Portions presented at DesignCon98,1998 International Verilog Conference, & 1998 FPGA Configurable Computing Machine Conference

Implementing C Designs in Hardware:

Donald Soderman

Abstract

The capability to compile behavioral C-language designs into RTL Verilog for implementing control, data-path, and DSP algorithms in FPGA/ASIC hardware is described. With specific RTL edits, high level system algorithms have been implemented in hardware with efficiencies approaching that of manually written Verilog descriptions. Global analysis allows multiple Verilog concurrent and sequential "always" blocks with state machines for loops, shared arithmetic macros, and external memory interfaces to be created. Expert users can selectively unroll loops and specify register memory arrays using local and global compiler directives. Multiple C operations are then executed in parallel to achieve the desired system performance.

Integer bit widths, fixed-point types and sign-bit manipulations were individually specified to use the minimum hardware resources. Verilog test-bench simulation files were extracted from the C behavioral debug files to verify that the RTL behavior with clock and other hardware control signals corresponds to the sequentially executed C code. The Verilog hardware code executes in substantially fewer clock cycles (ranging from nearly 1/4 to 1/25) compared to software executing on a PentiumPro. This cycle efficiency is due to variable storage in simple registers, clock packing techniques, and functional level parallelism. Actual run time performances are dependent on the type of FPGA and ASIC hardware.

This design methodology is demonstrated by compiling Finite Response Filers, Compression-Decompression, Encryption, Prime Number, and a Sorting algorithm into ASIC/FPGA hardware. Using local compiler directives, FIR filters implemented with various resource sharing configurations and parallel execution have been created. Also, an RSA encryption algorithm has been compiled and synthesized into an FPGA.

1. INTRODUCTION

A full-featured ANSI C to synthesizable RTL Verilog compiler was used to implement several system-level algorithms in hardware. These designs were created in C and compiled into RTL Verilog. Impressive results were achieved for designs ranging from (sequential memory intensive) encryption, compression, and sorting algorithms to (combinatorial logic intensive) prime number generation and FIR filter algorithms.

Presently many large systems are initially described using an intuitive, high-level behavioral language, such as C. The system designer defines global functionality using data types, operators, expressions, and functions. System feasibility can be analyzed without worrying about specific timing considerations.

ANSI C is an ideal choice for high-level system behavioral design since it is familiar to the majority of system engineers. Also a wide variety of environments for code debug are available. Previously, such system specifications had to be manually re-coded into Verilog and verified using lengthy simulation prior to synthesis and hardware implementation.

The hierarchical design approach allows memory and other optimized arithmetic macros to be integrated in large ASIC and FPGA hardware, as shown in Figure 1. With such a compiler, the system architect can participate in the hardware design process and accelerate development schedules by avoiding the common “over-the-wall” transfer.

In addition to compiling logic functionality, this approach generates simulation test-benches from the C stimulus and debug files. Thus, the design behavior after compilation into RTL Verilog can be readily compared to the results of sequential C statement execution to accelerate design verification.

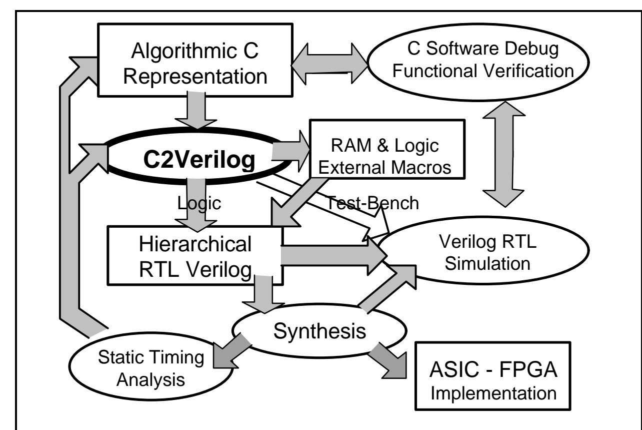

Figure 1: Design Flow

Copyright 1997-1998 ASIC DESIGN & MARKETING where majority of this work was performed

2. COMPILER OPERATION

The compiler accepted standard ANSI C data types, operations, variables, and functions. It performed global analysis to cluster operations into multiple Verilog always blocks and concurrent assignments to achieve parallel execution. State machines were created for loops, shared arithmetic macros, and external memory interfaces. Memory was needed to accommodate addressable variables, such as arrays. The compiler supported pointers, structures, loops and function calls, which are used in most C code.

<table>
<tr><td>Data Types</td><td>void, char, short, int, unsigned, long, float, intNNN, arrays, multi-dimensional arrays, pointers, structs, unions, enums</td></tr>
<tr><td>Operations</td><td>ANSI C usual integer promotion, call A++, A--, ++a, --a, pointer a++ and –a, &&, | |, <<, >>, <<=, >>=, sizeof, signed & unsigned < <= > >= == != =, + - & | ^ += -= &= |= ^= * *= / % /=, unary & and *</td></tr>
<tr><td>Optimization</td><td>optimizes integer promotion & enums, control flow, local expressions, constant variables, elimination of duplicate string literals, state machines, global optimization</td></tr>
<tr><td>Operators</td><td>simple & complex if, while, loops, do loops, for loops, simple & complex switch, goto labels, while- do- for- loop-break, switch break, return, while- do-for- loop-continue</td></tr>
<tr><td>Variables</td><td>register & memory based,
input & output wires</td></tr>
<tr><td>Functions</td><td>function and function calls, pipelining, functional level parallelism, synchronous event detection, and hierarchical functions</td></tr>
<tr><td>General Features</td><td>external memory, on-chip static memory for FPGAs and for Synopsys DesignWare, state machines, memory data bit widths, initialization, input/output declarations</td></tr>
<tr><td>Output</td><td>RTL Verilog for synthesis and simulation, executions report identifying variable and function usage</td></tr>
</table>

The compiler handled both concurrent and sequential statements by dividing the functions into clusters, which were executed in parallel using multiple *always* blocks.

Functions were grouped into the same cluster if 1) multiple functions write into the same variable, 2) one function calls another directly or indirectly, or 3) they share memory, multipliers, dividers, or other external functions.

Sequential Verilog statements are executed within *always* blocks using blocking RTL assignments. Thus, functions could be compiled into: 1) simple continuous assignments, 2) sequence of statements initiated by a clock, or 3) sequence of statements implemented using a state machine architecture.

Functions were implemented as state machines when one of the following circumstances occurred:

- Pointer/array indirection requiring addressable memory access
- *while, do,* or *for* loop or *goto* statements
- Calls to another function implemented using a state machine

Memory and other optimized arithmetic macros were incorporated in the hierarchical Verilog. Additional interface signals, such *as run* and *ready,* were created for function interface with the external system. Memory access utilizes signals, such as *RAM_data, RAM_out, RAM_addr*, and *RAM_we*.

Additional clock cycles in a state machine were manually introduced for pipelining using the pseudo-function *next_clock*. This "user-driven" capability was important for balancing the logic execution times in each operation. The optimum hardware efficiency and performance was obtained by iterating through various C code enhancements, compiler options, and logic synthesis preferences. This allowed insertion of registers and wait states to balance propagation delays.

The compiled RTL Verilog was then be synthesized using any one of a number of products from Synopsys, Exemplar, Synplicity, etc., to create a netlist for specific ASIC or FPGA target hardware. Various physical design tools were then be employed to place and route the design for the hardware implementation.

A Verilog simulation test-bench was compiled from the C code used to debug the design. Thus, the behavior generated from the sequential execution of C statements could be readily compared to that generated by the execution of RTL Verilog statements in a digital simulator. This test-bench design verification could be performed on a Verilog module created by the compiler or a manually generated module.

3. SIMPLE EXAMPLE

The following examples illustrate compiler operation. Figure 2 shows a simple C example of a function returning the AND of two variables *a* and *b*. As shown in Figure 3, the compiler generates RTL Verilog using continuous assignments with signal names corresponding to the C variable names.

```
int demoAnd (int a, int b)
{
  return a & b;
}
```

Figure 2: Simple C Example

```
module main (result_demoAnd, in_a, in_b);

input  [15:0] in_a, in_b;
output [15:0] result_demoAnd;

wire   [15:0] result_demoAnd = in_a & in_b;

endmodule
```

Figure 3: RTL Compiled Verilog Continuous Assignments - Variables Specified as 16b

Specific global bit widths, memory implementation, options for standard C functions, initialization, left and right shifts, and multiplication-divider options using a graphical user interface, as illustrated in Figure 4 were specified

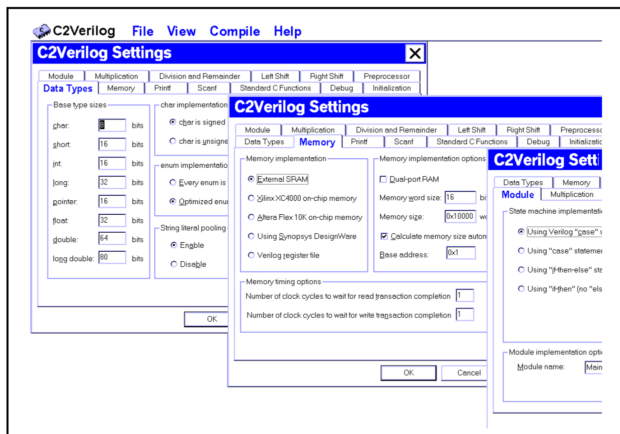

Figure 4: Compiler Settings

C function parameters infer module inputs. C function return values infer module outputs, unless the C function is static in which case no ports are generated for this function. Variables that are "read-only" are considered inputs, while those "written-to" are considered outputs. The compiler optionally implements variables in scanf and printf as module input and output wires.

4. SIMPLE PARALLEL EXECUTION EXAMPLE

Figure 5 illustrates how a simple C description was compiled into parallel execution *always* blocks for multiple-stage pipelines. The first stage adds "1" to the value of *data_in* and stores the value as *out1*. The second stage adds the value of *out1* to itself and stores the result as *out2*. The third stage performs an "OR" of *out2* with the constant "1234" and stores the result as *data_out*. The three *always* blocks allow these operations to occur in parallel.

```
int data_in; int out1; int out2; int
data_out;
void pipeline_stage_1 ()
{   out1 = data_in + 1;   }
void pipeline_stage_2 ()
{   out2 = out1 + out1;   }
void pipeline_stage_3 ()
{   data_out = out2 ^ 1234;   }

void main ()
{ for (;;)
    { scanf ("%d", & data_in);
      pipeline_stage_1 ();
      pipeline_stage_2 ();
      pipeline_stage_3 ();
      printf ("%d", data_out);
    }
}
```

Figure 5: C Example Illustrating Parallelism

```
module main(clock, data_in, data_out,
   run_pipeline_stage_1, run_pipeline_stage_2,
   run_pipeline_stage_3);
  input          clock, run_pipeline_stage_1, . . .
  input  [15:0] data_in;
  output [15:0] data_out;
  reg    [15:0] data_out, out1, out2;

  always @(posedge clock)
  begin
    if (run_pipeline_stage_1)
        out1 = (data_in + 16'd1);
  end

  always @(posedge clock)
  begin
    if (run_pipeline_stage_2)
        out2 = (out1 + out1);
  end

  always @(posedge clock)
  begin
    if (run_pipeline_stage_3)
        data_out = (out2 ^ 16'd1234);
  end
endmodule
```

Figure 6: RTL Verilog for Parallel Execution of 3 Stage Pipeline

5. LOOPS IMPLEMENTED WITH VERILOG STATE-MACHINES

The C description in Figure 7 has multiple *for* loops. The first function *sum1* with local variable input *n* returns a value for the sum of the first *n* integers. The second function *sum2* returns a value for the sum of integers in *array* of size *size*. The main function loops through the values of *i* ranging from 0 to *size* and squares the value in the array *array*. Then it adds the two integers returned from functions *sum1* and *sum2*.

```
int sum1 (int n)
 {
  int i, sum = 0;
  for (i = 0; i < n; i ++)
     sum += i;
  return sum;
 }
int sum2 (int array [], int size)
 {
  int i, sum = 0;
  for (i = 0; i < size; i ++)
      sum += array [i];
  return sum;
 }
int main ()
 {
  int i;
  int array [10];
  int size = sizeof (array) / sizeof (*array);
  for (i = 0; i < size; i++)
      array [i] = i * 2;
  return sum1 (size) + sum2 (array, size);
 }
```

Figure 7: Example with Simple “For” Loops

A state machine was created to sequence through the various memory access and logic operations, as illustrated in Figure 8. The first group of statements in state *0* resets the variables stored in memory. The subsequent transitions through states named *sum1*, *2*, *3*, *4*, *sum2*, … perform the intended operation. The Verilog state machine transitions first clear the return values of *sum1*, then test the inequality and branch to states *3* or *4*.

In state *3,* the index is incremented and the value of *sum1* is added to itself. Then it returns to state *2* and retests the inequality statement. In state *4*, the ready signal is enabled, indicating completion of this loop and memory storage, followed by a transition to state *0* (start state) for evaluating other functions, such as *sum2*.

6. PARALLELISM & PIPELINING

A more complicated example combining parallel execution with loops in multiple functions is shown in Figure 9. The first pipeline stage generates the value of *out1* depending on the value of the input *data_in*. The second stage generates the value of *out2* to be the "OR" of *out1* and constant "0x1EE". The third stage generates the value of *data_out* to be either "2" or *out + 1*, depending on the value of *out2*. This example illustrates how two state machines working in parallel and exchanging data through common registers form a data pipeline.

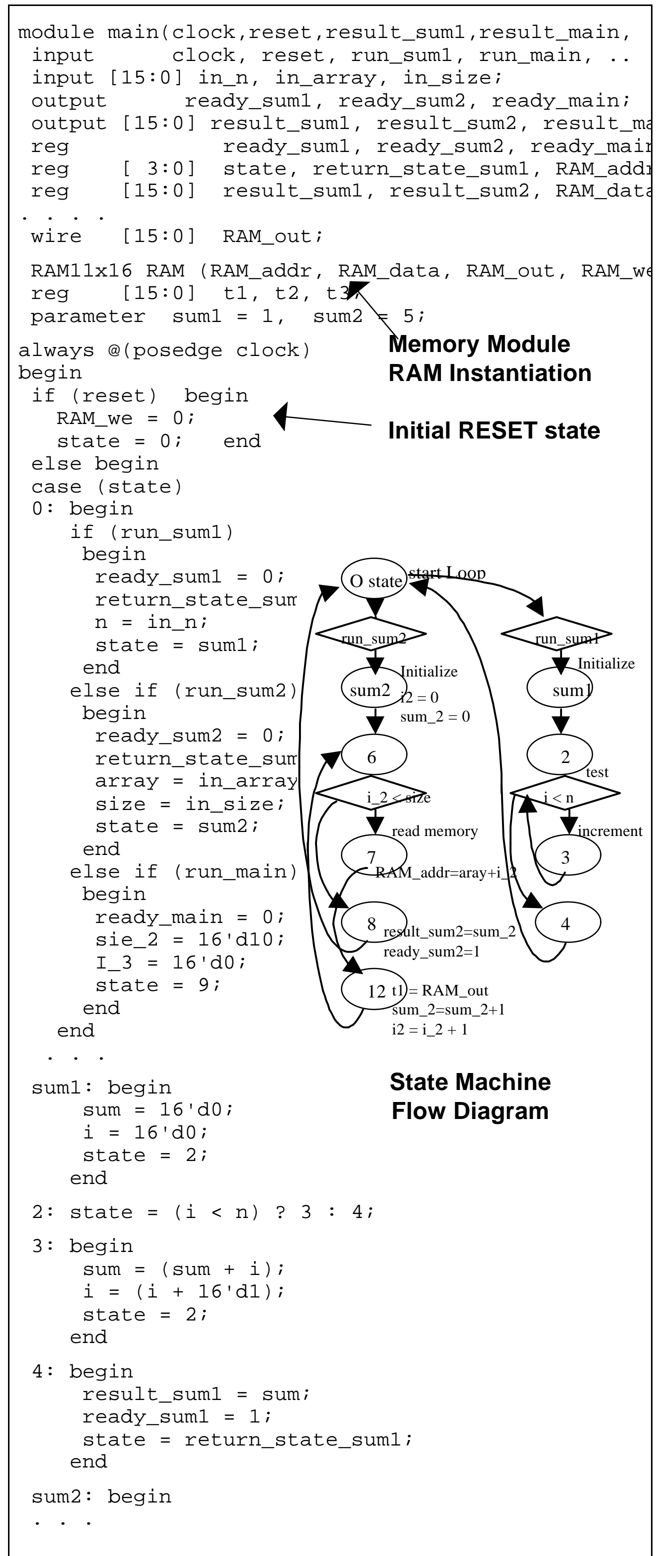

Figure 8: RTL Verilog for “For” Loop Example Using State-Machine

The compiled Verilog for this example is shown in Figure 10. The first and third functions are implemented as state machines because of loops, while the second can be implemented as a simple clock assignment. The register output of the first stage *out1* drives the input of the second function. The register output of the second stage *out2* drives the input of the third function.

Functions can be run in parallel if and only if they are from two different clusters. Two functions from the same cluster run sequentially only after the completion of each other. An unlimited number of C functions can be run in parallel, provided they are all from different clusters. There are no restrictions on reading from the same variable by two functions from two different clusters; thus, it is possible to program hardware pipelining in C.

```
int data_in;      int out1;
int out2;         int data_out;

void pipeline_stage_1 ()
{
   while (data_in == 0)
      out1 = 0;
   if (data_in == 1)
      while (data_in == 1)
         out1 = 2;
}

void pipeline_stage_2 ()
{
   out2 = out1 ^ 0x1EE;
}

void pipeline_stage_3 ()
{
   if (out2 == 1)
      while (out2 == 1)
            data_out = 2;
   while (out2 != 0)
      data_out = out2 + 1;
}
```

Figure 9: Multiple Functions with Loops

7. COMPLICATED EXAMPLE

A more complex C example using pointers, structures, user-defined types, loops, and function calls is shown in Figure 11. In this example, the C description (file named tree.c) performs a binary search in structure trees using pointers and loops. The structures/functions have global and local variables, requiring local (Verilog register) and external memory data storage.

A report file summarized the compiler actions as shown in Figure 12. The resulting Verilog (partially shown in Figure 13) uses a state machine and a single *always* statement for memory interface to retain values in loop operations. The compiler introduced several control signals: global *clock* and *reset* and multiple *run* and *ready* signals to sequence through the state machine for logic execution and memory interface.

```
module main(clock, reset, data_in ,data_out,
run_pipeline_stage_1,
 . . .

always @(posedge clock)
 begin
    if (reset)     state =
    else    begin
      case (state)
      0: begin
           if (run_pipeline_stage_1)
           begin
             ready_pipeline_stage_1 = 0;
             state = 1;
           end
         end

      1: state = (data_in == 16'd0) ? 2 : 3;

      2: begin
           out1 = 16'd0;
           state = 1;
         end

      3: state = (data_in == 16'd1) ? 4 : 5;

      4: begin
           out1 = 16'd2;
           state = 3;
         end

      5: begin
           ready_pipeline_stage_1 = 1;
           state = 0;
         end

       default:    ;
       endcase
      end
 end

 always @(posedge clock)
 begin
    if (run_pipeline_stage_2)
       out2 = (out1 ^ 16'd494);
 end

 always @(posedge clock)
 begin
   if (reset)    state_2 =
   else
   begin
     case (state_2)
      0: begin
           if (run_pipeline_stage_3)
           begin
              ready_pipeline_stage_3 = 0;
              state_2 = 1;
            end
          end

      1: state_2 = (out2 == 16'd1) ? 2 : 3;

      2: begin
            data_out = 16'd2;
            state_2 = 1;
         end
 . . .
```


Figure 10: Compiled Verilog for Parallel Loop Execution

```
struct _Node
 {  struct _Node * pLeft;
    struct _Node * pRight;
    int           nKey;
    intnValue;
 };
typedef struct _Node NODE;

static NODE Nodes [MAX_NODES];
static int nNodes;

NODE * pTree;

void Initialize (void)
 { nNodes = 0;
   pTree = NUL; }

static NODE * NewNode (int nKey, int nValue)
 {. . .}

NODE * FindNode (int nKey)
 {. . .}

int FindValue (int nKey)
 { NODE * p;
   return(p = FindNode(nKey))== NULL ? -1 :
          p -> nValue;
 }

NODE * FindOrAddNode (int nKey, int nValue)
 { NODE * * p = & pTree;
   p = & pTree;
   while (*p != NULL)
    { if ((*p) -> nKey < nKey)
            p = & (*p) -> pLeft;
      else if ((*p) -> nKey > nKey)
            p = & (*p) -> pRight;
      else
            return (*p);
    }
    return *p = NewNode (nKey, nValue);
 }
. . .
```

Figure 11: Complicated C example tree.c user defined types, loops, & function calls

```
      Function Translation Report
-----------------------------------------------
-
Function    Implementation Run Ready Result Other
Name        Type                Signals  Features
-----------------------------------------------
FindNode   state machine run ready result memory
Initialize state machine run ready        memory
NewNode    state machine             result static
FindOrAddNode stat mach run ready result memory
. . .
-----------------------------------------------
       Variable Translation Report
-----------------------------------------------
Variable Funct  In Out Reg I/O/Reg/Addr Addr Size
Name     Name     Wires Mem  Name     Decimal Words
-----------------------------------------------
Nodes                      mem  Nodes        1 40
nNodes                     reg  nNodes          1
pTree                      mem  pTree       41  1
nKey        NewNode        reg  nKey            1
nValue      NewNode        reg  nValue          1
nKey        FindNode       reg  nKey_2          1
                in          in_nKey_2
p           FindNode       reg  p               1
nKey        FindValue  in       nKey_3          1
. . .
-----------------------------------------------
       Function Call Report
-----------------------------------------------
```

Figure 12: Compilation Report

```
moduleTree(clock, reset, FindOrAddNode, . . .
 . . .
always @(posedge clock)
begin
 if (reset)  begin
   RAM_we = 0;  . . .
 else   begin
  case (state)
 0: begin
   if (run_Initialize)   begin
     ready_Initialize = 0;
     nNodes = 16'd0;
     RAM_addr = `pTree;  RAM_data [15:10] = 16'd0;
      RAM_we = 1;          state = 23;        end
    else if (run_FindNode)  begin  . . .
 . . .
 23: begin
   RAM_we = 0;  ready_Initialize = 1;  state = 0;
 . . .
 FindNode: begin
          RAM_addr = `pTree;  state = 28;   end
 28: begin t2 = RAM_out [15:10];
           p = t2;          state = 6;    end
 6: state = (p != 16'd0) ? 7 : 12
 7: begin  RAM_addr = (p + 6'd2); state = 29;
29: begin  t3=RAM_out; state=(t3 < nKey_2) ? 8:
 8: begin  RAM_addr = p;       state = 30;
 FindOrAddNode: beg
            p_3 = `pTre       state = 15; end
15: begin  RAM_addr = p_3;  state = 38;  end
38: begin  t10 = RAM_out [15:10];
          state = (t10 != 16'd0) ? 16 : 21;  end
16: begin  RAM_addr = p_3;  state = 39;   end
 . . .
21: begin  nKey = nKey_4;   nValue = nValue_2;
    return_state_NewNode=46; state=NewNode; end
46: begin  t18 = result_NewNode;
    RAM_addr = p_3;  RAM_data [15:10] = t18;
    RAM_we = 1;   state = 47;   end
47: begin  RAM_we = 0;
    result_FindOrAddNode=t18;  state=22;  end
22: begin  ready_FindOrAddNode = 1;
    state=return_state_FindOrAddNode;  end
 . . .
```


Figure 13: RTL Verilog Compiled from tree.c

There are five structures/functions requiring common memory access: *NewNode, FindNode, FindValue, FindOrAddNode*, and *FindOrAddValue*. The variables *nKey* and *nValue* in the *NewNode* structure are compiled into Verilog registers with names *nKey* and *nValue*, while the variables *p* and *nKey* in the *FindNode* structure are compiled into Verilog registers with names *p_2* and *nKey_2*. The Verilog register name is the concatenation of the variable name with the instance number to improve designer understanding and avoid confusion with duplicate local variable names. Since *nKey* is an input to all three functions, it is compiled as multiple input signals with multiple names, such as *nKey*, which are merged during logic synthesis. Because these variables are interdependent, they must be evaluated sequentially, and the entire design with multiple loops is compiled into one state machine.

This tree example uses a variable as a pointer to a pointer and uses pointers to structures. Thus, a complicated state machine is required to access temporary variables stored in external memory. This design requires multiple read and write operations to variables stored in a 42 word by 16 bit RAM. During a read operation, the RAM address is produced and the RAM data read on the subsequent clock cycle. During a write operation, both the data and address are produced on the same clock cycle along with a *RAM_we* signal, as illustrated in state *0* after *if(run_Initialize)* and state *3*.

The five loop operations in this example require five sequences of state transitions, starting-at and returning-to the initial state *0*, as shown in Figure 13. The conditional *if* clauses are used to select the appropriate state transition path from this initial state *0* and the transitions within each group of states.

8. LZW COMPRESSION/DECOMPRESSION ALGORITHM

The C to Hardware design flow has been used for several real algorithms. The first design example is the LZW compression and decompression algorithm which uses arrays, indirections, pointers, loops, and functions. Portions of the C code illustrating the usage of these operations are shown in Figure 14.

The LZW algorithm was initially compiled using Visual C++ with a test-bench for software execution on Pentium micro-processors. The compression and decompression operation is demonstrated using a test pattern of 1020 characters. The compression ratio for this string is 55% = 561/1020 and requires 144,954 Pentium Pro clock cycles.

This C design was compiled into Verilog for digital simulation along with the test-bench. A portion of the initial ASCII character string with hex equivalent and its compressed string are shown in Figure 15.

```
Initial Text length: 1020 characters
ASCII =  A  A  A  A  A  F  F  F  F  B  K  ...
HEX   = 41 41 41 41 41 46 46 46 46 42 4B  ..
        ...  B  A  B  A  H  H  B
        ... 42 41 42 41 48 48 42

Compressed length: 561 characters  55% compressio
ASCII =  A  -  -  F  e  F  B  K  D  K  K  A  C ..
HEX   = 41 7F 7F 46 82 46 42 4B 44 4B 4B 41 43 ..
        ...  -  -  -  -  -
        ... D1 9B D4 C6 C7

Number of Pentium clock cycles:  144,954
Number of hardware clock cycles:  40,684
Software Requires 3.5x More Cycles than Hardware
```

Figure 15: Verilog Simulation Results of LZW

```
static int find_match
( int hash_prefix,      unsigned hash_character
void compress (void)
{ unsigned next_code;    unsigned character;
  unsigned string_code;  unsigned index;  int i;
  rewind_text ();
  rewind_compressed ();
  next_code = FIRST_CODE;
  for (i = 0; i < TABLE_SIZE; i++)
    code_value [i] = -1;
  i = 0;
  string_code = input_text ();
  while ((character = input_text ()) != '\0')
    {index=find_match (string_code, character);
     if (code_value [index] != -1)
      {string_code = code_value [index]; }
     else {if (next_code <= MAX_CODE)
          {code_value [index] = next_code++;
           prefix_code [index] = string_code;
           append_character [index] = character;
           output_compressed (string_code);
           string_code = character; } }
      output_compressed (string_code);
      output_compressed (MAX_VALUE);
      output_compressed (0);
}  . . .
static int find_match
( int hash_prefix, unsigned hash_character )
{ int index;       int offset;
  index=(hash_character<<HASHING_SHIFT) ^ hash_p
  if (index == 0)  offset = 1;
  else             offset = TABLE_SIZE - index;
    while (1)
     {if (code_value [index] == -1)
      return index;
      if ( prefix_code [index] == hash_prefix
         && append_character[index] == hash_chara
      {return index; }
      index -= offset;
      if (index < 0)  index += TABLE_SIZE;  }
}
void expand (void)
{ unsigned next_code;      unsigned new_code;
  unsigned old_code;       int character;
  unsigned char * string;
  static unsigned char  * decode_string ();
  rewind_compressed ();
  rewind_expanded   ();
  next_code = FIRST_CODE;
  old_code=input_compressed ();
  character=old_code;
  output_expanded (old_code);
  while ((new_code=input_compressed ()) != MAX_V
   {if (new_code >= next_code)
    { *decode_stack = character;
      string = decode_string (decode_stack + 1, o
    }
    else
    { string = decode_string (decode_stack, new_c
    character = *string;
    while (string >= decode_stack)
       output_expanded (*string);
  . . .
```

Figure 14: LZW Compression/Decompression C Algorithm

Software execution requires more clock cycles than hardware execution for the following reasons:

1. Most C variables are memory based, requiring sequential read/modify/write operations with multiple clock cycles for software execution. Hardware implementations use simple registers which can be read, modified, and written in a single cycle.
2. Hardware implementations are able to use a clock packing technique where several C expressions can be executed during each cycle.
3. Hardware implementation allows several functions to be implemented in parallel, whereas software implementation sequentially evaluates each function.

Although this example represents a worst-case condition because of the excessive memory accesses, the compiled RTL Verilog requires only 40,684 clock cycles (as determined by Verilog simulation) compared to 144,495 clock cycles (as estimated by the execution time minus overhead for the Pentium Pro 180MHz). The number of clock cycles for software execution would be substantially larger for a 486 type micro-processor without a large cache. Thus, for this algorithm, software executing on a 180 MHz Pentium (144,495 cycles/180MHz = 0.8 ms) is nearly twice the speed of XC4013-3 FPGA hardware execution (40,684 cycles/20MHz = 2.0 ms) but nearly half the speed of ASIC hardware execution (40,684 cycles/80MHz = 0.5 ms).

The LZW report file shown in Figure 16 indicates that length of characters, 1189 x 16 bits of external SRAM memory are required. The 4 functions (*find_match, compress, expand,* and *decode_string*) are compiled into a state machine. This report also contains the list of variable/function names, type, decimal address, and word size. The state machines sequence through memory IO and arithmetic operations as shown in portions of the generated RTL Verilog code in Figure 17.

9. HARDWARE IMPLEMENTATION

Figure 18 shows the simple structural Verilog template instantiating the RTL Verilog module compiled from the LZW C code for implementation in APSx84 FPGA hardware. This hierarchical structure allows other external memory and arithmetic macro modules to be added if desired. The FPGA input-output signal pin assignments to the PC interface and external SRAM memory are defined within this top hierarchical module. A C interface program downloads the FPGA configuration data stream and ports data on/off the ISA bus.

```
Command Line c2vlog.exe -bsp 11 LZW.c
 Function Translation Report   1189 x 16 bit SRAM
-------------------------------------------------
Function  Implementation  Run Ready Result  Other
Name      Type             Signals        Features
-------------------------------------------------
decode_string state machine      result static mem
find_match    state machine      result static mem
compress      state machine run  ready     memory
expand        state machine run  ready     memory
-------------------------------------------------
   Variable Translation Report
-------------------------------------------------
Variable Function  In Out Reg I/O/Reg/Addr Addr Si
Name     Name      Wires Mem   Name        Dec  Wd
-------------------------------------------------
text                    mem  text          1   80
next_text               reg  next_text          1
next_compressed         reg  next_compressed    1
expanded                mem  expanded      81  80
next_expanded           reg  next_expanded      1
code_value              mem  code_value   161  257
prefix_code             mem  prefix_code  418  257
append_character        mem  append_char> 675  257
decode_stack            mem  decode_stack 932  257
next_code  compress     reg  next_code          1
character  compress     reg  character          1
string_co> compress     reg  string_code        1
index      compress     reg  index              1
i          compress     reg  i                  1
hash_pref> find_match   reg  hash_prefix        1
hash_char> find_match   reg  hash_character     1
 . . .
------------------------------------------------
   Function Call Report
------------------------------------------------
compress:   find_match
expand:     decode_string
```

Figure 16: Compiler Report for LZW Example

```
module LZW(clock,reset,result_read_output_bit, ..
 input     clock, reset, write_input, run_compress,
 output    RAM_we,ready_compress, . . .
 output [15:0] RAM_data,
 output [10:0] RAM_addr;
 reg [15:0] result_find_match,character,index,t1,
  . . .
always @(posedge clock)
 begin
  if (reset)
    begin      RAM_we = 0;     state = 0;     end
  else
   begin  case (state)
    0: begin
     if (run_compress)   begin  ...
     else if (run_expand)   begin  ...
   find_match: begin
     index_2=((hash_character<<16'd1) ^ hash_pre
     if ((index_2 == 16'd0))    offset = 16'd1;
     else offset=(16'd257 - index_2); state = 2;

   2: state = 16'd1 ? 3 : 8;

   3: begin
       RAM_addr = (`code_value + index_2);
       state = 34;
       . . .
  34: begin
       t1 = RAM_out;
       state = (t1 == - 16'd
       . . .
    4: begin
         result_find_match = index_2;
. . .
```


Figure 17: Portions of RTL Verilog for LZW Example

The RTL Verilog for this design synthesizes into ~390 CLB Xilinx FPGA logic resources using Synopsys FPGA Express, Exemplar Leonardo, and Synplicity Synplify EDA products. This 97% utilization exceeds the routing capabilities of the XC4010e, but easily fits into a 4013e and operates with a 20 MHz clock. Unfortunately, the XC4013 is not packaged in an 84 pin PLCC; thus, the combined compression-decompression algorithm cannot be implemented on the standard APSx84 board.

```
module APSX84(clock, in_data, reset, out_data,
  LED, SRAM_A, SRAM_IO, SRAM__CE, SRAM__OE, . . .
 input          clock      /* xc_loc=P13  */ ;
 input   [ 7:0] in_data    /* xc_loc="P10,P9,P8,P7,
 output  [ 7:0] out_data   /* xc_loc="P20,P19,P18, .
 output         LED        /* xc_loc=P35  */ ;
 output  [14:0] SRAM_A     /* xc_loc="P81,P78,P72, .
 inout   [ 7:0] SRAM_IO     /* xc_loc="P60,P58,P82,
 output         SRAM__CE    /* xc_loc=P51   */ ;
 output         SRAM__OE    /* xc_loc=P50   */ ;
 output         SRAM__WE    /* xc_loc=P77   */ ;

 wire          clock_select = in_data [0];
 wire          soft_clock   = in_data [1];
 wire          reset        = in_data [2];
 . . .
 assign        out_data [0] = ready_reset;
 assign        out_data [1] = ready_compress;
 assign        out_data [2] = ready_expand;
 assign        out_data [3] =ready_write_input_bit
 assign        out_data [4] =ready_read_output_bit
 assign        out_data [5]=result_read_output_bit
 . . .
 LZW LZW ( .clock(clock), .reset(reset),
.run_compress(.run)
   . . .
```

Figure 18: Structural Verilog Hierarchy Shell for APSx84 FPGA Hardware Implementation

This design synthesizes easily in a Lucent Technologies OR2C15A FPGA and exhibits similar 20 MHz performances. As with all FPGAs, the best performance is obtained using timing-driven place and route with time specifications generated by the synthesis and PPR system. Using the Exemplar Leonardo synthesis software, this design requires approximately 6,500 LSI Logic 300K ASIC gates and is estimated to run at nearly 80 MHz using the default wiring delay models.

- **3077x16 external SRAM Memory**
- **Synthesized into Xilinx XC4000 FPGA using Synopsys, Exemplar, & Synplicity XC4013e 390/576 = 72% CLB utilization max clock 20 MHz, 290 flip-flops**
- **Synthesized into Lucent FPGA 2C15 = 50% PFU utilization estimated max clock 20 MHz**
- **Synthesized into LSI 300K ASIC ~6,500 gates estimated max clock 80 MHz**

Figure 19: Hardware Implementation Results

10. LZW IMPLEMENTATION IN FPGA ON X84 SYSTEM FOR PC HARDWARE EXECUTION

Dynamically re-configurable FPGA hardware boards, developed by Associated Professional Systems (APS) and modified by ASIC DESIGN & MARKETING, have been used to demonstrate hardware execution of C algorithms. The modified APSx84 boards contain a Xilinx XC4010e or Lucent OR2C15A FPGA, 82C55 Interface IC, 32K of SRAM memory, and a decode PAL to interface to the PC via the ISA bus.

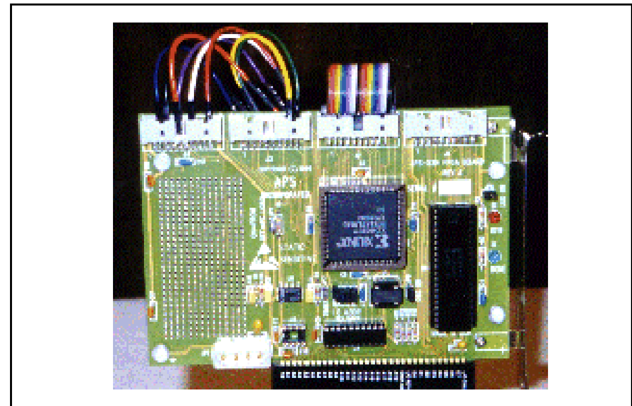

Figure 20: FPGA Hardware Execution Board XC4010e, 82C55, & SRAM Interfaced to PC via ISA bus

Compiling only the compression portion of the LZW algorithm reduces complexity and allows the design to fit into the XC4010e-3 on the APSx84 board, as shown in Figure 20. The SRAM capacity is reduced to 1796x16 for this “test-bench”. Again, the various synthesis systems create an XNF netlist which utilizes approximately the same number of CLBs (261=67%) and operates at a 24MHz maximum clock frequency determined by the Xilinx XACT Static Path Analyzer.

11. PRIME NUMBER DESIGN EXAMPLE

In another real example, a computationally intensive prime number algorithm (shown in Figure 21), containing multiple nested *if* statements, was compiled into RTL Verilog. No state machine or external SRAM memory is required. *clock* and *reset* input signals initiate the creation of a series of prime number outputs (16 bits).

This RTL Verilog executes approximately 25 times faster (in clock cycles) compared to Pentium software execution. Again, similar hardware logic resources were obtained by synthesizing the RTL Verilog into hardware using Synopsys FPGA Express, Exemplar Leonardo, and Synplicity Synplify. This design utilizes 37% of the XC4010-3 (149 CLBs, 166 flops) and can operate at a 25 MHz maximum clock frequency.

```
#include <stdio.h>
unsigned int p2 = 1, p3, p5, p7, p11, p13, p17,..
void main ()
{
 for (;;)
 {if ( p2  && ! p3  && ! p5 && ! p7  && ! p11 &&
     && ! p17 && ! p19 && ! p23 && ! p29)
   { printf("%d\n", p2);
     fflush (stdout); }
   if (p7 && p13) { p7--, p13--, p17++; }
   else if (p5 && p17) { p5--, p17--, p2++, p3++,
   else if (p3 && p17) { p3--, p17--, p19++; }
   else if (p2 && p19) { p2--, p19--, p23++; }
   else if (p3 && p11) { p3--, p11--, p29++; }
   else if (p29)       { p29--, p7++, p11++; }
   else if (p23)       { p23--, p5++, p19++; }
   else if (p19)       { p19--, p7++, p11++; }
   else if (p17)       { p17 = 0; }
   else if (p13)       { p13--, p11++; }
   else if (p11)       { p11--, p13++; }
   else if (p2 && p7)  { p2--, p7--, p3++, p5++;}
   else if (p2)        { p2--, p3++, p5++; }
   else { p5++, p11++; }
 }
}
```

Figure 21: Computationally Intensive Prime Number C Algorithm

```
module Prime(clock, reset, printf_1_1_line_29_p2,
 input         clock, reset, run_main;
 output [15:0] printf_1_1_line_29_p2;
 reg    [15:0] printf_1_1_line_29_p2, p2, p3,...

always @(posedge clock)
begin
 if (run_main)
 begin
  if(((((((((p2 && (! p3)) && (! p5)) &&.(! p7)) && (
  begin
    printf_1_1_line_29_p2 = p2;
    // User Verilog code
    $write ("%d\n", printf_1_1_line_29_p2);
  end
  if ((p7 && p13))
  begin   p7 = (p7 - 16'd1);
    p13 = (p13 - 16'd1);
    p17 = (p17 + 16'd1);
  end
  else  begin
     if ((p5 && p17))...  else
       if ((p3 && p17)) ...  else
         if ((p2 && p19)) ...  else
           if ((p3 && p11)) ...  else
             if (p29)           ...  else
                 . . .
  end
  if (reset)   . . .    end
endmodule
```

Figure 22: RTL Verilog for Prime Number Example

Thus, software execution on a 180 MHz Pentium Pro (25x cycles/180MHz = 0.13) is nearly 3 times faster than FPGA hardware execution (1x cycles/25MHz = 0.04) on a XC4010-3 FPGA, primarily due to the large cache memory in the Pentium. If the hardware is implemented in a LSI Logic 300K ASIC, nearly an order of magnitude faster execution speed is obtained. This shows the better relative performances possible in designs with fewer sequential evaluations.

- **No External SRAM or State Machine**
- **25x fewer clock cycles compared to Pentium software execution**
- **Synthesized & PPR in XC4010-3**
  37 % (149=CLB, 166=FF) utilization
  25 MHz max clock frequency
- **Synthesized in LSI 300K ASIC**
  3,466 gates
  ~ 120 MHz max clock frequency

Figure 23: Prime Number Hardware Summary

12. ABC SORT ALGORITHM

The following example illustrates the application of this design methodology and the results achieved by a software system designer, Lynn Yarbrough, at Mathematical and Computational Sciences.

ABCsort is a patented software algorithm for internal sorting using a table-driven variant of Radix sort. This algorithm has been implemented in hardware for increased execution speed with the Reconfigurable Programming Unit (RPU) results shown in Figure 24.

Software version
2.5x speed of Quicksort (30 char ASCII keys)
10x speed of QS (sorting multiple keys in parrallel)

Hardware version
~5000 gate ASIC   ~66% CLBs in XC4013
+ 512x32 bit SRAM
Performance estimated from Verilog simulation

FPGA XC4013-3	0.36ms (@25MHz)
Pentium II	0.20ms (@180MHz)
ASIC LSI 300K	~0.07ms (est @120 MHz)

Figure 24: ABCsort Hardware Summary

This Verilog simulation indicates that a 25 MHz FPGA runs 1.7x slower than a Pentium II; however, a 120 MHz ASIC runs 3x faster. Better hardware performances can be obtained by balancing the delays for each state machine cycle.

The maximum propagation delays within each clock cycle are determined by invoking Static Path Timing Analysis on the synthesized netlist for a specific hardware technology. The designer iterates the compilation with additional directives to insert pipeline registers and wait states to optimize the performance. The pseudo function *next_clock()* inserted in the C code automatically steps the clock.

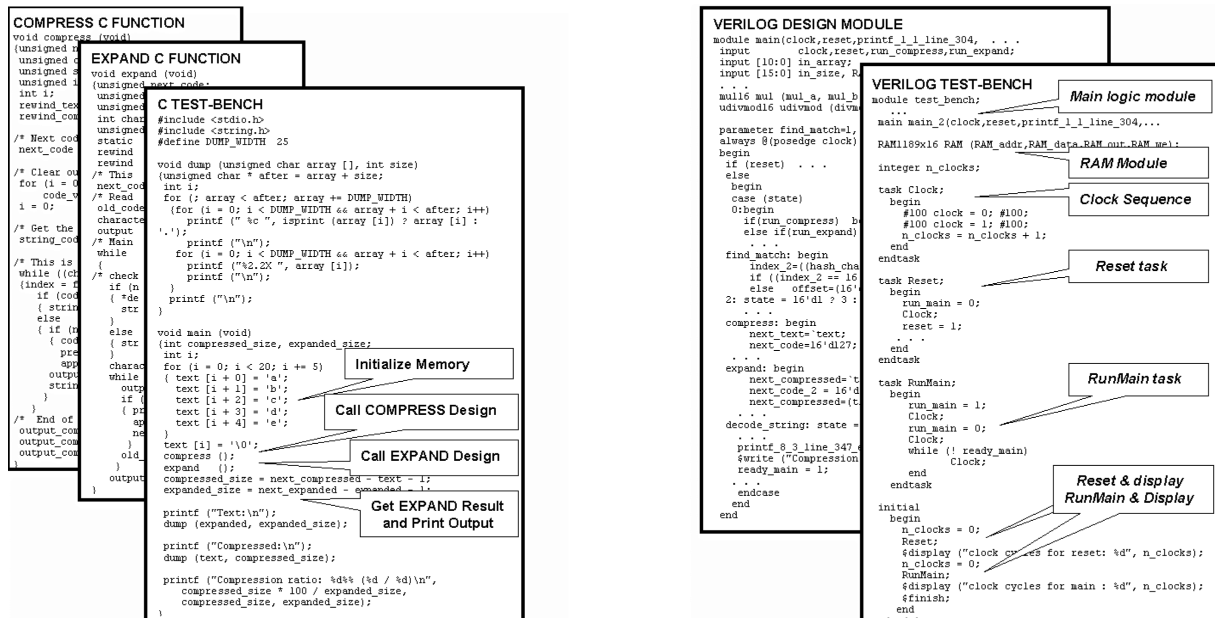

Figure 25 LZW C Test-Bench & Compiled Verilog

13. TEST-BENCH GENERATION

C programs with *#if* statements allow conditional compilation for creating RTL Verilog files for synthesis or for “test-bench” Verilog simulation. Figure 25 shows portions of the LZW compression-decompression C code using *#if* compilation controls and the resulting Verilog design and test-bench created by the procedures shown in Figure 1.

14. RSA Encryption Algorithm

Many real C algorithms have been compiled into Verilog and implemented in FPGAs to allow rapid hardware verification ranging from MPEG compression to encryption. This code is an ANSI C rewrite of the RC5REF.C reference implementation of the RC5-32/12/16 cryptographic algorithm rewritten by Ian Kaplan. This code has been written as a model for a Hardware Design Language implementation of this algorithm. To make hardware implementation more practical, the algorithm has been reduced to shifts and adds. The number of "rounds" has been increased from 12 to 15, to get rid of a mod "%" operation in the setup function.

Figure 26 shows the result of compiling the RSA code with different compiler directives. The user is able to select the sharing of resources and combinatorial logic vs. RAM-based Table implementation. The results of synthesizing RTL Verilog into QuickLogic FPGA netlists and LSI Logic 300k library are shown along with the internal register-to-register delay predicted by the Synplicity tool.

	QuickLogic FPGA	LSI300K ASIC
Approach #1 Multiple shifters & RAM-based table state-machine because memory access, fewest clock cycles		
Hardware Resources	1,429 cells	9,983 gates
Internal DFF - DFF delay	83 ns	15.3 ns
Approach #2 Single shared left & right shifter and combinatorial (not RAM-based) table requiring more clock cycles to execute		
Hardware Resources	1,683 cells	
Internal DFF - DFF delay	97 ns	
Approach #3 Single shared left & right shifter and RAM-based table shared memory accesses, largest number clock cycles to execute		
Hardware Resources	1,241 cells	10,610 gates
Internal DFF - DFF delay	69 ns	13.4 ns

Figure 26 RSA Encryption Implementation

15. FIR Filer Algorithm

A 18 TAP Finite Impulse Filter algorithm has been compiled with local compiler directives for unrolling loops as shown in Figure 28. Various implementation strategies utilizing a Verilog state-machine to share common functions, such as a 16b multiplier are also shown. With only one multiplier, the design requires less hardware resources, but requires 18 clock cycles to sequence through various multiply operations as verified by simulation using the generated test-bench. Additional hardware resources are required when the *for* loop is unrolled and more operations are performed in parallel, requiring fewer clock cycles to execute the various COEFF and TAP vector multiply operations. The state-machine introduces some hardware resources; thus, the approach with full unrolling (complete parallel execution) requires less hardware resources than the case with 9 shared multipliers. Thus, the user is able to guide the compiler to obtain the desired design implementation.

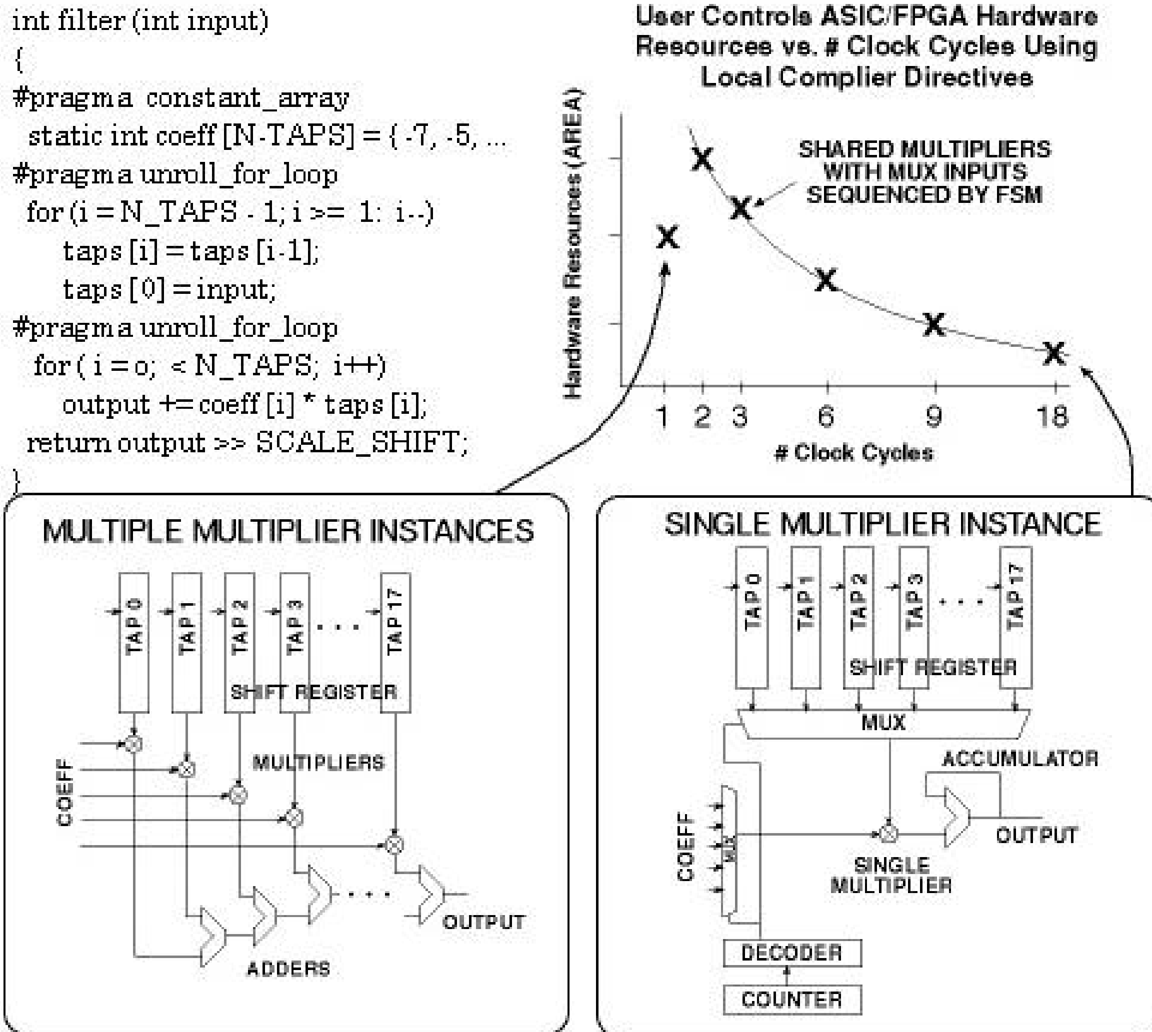

Figure 26: User Control of Local Directives for Optimum Hardware Implementation of 18 TAP Filter

Approach	Hardware Resources	F/H Maps	DFFs
Approach #1 18 unique Multipliers and adders (execute in 1 clock cycle)	3,011	F/H Maps	272 DFFs
Approach #2 9 Shared Multpliers muxed using state-machine (2 clk cycles)	4,599	F/H Maps	339 DFFs
Approach #3 6 Shared Multipliers muxed using state-machine (3 clk cycles)	3,132	F/H Maps	339 DFFs
Approach #4 3 Shared Multipliers muxed using state-machine (6 clk cycles)	1,591	F/H Maps	339 DFFs
Approach #5 2 Shared Multipliers muxed using state-machine (9 clk cycles)	1,109	F/H Maps	339 DFFs
Approach #6 1 Shared Multipliers muxed using state-machine (18 clk cycles)	541	F/H Maps	339 DFFs

Figure 27: 18 TAP (FIR) Finite Impulse Response Filter Implementation in Xilinx FPGA

14. SUMMARY

In conclusion, encryption, filter, compression, prime number, and sorting C algorithms have been implemented in hardware using a compiler, conventional logic synthesis and FPGA products. Although these examples have very sequential data flows, competitive execution speeds were obtained using hardware implementation versus software execution on high performance Pentium Pro PC systems. Larger performance gains are possible for more parallel algorithms using simple and less expensive hardware implementation. Thus, this design methodology improves the productivity of system architects as well as experienced hardware designers.

Generating Verilog test-benches automatically from C code to simulate the behavior of hand-coded RTL or compiled Verilog substantially accelerates the design verification process. This approach automatically generated in a few minutes a Verilog test-bench from a complicated ARM C behavior model which previously required a month to manually code for hardware assisted simulation and emulation.