Reconfigurable Cryptography

A Hardware Compiler for Cryptographic Applications

C. Scott Ananian

May 12, 1997

Contents

Implementation	Year	Cost per chip	Blocks/s	Chips per $1M	Time, given $1M	Ref
Diffie-Hellman†	77	≈$20	1M	50k	17 days	[DH77]
Hoornaert, et al†	84	≈$40	1.1M	25k	30 days	[HGD85]
AMD	84	$19	218k	53k	72 days	[AMD84]
Wayner†	92	≈$30	448k	31k	30 days	[Way93]
VLSI Technology	92	$170	3M	6k	47 days	[VLS91]
DEC	92	$300	16M	3k	16 days	[Ebe93]
Wiener	93	$11	50M	58k	4 hours	[Wie94]

† Paper study. These tend to be rather optimistic.

Table 1: Cost and Time Estimates to Break DES.

1 Introduction

The United States' key-length limit of 40-bits for exportable cryptography is laughably small: Ian Goldberg at the University of California at Berkeley needed fewer than 4 hours of compute-time to brute-force the key space of 40-bit RC5. Forty-eight bit algorithms are small improvement; a European team led by the Swiss Federal Institute of Technology in Zurich exhausted the key-space of 48-bit RC5 in 13 days. The 56-bit key length of the Data Encryption Standard, DES, has likewise been claimed too small; Diffie and Hellman objected at the time of the standard's adoption, in 1977 [DH77].

A number of papers have provided estimates of the cost and time of breaking DES using brute-force search. Custom hardware invariably performs much better than software for this task; DES is not particularly suited to software implementation due to its employment of bit-permutations and variable word lengths.[1] Table 1 summarizes the costs and speeds of hardware implementations proposed from the time of DES' first adoption.

By Garon and Outerbridge's estimates, DES chips are increasing in speed by a factor of eight every five years [GO91]. Thus, Wiener's 1993 0.8 μm CMOS design using a 50M block/s DES chip [Wie94], could now be translated into a $10,000 machine that would extract a DES key in 44 hours.[2] If one is going to spend money on a cracking machine, one might wisely ask if, for a small additional expenditure, the machine may be made flexible enough to accomodate multiple algorithms. This paper attempts to more quantitatively assess that possibility. In particular, we will discuss the creation of an optimizing compiler to create hardware structures for cryptographic algorithms, and the results of a chip-level design of an FPGA-based brute-force search engine.

[1] A table of DES speeds for various processor platforms is given in [Sch94, p 131].

[2] The current RSA challenge offers a reward of $10,000 for the successful brute-force solution of a posted ciphertext/plaintext pair. Current co-operative software-only approaches seem to require at least 4 years of processing time to achieve a solution.

2 Related Work

Peter Wayner describes the use of a content-addressable memory to attack DES in [Way93]. The content-addressable memory is used as an array of bit-level processors; they could be reprogrammed, as we propose, to attack algorithms which differ slightly from DES. The processing elements are sufficiently simple that it would be very hard to implement the more "modern" software-oriented algorithms which rely an arithmetic operators rather than boolean operations and bit permutations. FPGAs have no such limitation. In addition, Wayner's DES algorithm is coded by hand; he does not address automatic code generation for his machine from a high-level algorithm description. Finally, his results are more than an order of magnitude slower than rival custom ASIC implementations.

Wiener describes a hardware implementation of DES in detail in [Wie94]. The design is for 0.8 μm standard-cell CMOS, clocked at 50 MHz. His custom chip achieves the highest speed-to-price ratio of any hardware implementation to date; our implementation success in FPGA technology will be measured against his standard. Dave Wagner describes an extension to Wiener's work to allow ciphertext-only attacks on DES for an order of magnitude more cost[WB94]; our current work concerns itself with Wiener's baseline design only.

The idea of utilizing configurable computing devices cryptographically was first proposed by [V+96] and [ACC+95], who studied long-integer arithmetic circuits suitable for public-key cryptography. These results have little relevance to the secret-key systems we consider in this paper. Implementations of microprocessors with reconfigurable functional units would be well suited to attacking cryptographic algorithms with complex boolean operations and bit permutations; however, the published literature [AS93, WH95] does not address this issue.

3 Methodology

A compiler, a general hardware design, and several benchmarks were created to evaluate programmable hardware's suitability for brute-force key search. The cryptographic algorithm was expressed in a high-level language and compiled to produce behavioral VHDL. The VHDL description was analyzed by Synopsys tools and targeted to the Xilinx XC4010 FPGA. Xilinx place-and-route tools were used on the final net-list.

4 Compiler

An optimizing compiler was written for the TIGER programming language. The compiler translates algorithm descriptions into behavioral VHDL. A subset of TIGER is supported; explicitly omitted are arrays, strings, and functions. Looping constructs are supported. The optimization phase of the compiler is designed to target hardware; for example, copy propagation is omitted because it disappears into a net-list once hardware translation is complete.

4.1 Front-end

The source language for the compiler is described in [App97]. TIGER is a "simple but non-trivial language of the Algol family," lacking only the wide variety of data types that categorize more familiar languages such as C. A pre-existing front-end was modified to implement the bit-level operations[3] needed to support most cryptographic algorithms. Several "pseudo-functions" were also added to the language to make available the values of the hardware key registers. The output of the front-end is an Intermediate Representation Tree (IR tree). It is possible to rewrite the front-end to generate IR trees from another source language (say, a C subset) with minimal changes to the back-end implemented in this project.

4.2 Optimizer

A number of optimizations were implemented in order to generate efficient hardware. Perhaps the most important of these is loop-unrolling, which can replace sequential circuitry with combinational logic when successful. In order to recognize when unrolling is possible, constant propagation and folding are done. Constant propagation, constant folding, and dead code elimination also reduce the amount of unnecessary hardware generated.[4]

4.2.1 Quadruples

The first step of the optimization phase is conversion of the IR tree to *quadruples*, simple statements computing an operation of no more than two operands. The conversion to quadruples involves flattening the IR tree through the introduction of new temporary variables. The converted IR tree is a list consisting of only nine types of simple statements:

MOVE	$a \leftarrow b$	$a \leftarrow b \; binop \; c$	BINOP
LOAD	$a \leftarrow M[b]$	$M[a] \leftarrow b$	STORE
CALLSUB	$f(a_1, \ldots, a_n)$	$a \leftarrow f(b_1, \ldots, b_n)$	CALLFUN
LABEL	L:	goto L	GOTO
COND.	if $a \; relop \; b$ goto L_1 else goto L_2		

Once the quadruples have been generated, the code is converted to Static Single-Assignment (SSA) form [C+91] for optimization.

4.2.2 Static Single-Assignment Form

Static Single-Assignment form is an intermediate format that allows optimizations to be done efficiently and easily. Every variable receives exactly one assignment during its lifetime, and *ϕ-functions* are added at places where program flow joins. The value of the ϕ-function "magically" depends on the path the program has taken; in practice,

[3] Bit-wise AND, OR, XOR, shift and rotation

[4] The Synopsys compiler tends to be rather literal with its input stream; dead code will be translated into hardware despite being computationally useless.

$$\begin{array}{rcl} V & \leftarrow & 4 \\ & \leftarrow & V+5 \\ V & \leftarrow & 6 \\ & \leftarrow & V+7 \end{array} \qquad \begin{array}{rcl} V_1 & \leftarrow & 4 \\ & \leftarrow & V_1+5 \\ V_2 & \leftarrow & 6 \\ & \leftarrow & V_2+7 \end{array}$$

Figure 1: Straight-line code and its single assignment version.

```
if  P                                  if  P
    then V ← 4                             then V1 ← 4
    else V ← 6                             else V2 ← 6
                                           V3 ← φ(V1, V2)
/* Use V several times */                 /* Use V3 several times */
```

Figure 2: If-expression code and its single assignment version.

a move-insertion on each entrance path implements the ϕ-function when coverting out of SSA form. An example, taken from [C+91], is shown in figures 1 and 2.

The use of ϕ-functions simplifies the book-keeping for various optimizations, and by maintaining a single point of definition for every variable allows the algorithms to execute in linear, rather than quadratic, time. Furthermore, this work has disovered that the ϕ-function notation allows concise and accurate identification of state-machine registers in the translation of loop constructs. This application will be further discussed in section 4.3.2.

The translation into SSA form uses the algorithms discussed in [App97, C+91, LT79]. The dominator tree is computed using the Lengauer-Tarjan algorithm and path-compression, and is then used to compute the dominance frontier using Cytron's two-pass algorithm. After adding ϕ-functions for the variable a at the dominance frontier of every node where a is defined, we walk the dominator tree to rename variables so that every variable is defined exactly once. There is a simpler algorithm for SSA form translation that utilizes source-language information to aid placement [BM94], but the more complicated algorithm implemented here works on simple quadruples, and thus allows front-end (source language) modification or replacement without necessitating changes to the back-end implemented in this project.

The ϕ-functions of the SSA form were implemented as a tenth type of quadruple, and a flowgraph of the SSA-format quadruple list was input to the optimization routines.

4.2.3 Conditional Constant Propagation

Wegman and Zadeck's Sparse Conditional Constant (SCC) algorithm was used to find constant expressions, constant conditions, and unreachable code [WZ91]. Figure 3 shows the optimization extent possible.

The output of the SCC algorithm is an association of variables to one of $\{\bot, c, \top\}$, where $\bot$ marks a variable that is never defined, c indicates a constant value, and $\top$

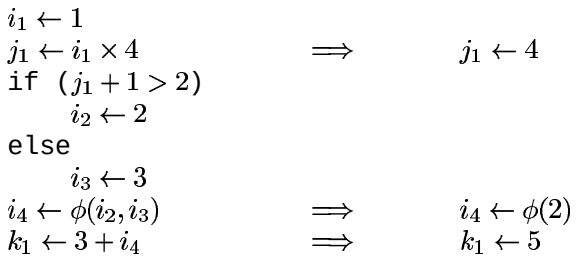

Figure 3: SCC code optimization.

signifies an over-defined variable (which may be assigned any one of a number of values). In addition, every flow-graph node (corresponding to a quadruple) is marked as executable or non-executable. We then walk the flow-graph, eliminating dead-code (quadruples marked non-executable), replacing constant variables with their values, and changing constant conditional branches to `goto` statements.

4.2.4 Code motion

Maximal loop-unrolling is possible after constant propagation. The code motion analysis implemented was very simple, and relied on source-language information from the abstract-syntax tree. It was able, however, to fully unroll the simple loops found in the algorithms under investigation. Once the loop was fully unrolled, the above optimization algorithms render more sophisticated code motion analysis (for example, code hoisting outside the loop) unnecessary. Further work on code motion optimizations is possible, especially in light of the recent ability to generate sequential circuitry from loop constructs. Code motion when sequential circuits are targetted allows us to reduce the amount of state, and hence, registers, necessary to implement the state machine.

4.3 VHDL generation

The optimized quadruples were used to generate behavioral VHDL code which could be compiled to hardware. The load and store operations accessing memory were unsupported, but the binary operation quadruples could be translated fairly directly to VHDL. Properly translating branches and conditionals was more difficult.

4.3.1 Branch-compression

For code without loops, the conditional branches and `goto`s need to be translated into `if-then-else` statements, from which the VHDL compiler will create combinational logic. The trouble is that constructs such as the one shown in figure 4 do not have equivalent if-constructs. These control-flow patterns can be generated by source-language `goto` statements or short-circuit logical operators. The algorithm devised in this work combines dominator tree and flow-graph information to define a “merge

```
     if a then goto L1 else goto L3
L1:  if b then goto L2 else goto L3
L2:  c ← d + e
     goto L4
L3:  f ← g + h
     goto L4
L4:
```

Figure 4: A flowgraph which can not be represented as an if-then-else statement without quadruple duplication.

```
if a then
  if b then
    c := d + e;
  else
    f := g + h;
  end if;
else
  f := g + h;
end if;
```

Figure 5: Conversion of the program of figure 4 to VHDL.

node," where the two control flows of the conditional will merge (if ever).[5] Statements must then be duplicated along each side of the conditional, until the merge node is reached, or all statements have been translated. Figure 5 shows the resultant VHDL for the quadruples in figure 4.

4.3.2 Loop handling

The original compiler relied on maximal loop unrolling to eliminate looping constructs. It was realized that the SSA form dictated a precise method of converting a quadruple list with ϕ-functions to an equivalent state machine. Therefore the code written to translate back from SSA form after code optimization was removed, and a new version of the VHDL generator was written, using the SSA form directly as input.

Loop analysis was performed on the dominator tree using the algorithm in [ASU85]. This yielded a list of loops and their headers. All flow into a loop must be through its header (the header must *dominate* all the nodes in the loop). Our insight, simply stated, was that the list of ϕ-functions in the header of the loop exactly defined the required registers for a state-machine implementing that loop. For example, the simple while-loop in figure 6 needs only one register, to store the value of i_2. On state transitions, i_2 would be loaded with the value of either i_1 or i_3. To simplify circuitry, these registers

[5] The "merge node" is the dominator tree child of the conditional branch, which is reached last in a post-order depth-first-search of the control flow graph.

```
let                          i1 ← 1
  var i:=1            L1:    i2 ← φ(i1,i3)
in                           if i2 < 16 goto L2 else L3
  while (i<16) do     L2:    i3 ← i2 + 1
    i:=i+1                   goto L1
end                  L3:
```

Figure 6: A simple while-loop in TIGER, left, and SSA form, right.

```
architecture BEHAVIOR of whiletest is
-- register definitions
  signal t8,t11:INT32;
-- state definitions
  signal State2, nextState2:BIT;
  signal State1, nextState1:BIT;
  signal State0, nextState0:BIT;
begin
  CLOCKSTATE: process
  begin
    wait until (CLOCK'event) and (CLOCK='1');
    -- new state:
    State2 <= nextState2;
    State1 <= nextState1;
    State0 <= nextState0;
    -- new registers:
    t8 <= t11;
  end process;
  NEXTSTATE: process(
               State2,
               State1,
               State0,
               t8)
    variable t9:INT32;
  begin
    DONE <= '0';
    -- default state:
    nextState2 <= '0';
    nextState1 <= '0';
    nextState0 <= '0';
    -- default registers:
    t11 <= t8;
    if RESET = '1' then
      nextState1 <= '1'; -- initial state.
    -- STATE MACHINE
    elsif State1 = '1' then
        nextState0 <= '1'; -- goto
        t11 <= To_INT32(1); -- phi1
    elsif State0 = '1' then
        if
          t8 < To_INT32(16)
        then
          t9 := t8 + To_INT32(1);
          nextState0 <= '1'; -- goto
          t11 <= t9; -- phi1
        else
          nextState2 <= '1'; -- goto
        end if;
    elsif State2 = '1' then
        nextState1 <= '1';
        DONE <= '1';
    end if;
  end process;
end;
```

Figure 7: Excerpted VHDL code for the while-loop in figure 6.

were augmented with a simple one-hot state-encoding,[6] and a new variable is created to hold the "next" value of the state register after the transition.

The allocation into states was follows Galloway's work on Transmogrifier C [Gal95]. The initial state comprises the code prior to the loop header, another state indicates execution of the loop body, and a final state is reached on exit from the loop, for a total of three bits of state per loop. Loops can be nested to arbitrary depth. Multiple exit points for the loop are supported. The VHDL code output for the while-loop in figure 6 is shown in figure 7.

[6] Galloway indicates in [Gal95], citing [BFS94], that, despite its simplicity, one-hot encoding may be the best encoding to use on FPGAs. See also [POA96], who reference [Wak90].

Algorithm	Keys tested/s
DES	90k
RC5	150k
TEA	357k

Table 2: Representative algorithm speeds in software. Speeds are reported for a Sun Ultra-2 machine, using a 168MHz UltraSPARC CPU.

5 Hardware Design

Basic hardware using FPGAs to form a brute-force cracking machine was designed. The basic idea has not changed fundamentaly from that proposed by Diffie and Hellman in 1977 [DH77]: design a chip to test keys as quickly as possible, and use as many as possible of them in parallel. Very little inter-chip communication is necessary, besides initial set-up.

Wiener describes a detailed board layout for the DES-cracking chip he designed [Wie94], using a 8-bit data bus for chip interconnect. His chip requires 27 I/O pins for the interface. We feel that a parallel bus is over-kill for this application; we propose a simple daisy-chained serial bus instead, requiring only 5 pins, not including clocks. This should allow the printed-circuit board layout of the FPGA array to be extremely simple. The chip I/O interface is defined by the behavioral VHDL in `driver.vhdl`, found in appendix B.2.

The interface is based on a 65-bit shift register, which holds the current 64-bit key under test and a one bit search status flag. To save logic, we follow Wiener in advancing the key using a linear feedback shift register, instead of a 64-bit ripple-carry adder. The generating polynomial for the 64-bit LFSR is $x^{64} + x^4 + x^3 + x^1 + 1$. This is a maximal-length LFSR, meaning that it will step through all of the $2^{64} - 1$ possible states [Sch94]. The LFSR will not step through the all-zero key; this must be tested separately.

In addition, an internal driver-to-cryptographic engine interface has been defined; the behavioral VHDL describing it is in `crypt_pack.vhdl`. Details vary slightly for pipelined and non-pipelined versions, but both types have two 32-bit key-inputs, and a single bit result output in common.

A design cycle using a single adder as the “crypto engine” component reveals that the driver-stage uses about 20% of the available CLBs (mostly for the 65-bit shift-register), and can be clocked at over 18MHz. Obviously, the introduction of a “real” cryptographic component will limit us to far below this maximum possible speed.

6 Algorithm Selection

Three different cryptographic algorithms were investigated: DES, the Data Encryption Standard of the National Bureau of Standards, defined formally in [NBS88] and informally in [Sch94]; RC5, designed by Ron Rivest, defined in [Riv95]; and TEA, the Tiny Encryption Algorithm, designed by Wheeler and Needham and described in [WN95].

DES is the most important commercially of the three algorithms; it is used every day by the financial industry despite dire warnings [Hel79] of its imminent insecurity. It is implemented much more efficiently in hardware than in software, due to its dependence on bit-level permutations and other constructs not easily described in "conventional" high-level programming languages. The typical software speeds listed in table 2 may be compared against Wiener's 50 million key-per-second 1993 hardware implementation [Wie94].

DES is very difficult to express in TIGER until source-language constructs to support bit-permutations are added, and so a reconfigurable implementation of DES was not possible within the scope of this present work. Because of its importance to the financial industry and others, it has commerically-obtainable hardware implementations to which we can compare our paper-study results; our assumption is that DES is not easier to implement than TEA.

RC5 is a cipher proposed by Ron Rivest, which is used in RSA Data Securities' products and in a number of Internet protocols. It was explicitly designed to be simple to implement in software; i.e. no bit permutations are used. Even so, table 2 demonstrates only a 50% speed increase over DES, using highly-optimized versions of both algorithms. The lackluster performance is likely due to key-setup times; Rivest designed a lengthy sub-key generation phase into the algorithm to make brute-force key-searching substantially more difficult without slowing down conventional one-key uses of RC5.

RC5 is a 12-round algorithm, not counting 78 rounds of key setup, making this algorithm difficult to implement non-iteratively. In addition, the wide addition steps required by RC5 are costly to implement quickly in an FPGA; the RC5 algorithm will not fit in the Xilinx XC4010 we are targeting.

TEA is the simplest algorithm of the three; it uses many rounds to counter the simplicity of its round function. The author's claim of triple the speed of DES in software seems to be warranted; table 2 shows a speed-up of closer to four. The wide additions of the algorithm make it likely that the hardware complexity of TEA is comparable to that of DES. The TEA algorithm very nearly fills a Xilinx XC4010 chip; this puts its complexity as (very roughly) 10,000 gates. Eberle's GaAs DES implementation [Ebe93] uses somewhere between 4,000 and 15,000 gates.

7 Benchmark designs

In order to have a standard to which to compare the compiler-generated designs, a two manual crypto-engine implentations were done, using the TEA algorithm. One used a VHDL `for`-loop which was automatically unrolled by the Synopsis tools to generate straight-line code; the other was a hand-tweaked state machine that did one round of the algorithm per iteration.

In addition, software benchmarks were created for the three cryptographic algorithms under consideration. Software optimization techniques were studied, and loops were unrolled and memory accesses eliminated as much as possible.

It was recently brought to my attention that an efficient parallel implementation of DES is possible on general-purpose machines by testing multiple keys in parallel; each

Algorithm	Code Gen.	Cycles/key	Clock Rate	Keys/s
Baseline	None	1	18 MHz	—
`Tea1`	Automatic	1	—	—
`Tea2`	Manual	1	514 kHz	514k
`Tea3`	Manual	34	13.5 MHz	397k
`Tea4`	Automatic	34	10.1 MHz	297k
RC5	Automatic	5	—	—

Table 3: Design Speeds (as reported by the Synopsys tools).

32-bit machine register could represent thirty-two parallel copies of a single bit in the algorithm. This allows efficient representation of bit-level operations, but eliminates the use of lookup tables. The S-boxes of DES have to be represented by a logical expression to enable their implementation. The complexity of the algorithm decomposition necessary for this approach puts it outside the scope of this research; it would, however, enable the expression of DES in TIGER without source-language enhancements.

8 Results

Four complete designs for the TEA-algorithm and a implementation of RC5 were evaluated, in addition to a base-line design with key-search hardware but no cryptographic algorithm. The results are shown in tables 3 and 4.

`Tea1` is automatically generated fully-unrolled code generated from the TIGER-language program of appendix A.1. The Synopsys tools were not able to properly handle the large number of temporary variables generated, and so there are no detailed speed or complexity figures for this design.

`Tea2` is hand-written code which uses a VHDL for-loop to allow the Synopsys tools to unroll the 32 TEA rounds. The Synopsys tools were able to evaluate the design and give a clock speed estimate, but the design was too large to fit in a single Xilinx XC4010.

`Tea3` is hand-written code for an iterative state-machine implementation of TEA, used as a feasibility test for automatic generation of sequential circuits. The resulting design was able to be squeezed into a Xilinx XC4010, with only 10 CLBs to spare.

`Tea4` is an automatically generated state machine compiled from the same TIGER program as `Tea1`, with loop unrolling turned off. It also fit into a Xilinx XC4010, barely. Performance is roughly equivalent to `Tea3`.

RC5 is an automatically generated state machine compiled from the TIGER program of appendix A.2. The TIGER program needed to be unrolled to eliminate array references, but still uses a iterative state machine in subkey generation. The design does not fit in a Xilinx XC4010.

Algorithm	Code Gen.	Total CLBs†	Xilinx XC4010 CLB Use‡	
Baseline	None	57	95/400	23%
`Tea1`	Automatic	—	—	—
`Tea2`	Manual	5814	—	—
`Tea3`	Manual	2187	390/400	97%
`Tea4`	Automatic	364	399/400	99%
RC5	Automatic	—	—	—

† Reflects Synopsys figures before target-specific optimization and routing.

‡ Reported by the Xilinx tools after successful place-and-route.

Table 4: Hardware complexity of hand-written and compiled designs.

Technology	Speed (keys/s)	Notes
FPGA	514k	Best TEA results
Software†	1,660k	175MHz MIPS R10000, bitsliced
Custom hardware‡	50,000k	0.8 μm standard-cell CMOS

† From author's experiments with the DES Challenge client from `http://www.frii.com/~rcv/deschall.htm`.

‡ From [Wie94]

Table 5: Realistic speed comparison.

9 Conclusions

The speed and density of the compiled algorithms compares well with hand-written VHDL. The Synopsys tools are very sensitive to the manner in which the hardware description is expressed [SWA95], and we are confident that closer examination of the generated code will disclose ways to make up the current 20% performance penalty of the compiled code. The state-machine generation algorithm utilizing the SSA form information appears to be robust and efficient.

The future of FPGAs in brute-force cracking machines does not appear as rosy. Although the 514,000 key per second cracking rate of the `Tea2` design (and the presumably similar speed of `Tea1`) compares favorably with the software speeds listed in table 2, it is unimpressive compared to more sophisticated implementations on faster processors or custom hardware. Table 5 shows that the FPGA implementations discussed here are a hundred times slower than a custom CMOS design. Wiener's 50 million key/s design was manufacturable for less than $11/chip; the Xilinx XC4010 considered here costs about $100.[7] The added algorithmic flexibility of the FPGA approach does not seem to justify the three orders of magnitude speed-cost ratio disadvantage. Wiener's $1 million brute-force cracking machine would cost $1 *billion* if it were to use reconfigurable devices. Wayner's content-addressable memory scheme [Way93] seems more practical; however it is unclear whether it can handle addition-based algorithms such as TEA and RC5.

[7] Wiener assumes quatities of 10,000; the Xilinx price quote is for a -2 grade part in quantity 100.

10 Future Work

The current 20% speed disadvantage of compiled code can be remedied. There are hopes as well that closer work with the Synopsys tools may reveal methods to speed up primitive operations, such as the 32-bit add, as well.

Loop unrolling is very successful as an optimization technique, allowing a 30% speed increase over an iterative implementation. The automatic insertion of pipeline registers into an unrolled algorithm promises further speed improvement; the methods of [AS93, POA96] might prove useful.

Remaining compiler work may include retargeting the front-end to a C subset, and implementing more optimization stages to perform strength-reduction and copy-propagation. Better support for arrays (and their decomposition into register variables) may enable us to express the RC5 algorithm in a manner that does not require substantial manual unrolling.

Finally, the possibility of compiling directly to *structural* VHDL remains to be considered; ϕ-functions correspond neatly to multiplexors required in a hardware implementation.

References

[ACC+95] Rick Amerson, Richard J. Carter, W. Bruce Culbertson, Phil Kuekes, and Greg Snider. Teramac—configurable custom computing. In *Proceedings, FCCM*, pages 32–38, April 1995.

[AMD84] Advanced Micro Devices. *AmZ8068/Am9518 Data Ciphering Processor*, July 1984. Datasheet.

[App97] Andrew W. Appel. *Modern Compiler Implementation in Java*. Cambridge University Press, 1997.

[AS93] Peter M. Athanas and Harvey F. Silverman. Processor reconfiguration through instruction-set metamorphosis. *IEEE Computer*, 26(3):11–18, March 1993.

[ASU85] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1985.

[BFS94] H. Belhadj, A. Fortas, and G. Saucier. State assignment selection for FPGAs and CPLDs. In *Second ACM/SIGDA Workshop on FPGAs, FPGA '94*, February 1994.

[BM94] Marc M. Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Transactions on Programming Languages and Systems*, 16(6):1684–1698, November 1994.

[C+91] Ron Cytron et al. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, October 1991.

[DH77] W. Diffie and M. Hellman. Exhaustive cryptanalysis of the NBS data encryption standard. *Computer*, 10(6):74–84, June 1977.

[Ebe93] H. Eberle. A high-speed DES implementation for network applications. In *Advances in Cryptology: CRYPTO '92 Proceedings*, pages 527–545. Digital Equiptment Corporation, Springer-Verlag, 1993.

[Gal95] David Galloway. The Transmogrifier C Hardware Description Language and compiler for FPGAs. In *Proceedings, FCCM*, pages 136–144, April 1995.

[GO91] G. Garon and R. Outerbridge. DES watch: An examination of the sufficiency of the data encryption standard for financial institution information security in the 1990's. *Cryptologia*, XV(3):177–193, July 1991.

[Hel79] M. Hellman. DES will be totally insecure within ten years. *IEEE Spectrum*, 16:32–39, July 1979.

[HGD85] F. Hoornaert, J. Goubert, and Y. Desmedt. Efficient hardware implementation of the DES. In *Advances in Cryptography: Proceedings of Crypto 84*, pages 147–173. Springer-Verlag, 1985.

[LT79] T. Lengauer and R. E. Tarjan. A fast algorithm for finding dominators in a flowgraph. *ACM Transactions on Progamming Languages and Systems 1*, 1:121–141, 1979.

[NBS88] National Bureau of Standards. *Data Encryption Standard*, January 1988. Federal Information Processing Standards Publication FIPS PUB 46-1 (supercedes FIPS PUB 46, January 1977).

[POA96] James B. Peterson, R. Brendan O'Connor, and Peter M. Athanas. Scheduling and partitioning ANSI-C programs onto multi-FPGA CCM architectures. In *Proceedings, FCCM*, pages 178–187, 1996.

[Riv95] Ronald L. Rivest. The rc5 encryption algorithm. In B. Preneel, editor, *Proceedings of the 1994 K. U. Leuven Workshop on Cryptographic Algorithms*. Springer-Verlag, 1995.

[Sch94] Bruce Schneier. *Applied Cryptography*. John Wiley and Sons, first edition edition, 1994.

[SWA95] Nabeel Shirazi, Al Walters, and Peter Athanas. Quantitative analysis of floating point arithmetic on FPGA-based custom computing machines. In *Proceedings, FCCM*, pages 155–162, April 1995.

[V+96] Jean E. Vuillemin et al. Programmable Active Memories: Reconfigurable systems come of age. *IEEE Transactions on VLSI Systems*, 4(1):56–69, March 1996.

[VLS91] VLSI Technology. *VM007 Data Encryption Processor*, October 1991. Datasheet (Advance Information).

[Wak90] J. Wakerly. *Digital Design Principles and Practices*. Prentice-Hall, Englewood Cliffs, New Jersey, 1990.

[Way93] Peter C. Wayner. Using content-addressable search engines to encrypt and break DES. In *Advances in Cryptology: CRYPTO '92 Proceedings*. Springer-Verlag, 1993.

[WB94] David Wagner and Steven M. Bellovin. A programmable plaintext recognizer. Unpublished manuscript. Available from `http://www.cs.berkeley.edu/~daw/recog.ps`., September 1994.

[WH95] Michael J. Wirthlin and Brad L. Hutchings. A Dynamic Instruction Set Computer. In *Proceedings, FCCM*, pages 99–107, April 1995.

[Wie94] Michael J. Wiener. Efficient DES key search. Technical Report TR-244, School of Computer Science, Carleton University, Ottawa, Canada, May 1994. Presented at Rump Session of Crypto '93. Available through anonymous ftp from `ripem.msu.edu:/pub/crypt/docs/des-key-search.ps`.

[WN95] David Wheeler and Roger Needham. TEA, a Tiny Encryption Algorithm. In B. Preneel, editor, *Proceedings of the 1994 K. U. Leuven Workshop on Cryptographic Algorithms*. Springer-Verlag, 1995.

[WZ91] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems*, 13(2):181–210, April 1991.

A Tiger code

A.1 The TEA algorithm

```
let
  var c0:=12  /* cipher text */
  var c1:=23

  var p0:=45  /* plain text */
  var p1:=56
in
    let
      var delta := 0x9e3779b9
      var sum := delta << 5
      var y := c0
      var z := c1

    in ( for n:= 1 to 32 do (
            z:=z - (((y << 4) + k0()) ^
                    ( y       + sum ) ^
                    ((y >> 5) + k1()) );
            y:=y - (((z << 4) + k0()) ^
                    ( z       + sum ) ^
                    ((z >> 5) + k1()) );
            sum:= sum - delta
         );
         c0:=y;
         c1:=z;
         (c0=p0)&&(c1=p1)
    )
    end
end
```

A.2 The RC5 algorithm

```
let
  /* RC5 algorithm "magic numbers" */
  var P:= 0xb7e15163     /* base of ln   - 2 */
  var Q:= 0x9e3779b9     /* golden ratio - 1 */

  var L0:=k0()             /* Keys to be tested */
  var L1:=k1()

  var P0:= 0x20656854 ^ 0xC93C8C23        /* plain text */
  var P1:= 0x6e6b6e75 ^ 0x9e9ffdb0

  var C0:= 0xD28688BF             /* Cipher text */
  var C1:= 0x1C8450A9

  var S00:=P             /* initialize constant array */
  var S01:=S00+Q
  var S02:=S01+Q
  var S03:=S02+Q
  var S04:=S03+Q
  var S05:=S04+Q
  var S06:=S05+Q
  var S07:=S06+Q
  var S08:=S07+Q
  var S09:=S08+Q
  var S10:=S09+Q
  var S11:=S10+Q
  var S12:=S11+Q
  var S13:=S12+Q
  var S14:=S13+Q
  var S15:=S14+Q
  var S16:=S15+Q
  var S17:=S16+Q
  var S18:=S17+Q
  var S19:=S18+Q
  var S20:=S19+Q
  var S21:=S20+Q
  var S22:=S21+Q
  var S23:=S22+Q
  var S24:=S23+Q
  var S25:=S24+Q
             /* calculate key expansion */
  var A:=0
  var B:=0
in
  ( for i:=1 to 3 do (
     S00 := (S00+(A+B)) <<< 3;            A := S00;
     L0  := (L0 +(A+B)) <<< (A+B);        B := L0 ;
```

```
     S01 := (S01+(A+B)) <<< 3;           A := S01;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S02 := (S02+(A+B)) <<< 3;           A := S02;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S03 := (S03+(A+B)) <<< 3;           A := S03;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S04 := (S04+(A+B)) <<< 3;           A := S04;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S05 := (S05+(A+B)) <<< 3;           A := S05;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S06 := (S06+(A+B)) <<< 3;           A := S06;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S07 := (S07+(A+B)) <<< 3;           A := S07;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S08 := (S08+(A+B)) <<< 3;           A := S08;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S09 := (S09+(A+B)) <<< 3;           A := S09;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S10 := (S10+(A+B)) <<< 3;           A := S10;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S11 := (S11+(A+B)) <<< 3;           A := S11;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S12 := (S12+(A+B)) <<< 3;           A := S12;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S13 := (S13+(A+B)) <<< 3;           A := S13;
     L1  := (L1 +(A+B)) <<< (A+B);       B := L1 ;

/**/
     S14 := (S14+(A+B)) <<< 3;           A := S14;
     L0  := (L0 +(A+B)) <<< (A+B);       B := L0 ;

     S15 := (S15+(A+B)) <<< 3;           A := S15;
```

```
     L1  := (L1 +(A+B)) <<< (A+B);        B := L1 ;

/**/
     S16 := (S16+(A+B)) <<< 3;            A := S16;
     L0  := (L0 +(A+B)) <<< (A+B);        B := L0 ;

     S17 := (S17+(A+B)) <<< 3;            A := S17;
     L1  := (L1 +(A+B)) <<< (A+B);        B := L1 ;

/**/
     S18 := (S18+(A+B)) <<< 3;            A := S18;
     L0  := (L0 +(A+B)) <<< (A+B);        B := L0 ;

     S19 := (S19+(A+B)) <<< 3;            A := S19;
     L1  := (L1 +(A+B)) <<< (A+B);        B := L1 ;

/**/
     S20 := (S20+(A+B)) <<< 3;            A := S20;
     L0  := (L0 +(A+B)) <<< (A+B);        B := L0 ;

     S21 := (S21+(A+B)) <<< 3;            A := S21;
     L1  := (L1 +(A+B)) <<< (A+B);        B := L1 ;

/**/
     S22 := (S22+(A+B)) <<< 3;            A := S22;
     L0  := (L0 +(A+B)) <<< (A+B);        B := L0 ;

     S23 := (S23+(A+B)) <<< 3;            A := S23;
     L1  := (L1 +(A+B)) <<< (A+B);        B := L1 ;

/**/
     S24 := (S24+(A+B)) <<< 3;            A := S24;
     L0  := (L0 +(A+B)) <<< (A+B);        B := L0 ;

     S25 := (S25+(A+B)) <<< 3;            A := S25;
     L1  := (L1 +(A+B)) <<< (A+B);        B := L1
  );

  B:=C1; A:=C0;             /* now decrypt */

  B:= ((B - S25) >>> A) ^ A;
  A:= ((A - S24) >>> B) ^ B;

  B:= ((B - S23) >>> A) ^ A;
  A:= ((A - S22) >>> B) ^ B;

  B:= ((B - S21) >>> A) ^ A;
  A:= ((A - S20) >>> B) ^ B;

  B:= ((B - S19) >>> A) ^ A;
```

```
  A:= ((A - S18) >>> B) ^ B;

  B:= ((B - S17) >>> A) ^ A;
  A:= ((A - S16) >>> B) ^ B;

  B:= ((B - S15) >>> A) ^ A;
  A:= ((A - S14) >>> B) ^ B;

  B:= ((B - S13) >>> A) ^ A;
  A:= ((A - S12) >>> B) ^ B;

  B:= ((B - S11) >>> A) ^ A;
  A:= ((A - S10) >>> B) ^ B;

  B:= ((B - S09) >>> A) ^ A;
  A:= ((A - S08) >>> B) ^ B;

  B:= ((B - S07) >>> A) ^ A;
  A:= ((A - S06) >>> B) ^ B;

  B:= ((B - S05) >>> A) ^ A;
  A:= ((A - S04) >>> B) ^ B;

  B:= ((B - S03) >>> A) ^ A;
  A:= ((A - S02) >>> B) ^ B;

  (P1 = B-S01) && (P0 = A-S00)        /* test for result equality */
)
end
```

B VHDL code for brute-force attack on TEA

B.1 Data Types: `crypt_pack.vhdl`

```
library ieee;
    use ieee.std_logic_1164.all;

package crypt_pack is

 subtype INT32 is STD_LOGIC_VECTOR(31 downto 0); -- MSB is 31, LSB is 0

 component crypt           -- Crypto core
  port(        k0, k1: in  INT32;
         RV0:     out STD_LOGIC;
        CLOCK:  in  STD_LOGIC;
        DONE:   out STD_LOGIC;
        RESET:  in  STD_LOGIC);
 end component;

 FUNCTION "xor" ( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR;
 FUNCTION "xor" ( l:INTEGER; r:STD_LOGIC_VECTOR ) RETURN STD_LOGIC_VECTOR;
-- FUNCTION "sll" ( l,r:STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
-- FUNCTION "srl" ( l,r:STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR;
 FUNCTION SHL( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR;
 FUNCTION SHL( l:INTEGER; r:STD_LOGIC_VECTOR ) RETURN STD_LOGIC_VECTOR;
 FUNCTION SHR( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR;
 FUNCTION SHR( l:INTEGER; r:STD_LOGIC_VECTOR ) RETURN STD_LOGIC_VECTOR;

 FUNCTION To_INT32 ( l:INTEGER ) RETURN STD_LOGIC_VECTOR;
 FUNCTION To_INT32 ( b:BIT_VECTOR ) RETURN STD_LOGIC_VECTOR;
end;

use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
package body crypt_pack is

 FUNCTION To_INT32 ( l:INTEGER ) RETURN STD_LOGIC_VECTOR is
 begin
   return CONV_STD_LOGIC_VECTOR(l, 32);
 end;
 FUNCTION To_INT32 ( b:BIT_VECTOR ) RETURN STD_LOGIC_VECTOR is
 begin
   return To_StdLogicVector(b);
 end;

 FUNCTION "xor" ( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR is
 begin
   return l xor CONV_STD_LOGIC_VECTOR(r, l'length);
 end;
 FUNCTION "xor" ( l:INTEGER; r:STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR is
 begin
   return CONV_STD_LOGIC_VECTOR(l, r'length) xor r;
 end;

-- FUNCTION "sll" ( l,r:STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR is
-- begin
--   return SHL(l, r);
```

```
-- end;

-- FUNCTION "srl" ( l,r:STD_LOGIC_VECTOR) RETURN STD_LOGIC_VECTOR is
-- begin
--   return SHR(l, r);
-- end;

-- FUNCTION "sll" ( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR is
--   return l(r to l'length) & 'r' zeros.

 FUNCTION SHL( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR is
 begin
     return SHL(l, CONV_STD_LOGIC_VECTOR(r, l'length));
 end;
 FUNCTION SHL( l:INTEGER; r:STD_LOGIC_VECTOR ) RETURN STD_LOGIC_VECTOR is
 begin
     return SHL(CONV_STD_LOGIC_VECTOR(l, r'length), r);
 end;
 FUNCTION SHR( l:STD_LOGIC_VECTOR; r:INTEGER ) RETURN STD_LOGIC_VECTOR is
 begin
     return SHR(l, CONV_STD_LOGIC_VECTOR(r, l'length));
 end;
 FUNCTION SHR( l:INTEGER; r:STD_LOGIC_VECTOR ) RETURN STD_LOGIC_VECTOR is
 begin
     return SHR(CONV_STD_LOGIC_VECTOR(l, r'length), r);
 end;

end;
```

B.2 Driver chip: `driver.vhdl`

```
Library IEEE;
use IEEE.std_logic_1164.all;
use WORK.CRYPT_PACK.ALL;
use IEEE.std_logic_unsigned.all;

entity driver is -- Crypto machine pin-out
  port(        CRYPTCLOCK: in  STD_LOGIC; -- the results of the crypto computation.
        RESULTIN:   in  STD_LOGIC;
        RESULTOUT:  out STD_LOGIC;

        DATACLOCK:  in  STD_LOGIC;
        DATAIN:     in  STD_LOGIC;
        DATAOUT:    out STD_LOGIC;

        RUN:     in  STD_LOGIC);
end;

architecture BEHAVIOR of driver is
        signal key, nextkey:STD_LOGIC_VECTOR(63 downto 0);
        signal found, nextfound, cryptdone:STD_LOGIC;
        signal CLOCK:STD_LOGIC;
        signal CRYPTRESET:STD_LOGIC;
begin
  CLOCK <= CRYPTCLOCK when RUN = '1' else DATACLOCK;
  CRYPTRESET <= not RUN;

  MAINCLOCK: process
  begin
    wait until CLOCK'event and CLOCK = '1';
    if RUN = '0' then   -- process to perform serial i/o
      DATAOUT <= found;
      found   <= key(63);
      key     <= key(62 downto 0) & DATAIN;
    else                -- clock the computation
      if (cryptdone = '1') then
        found<= nextfound or found;
        if (found or nextfound) = '0' then
          key  <= nextkey;
        end if;
      end if;
    end if;
  end process;

  -- process to perform computation
  DOCRYPT: process(key, nextkey, found, nextfound)
    constant mask:BIT_VECTOR := X"8000_0000_0000_000D";
    variable temp:STD_LOGIC_VECTOR(63 downto 0);
  begin
    if key(0) = '0' then
      nextkey <= key(0) & key(63 downto 1);
    else
      temp := key xor To_StdLogicVector(mask);
      nextkey <= key(0) & temp(63 downto 1);
    end if;

  end process;
```

```
  CRYPT0: crypt
  port map (
      k0 => key(31 downto  0), -- lsb
      k1 => key(63 downto 32), -- msb
      RV0=> nextfound,
      CLOCK => CRYPTCLOCK,
      DONE=>cryptdone,
      RESET=>CRYPTRESET
  );

  -- external result flag
  RESULT: process(RESULTIN, found)
  begin
    RESULTOUT <= RESULTIN or found;
  end process;
end BEHAVIOR;
```

B.3 Cryptographic Engine: `crypt.vhdl`

```
Library IEEE;
    use IEEE.std_logic_1164.all;
    use IEEE.std_logic_signed.all;
    use WORK.CRYPT_PACK.ALL;

entity crypt is         --Computational core
  port( k0, k1: in  INT32;
        RV0:   out STD_LOGIC;
        CLOCK: in  STD_LOGIC;
        DONE:  out STD_LOGIC;
        RESET: in  STD_LOGIC);
end;

architecture BEHAVIOR of crypt is
-- register definitions
  signal t74,t171:INT32;
  signal t80,t170:INT32;
  signal t132,t175:INT32;
  signal t59,t173:INT32;
  signal t144,t174:INT32;
  signal t68,t172:INT32;
-- state definitions
  signal State2, nextState2:BIT;
  signal State0, nextState0:BIT;
  signal State1, nextState1:BIT;
begin
  CLOCKSTATE: process
  begin
    wait until (CLOCK'event) and (CLOCK='1');
    -- new state:
    State2 <= nextState2;
    State0 <= nextState0;
    State1 <= nextState1;
    -- new registers:
    t74 <= t171;
    t80 <= t170;
    t132 <= t175;
    t59 <= t173;
    t144 <= t174;
    t68 <= t172;
  end process;
  NEXTSTATE: process(k0,k1,
         State2,
         State0,
         State1,
         t74,
         t80,
         t132,
         t59,
         t144,
         t68,
         RESET)
    variable t105:INT32;
    variable t97:INT32;
    variable t100:INT32;
    variable t158:INT32;
```

```
  variable t92:INT32;
  variable t118:INT32;
  variable t154:INT32;
  variable t159:INT32;
  variable t119:INT32;
  variable t101:INT32;
  variable t114:INT32;
  variable t93:INT32;
  variable t161:INT32;
  variable t106:INT32;
  variable t120:INT32;
  variable t110:INT32;
  variable t102:INT32;
  variable t98:INT32;
  variable t94:INT32;
  variable t90:INT32;
  variable t121:INT32;
  variable t111:INT32;
  variable t103:INT32;
  variable t155:INT32;
  variable t112:INT32;
  variable t115:INT32;
  variable t104:INT32;
  variable t107:INT32;
  variable t99:INT32;
  variable t91:INT32;
  variable t95:INT32;
  variable RV:INT32;
  variable t116:INT32;
  variable t108:INT32;
  variable t96:INT32;
  variable t117:INT32;
  variable t113:INT32;
  variable t109:INT32;
begin
  DONE <= '0';
  RV0  <= '0';
  -- default state:
  nextState2 <= '0';
  nextState0 <= '0';
  nextState1 <= '0';
  -- default registers:
  t171 <= t74;
  t170 <= t80;
  t175 <= t132;
  t173 <= t59;
  t174 <= t144;
  t172 <= t68;
  if RESET = '1' then
    nextState1 <= '1';
  -- STATE MACHINE
  elsif State2 = '1' then
      t154 := t144; -- move
      t155 := t132; -- move
      if
        t154 = To_INT32(45)
      then
```

```
        if
          t155 = To_INT32(56)
        then
        t158 := To_INT32(1); -- phi2
        t159 := t158; -- move
        t161 := t159; -- phi2
        else
        t158 := To_INT32(0); -- phi2
        t159 := t158; -- move
        t161 := t159; -- phi2
        end if;
        else
        t161 := To_INT32(0); -- phi2
        end if;
        RV := t161; -- move
        nextState1 <= '1';
        DONE <= '1';
        if RV=0 then RV0<='0'; else RV0<='1'; end if;
    elsif State0 = '1' then
        t90 := t68; -- move
        t91 := SHL(t80, To_INT32(4));
        t92 := t91; -- move
        t93 := k0; -- ARG
        t94 := t92 + t93;
        t95 := t94; -- move
        t96 := t80 + t59;
        t97 := t95 xor t96;
        t98 := t97; -- move
        t99 := SHR(t80, To_INT32(5));
        t100 := t99; -- move
        t101 := k1; -- ARG
        t102 := t100 + t101;
        t103 := t98 xor t102;
        t104 := t90 - t103;
        t105 := t80; -- move
        t106 := SHL(t104, To_INT32(4));
        t107 := t106; -- move
        t108 := k0; -- ARG
        t109 := t107 + t108;
        t110 := t109; -- move
        t111 := t104 + t59;
        t112 := t110 xor t111;
        t113 := t112; -- move
        t114 := SHR(t104, To_INT32(5));
        t115 := t114; -- move
        t116 := k1; -- ARG
        t117 := t115 + t116;
        t118 := t113 xor t117;
        t119 := t105 - t118;
        t120 := t59 - To_INT32(654329);
        if
          t74 < To_INT32(32)
        then
        t121 := t74 + To_INT32(1);
        nextState0 <= '1'; -- goto
        t170 <= t119; -- phi1
        t171 <= t121; -- phi1
```

```
        t172 <= t104; -- phi1
        t173 <= t120; -- phi1
        else
        nextState2 <= '1'; -- goto
        t174 <= t119; -- phi1
        t175 <= t104; -- phi1
        end if;
    elsif State1 = '1' then
        nextState0 <= '1'; -- goto
        t170 <= To_INT32(12); -- phi1
        t171 <= To_INT32(1); -- phi1
        t172 <= To_INT32(23); -- phi1
        t173 <= To_INT32(20938528); -- phi1
    end if;
  end process;
end;
```