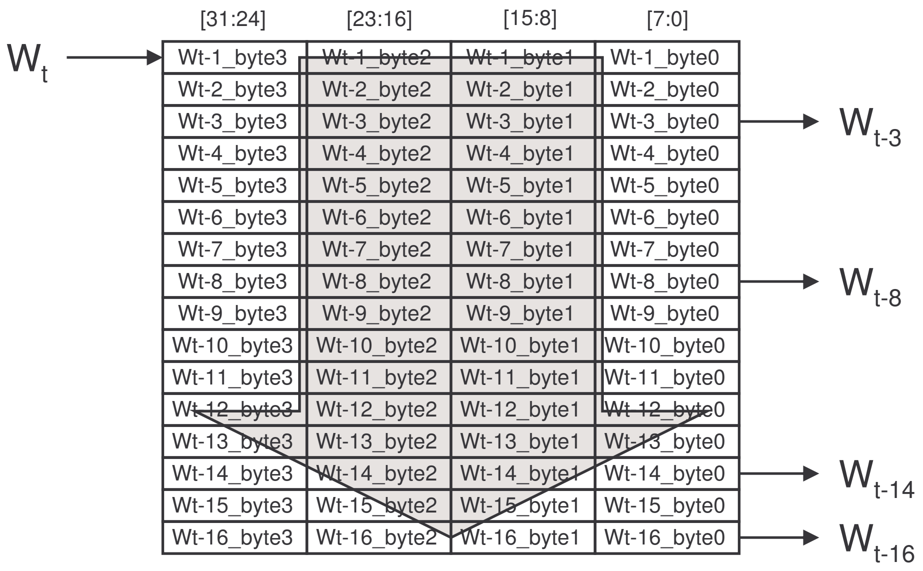
Each iteration of SHA-1 algorithm generates a ‘Wt’ word (32-bits) and we have seen that the first 16 of these are just copies of the original table of ‘M’ words. The remaining 64 ‘Wt’ words involve some computation and a history of previous values of ‘Wt’.

For iterations t=16 to t=79

$$W_t = \text{rotate\_left\_1\_place}(\ W_{t-3} \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16}\ )$$

e.g. $W_{24} = \text{rotate\_left\_1\_place}\ (\ W_{21} \text{ XOR } W_{16} \text{ XOR } W_{10} \text{ XOR } W_8\ )$

The EXCLUSIVE-OR operations are easy enough to achieve, as is the rotate left, but the key is to access the older ‘W’ words. As the suffix suggests in each case, it is necessary to access the value of Wt which were generated 3, 8, 14 and 16 iterations before. This again leads to the fundamental requirement for a buffer of 16 words or 64-bytes. Fortunately there is an opportunity to use the same buffer storage as that used to hold the 16 initial ‘M’ words because after they have been read during the fist 16 iterations they are not read directly again. Of course, the values of W_0 through W_{15} are actually the same as the original M0 through M15 values and will be used during iterations 16 to 31.

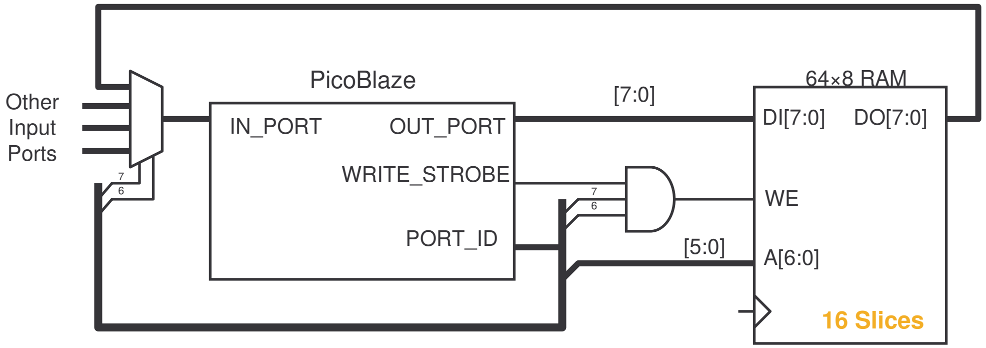


[31:24]	[23:16]	[15:8]	[7:0]
Wt-1_byte3	Wt-1_byte2	Wt-1_byte1	Wt-1_byte0
Wt-2_byte3	Wt-2_byte2	Wt-2_byte1	Wt-2_byte0
Wt-3_byte3	Wt-3_byte2	Wt-3_byte1	Wt-3_byte0
Wt-4_byte3	Wt-4_byte2	Wt-4_byte1	Wt-4_byte0
Wt-5_byte3	Wt-5_byte2	Wt-5_byte1	Wt-5_byte0
Wt-6_byte3	Wt-6_byte2	Wt-6_byte1	Wt-6_byte0
Wt-7_byte3	Wt-7_byte2	Wt-7_byte1	Wt-7_byte0
Wt-8_byte3	Wt-8_byte2	Wt-8_byte1	Wt-8_byte0
Wt-9_byte3	Wt-9_byte2	Wt-9_byte1	Wt-9_byte0
Wt-10_byte3	Wt-10_byte2	Wt-10_byte1	Wt-10_byte0
Wt-11_byte3	Wt-11_byte2	Wt-11_byte1	Wt-11_byte0
Wt-12_byte3	Wt-12_byte2	Wt-12_byte1	Wt-12_byte0
Wt-13_byte3	Wt-13_byte2	Wt-13_byte1	Wt-13_byte0
Wt-14_byte3	Wt-14_byte2	Wt-14_byte1	Wt-14_byte0
Wt-15_byte3	Wt-15_byte2	Wt-15_byte1	Wt-15_byte0
Wt-16_byte3	Wt-16_byte2	Wt-16_byte1	Wt-16_byte0

So in some way the 64-byte buffer must be arranged and controlled in such a way that it can be initialised with the ‘M’ words and then always hold the last 16 ‘Wt’ words in such a way that 4 of them can be read. As this diagram indicates, it is as if the contents of the memory must all move down by one place such that the new value can be stored and the oldest data is lost. This is very similar to a FIFO, but always with the same number of words stored and the ability to snoop internal values as well as the first to be written (oldest).

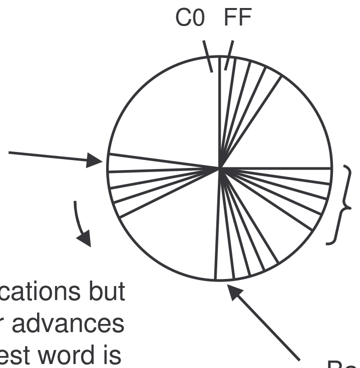
W_t Buffer using RAM

PicoBlaze is an 8-bit processor and the communication protocol of the DS2432 is also byte based (across a serial connection). It therefore makes total sense to implement the buffer as 64-bytes rather than 16-words. Memory (RAM) provides the most efficient way to implement the 16 word buffer (64-bytes). Spartan devices contain Block Memories each of which provide 2k-bytes and would be excessive and wasteful in this application. So in this case, the smaller distributed memory option is far more attractive and efficient. In fact this is the very same way in which the scratch pad memory inside PicoBlaze is implemented. Distributed memory is ability to use the look-up tables (LUTs) contained in the slices of a Configurable Logic Blocks (CLB) as RAM. Each LUT can implement a 16× 1-bit memory so when the four LUTs contained in two 'slices' are combined you have a 64× 1-bit memory. Therefore 16 slices can implement a 64 byte RAM and would be ideal for the W_t buffer.



A simple 64-byte memory has 6 address lines. This simplified circuit diagram shows how this can be mapped to the PicoBlaze port address range C0 to FF. The use if input/output instructions with indirect addressing (e.g. OUTPUT s5,(s8)) will need to be used to write and read to different locations during each iteration of the algorithm.

There is absolutely nothing wrong with this approach although it does becomes the responsibility of the PicoBlaze software to keep track of where each data word was stored. A 'cyclic buffer' would be implemented in which the newest data is always written to the location of the oldest data. This requires a 'memory pointer' to be maintained for which a single register would be suitable. Adding to the software complicity is the requirement to read back the 'W' words written 3, 8, 14 and 16 iterations before which requires some address computation in order to access the correct data.



Base
Memory
pointer to
W_{t-16}

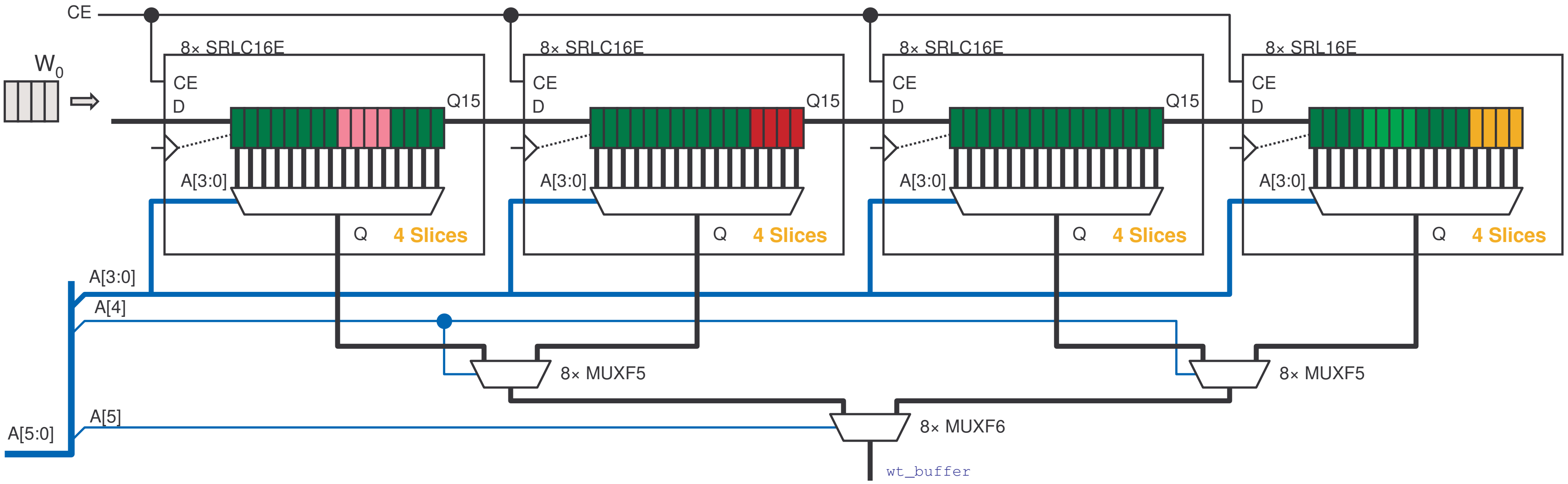
4 locations hold the bytes
for a given 'W_{t-n}' word

Data remains in same locations but the base memory pointer advances by 4 locations as the oldest word is over-written by the newest word.

Base + offset
allows older 'W_t' value to be read.
This must wrap round to C0 from
FF in this memory map.

W_t Buffer Using SRL Technique

Although the simple memory approach is valid, it is possible to exploit distributed memory to provide PicoBlaze with an application specific memory structure. In this alternative implementation, the 64-bytes will be organised as a shift register which completely obviates the requirement to address the memory structure; it will naturally allow each new ‘Wt’ word to be shifted in and cause the oldest ‘Wt-16’ word to be pushed out (and lost). Spartan LUTs can be configured to provide a shift register up to 16-bits long using the SRL16E mode. Additionally, the SRLC16E mode implements a 16-bit shift register with a secondary tapping point. The diagram below shows how these are combined with dedicated multiplexers to implement a byte-wide shift register with a length of 64 which can be read from any location. This takes the same 16-slices as normal RAM.



W_{t-3}
A[5:0] = 08 to 0B

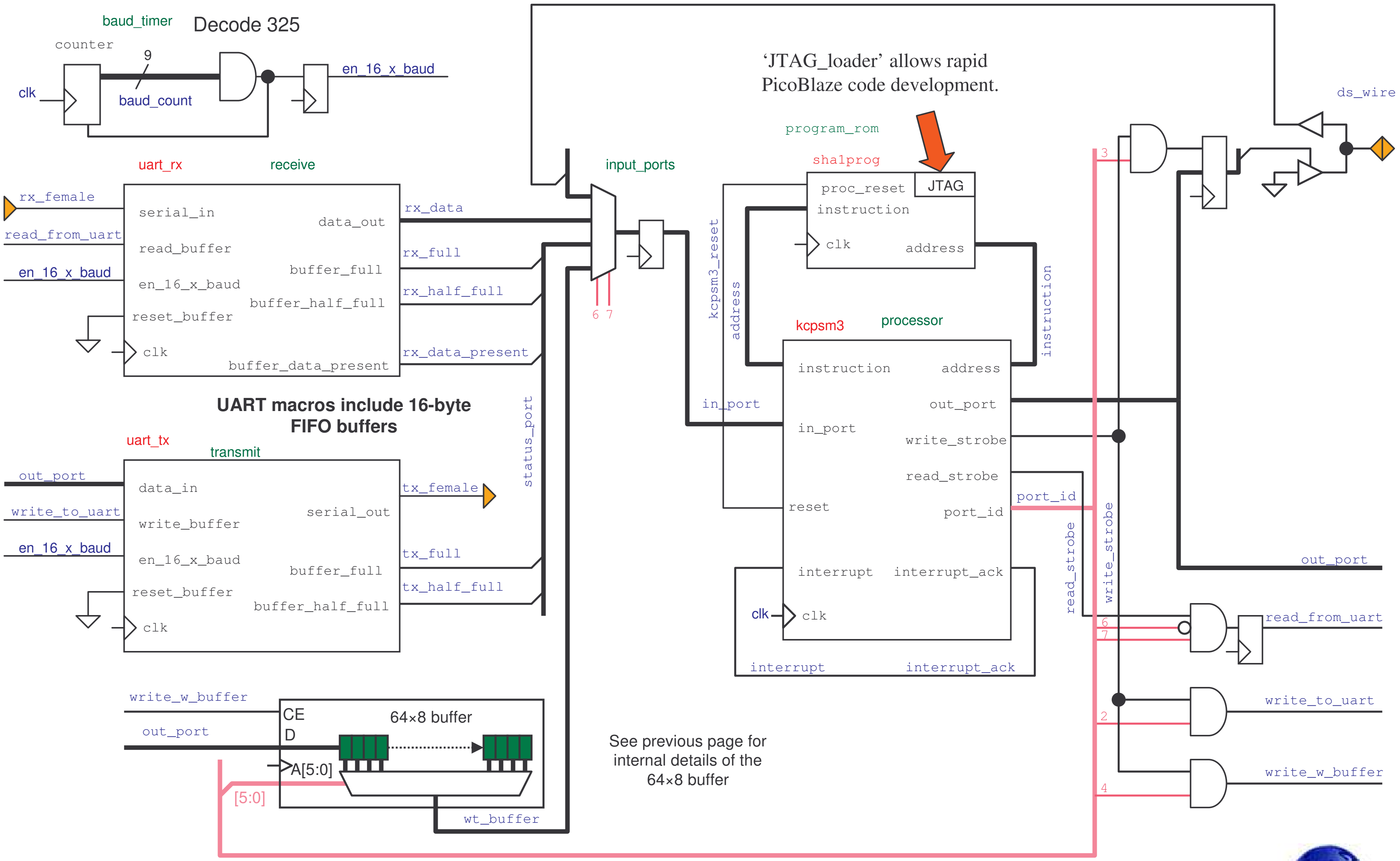
W_{t-8}
A[5:0] = 1C to 1F

W_{t-14}
A[5:0] = 34 to 37

W_{t-16}
A[5:0]= 3C to 3F

Each ‘W’ word is stored as 4 bytes. Providing that all 4 bytes of the new ‘W_0’ word are shifted in, then all the older words move along so that they can always be read from the same physical addresses. The initial ‘M’ words will also need to be stored by writing them in ascending order. In fact, this is a direct implementation of the buffer shown as a table on page 15 and can be called a ‘history buffer’.

PicoBlaze Circuit Diagram



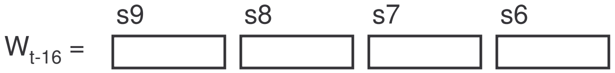
Software Access to W_t Buffer

```
CONSTANT W_word_write_port, 10          ;Write byte
;
CONSTANT Wt_minus3_byte0_read_port, 08  ;Read of Wt-3
CONSTANT Wt_minus3_byte1_read_port, 09
CONSTANT Wt_minus3_byte2_read_port, 0A
CONSTANT Wt_minus3_byte3_read_port, 0B
;
CONSTANT Wt_minus8_byte0_read_port, 1C  ;Read of Wt-8
CONSTANT Wt_minus8_byte1_read_port, 1D
CONSTANT Wt_minus8_byte2_read_port, 1E
CONSTANT Wt_minus8_byte3_read_port, 1F
;
CONSTANT Wt_minus14_byte0_read_port, 34 ;Read of Wt-14
CONSTANT Wt_minus14_byte1_read_port, 35
CONSTANT Wt_minus14_byte2_read_port, 36
CONSTANT Wt_minus14_byte3_read_port, 37
;
CONSTANT Wt_minus16_byte0_read_port, 3C ;Read of Wt-16
CONSTANT Wt_minus16_byte1_read_port, 3D
CONSTANT Wt_minus16_byte2_read_port, 3E
CONSTANT Wt_minus16_byte3_read_port, 3F
```

The PicoBlaze software completes the processing of ‘Wt’ words by interacting with the external history buffer.

The SRL16E approach to implementing the buffer means that input/output instructions using direct addressing are possible. The code provides defines each address of interest as a constant to make the process code easier to understand and to make the whole design easier to maintain.

Using the constants, it is a simple task to copy ‘Wt’ words from the external buffer into a set of internal registers. In this case the set of registers [s9,s8,s7,s6] are used to read the oldest word W_{t-16}



```
          INPUT s9, Wt_minus16_byte3_read_port
          INPUT s8, Wt_minus16_byte2_read_port
          INPUT s7, Wt_minus16_byte1_read_port
          INPUT s6, Wt_minus16_byte0_read_port
          COMPARE sE, 10
          JUMP C, store_Wt
```

The first 16 iterations only require the ‘M’ word to be copied directly to form W_0 through to W_{15}. The register ‘sE’ is used to count the iterations and enables this selection to be made.

```
store_Wt: OUTPUT s9, W_word_write_port
          OUTPUT s8, W_word_write_port
          OUTPUT s7, W_word_write_port
          OUTPUT s6, W_word_write_port
```

Storing the new ‘Wt’ word in the buffer only requires the 4 bytes to be written to the single port. It is vital that the 32-bit word is consistently written with the significant byte first for the read locations to correlate.

Computing W_t Values

[s9,s8,s7,s6]

```
INPUT s0, Wt_minus14_byte3_read_port
XOR s9, s0
INPUT s0, Wt_minus14_byte2_read_port
XOR s8, s0
INPUT s0, Wt_minus14_byte1_read_port
XOR s7, s0
INPUT s0, Wt_minus14_byte0_read_port
XOR s6, s0
INPUT s0, Wt_minus8_byte3_read_port
XOR s9, s0
INPUT s0, Wt_minus8_byte2_read_port
XOR s8, s0
INPUT s0, Wt_minus8_byte1_read_port
XOR s7, s0
INPUT s0, Wt_minus8_byte0_read_port
XOR s6, s0
INPUT s0, Wt_minus3_byte3_read_port
XOR s9, s0
INPUT s0, Wt_minus3_byte2_read_port
XOR s8, s0
INPUT s0, Wt_minus3_byte1_read_port
XOR s7, s0
INPUT s0, Wt_minus3_byte0_read_port
XOR s6, s0
CALL rotate_word_left
```

XOR W_{t-14}

XOR W_{t-8}

XOR W_{t-3}

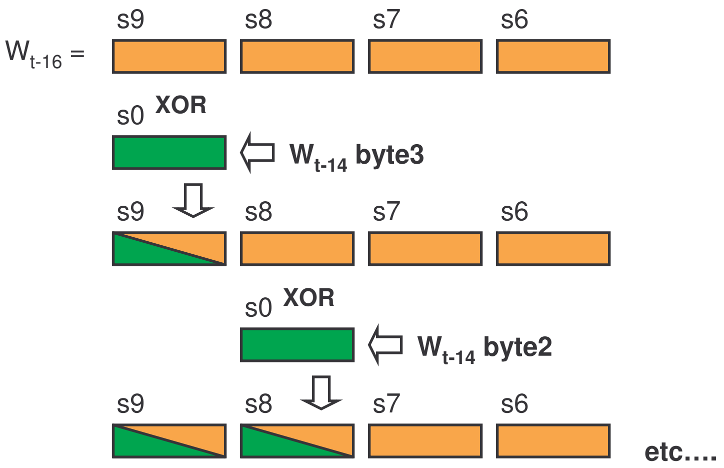
[s9,s8,s7,s6]

```
rotate_word_left: TEST s9, 80
                  SLA s6
                  SLA s7
                  SLA s8
                  SLA s9
                  RETURN
```

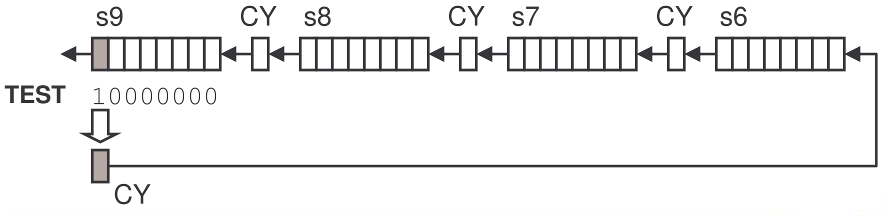
During iterations t=16 to t=79 (sE register contents = 11 to 4F hex) the following equation needs to be implemented…

$$W_t = \text{rotate\_left\_1\_place}(\ W_{t-3} \text{ XOR } W_{t-8} \text{ XOR } W_{t-14} \text{ XOR } W_{t-16}\ )$$

The Exclusive-OR is a bitwise operation so this can be conducted separately on each byte of the 32-bit word. From the previous page we saw that the value of W_{t-16} was loaded into the register set [s9,s8,s7,s6]. It is in these registers that the final value of W_t will be formed by repeatedly reading a byte from the external buffer and executing an XOR with the appropriate register. The following diagram illustrates the first four instructions of this process.



The 32-bit rotate left is achieved by first copying the MSB into the carry flag using a TEST instruction and then performing a shift left (with carry) on each byte starting with the least significant.

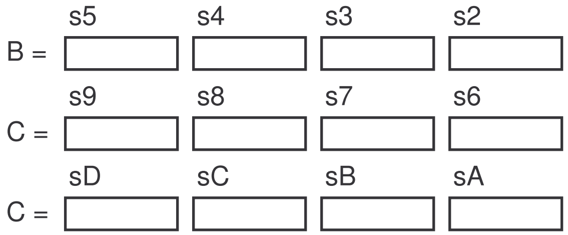


MAC Variables and Processing

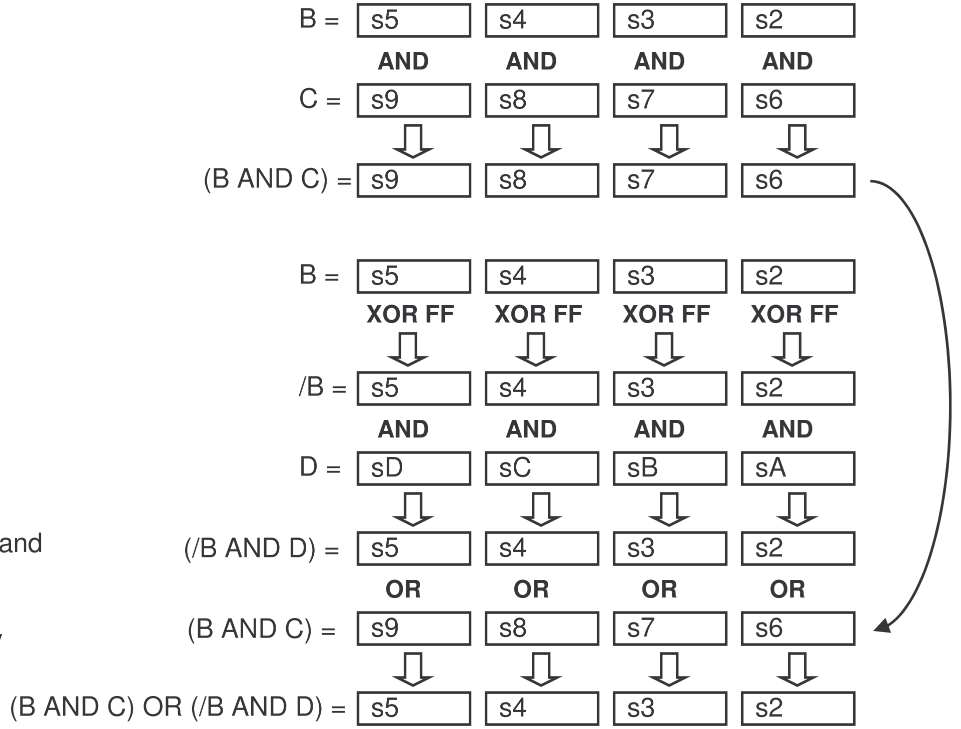
The 5 words used to form the message Authentication Code (MAC) are stored in PicoBlaze Scratch pad memory. Duing the implementation of the main SHA-1 process, these values are copied into registers where they are manipulated. Since each word is 32-bit, the temporary copies and all manipulations are carried out using sets of 4 registers. This is clearly seen in the function ‘f(B,C,D)’ which takes 3 different forms depending on the iteration (t).

```
CONSTANT var_A0, 08   ;Variable 'A'
CONSTANT var_A1, 09
CONSTANT var_A2, 0A
CONSTANT var_A3, 0B
;
CONSTANT var_B0, 0C   ;Variable 'B'
CONSTANT var_B1, 0D
CONSTANT var_B2, 0E
CONSTANT var_B3, 0F
;
CONSTANT var_C0, 10   ;Variable 'C'
CONSTANT var_C1, 11
CONSTANT var_C2, 12
CONSTANT var_C3, 13
;
CONSTANT var_D0, 14  ;Variable 'D'
CONSTANT var_D1, 15
CONSTANT var_D2, 16
CONSTANT var_D3, 17
;
CONSTANT var_E0, 18  ;Variable 'E'
CONSTANT var_E1, 19
CONSTANT var_E2, 1A
CONSTANT var_E3, 1B
```

```
FETCH s9, var_C3
FETCH s8, var_C2
FETCH s7, var_C1
FETCH s6, var_C0
```



The required variation of the function is then selected (using value in ‘sE’)and decomposed into sections. In this example the function is (B and C) or ((not B) and D) is shown…



```
AND s9, s5
AND s8, s4
AND s7, s3
AND s6, s2




XOR s5, FF
XOR s4, FF
XOR s3, FF
XOR s2, FF

AND s5, sD
AND s4, sC
AND s3, sB
AND s2, sA

OR s5, s9
OR s4, s8
OR s3, s7
OR s2, s6
```

Hint – The code supplied contains many comments and descriptions for the whole algorithm. It is hoped that these pages have introduced the coding style and special requirements of the ‘Wt’ buffer such that they can be understood fully.