DES in Four Days using Behavioural Modeling & Synthesis

Peter R. Wilson and Andrew D. Brown

School of Electronics and Copmputer Science, The University of Southampton, Southampton, UK

prw@ecs.soton.ac.uk & adb@ecs.soton.ac.uk

Abstract—**The experience of implementing the Data Encryption Standard (DES) using high level VHDL and behavioral synthesis is described. It is shown that it is possible to describe an algorithm which is notoriously low-level such that it is both readable and synthesizable using behavioral modelling appropriate for behavioural synthesis. The paper also discusses typical design issues that arise when working at the behavioral level and shows that human insight is still necessary to achieve the best possible results. However, this insight is brought to bear at a high level — which is what humans are good at — whilst the synthesis system provides "decision support" and optimisation — which is what software and computers are good at.**

I. INTRODUCTION

This paper describes the experience of designing a Data Encryption Standard (DES) core [1] in Electronic Code Book (ECB) mode [2] using behavioural VHDL and the MOODS behavioral synthesis system [3].

The main objective was to write a high-level language description that was both readable and synthesizable. The secondary objective was to explore the area/delay design space of both single and triple DES. The whole exercise took approximately four full days.

All designs have been tested against the National Institute of Standards and Technology (NIST) standard set of test vectors for the DES ECB implementation [4]. The designs were simulated using both the pre-synthesis (behavioral) and post-synthesis (RTL) VHDL simultaneously, verifying that the outputs were not only the same, but were the expected outputs defined in the test set.

While the software used for synthesis is the research platform at the University of Southampton, the concepts and behavioural modelling approach are fundamental.

II. THE DATA ENCRYPTION STANDARD (DES)

The Data Encryption Standard, usually referred to by the acronym DES, is a well-established encryption algorithm which was first standardized in 1988. The standard is maintained by the NIST [1]. DES is a symmetrical private-key cipher. This means that the same key is used to encrypt and to decrypt. It is therefore only suitable for applications where the key can be kept secure.

A key consists of 64 bits — however, only 56 bits are used in DES and the other 8 bits are parity bits.

Many people now consider DES's 56-bit key to be too short and therefore capable of being cracked using a simple brute force attack. This argument is shallower than it first appears. Although it is becoming easier to search the entire key space, the problem of recognizing that a solution has been found is still severe. The DES cracker promulgated by the Electronic Frontier Foundation (EFF) [5], for example, relies on the plaintext being ASCII. If any form of pre-encryption encoding is used, or even if the plaintext is in binary form (such as a word-processing document) then the plaintext cannot reliably be recognized without *a priori* knowledge of the contents.

Nevertheless, it is now common to find the algorithm being used in triplicate — an algorithm known as Triple-DES or TDES for short. This algorithm uses the same DES core, but uses three passes with different keys. A common form of TDES is EDE2, which encrypts, decrypts and then encrypts again using two different keys. This form of TDES can be made backwards compatible with DES simply by making the two keys identical.

DES was designed to be small and fast. It was designed in the mid-1980s and this is reflected in the fact that the algorithm is mainly based on shuffling and substitution — there is very little computation involved. Thus it is well-suited to hardware implementation. It is described in detail in [1], [2], [4], [6]–[8].

III. MOODS

MOODS (**M**ultiple **O**bjective **O**ptimization in Control and **D**atapath **S**ynthesis) is a high-level behavioral synthesis suite developed at the University of Southampton. It takes as input behavioral VHDL [9], [10]. and transforms this into structural VHDL that is behaviorally equivalent. MOODS is an aggressively optimizing behavioral synthesis suite (hardware compiler). The internal workings and algorithms have been described in detail elsewhere [3], [11]–[20], but in essence the operation of the system is as follows:

- The initial high-level VHDL description is decomposed into ICODE, which is a hardware assembly language. A dataflow graph is extracted

0-7803-9352-X/05/$20.00 © 2005 IEEE.

from this and a naive control graph constructed with one dataflow operation per cycle. Thus the mapping from control graph to dataflow graph at this point is 1:1. The dataflow graph describes the connectivity of the instructions and the control graph is a state machine that enables and disables data operations in the dataflow graph as needed to implement the algorithm.

- Optimization with respect to area is achieved by physically sharing datapath units (via multiplexors) in which case the control graph is restructured to guarantee that data collisions do not occur.
- Optimization with respect to delay is achieved by allowing datapath units to be connected combinatorially so that they can perform a combined operation in one clock cycle — effectively merging together control states.

IV. INITIAL DESIGN

The overall structure of the DES algorithm is shown in fig1.

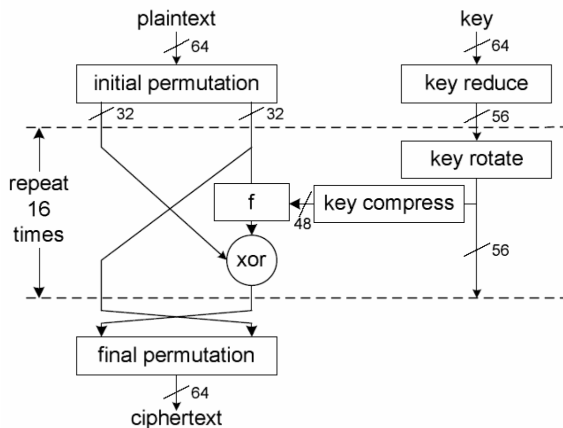

Fig. 1. Overall Structure of the DES Algorithm

The core algorithm is repeated 16 times with a different subkey for each round. These subkeys are 48 bits long and are generated from the original 56-bit key. The first attempt at designing DES was based on the description of the DES algorithm in [21], which in turn is a summary of the algorithm as described in [8] and [7]. This description of the algorithm was converted directly to VHDL using a functional decomposition style.

A. Overall Structure

The first stage in this design was to create an entity and architecture with the required inputs and outputs and a single process containing the overall algorithm. This resulted in the VHDL process below:

```
process
begin
  wait until go = '1';
  done <= '0';
  wait for 0 ns;
  ciphertext <=
    des_core(plaintext, key_reduce(key),
encrypt);
    done <= '1';
  end process;
```

This process is a direct implementation of the main DES routine. This algorithm requires the two functions `key_reduce` and `des_core`. The former strips the parity bits from the key and the latter then implements the whole DES algorithm. The `key_reduce` function reduces the key from 64 to 56 bits and permutes the bits to form the initial state of the subkey:

```
function key_reduce(key : in vec64) return
vec56 is
  --moods inline
  begin
    return
    key(57) & key(49) & key(41) & key(33) &
      ...
    key(28) & key(20) & key(12) & key(4);
  end;
```

The compiler directive `--moods inline` causes the synthesizer to inline the function. This allows the optimizer more scope for optimization of the circuit.

The `des_core` function applies the basic DES algorithm 16 times on a slice of the data using a different subkey on each iteration:

```
function des_core
  --moods inline
  (plaintext : vec64;
  key : vec56;
  encrypt : std_logic)
return vec64
is
  variable data : vec64;
  variable working_key : vec56 := key;
begin
  data := initial_permutation(plaintext);
  for round in 0 to 15 loop
    working_key :=
    key_rotate(working_key,round,encrypt);
    data := data(33 to 64) &
     (f(data(33 to
64),key_compress(working_key))
     xor
     data(1 to 32));
  end loop;
  return
    final_permutation(data(33 to 64) & data(1
to 32));
  end;
```

The DES algorithm is made up of the key transformation functions `key_rotate` and `key_compress`, and the data transformation functions `initial_permutation`, `f` and `final_permutation`.

B. Data Transformations

The data transformations `initial_permutation` and `final_permutation` are simply hard-wired bit-swapping routines which are most easily implemented as concatenations. These two functions are symmetrical, so if you pass data through both functions, the result is the same as the input. The `f` function is the main data transform which is applied 16 times to the rightmost half — a 32-bit slice — of the data path. It takes as its second argument a 48-bit subkey generated by the `key_compress` function.

The function first takes the 32-bit slice of the datapath and expands it into 48 bits using the `expand` function. The `expand` function is again just a rearrangement of bits — input bits are replicated in a special pattern to expand the 32-bit input to the 48-bit output [1].

This expanded word is then exclusive-ored with the subkey and fed into a substitute block. This substitutes a different 4-bit pattern for each 6-bit slice of the input pattern (remember that the original input has been expanded from 32 bits to 48 bits, so there are eight substitutions in all). The substitution also has the effect of reducing the output back to 32 bits again.

The substitute algorithm first splits the input 48 bits into eight 6-bit slices. Each slice is then used to lookup a substitution pattern for that 6-bit input. This structure is known as the S-block. In the initial implementation, a single ROM is used to store all the substitution patterns.

The substitution combines a block index with the input data to form an address which is then used to lookup the substitution value in the S-block ROM. This address calculation is encapsulated in the `smap` function. The 8 substitutions required are carried out by the 8 calls to `smap` in the `substitute` function. The final stage of the datapath transform is the permute function which is another bit-swapping routine [1].

These functions define the whole of the datapath part of the algorithm – with the majority of the code omitted for brevity.

C. Key Transformations

The encryption key also needs to be transformed a number of times - specifically, before each data transformation, the key is rotated and then a smaller subkey is extracted by selecting 48 of the 56 bits of the key. The rotation is the most complicated part of the key transformation. The 56-bit key is split into two halves and each half rotated by 0, 1 or 2 bits depending on which round of the DES algorithm is being implemented. The direction of the rotation is to the left during encryption and to the right during decryption. The algorithm is split into two functions — `do_rotate` which, as the name suggests, does the rotation and `key_rotate` which calls `do_rotate` twice, once for each half of the key. The `do_rotate` function uses a ROM to store the rotate distances for each round, numbered from 0 to 15.

This, then, was the initial realization of DES in behavioural VHDL. Most of the low level functions have been omitted in this paper to save space, but can be easily derived from [1].

V. SYNTHESIS

A. Initial Synthesis

The design was synthesized by MOODS with delay prioritized first and area prioritized second. The target technology was the Xilinx Virtex library.

Fig 2 shows the control state machine of the synthesized design. The whole state sequence represents the process, which is a loop as shown by the state transition from the last state (c11) back to the first (c1).

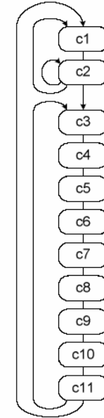

Fig. 2. Control State Machine for Initial Synthesis

The first two states c1 and c2 implement the input handshake on signal `go` to trigger the process. The DES core is implemented by the remaining states, namely states c3 to c11, which are in the main loop as shown by the state transition back from c11 to c3, so are executed 16 times. There are 9 states in this inner loop, giving a total algorithm length of 146 cycles including the 2 states required for the input handshake and 144 for the DES core itself. However, an inspection of the original structure shown in fig 1 suggests that a reasonable target for the inner loop is 2 cycles per round with an optimistic target of 1 cycle. Clearly there is a problem with this design. MOODS predicts that this design has the area and delay characteristics shown in Table I in the line labeled (1).

B. Optimizing the Datapath

Examining the 9 control states in the main loop and relating these to the mapping of the control graph to the dataflow graph showed that the last 8 cycles were performing the Sblock and the first 2 cycles were mainly related to transforming the key. The second state is an overlap state where both key and data transforms are taking place. The problem with the last 8 cycles was fairly self-evident since there are eight substitutions and there are eight control states to perform them. Clearly there was something causing each substitution to be locked into a separate control state and therefore preventing optimization with respect to latency. It wasn't difficult to see what — each of these states contained just register assignments, concatenations and a ROM read operation. It is the last of these that is the problem — the ROM implementation being targeted is a synchronous circuit, so the S-block ROM can only be accessed once per clock cycle — in other words once per control state. It is this that is preventing the datapath operations from being performed in parallel.

Attacking this problem is beyond the capabilities of behavioural synthesis because it requires knowledge of the dataflow at a much higher level than can be automatically

extracted. The solution therefore requires modification of the original design.

There are two obvious solutions to this problem — either split the S-block into eight smaller ROMs that can therefore be accessed in parallel or make the S-block a non-ROM so that the array gets expanded into a decoder block once for each access, giving eight decoders.

The latter solution appears simplest, but it will result in eight 512-way decoders, which will be a very large implementation.

The solution of splitting the ROMs is more likely to yield a useful solution. The substitute function was rewritten to have 8 mini-ROMs instead.

This was resynthesized and resulted in the control graph shown in fig 3. The inner loop was found to have been reduced to 2 states, and examination of the last state confirmed that all of the S-block substitutions were being carried out in the one state c4. The key transformations were still split across the two inner states c3 and c4.

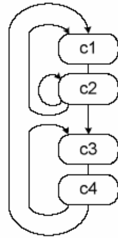

Fig. 3. Control State Machine for Optimized S-blocks

One interesting side-effect of this optimization is that it is also a smaller design. MOODS predicts that this design has the area and delay characteristics shown in Table I in the line labeled (2).

C. Optimizing the Key Transformations

Examination of the 2 control states in the main loop, which both contain key transformations, showed that both of these states were performing ROM access and rotate operations.

Examination of the original `key_rotate` function showed that the shift distance ROMs are accessed twice per call, so this turned out to be exactly the same problem as with the Sblock ROM. Since ROMs are synchronous, they can only be accessed once per cycle and this forces at least two cycles to be used for the rotate. To solve this, the function can be rewritten to only access the ROMs once per call.

This was resynthesized and resulted in the control graph shown in fig 4. The inner loop was found to have been reduced to 1 state (c3) containing both the key and data transformations which are repeated 16 times. As before, states c1 and c2 implement the input handshake.

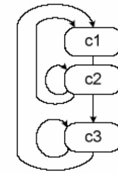

Fig. 4. Control State Machine for Optimized Key Rotate

So, this optimization means that the target of 1 clock cycle per round of the core was achieved. MOODS predicts that this design has the area and delay characteristics shown in Table I in the line labeled (3).

D. Final Optimization

It was recognized that the `key_rotate` function could be simplified by rethinking the rotate algorithm such that a right rotate of 1 bit was replaced by a left rotate of 27 bits (for a 28-bit word). This eliminates a conditional statement, which it was felt could be preventing some optimizations from taking place.This means that there was no need to have a different algorithm for encryption and decryption.

The state machine for this design was basically the same as for the previous design as shown in fig 4. It was found that this version was slightly slower than the previous design but significantly smaller.

MOODS predicts that this design has the area and delay characteristics shown in Table I in the line labeled (4).

E. Results

The results predicted by MOODS for all the variations of the design discussed so far are summarized in Table I.

TABLE I
PHYSICAL METRICS FOR SINGLE DES DESIGNS

Design	Area (slices)	Latency (cycles)	Clock (ns)	Throughput (MB/s)
(1) Initial Design	552	146	7.8	7.12
(2) Optimised S blocks	426	34	7.1	35.2
(3) Optimised Key	489	18	7.1	62.6
(4) Optimised Branch	307	18	8.4	52.9

It can be seen that design (3) is the fastest, but design (4) is the smallest. Fig 5 plots area versus throughput for these 4 designs. The X-axis represents the area of the design and the Y-axis the throughput.

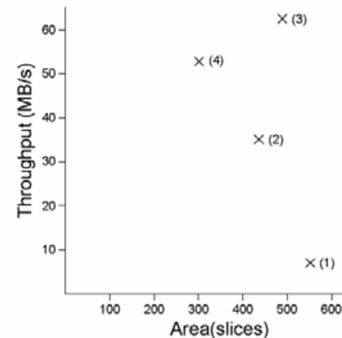

Fig. 5. Area versus Throughput for all DES Designs

VI. TRIPLE DES

Building on this, the DES core developed above was used as the core for a Triple-DES implementation. The idea of triple DES is that data is encrypted three times. The rationale for choosing three iterations and the advantages and disadvantages of this are explained in [8]. A common form of Triple DES is known as EDE2, which means data is encrypted, decrypted and then encrypted again using two different keys. The first key is used for both encryptions and the second key for the decryption.

There are obviously a number of different trade-offs that can be made in this design. Each of these is examined in the following sections. In all cases, the smallest implementation (design (4)) was used as the DES core.

A. Minimum Area — Iterative

To achieve a minimum area implementation, a single DES core is used for all three stages. The data is passed through this core three times with the different permutations of keys and encryption mode to achieve the EDE2 algorithm.

Two different styles of VHDL were tried. These differed in the method used to select the different inputs for each encryption step. The first style used a case statement and the second style used indexed arrays. The case statement style results in the following VHDL architecture:

```
architecture behavior of tdes_ede2_iterative is
  ...
begin
  process
    variable data : vec64;
    variable key : vec56;
    variable mode : std_logic;
  begin
    wait until go = '1';
    done <= '0';
    wait for 0 ns;
    data := plaintext;
    for i in 0 to 2 loop
      case i is
        when 1 =>
          key := key_reduce(key2);
          mode := not encrypt;
        when others =>
          key := key_reduce(key1);
          mode := encrypt;
      end case;
      data := des_core(data,key,mode);
    end loop;
    ciphertext <= data;
    done <= '1';
  end process;
end;
```

It can be seen that this uses a case statement to select the appropriate key and encryption mode for each iteration. The characteristics of the case statement solution are shown in Table II in the line labeled (5). The core DES algorithm accounts for 48 cycles (3 iterations of 16 rounds with 1 cycle per round), leaving an additional overhead of 3 cycles, due to the case statement selection of the key which adds an extra cycle per iteration of the core. The second style used arrays to store the keys and modes and then indexes these arrays to set the key and mode for each iteration. It was found that the latency was the same as the case statement solution but the area was approximately 25% larger. This overhead is mostly due to the use of the register arrays which add up to about 200 extra flip-flops. Clearly the case statement design is the most efficient of the two and so this solution was kept and the array style solution discarded.

B. Minimum Latency — Pipelined

To achieve minimum latency between samples, three DES cores are used to form a pipeline. Data samples can then be fed into the pipeline every 18 cycles (the latency of the single core), although the time taken for a result to be generated is 50 cycles because of the pipeline length. The circuit is simply three copies of the single-DES process:

```
architecture behavior of tdes_ede2_pipe is
  ...
  signal intermediate1, intermediate2 : vec64;
begin
  process
  begin
    wait until go = '1';
    intermediate1 <=
des_core(plaintext,key_reduce(key1),encrypt);
  end process;
  process
  begin
    wait until go = '1';
    intermediate2 <=
    des_core(intermediate1,key_reduce(key2),not
    encrypt);
  end process;
  process
  begin
    wait until go = '1';
    done <= '0';
    wait for 0 ns;
    ciphertext <=
    des_core(intermediate2,key_reduce(key1),
    encrypt);
    done <= '1';
  end process;
end;
```

Note how the `done` output is driven only by one of the cores — this will give the right result provided all three cores synthesize to the same delay, which in practice they will. This design decision alleviates the need to have handshaking between the cores. MOODS predicts that this design has the area and delay characteristics shown in Table II in the line labeled (6). The state machine shown in fig 6 shows the three independent processes. For example, the first process is represented by states c2, c3 and c4. The first two states perform the handshaking on `go` and c4 implements the DES core with its 16 iterations. State c7 is the second DES core and c10 the third.

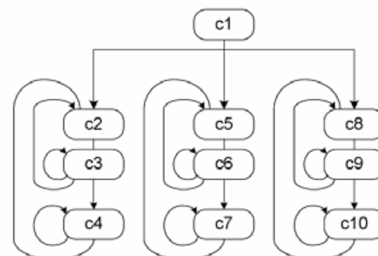

Fig. 6. Control State Machine for Pipelined Triple-DES

VII.COMPARING THE APPROACHES

The physical metrics of the previous section are the predicted values given by MOODS. To get a more accurate assessment of the design, RTL synthesis of the structural VHDL output of MOODS is required. This was carried out using Mentor Graphics' Leonardo Spectrum RTL synthesis suite. These results can be finessed further by carrying out placement and routing using the Xilinx Integrated Software Environment (ISE) Foundation suite. The results predicted by all three tools (MOODS, Leonardo and Foundation) for the three approaches (DES, Iterative TDES and Pipelined TDES) are shown in Table II. In all cases, the design was optimized during RTL synthesis using the vendor's default optimization settings — a combination of area and delay optimization — with maximum optimization effort. Placement and routing was performed with an unreachable clock period to force Foundation to produce the fastest design.

Table II shows that MOODS tends to overestimate the area of the design and underestimate the delay. Both of these are expected outcomes. The tendency to overestimate area is because it isn't possible to predict the effect of logic minimization when working at the behavioural level. The tendency to underestimate delay is because it isn't possible to predict routing delays.

TABLE II
PHYSICAL METRICS PREDICTED BY EACH TOOL

Design	Tool	Area (slices)	Latency (cycles)	Clock (ns)	Throughput (MB/s)
(4)	MOODS	307	18	8.4	52.9
DES	Leonardo	258		13.4	33.2
	Foundation	274		18.4	24.2
(5)	MOODS	500	53	8.4	18.0
Iterative	Leonardo	381		13.7	11.0
TDES	Foundation	422		17.8	8.5
(6)	MOODS	920	18	8.4	52.9
Pipelined	Leonardo	774		13.7	32.4
TDES	Foundation	826		18.4	24.2

VIII. CONCLUSION

This paper has shown that it is possible to design and analyse complex algorithms such as DES using the abstraction of high-level VHDL and get a synthesizable design.

However, the synthesis process is not and cannot ever be fully automated - human guidance is still necessary to optimize the design's structure to get the best from the synthesis tools.

Nevertheless the modifications are high-level design decisions and the final design is still readable and abstract. There has been no need to descend to low-level VHDL to implement DES. The implementations of Triple-DES show how VHDL code can easily be reused when written at this level of abstraction.

It is quite an achievement to implement the DES and two implementations of the Triple-DES algorithm in four working days including testing and this demonstrates the kind of productivity that result from the application of behavioural modelling and behavioural synthesis tools.

IX. REFERENCES

[1] "Data encryption standard," National Institute of Standards and Technology (NIST), Federal Information Processing Standard (FIPS) Publication 46-3, Oct. 1999.

[2] "DES modes of operation," National Institute of Standards and Technology (NIST), Federal Information Processing Standard (FIPS) Publication 81, Dec. 1980.

[3] A. D. Brown and A. C. Williams, "The MOODS behavioural synthesis system," in *Proceedings of the 3rd International Forum on Design Languages (FDL)*, Tubingen, Germany, Sept. 2000, pp. 17–21.

[4] "Validating the correctness of hardware implementations of the NBS data encryption standard," National Institute of Standards and Technology, Tech. Rep. 500-20, 1980.

[5] Electronic Frontier Foundation, *Cracking DES*. O'Reilly and Associates Inc., July 1998.

[6] "Guidelines for implementing and using the NBS data encryption standard," National Institute of Standards and Technology (NIST), Federal Information Processing Standard (FIPS) Publication 74, Apr. 1981.

[7] A. Menezes, P. van Oorschot, and S. Vanstone, *Handbook of Applied Cryptography*. CRC press, 1996.

[8] B. Schneier, *Applied Cryptography*, 2nd ed. Wiley press, 1996.

[9] A. J. Rushton, *MOODS VHDL Style Guide*, MOODS v1.2 ed., LME Design Automation Ltd., Chilworth Park, Southampton, UK, Aug. 2001.

[10] M. Zwoli˜nski and A. J. Rushton, *MOODS VHDL Reference*, MOODS v1.2 ed., LME Design Automation Ltd., Chilworth Park, Southampton, UK, Aug. 2001.

[11] A. D. Brown, A. C. Williams, and Z. A. Baidas, "Hierarchical module expansion in a VHDL behavioural synthesis system," in *Electronic Chips and Systems Design Languages*, J. Mermet, Ed. Kluwer Academic Publishers, 2001, pp. 249–260.

[12] K. R. Baker and A. J. Currie, "Multiple objective optimisation in a behavioural synthesis system," *IEE Proceedings - G*, vol. 140, no. 4, pp. 253–260, 1993.

[13] Z. A. Baidas, A. D. Brown, and A. C. Williams, "A VHDL behavioural synthesis system with floating point support," in *Forum on Design Languages 2000*, Tubingen, Germany, 2000, pp. 31–36.

[14] ——, "Floating point behavioural synthesis," *IEEE Transactions on Computer Aided Design*, vol. 20, no. 7, pp. 828–839, 2001.

[15] A. C. Williams, A. D. Brown, and M. Zwoli˜nski, "Simultaneous optimisation of dynamic power, area and delay in behavioural synthesis," *IEE Proceedings on Computers and Digital Techniques*, vol. 147, no. 6, pp. 383–390, 2000.

[16] A. C. Williams, A. D. Brown, and Z. A. Baidas, "Hierarchical module expansion in a VHDL behavioural synthesis system," in *Forum on Design Languages*, 1998.

[17] A. D. Brown, K. R. Baker, and A. C. Williams, "Online testing of statically and dynamically scheduled synthesized systems," *IEEE Transactions on Computer Aided Design*, vol. 16, pp. 47–57, 1997.

[18] K. R. Baker, "Multiple objective optimisation of data and control paths in a behavioural silicon compiler," Ph.D. dissertation, University of Southampton, Southampton, England, Sept. 1992.

[19] D. J. D. Milton, "Dynamic memory allocation within a behavioural synthesis system," Ph.D. dissertation, University of Southampton, Southampton, England, Jan. 2002.

[20] A. C. Williams, "A behavioural VHDL synthesis system using data path optimisation," Ph.D. dissertation, University of Southampton, Southampton, England, Oct. 1997.

[21] A. D. Brown, "Application note - DES core," LME Design Automation, Southampton, UK, Tech. Rep., Dec. 2000.