Overview of a Compiler for Synthesizing MATLAB Programs onto FPGAs

P. Banerjee, *Fellow, IEEE*, M. Haldar, A. Nayak, V. Kim, V. Saxena, S. Parkes, D. Bagchi, S. Pal, N. Tripathi, D. Zaretsky, R. Anderson, J. R. Uribe

Abstract— This paper describes a behavioral synthesis tool called AccelFPGA which reads in high-level descriptions of DSP applications written in MATLAB, and automatically generates synthesizable RTL models and simulation testbenches in VHDL or Verilog. The RTL models can be synthesized using commercial logic synthesis tools and place and route tools onto FPGAs. The paper describes how powerful directives are used to provide high-level architectural tradeoffs for the DSP designer. Experimental results are reported on a set of eight MATLAB benchmarks that are mapped onto the Xilinx Virtex II and Altera Stratix FPGAs.

Index Terms— **High level synthesis, MATLAB, RTL, VHDL, Verilog, FPGAs.**

Manuscript received March 5, 2003; revised August 1, 2003 and September 15, 2003.

Prith Banerjee is the Founder of AccelChip, Inc., He is currently with the Electrical and Computer Engineering Department, Northwestern University, Evanston, IL-60208, USA (email: banerjee@ece.northwestern.edu).

Malay Haldar was with AccelChip, Inc. He is now working for Calypto Design Systems, 2903 Bunker Hill Lane, Suite 208, Santa Clara, CA 95054 (email: malay@calypto.com).

Anshuman Nayak was with AccelChip, Inc. He is now working for Atrenta (I) Pvt. Ltd., A 10 Ground Floor, Sector 9, Noida, 201 301, UP, India (email: nayak@atrenta.com).

Victor Kim was with AccelChip, Inc. He is now working for Calypto Design Systems, 2903 Bunker Hill Lane, Suite 208, Santa Clara, CA 95054 (email: vkim@calypto.com)

Vikram Saxena was with AccelChip, Inc. He is now working for Synopsys, Inc., 700 East Middlefield Road, Mountain View, CA 94043, USA (email: vikram@synopsys.com).

Steven Parkes was with AccelChip, Inc. He is now working for IBM Almaden Research Center, 650 Harry Road, San Jose, CA-95120, USA. (email: steven@almaden.ibm.com).

Debabrata Bagchi was with AccelChip, Inc. He is now working for Atrenta (I) Pvt. Ltd., A 10 Ground Floor, Sector 9, Noida, 201 301, UP, India (email: bagchi@atrenta.com)

Satrajit Pal was with AccelChip, Inc. He is now working for Atrenta (I) Pvt. Ltd., A 10 Ground Floor, Sector 9, Noida, 201 301, UP, India (email: satrajit@atrenta.com).

Nikhil Tripathi was with AccelChip, Inc. He is now working for Atrenta (I) Pvt. Ltd., A 10 Ground Floor, Sector 9, Noida, 201 301, UP, India (email: nikhil@atrenta.com).

David Zaretsky was with AccelChip. He is currently with the Electrical and Computer Engineering Department, Northwestern University, Evanston, IL-60208, USA (email: dcz@northwestern.edu).

Robert Anderson is with AccelChip, Inc., 1900 McCarthy Blvd., Suite 204 , Milpitas, CA 95035 USA (email: robert@accelchip.com).

Juan Ramon Uribe is with AccelChip, Inc., 1900 McCarthy Blvd., Suite 204 , Milpitas, CA 95035 USA (email: uribe@accelchip.com).

I. INTRODUCTION

THE performance requirements of today's communication systems, such as 3G and 4G wireless communication systems, MPEG4 video and Video over IP, now exceed the capabilities of general-purpose processors. With the introduction of advanced Field-Programmable Gate Array (FPGA) architectures such as the Xilinx Virtex-II [14], and the Altera Stratix [2], a new hardware alternative is available for DSP designers that combines all the benefits of general-purpose processors with the performance advantage of ASICs.

DSP design has traditionally been divided into two types of activities – systems/algorithm development and hardware/software implementation. The majority of DSP system designers and algorithm developers use the MATLAB language [9]. The first step in this flow is the conversion of the floating point MATLAB algorithm into a fixed point version using quantizers from the Filter Design and Analysis (FDA) Toolbox for MATLAB. Algorithmic tradeoffs such as the precision of filter coefficients, rounding modes, and the number of taps used in a filter are performed at the MATLAB level. Hardware design teams take the specifications created by the systems engineers and algorithm developers (in the form of a fixed point MATLAB code) and create a physical implementation of the DSP design. If the target is an FPGA, or PLD, the first task is to create a register transfer level (RTL) model in a hardware description language (HDL) such as VHDL and Verilog. The RTL HDL is synthesized by a logic synthesis tool, and placed and routed onto an FPGA using backend tools. The process of creating an RTL model and a simulation testbench takes about one to two months with the tools currently used today.

This paper described the AccelFPGA compiler which directly reads in fixed point MATLAB behavioral models and automatically outputs synthesizable RTL models and simulation testbenches in VHDL or Verilog. The resultant RTL is bit-true with the original fixed point MATLAB specification. The current manual and new automated flow is shown in Figure 1. AccelFPGA also allows users to perform quick iterations of hardware designs, allowing area and speed trade offs and architecture exploration.

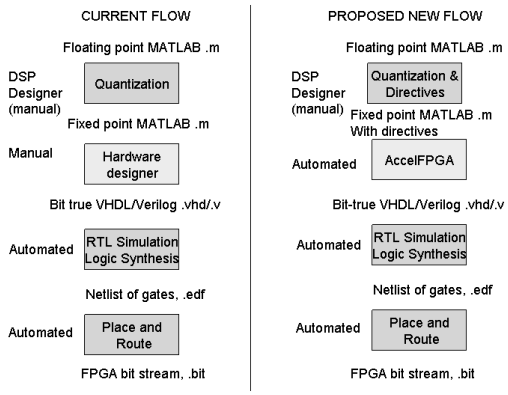


Figure 1. Automated design using AccelFPGA.

II. RELATED WORK

The problem of translating a high-level or behavioral language description into a register transfer level representation is called high-level synthesis [6]. Synopsys developed one of the first successful commercial behavioral synthesis tools in the industry, the Behavioral Compiler [12], which took behavioral VHDL or Verilog and generated RTL VHDL or Verilog. Recently, there have been several efforts to develop compilers that compile high-level languages such as C into VHDL or Verilog [7,10]. Commercial products are offered by companies such as Adelante [1], Celoxica [3], C Level Design [4] and Cynapps [5]. SystemC is a new language developed by the SystemC consortium which allows users to write hardware system descriptions using a C++ class library [11]. Synopsys has a tool called Cocentric which takes SystemC and generates RTL VHDL/Verilog. There have been some system level tools that take graphical descriptions of systems and generate HDL code. Examples include SPW from Cadence [13], System Generator from Xilinx [14], and DSP Builder from Altera [2].

While there are some related tools that convert C or C++ into VHDL and Verilog, this paper describes a compiler that takes behavioral MATLAB descriptions (the default language of DSP design) and generates RTL VHDL and Verilog for FPGA design. Some of the unique and challenging features of the MATLAB language are the support for array operations (operating on matrices instead of scalars), an interpretive environment where the types and shapes of variables are not declared at compile time but inferred at runtime, and a very powerful set of built in library functions. Examples of such functions are ""filter" and "fft". AccelFPGA supports a subset of the fixed point built-in functions of MATLAB in hardware in the form of AccelWare functions. The MATCH compiler project [8] at

Northwestern University has built a compiler that took the MATLAB applications and produced synthesizable RTL VHDL. The technology for the AccelFPGA compiler described in this paper is an outgrowth of the MATCH Compiler project.

III. AN EXAMPLE DESIGN IN MATLAB

We will illustrate the use of the AccelFPGA compiler using a 16 tap finite impulse response (FIR) filter example. Let us first look at how such a filter can be represented in MATLAB using a floating point representation.

```
%   16-tap FIR filter demonstration - Floating point
version.
% read input from file
x = load('sines.txt'); % input data
indata = x;
NUMTAPS = 16;  % number of taps
NUMSAMPS = length(x); % number of samples
% define filter coefficients
% 16-tap low-pass filter; sampling rate 8000 Hz, bandpass
cutoff 2000 Hz; bandstop start 3000 Hz
coeff = 100 * [-0.024750172265052, -
0.030362659582556, 0.037764386593039,
0.048119075484636, ...
              -0.063925788455935, -0.091690255759161,
0.155281320470888, 0.469564093514142, ...
               0.469564093514142, 0.155281320470888, -
0.091690255759161, -0.063925788455935, ...
               0.048119075484636, 0.037764386593039, -
0.030362659582556, -0.024750172265052];
% apply filter to each input sample
for n = NUMTAPS:NUMSAMPS
  % initialize sum of products
    sum = 0;
    % perform sum of products
    for k = 1:16,
      mult = indata(n-k+1) * coeff(k);
      sum  = sum + mult;
    end
    % store output
    outdata(n) = sum;
end
```

The first step in this flow is the conversion of the floating point MATLAB algorithm into a fixed point version using "quantizers" from the Filter Design and Analysis (FDA) Toolbox for MATLAB [9]. One specifies a quantizer as

qresults = quantizer('fixed','floor','wrap',[16,0]);

The quantizer is actually used in a subsequent multiplication computation as

mult = quantize(qresults , (indata(n-k+1) * coeff(k)));

which means that the multiplication operation is performed, and the output is stored in 16 bit precision, with 0 bits after the decimal point, and uses the "floor" mode for rounding, and "wrap" mode for overflow computations.

Algorithmic tradeoffs such as the precision of filter coefficients and the number of taps used in a filter are performed at the MATLAB level. We now show the fixed point version of the MATLAB algorithm for a FIR filter below.

```
% Example MATLAB code for 16 tap FIR filter with
fixed point quantization
% read input from file
x = load('sines.txt'); % input data
NUMTAPS = 16;  % number of taps
NUMSAMPS = length(x); % number of samples
% define fixed-point parameters
qpath = quantizer('fixed','floor','wrap',[8,0]); %
quantization of inputs in data path
qresults = quantizer('fixed','floor','wrap',[16,0]); %
quantization of operation results
indata = quantize(qpath,x);
% define filter coeffients
% 16-tap low-pass filter; sampling rate 8000 Hz, bandpass
cutoff 2000 Hz; bandstop start 3000 Hz
coeff = quantize(qpath, [-2.4750172265052; -
3.0362659582556; 3.7764386593039; 4.8119075484636;
...
              -6.3925788455935; -9.1690255759161;
15.5281320470888; 46.9564093514142; ...
               46.9564093514142; 15.5281320470888; -
9.1690255759161; -6.3925788455935; ...
               4.8119075484636; 3.7764386593039; -
3.0362659582556; -2.4750172265052]);
for n = NUMTAPS:NUMSAMPS
    sum = 0;
    % perform sum of products
    for k = 1:16
      mult = quantize(qresults , (indata(n-k+1) *
coeff(k)));
      sum  = quantize(qresults , (sum + mult));
    end
    outdata(n) = quantize(qresults,sum);
end
```

We now apply various AccelFPGA directives to the fixed point MATLAB. AccelFPGA compiler directives

are used to bridge the gap between the MATLAB source and the synthesis of the computational structures created by AccelFPGA. Every compiler directive is prefixed by "%!ACCEL". This makes the directives appear as comments to other environments dealing with MATLAB since all comments in MATLAB start with %.

The first step in the process of hardware design is to add a TARGET directive which tells the compiler that it needs to generate HDL code for specific FPGA device. AccelFPGA uses a Resource Description Language that describes the architecture, routing and internal execution resources of FPGAs from all the major FPGA vendors. AccelFPGA uses the RDL to create synthesizable RTL code that is uniquely optimized for each device. AccelFPGA supports various FPGA devices using the TARGET directive. By specifying the %!ACCEL TARGET XC2V250 directive, the compiler becomes aware of the characteristics of that target Virtex II architecture, namely that it can support 1536 Combinational Logic Blocks, 48 Kbits of distributed RAM, 24 embedded multipliers, 24 embedded RAM blocks.

The next step is to use hardware partitioning directives. AccelFPGA allows the user to use hardware partitioning directive to demarcate parts of the input source that are targeted for hardware synthesis and parts that are not. The BEGIN_HARDWARE and END_HARDWARE directives indicate a section of MATLAB code that is intended for hardware synthesis. The BEGIN-HARDWARE directive takes as a parameter the input data port for the hardware part. The END-HARDWARE directive takes as a parameter the output data port for the hardware part.

The next step is to use a STREAM directive. The purpose of the STREAM directive is the specification of the type of data flow that inputs and outputs of the synthesized hardware will handle. Streaming data is defined as data with a regular rate of flow through the hardware. For systems that will handle streaming data, AccelFPGA supports the automatic creation of ports with the required buffering mechanisms to sustain the regular flow of data with the use of the STREAM directive. These mechanisms include 'double-buffering' to allow concurrent processing of data and buffering of new data samples. It should also be noted that the fixed point code got modified to add some explicit buffering of input and output data for the parts of the code in the hardware component. For example the input data "indata" gets copied to "indatabuf" before the actual computation of the FIR computation.

The resultant FIR filter code with AccelFPGA directives is shown below.

```
% Example MATLAB code for 16 tap FIR filter with
basic AccelFPGA directives
%  Specif\y hardware target Xilinx Virtex2 FPGA
%!ACCEL TARGET XC2V250
% read input from file
x = load('sines.txt'); % input data
NUMTAPS = 16;  % number of taps
NUMSAMPS = length(x); % number of samples
% define fixed-point parameters
qpath = quantizer('fixed','floor','wrap',[8,0]); %
quantization of inputs in data path
qresults = quantizer('fixed','floor','wrap',[16,0]); %
quantization of operation results
indata = quantize(qpath,x);
% define filter coeffients
% 16-tap low-pass filter; sampling rate 8000 Hz, bandpass
cutoff 2000 Hz; bandstop start 3000 Hz
coeff = quantize(qpath, [-2.4750172265052; -
3.0362659582556; 3.7764386593039; 4.8119075484636;
...
            -6.3925788455935; -9.1690255759161;
15.5281320470888; 46.9564093514142; ...
             46.9564093514142; 15.5281320470888; -
9.1690255759161; -6.3925788455935; ...
             4.8119075484636; 3.7764386593039; -
3.0362659582556; -2.4750172265052]);
% apply filter to each input sample
%!ACCEL STREAM n
for n = NUMTAPS:NUMSAMPS
   %!ACCEL BEGIN_HARDWARE indata
   indatabuf = quantize(qpath, indata(n-15:n));
   % initialize sum of products
   sum = quantize(qresults,0);
   % perform sum of products
   for k = quantize(qpath,1:16),
     mult = quantize(qresults , (indatabuf(k) * coeff(k)));
     sum  = quantize(qresults , (sum + mult));
   end
   outdatabuf = quantize( qresults, sum );
   % store output
   outdata(n) = quantize(qresults,outdatabuf);
   %!ACCEL END_HARDWARE outdata
end
```

The basic MATLAB FIR filter algorithm performs the 16 tap FIR filter operation using 16 iterations using one adder and one multiplier per iteration. This particular design with these AccelFPGA directives when compiled by the AccelFPGA compiler requires 143 LUTs, 75 multiplexers, 1 multiplier, 8 ROMs, runs at 82.9 MHz, and has a latency of 23 cycles, and an initiation rate

of 19 cycles (one new data sample every 19 cycles). In terms of DSP filter performance, this design performs FIR filtering at a rate of 82.9/19*1000 = 4363 Kilo-samples per second (KSPS) as shown in Table 2. We will show in the next section how the performance of this filter can be improved.

IV. DESIGN SPACE EXPLORATION USING THE ACCELFPGA COMPILER

AccelFPGA allows the user to use compiler directives to perform design space exploration of various area-performance tradeoffs. AccelFPGA performs high-level estimates of area by counting the number of functional unites such as adders, multipliers, multiplexers and registers and reporting that to the user during synthesis. We combine this unit count with area models for functional units that are parameterized with respect to the bit widths of the devices [15]. AccelFPGA estimates performance (latency, throughput) in terms of clock cycles used in the scheduling. The latency is measured by the number of clock cycles needed to generate an output in response to an input. The initiation rate is the number of clock cycles between successive inputs. Throughput is defined as the worst-case clock frequency of the design divided by the initiation rate. The latency, initiation rate and throughput numbers are reported to the user as part of the high-level synthesis. We will now describe the various directives available for the user to perform these tradeoffs.

A. UNROLL Directive

The UNROLL directive is a mechanism to expand the source MATLAB to create more copies of loop bodies, thereby increasing performance optimizations as illustrated below.

Let us consider the for loop in the example MATLAB code for the FIR filter.

```
sum = 0;
 for k = 1:16
  mult = quantize(qresults , (indatabuf(k) * coeff(k)));
  sum  = quantize(qresults , (sum + mult));
 end
```

The MATLAB code has one addition and one multiplication operation in the data flow graph of its basic block hence the AccelFPGA compiler will generate an RTL VHDL or Verilog which will use one adder and one multiplier to schedule this computation which will take 16 cycles.

If the code were to be unrolled as shown below

```
sum = 0;
 %!ACCEL UNROLL 4
 for k = 1:16
  mult = quantize(qresults , (indatabuf(k) * coeff(k)));
  sum  = quantize(qresults , (sum + mult));
 end
```

The loop body will be replicated 4 times and the loop indices in successive iterations are incremented. In addition, scalars that carry values from one iteration to another iteration are renamed. For example, the scalar "sum" and "mult" would be renamed in successive copies. This exposes opportunities to chain operations to the compiler.

```
sum=0;
for i = 1:4:16
    mult1 = quantize(qresults , (indatabuf(k) *
coeff(k)));
  sum1  = quantize(qresults , (sum + mult1));
  mult2 = quantize(qresults , (indatabuf(k+1) *
coeff(k+1)));
  sum2  = quantize(qresults , (sum1 + mult2));
  mult3 = quantize(qresults , (indatabuf(k+2) *
coeff(k+2)));
  sum3  = quantize(qresults , (sum2 + mult3));
  mult4 = quantize(qresults , (indatabuf(k+3) *
coeff(k+3)));
  sum4  = quantize(qresults , (sum3 + mult4));
end;
```

AccelFPGA now recognizes four addition and four multiplication operations in each basic block hence it will schedule it across four cycles using four adders and four multipliers in parallel.

If the code were to be unrolled fully as shown below

```
sum = 0;
 %!ACCEL UNROLL 16
 for k = 1:16
  mult = quantize(qresults , (indatabuf(k) * coeff(k)));
  sum  = quantize(qresults , (sum + mult));
 end
```

The loop body will be replicated completely 16 times and the for loop will be eliminated.

```
sum=0;
```

```
      mult1 = quantize(qresults , (indatabuf(k) * coeff(k)));
    sum1  = quantize(qresults , (sum + mult1));
    mult2 = quantize(qresults , (indatabuf(k+1) * coeff(k+1)));
    sum2  = quantize(qresults , (sum1 + mult2));
  …
  …
    mult15 = quantize(qresults , (indatabuf(k+14) * coeff(k+14)));
    sum15  = quantize(qresults , (sum14 + mult15));
    mult16 = quantize(qresults , (indatabuf(k+15) * coeff(k+15)));
    sum16  = quantize(qresults , (sum15 + mult16));
  end;
```

For this particular choice of UNROLL 16, AccelFPGA produces a design that requires 259 LUTs, 399 multiplexers, 16 multipliers, 8 ROMs. This design has a reduced latency of 5 cycles, and initiation rate of 1 cycle, however, it operates at a frequency of 76.9 MHz owing to a large critical path involving 16 adders and 1 multiplier in one cycle. In terms of FIR filter performance, even though the clock frequency has gone down, the throughput has gone up to 76,900 Kilo-samples per second as shown in Table 2.

The UNROLL directive is therefore used by the user to generate different area-delay hardware alternatives.

B. PIPELINE Directive

Pipelining increases the throughput of a datapath by introducing registers in the datapath. This increase in throughput is particularly important when the datapath is iterated in the overall design. The PIPELINE directive is placed just before the loop, whose body is to be pipelined. For pipelining function bodies the directive is placed just above the function definition. Let us consider the for loop in the example MATLAB for the FIR filter.

```
sum = 0;
  for k = 1:16
    mult = quantize(qresults , (indatabuf(k) * coeff(k)));
    sum  = quantize(qresults , (sum + mult));
  end
```

If the code were to be pipelined as shown below

```
sum = 0;
  %!ACCEL PIPELINE
  for k = 1:16
    mult = quantize(qresults , (indatabuf(k) * coeff(k)));
    sum  = quantize(qresults , (sum + mult));
  end
```

AccelFPGA will now unroll the 16 tap for loop into a data flow graph consisting of 16 multipliers and 16 adders, and breaks off the data flow graph into 16 stages of a pipeline, with each stage having one multiplier and one adder, amnd insert registers between each stage

For our FIR filter example, AccelFPGA now produces a design which improves the frequency of operation of the design to 134.2 MHz, but suffers a large latency of 20 cycles, however, the initiation rate is now at 1 cycle per operation. This design therefore works at 134,200 Kilo-samples per second as shown in Table 2.

C. MEM_MAP Directive

The AccelFPGA compiler by default maps all variables to registers in the hardware implementation. In many cases if the variables are arrays of large size, this may results in large hardware resources in the forms of large multiplexers. The memory map directive indicates that the given array variable should be mapped to a specific memory resource in the target architecture. The MEM_MAP directive can be used to map array variables to embedded RAMs on a Xilinx Virtex II or Virtex-E device or Altera APEX or Stratix device.

Let us consider the MATLAB code for the FIR filter which illustrates the MEM_MAP directive.

```
  for n = NUMTAPS:NUMSAMPS
  %!ACCEL MEM_MAP indatabuf TO ram_s9_s9(0) AT 0
  %!ACCEL BEGIN_HARDWARE indata
    indatabuf = quantize(qpath, indata(n-15:n));
    sum = 0;
    for k = 1,16
      mult = quantize(qresults , (indatabuf(k) * coeff(k)));
      sum  = quantize(qresults , (sum + mult));
    end
    outdatabuf = quantize( qresults, sum );
    outdata(n) = quantize(qresults,outdatabuf);
    %!ACCEL END_HARDWARE outdata
  end
```

In this example, the user wants to map the array **indatabuf** to the embedded memory on an Xilinx VirtexII device named "ram_s9_s9" with instance "0" starting at memory location "0" using the following directive:

%!ACCEL MEM_MAP indatabuf TO ram_s9_s9(0) AT 0

For our running design of the 16 tap FIR filter, AccelFPGA produces a design that requires only 126 LUTs, 47 multiplexers, 1 multiplier, 8 ROMs and 1 BlockRAM. However, the latency goes up to 73 cycles, and the throughput goes to 67 cycles between consecutive data streams. Hence even though the number of multiplexers has gone down from 75 to 47, and the number of LUTS has gone down from 143 to 126, the FIR filter throughput is 115.1/67 = 1717 Kilosamples per second. This is clearly not a good choice of the directive. However, as we will show later on, this will be a good choice if the FIR were a 64 tap filter.

D. TILE Directive

Tiling enables the user to specify an array of computation that happens concurrently on data that is stored in a distributed manner across multiple memories to provide higher memory bandwidth. Independent memory and resources can be specified for each tile concisely. The compiler generates hardware where each of the "tile" is scheduled concurrently. This allows the user to exploit large amount of data parallelism typically present in DSP applications. Let us consider the case of a 64 Tap FIR, as shown below.

```
sum = 0;
for k = 1:64
   mult = quantize(qresults , (indatabuf(k) * coeff(k)));
   sum  = quantize(qresults , (sum + mult ));
end
```

No memory maps are provided for the indatabuf() and coeff() arrays, implying they are mapped to registers. However, to access a register array of 64 elements, one needs a multiplexer (mux) with 64 inputs. Each input to the mux corresponds to a register of the array, and the mux is controlled by the array index. Hence, for generating this hardware, the arrays in the above example need to be mapped to memories. For the sake of illustration, let us assume that there is a RAM of 64 entries with a single read port, called ram_s9_s9 , on the architecture on which we want to map the 64 Tap FIR. The modified code is shown below which maps the **indatabuf** to the RAM instance called ram_s9_s9(0) on the Xilinx Virtex II device.

```
sum = 0
```

```
%!ACCEL MEM_MAP indatabuf() TO ram_s9_s9(0) AT 0
for k = 1:64
   mult = quantize(qresults , (indatabuf(k) * coeff(k)));
   sum  = quantize(qresults , (sum + mult ));
end
```

However, this now restricts the inner loop to only one read access for indatabuf per cycle assuming that the RAM allows a single read operation per port per cycle. In order to get more reads accomplished per cycle, one needs to use multiple memory banks.

One can rewrite the above loop as follows.

```
for t = 1:16
  sum(t) = 0;
end;

for k = 1:4
  for t1 = 1:16
     mult = quantize(qresults , (indatabuf(k,t1) * coeff(k,t1)));
     sum(t1)  = quantize(qresults , (sum(t1) + mult ));
   end
end;
final_sum = 0;
for t2 = 1:16
   final_sum = quantize(qresults, final_sum + sum(t2));
end;
```

The loop running over the number of tiles t1 is referred to as the Tiling Loop and is indicated by the TILE directive. Combining this with the memory maps split over 16 RAMS, we get a tiled version of the 64 tap FIR filter shown below.

```
%!ACCEL STREAM n
for n = 64:1024
   %!ACCEL BEGIN_HARDWARE indata
   indatabuf = quantize(qpath, reshape(indata(n-63:n),4,16));
   %!ACCEL TILE t
   % initialize sum of products
   for t = 1:16
      sum(t) = 0;
   end;
   %!ACCEL PIPELINE
   for k = 1:4
      %!ACCEL TILE t1
      %!ACCEL MEM_MAP indatabuf(:,t1) TO ram_s9_s9(t1) AT 0
```

```
        for t1 = 1:16
          mult = quantize(qresults , (indatabuf(k,t1) *
coeff(k,t1)));
          sum(t1)  = quantize(qresults , (sum(t1) + mult
));
        end
      end;
      final_sum = 0;
      for t2 = 1:16
        final_sum = quantize(qresults, final_sum +
sum(t2));
      end;
      outdatabuf = quantize( qresults, final_sum );
      % store output
      outdata(n) = quantize(qresults,outdatabuf);
      %!ACCEL END_HARDWARE outdata
   end
```

The impact of tiling can be shown as follows. The 64 tap FIR filter with pipelining and no memory mapping would require 2066 LUTs and 971 multiplexers, and operate at a frequency of 83.7 with a throughput of 2041.5 KSPS. However, if one were to tile the 64 tap filter and memory map the indatabuf buffer across 16 memories, the design requires 1654 LUTS, 330 muxes, operates at 79.7 MHz, and has a throughput of 1449 KSPS as shown in Table 2.

V. COMPILER OVERVIEW

AccelFPGA is built as a set of compiler passes operating on different intermediate representations as shown in Figure 2. The front-end of AccelFPGA reads in MATLAB m-files including various directives, performs various syntax analysis and error reporting, and translates the MATLAB program into a high level intermediate representation (HIR). The HIR representation is similar to Abstract Syntax Tree (AST) representation of a traditional compiler. It stores the information regarding a given source MATLAB program in a tree form with information about expressions, assignment statements, conditional statements, loop statements, functions and procedures. A set of compiler passes perform various transformations on the HIR representation. These passes are described briefly.

A. Shape Inferencing

The shape inferencing pass infers the shapes of most of the scalar and array variables in the program for parts of the MATLAB models that will be mapped to hardware (separated by BEGIN-HARDWARE and END-HARDWARE directives). The algorithm is based on a lattice theoretic shape algebra. Details of our automatic type/shape inferencing algorithm are outside the scope of this paper, and the readers are referred to [17].

B. Scalarization and Levelization

The scalarization pass takes a vectorized MATLAB statement and converts it into scalar form using enclosing FOR loops. The levelization pass takes a MATLAB assignment statement consisting of complex expressions on the right hand side and converts it into a set of statements each of which is in a single operator form. This pass operates on both scalar and array operations.

C. Auto-quantization

The auto-quantization phase takes in floating-point MATLAB computations for parts of the MATLAB models that will be mapped into hardware and translates them to fixed-point MATLAB design. The algorithms for auto-quantization are described in [16]. This algorithm takes in a scalarized and levelized MATLAB program with some of the quantizations of input variables specified, and computes the initial value ranges for each variable in the program. The next step takes the partial value ranges of the variables obtained at the previous step and propagates the value ranges in the forward direction using use-def analysis. The algorithm handles simple blocks of assignment statements, conditionals such as IF-THEN-ELSE, and FOR loops. The next step propagates the value ranges of variables in the backward direction. If the user has only specified the quantizers of the output variables, the back propagation pass will propagate the results to the right-hand side of an assignment statement. The algorithm is similar to Forward propagation. The final step assigns values of quantizers based on the value ranges of various variables.

D. Unrolling

The Unrolling pass interprets the UNROLL directives and generates an unrolled form of the HIR representation of a FOR loop or a vector statement of an original MATLAB program based on the UNROLL directive. It takes the data flow graph representation of the statement in a loop body and replicates it by an amount equal to the UNROLL factor as described in Section 4.1.

E. Streaming

The Stream Compiler pass handles the STREAM directive which is used by the user to represent streaming data as described in Section 3. The stream compiler takes care of creating corresponding input and output ports, creating buffers between functional blocks and manages the streaming of sample-based and frame-based data.

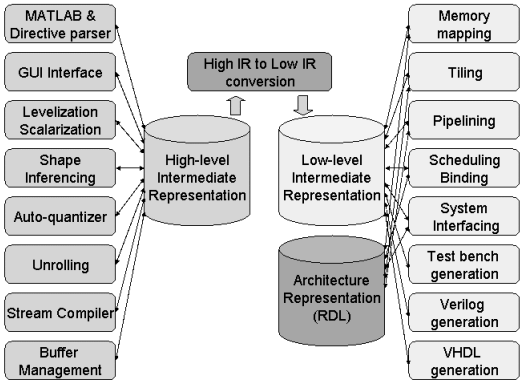


Figure 2. Overview of various compiler passes.

All these passes perform operation on the High-level intermediate representation.

F. Scheduling and Binding

The Scheduling and Binding pass performs behavioral synthesis on the HIR representation and generates a Low-level Intermediate Representation (LIR). The LIR representation has the notion of concurrency of processes, variables and signals, states and state transitions, similar to constructs in VHDL and Verilog. From the data flow graph of each basic block in the program, it schedules the computations onto various resources (adders, multipliers, etc). The type and quantity of each of these architectural resources are described using the Resource Description Language (RDL). Several well-known scheduling algorithms [6] such as As Soon As Possible (ASAP) and As Late As Possible (ALAP), Resource Constrained ASAP, and Resource Constrained ALAP scheduling based on List Scheduling, have been developed as part of this framework. Our high-level synthesis algorithms handle multi-cycle operators during scheduling, as well as multi-cycle memory read and write accesses. An example of a Resource Constrained ALAP scheduling on a CDFG is shown in Figure 3. In the figure, the CDFG graph shown on the left has 10 operation nodes. Using the RCALAP algorithm, these operations are scheduled onto limited resources of one adder and one multiplier using 6 scheduling steps as shown on the right. Once this scheduling is performed, RTL VHDL or Verilog can be generated using 6 RTL states in a state machine, and the corresponding RTL operations per state.

G. Memory Mapping and Tiling

The Memory Mapping pass interprets the MEM_MAP directive described earlier and maps data into memory structures on the target FPGA such as the BlockRAMs on

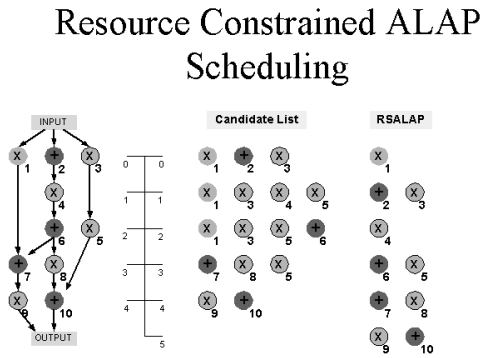


Figure 3. Example of Resource Constrained ALAP Scheduling on CDFG framework with constraints of one adder and one multiplier.

a Xilinx Virtex II FPGA as described in Section 4.3. An example of memory mapping is shown in Figure 4 for a system being synthesized for two IP blocks on a streaming data system. Two functions *compute1()* and *compute2()* are synthesized by the compiler into datapaths. The buffers from which the functions perform omputations are mapped onto the embedded RAMs with two ports (one for reading, one for writing) in order to allow parallel operations of the two IP blocks.

The tiling pass interprets the TILE directive and tiles an unrolled computation among multiple memories instantiated by the Memory Mapping pass and thereby exploits parallel memory accesses across multiple memories as described in Section 4.4.

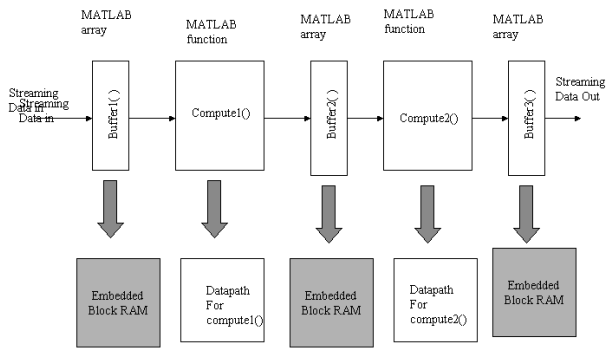


Figure 4. Example of memory mapping.

H. Pipelining

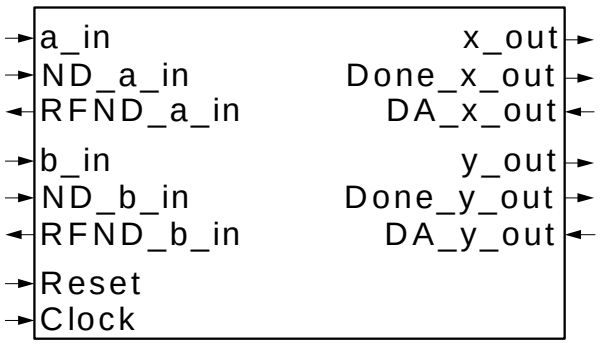
The Pipelining pass interprets the PIPELINE directive as described in Section 4.2 and performs block level pipelining of any computation block (for loops and functions) that is associated with the PIPELINE directive.

I. System Interfacing

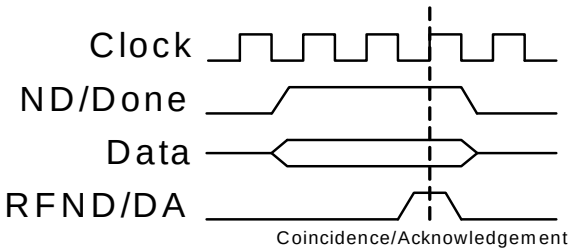
The System Interfacing pass generates a unified way to interconnect function blocks together using an asynchronous protocol which uses handshaking between consecutive blocks. It also generates the controller state machines to coordinate these asynchronous data transfers.

Figure 5(a) shows the interface for a hypothetical compiled device with two inputs, a and b, and two outputs, x and y. This device would be one of the IP blocks shown in Figure 4. The signals on the device fall into two groups, global device signals (Clock and Reset), and I/O specific signals. Each signal shown as a pin in the figure is represented by a port in the HDL. All the signals related to a MATLAB variable are termed *a channel*. The example has four channels, a, b, x, and y. The ports related to channel a are a_in, ND_a_in, and RFND_a_in.

The ND and Done signals indicate that the signals attached to the data port have been driven to stable, valid values and may be processed by the compiled device (ND) or by the embedding design (Done). ND, Done, and the data ports are all synchronous signals and as such are only defined at the rising edge of clock. Once ND/Done has been set high, it must be held high (and the data signals stable) until acknowledged (via RFND/DA, described later): ND/Done must/will be set low in the clock cycle following acknowledgement unless the design/device can support single-cycle I/O, that is, back-to-back I/O cycles. All channel ports are synchronous and are sampled on the rising edge of Clock. The Reset signal is asynchronous. When used as a simple acknowledgement, the device or design monitors the state of ND/Done, processing the data and setting RFND/DA in response: Figure 5(b) illustrates the timing diagram of the interfaces with the definition of acknowledgement: the coincidence of ND and RFND (or Done and DA) at a rising edge of clock. Following coincidence, a new I/O cycle begins.



(a) Interface signals



(b) Timing diagram

Figure 5 Example interface of hardware blocks synthesized by compiler.

J. VHDL and Verilog Generation

The VHDL generation pass converts the LIR representation into a synthesizable Register Transfer Level VHDL which can be synthesized by back-end logic synthesis tools. Similarly, the Verilog generation pass converts the LIR representation into a synthesizable Register Transfer Level Verilog.

K. Testbench Generation

The Testbench generation pass generates testbenches in VHDL and Verilog corresponding to the corresponding inputs and outputs at the MATLAB level. Given a

floating point MATLAB simulation model, AccelFPGA or the user creates a fixed point MATLAB model. This fixed point MATLAB model is executed again in the MATLAB environment, and two files are created: one file for the input vectors in fixed point, and another file for output vectors in fixed point. The same input vectors are read subsequently by the automatically generated VHDL or Verilog testbench and applied to the RTL simulator.

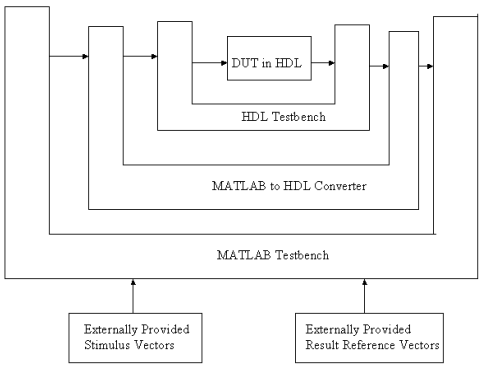


Figure 6. An overview of the automated testbench flow.

When the design under test (DUT) is compiled from MATLAB code into an hardware description language such as VHDL or Verilog, the testbench will be organized as shown in Figure 6. The features of the testbench are:

Design Under Test (DUT): Create a design behavior at the VHDL/Verilog level

MATLAB testbench: This comprises any required computations of inputs and capture of outputs of the DUT. Stimuli could come from any number of sources, most commonly being generated within the assembly/binary environment using functions/blocks and from files. Similarly, outputs could be analyzed via MATLAB functions or stored for later processing

VHDL/Verilog testbench: This procedure will instantiate the DUT at the HDL level and provides the environment needed for simulation by an HDL simulator. Primary function is stimulus application and result recording.

MATLAB to HDL converter: This procedure will capture stimuli independent of how it is generated and store it in a format that can be processed by the HDL testbench. Similarly, it provides for the reading of result vectors generated by MATLAB simulation and a comparison analysis against the HDL simulation results.

VI. EXPERIMENTAL RESULTS ON BENCHMARKS

We now report some experimental results on various benchmark MATLAB programs using the AccelFPGA compiler.

- A 16 tap Finite Impulse Response Filter
- A 64 tap memory mapped tiled FIR filter
- A Decimation in Time FIR filter
- A 64 point Fast Fourier Transform
- An Infinite Impulse Response Filter of type DF1
- An Interpolation FIR filter
- A Block Matching Algorithm
- A Digital Subscriber Line (DSL) algorithm

Table 1 shows some benchmark characteristics of the MATLAB programs. It can be seen that the MATLAB programs vary in size from 20 lines to 175 lines. We also show the number of directives used in the 9 benchmark programs. The corresponding synthesizable RTL Verilog versions of the designs are quite large, varying in size from 883 lines to 4188 lines. We also include the compile times of AccelFPGA version 1.5 for each of the benchmarks. All execution times were measured on a Dell Latitude Model C610 laptop with a 1.2GHz Pentium III CPU, 512 MB RAM, and 80 GB hard drive running Windows 2000. It can be seen that the execution times vary from 2.5 seconds to 39 seconds. We also include the compile times of the backend logic synthesis tool, namely, Synplify Pro 7.1 from Synplicity where the times vary from 2.1 seconds to 872.4 second.

Table 2 shows the results of using the AccelFPGA compiler to perform architectural tradeoffs for 8 benchmark examples for the Xilinx Virtex2 device. Results are given in terms of resources used, and performance obtained as estimated by the Synplify Pro 7.1 tool executed on the RTL Verilog that was output by AccelFPGA. The resource results are reported in terms of LUTS, Multiplexers, embedded multipliers, ROMS and BlockRAMS used. The performance was measured in terms clock frequency of the design as estimated by the internal clock frequency inferred by the Synplify Pro 7.1 tool, and the latency and initiation rate of the design in terms of clock cycles by using the ModelSim 5.5e RTL simulator. We also show the throughput of the design in Kilo-samples per second which is the frequency of the design divided by the initiation rate. For each benchmark, we show the base case on the first row (which is a design with only the information directives like TARGET, BEGIN_HARDWARE, END_HARDWARE, SHAPE and STREAM) included. We next show other designs using various performance directives such as UNROLL, PIPELINE, MEM_MAP and TILE. It can be seen that it is possible to obtain designs with widely varying resource requirements, latencies and throughputs.

Table 3 shows similar architectural tradeoffs for 8

MATLAB benchmark examples on an Altera Stratix EP1S10 device. Resources are measured in LUTS, ATOMS, MACs, and DSP Blocks, and performance is again measured in clock frequency, latency and throughput. It is therefore possible to compare the designs obtained on a Xilinx VirtexII device with an Altera Stratix device for the same choice of other performance directives such as UNROLL, PIPELINE, MEM_MAP and TILE. This is the power of the AccelFPGA tool.

We finally show a comparison of the AccelFPGA compiler with the Xilinx System Generator and the Altera DSP Builder tools. Table 4 shows a design of a 16 tap FIR filter designed using AccelFPGA and the Xilinx System Generator on a Xilinx VirtexII XC2V500 device. Table 5 shows the design of a 16 tap FIR filter using AccelFPGA and the Altera DSP Builder on an APEX EP20K device. It should be noted that this comparison is not really fair since the designs in the System Generator and DSP Builder are manually optimized libraries, while the AccelFPGA results are the output of a behavioral synthesis tool working on a MATLAB version of a filter. We attribute the differences between the compiler generated designs and the manual designs to some clever forms of implementing multipliers using constant coefficients based on efficient look-up table techniques. However it is interesting to note that the results are comparable.

VII. CONCLUSIONS

This paper described a behavioral synthesis tool called AccelFPGA which reads in high-level descriptions of DSP applications written in MATLAB, and automatically generates synthesizable RTL models and simulation testbenches in VHDL or Verilog. The RTL models can be synthesized using commercial logic synthesis tools and placement and routing tools onto FPGAs. By linking the two design domains of DSP and FPGA hardware design, AccelFPGA provides DSP design teams a significant reduction in design labor and time, elimination of misinterpretations and costly design rework, automatic verification of the hardware implementation, and the ability of systems engineers and algorithm developers to perform architectural exploration in the early phases of their development cycle. The paper described how powerful directives are used to provide high-level architectural tradeoffs for the DSP designer. Experimental results were reported on a set of 8 MATLAB benchmarks that are mapped onto the Xilinx Virtex II and Altera Stratix FPGAs.

Table 1. MATLAB Benchmark Characteristics.

Benchmark	fir16tap	fir64tap	fft64	dec_fir	lms	iirdf1	int_fir	bma	dsl
MATLAB Lines	20	40	98	38	39	33	38	63	175
Directives Used	6	8	9	6	6	6	7	10	9
Verilog Lines	957	1312	4188	1333	2219	883	1084	2758	5654
AccelFPGA Time (sec)	4.0	39.0	10.2	8.9	20.8	2.7	2.5	12.3	38.8
Synplify Time (sec)	3.6	248.7	698.8	32.6	872.4	2.1	9.5	11.9	382.1

Table 2. Results of AccelFPGA on the Xilinx Virtex II XCV250 device.

fir16tap	LUTS	MUX	Mult	ROMS	RAMS	Freq (MHz)	Latency (cycles)	Initiation rate (cycles)	Throughput (KSPS)
Base	143	75	1	8	0	82.9	23	19	4363.2
UNROLL 16	259	399	16	8	0	76.9	5	1	76900.0
PIPELINE	373	326	8	8	0	134	20	1	134200.0
MEMMAP	126	47	1	8	2	115	73	67	1717.9
PIPE+MEM	1256	565	0	0	1	131	94	54	2429.6
fir64tap									
Base	894	490	1	8	0	50.1	104	100	501.0
UNROLL 16	3172	740	16	8	0	58.7	44	40	1467.5
TILE+MEM+PIPE	1654	330	16	8	16	79.7	59	55	1449.1
dec_fir									
Base	516	197	1	6	0	66.6	74	71	938.0
UNROLL 64	1356	1209	0	0	0	61.2	8	5	12240.0
MEM+UNROLL64	3303	1963	0	0	1	96.9	207	193	502.1
iirdf1									
Base	119	47	2	0	0	107	11	7	15300.0
UNROLL 2	41	21	0	0	0	134	5	1	134200.0
int_fir									
Base	254	49	1	6	0	75.3	79	75	1004.0
UNROLL 16	446	231	16	12	0	56.8	11	7	8114.3
fft64									
Base	9882	3393	4	16	0	30.2	340	64	471.9
MEMMAP	4212	1473	4	16	2	66.8	5722	4	16700.0
dsl									
Base	7145	3055	5	16	0	38.8	3114	2883	13.5
UNROLL 16	19701	5953	20	24	0	29.4	394	227	129.5
bma									
Base	9349	3735	0	0	0	40.8	42297	42285	1.0
MEMMAP	929	512	0	0	3	72.3	230072	228342	0.3

Table 3. Results of AccelFPGA on the Altera EP1S10 Stratix device.

fir16tap	LUTS	ATOMS	MAC	DSP	ROMS	RAMS	Freq (MHz)	Latency (cycles)	Initiation rate (cycles)	System Throughput (KSPS)
Base	162	367	1	1	1	0	87.2	23	19	4589.5
UNROLL 16	302	475	8	1	1	0	82.9	5	1	82900.0
PIPELINE	287	547	4	1	1	0	129.8	20	1	129800.0
MEMMAP	169	225	1	1	1	1	95.4	73	67	1423.9
PIPE+MEM	1031	3583	8	1	1	1	123.4	94	54	2285.2

fir64tap										
Base	1244	1816	1	1	0	0	60.3	104	100	603.0
UNROLL 16	2307	2878	16	2	0	0	78.1	44	40	1952.5
TILE+MEM+PIPE	2125	2702	16	2	0	16	78.1	59	55	1420.0
dec_fir										
Base	570	1166	1	1	0	0	78.9	74	71	1111.3
UNROLL64	1090	1662	12	2	0	0	67.1	8	5	13420.0
MEM+UNROLL64	2536	2837	5	1	0	1	99.8	207	193	517.1
iirdf1										
Base	103	170	3	1	0	0	103	11	7	14771.4
UNROLL 2	21	63	1	1	0	0	130	5	1	129800.0
int_fir										
Base	311	578	1	1	0	0	67.6	79	75	901.3
UNROLL 16	840	1106	16	2	0	0	47.7	11	7	6814.3
fft64										
Base	10704	16730	4	1	0	0	46.1	340	64	720.3
MEMMAP	4439	8361	4	1	0	2	84.3	5722	4	21075.0
dsl										
Base	8514	19905	5	2	0	0	50.3	3114	2883	17.4
UNROLL 16	22487	33875	20	5	0	0	31.4	394	227	138.3
bma										
Base	9015	26362	1	1	0	0	47.9	42297	42285	1.1
MEMMAP	905	1037	0	0	0	3	57.4	230072	228342	0.3

Table 4. Comparison of AccelFPGA with System Generator for 16 tap FIR filter (XC2V500)

	Area (slices)	Frequency (MHz)	Throughput (MSPS)
AccelFPGA	386	163	163
Xilinx System Generator	587	205	205

Table 5. Comparison of AccelFPGA with DSP Builder for 16 tap FIR filter (EP20K)

	Area (Logic Cells)	Frequency (MHz)	Throughput (MSPS)
AccelFPGA	436	118	118
Altera DSP Builder	870	123	123

REFERENCES

[1] Adelante Technologies, A|RT Builder, www.adelantetechnologies.com
[2] Altera, Stratix Datasheet, www.altera.com
[3] Celoxica Corp, Handle C Design Language, www.celoxica.com
[4] System Compiler: Compiling ANSI C/C++ to Synthesis-ready HDL. Whitepaper. C Level Design Incorporated. www.cleveldesign.com
[5] CynApps Suite. Cynthesis Applications for Higher Level Design. www.cynapps.com
[6] G. DeMicheli, Synthesis and Optimization of Digital Circuits, McGraw Hill, 1994
[7] Esterel-C Language (ECL). Cadence website. www.cadence.com
[8] M. Haldar, A. Nayak, A. Choudhary, and P. Banerjee, "A System for Synthesizing Optimized FPGA Hardware from MATLAB," Proc. International Conference on Computer Aided Design, San Jose, CA, November 2001, See also www.ece.northwestern.edu/cpdc/Match/Match.html.
[9] Mathworks Corp, MATLAB Technical Computing Environment, www.mathworks.com
[10] De Micheli, G. Ku D. Mailhot, F. Truong T. The Olympus Synthesis System for Digital Design. IEEE Design & Test of Computers 1990.
[11] Overview of the Open SystemC Initiative. SystemC website. www.systemc.org
[12] Synopsys Corp, Behavioral Compiler Datasheet, www.synopsys.com
[13] Signal Processing Workbench (SPW) Datasheet, www.cadence.com.
[14] Xilinx, Virtex II Datasheet, www.xilinx.com
[15] A. Nayak, M. Haldar, A. Choudhary, P. Banerjee, "Precision And Error Analysis Of MATLAB Applications During Automated Hardware Synthesis for FPGAs," Proc. Design Automation and Test in Europe (DATE 2001), Mar. 2001, Berlin, Germany.
[16] P. Banerjee, D. Bagchi, M. Haldar, A. Nayak, V. Kim, R. Uribe, "Automatic Conversion of Floating Point MATLAB Programs into Fixed Point FPGA Based Hardware Design," Proc. FPGA based Custom Computing Machines (FCCM) (poster paper) , Apr. 2003, Monterey, CA.
[17] P. G. Joisha, and P. Banerjee, "The MAGICA Type Inference Engine for MATLAB," Proc. International Conference on Compiler Construction (CC 03), Warsaw, Poland, Apr. 2003.

Prithviraj Banerjee (F'94) is currently the Walter P. Murphy Professor and Chairman of the Department of Electrical and Computer Engineering at Northwestern University in Evanston, Illinois. Prior to that, he was the Director of the Computational Science and Engineering program, and Professor of Electrical and Computer Engineering at the University of Illinois at Urbana-Champaign. He is also the Founder and Chief Scientist of AccelChip which is developing system level electronic design tools to accelerate chip design. He founded the company in July 2000, and served as its President and CEO until June 2002.

Prith Banerjee's research interests are in VLSI computer aided design, and compilers, and is the author of about 300 research papers in these areas, and is the author of a book entitled "Parallel Algorithms for VLSI CAD". He has supervised 30 Ph.D. and 36 M.S. student theses thus far. Dr. Banerjee has served as the Program Chair, General Chair, and Program Committee of more than 50 conferences in the past 15 years and has served as Associate Editor of four journals.

Dr. Banerjee has received numerous awards and honors during his career. He received the IEEE Taylor L. Booth Education Award from the IEEE Computer Society in 2001. He became a Fellow of the ACM in 2000. He was the recipient of the 1996 Frederick Emmons Terman Award of ASEE's Electrical Engineering Division. He was elected to the Fellow grade of IEEE in 1995. He received the University Scholar award from the University of Illinois for in 1993, the Senior Xerox Research Award in 1992, the National Science Foundation's Presidential Young Investigators' Award in 1987, the IBM Young Faculty Development Award in 1986, and the President of India Gold Medal from the Indian Institute of Technology, Kharagpur, in 1981

Dr. Banerjee has been on the Technical Advisory Boards of many companies such as Atrenta, Calypto Design Systems, and Ambit Design Systems, and has served as consultants to many more.

Dr. Banerjee received his B.Tech. degree in Electronics and Electrical Engineering from the Indian Institute of Technology, Kharagpur, India, in August 1981, and the M.S. and Ph.D degrees in Electrical Engineering from the University of Illinois at Urbana-Champaign in December 1982 and December 1984

Malay Haldar was Co-Founder and Principal Software Engineer of AccelChip, Inc. He is currently a Senior Software Engineer at Calypto Design Systems. He received his B. Tech degree from the Indian Institute of Technology, Kharagpur, in 1998, his M.S. degree in Electrical and Computer Engineering from Northwestern University in 1999, and his Ph.D. in Electrical and Computer Engineering from Northwestern University in 2001.

Anshuman Nayak was Co-Founder and Principal Software Engineer of AccelChip, Inc. He is currently a Senior Software Engineer at Atrenta, India. He received his B. Tech degree from the Indian Institute of Technology, Kharagpur, in 1998, his M.S. degree in Electrical and Computer Engineering from Northwestern University in 1999, and his Ph.D. in Electrical and Computer Engineering from Northwestern University in 2001.

Victor Kim was Principal Software Engineer of AccelChip, Inc. He is currently a Software Engineer at Calypto Design Systems. He received his B. S. degree in Computer Engineering from Northwestern in 1997, and his M.S. degree in Electrical and Computer Engineering from Northwestern University in 1999. He is currently also working for his Ph.D. in Electrical and Computer Engineering at Northwestern University.

Vikram Saxena was Vice President of Engineering of AccelChip, Inc. He is currently a Senior Software Engineer at Synopsys, Inc.. He also worked for Synopsys during 1996 to 2000. He received his B. Tech degree from the Indian Institute of Technology, New Delhi in 1994, his M.S. degree in Electrical and Computer Engineering from University of Illinois at Urbana Champaign in 1996.

Steven Parkes was Principal Software Engineer of AccelChip, Inc. He is currently a Research Staff Member at IBM Almaden Research Center in San Jose, CA. He was the Founder, President and CEO of Sierra Vista Research during 1994 to 2001. He received his B. S. degree from the University of California, Davis in 1992, his M.S. degree in Electrical and Computer Engineering from the University of Illinois at Urbana Champaign in 1992, and his Ph.D. in Electrical and Computer Engineering from the University of Illinois at Urbana Champaign in 1994.

Debabrata Bagchi was Corporate Software Engineer of AccelChip, Inc. He is currently a Software Engineer at Atrenta, India. He received his B. Tech degree from the Indian Institute of Technology, Kharagpur, in 2000, and his M.S. degree in Electrical and Computer Engineering from Northwestern University in 2001.

Satrajit Pal was Corporate Software Engineer of AccelChip, Inc. He is currently a Software Engineer at Atrenta, India. He received his B. Tech degree from the Indian Institute of Technology, Kharagpur, in 2000, and his M.S. degree in Electrical and Computer Engineering from Northwestern University in 2001.

Nikhil Tripathi was Corporate Applications Engineer of AccelChip, Inc. He is currently a Software Engineer at Atrenta, India. He received his B. Tech degree from the Indian Institute of Technology, Kharagpur, in 2000, and his M.S. degree in Electrical and Computer Engineering from Northwestern University in 2001.

David Zaretsky was Principal Software Engineer of AccelChip, Inc. He received his B. S. degree in Computer Engineering from Northwestern in 2000, and his M.S. degree in Electrical and Computer Engineering from Northwestern University in 2001. He is currently also working for his Ph.D. in Electrical and Computer Engineering at Northwestern University.

Robert Anderson is a Senior Corporate Applications Engineer at AccelChip, Inc. He worked at Tellabs from 1997 to 2001, and at Lucent Technologies from 1986 to 1997. He received his B. S. degree in Computer Engineering from Devry Institute of Technology in 1986, and his M.S. degree in Electrical and Computer Engineering from Northwestern University in 2001.

Juan Ramon Uribe is a Senior Corporate Applications Engineer at AccelChip, Inc. He worked for Tellabs from 2000 to 2001, and Charles Industries from 1995 to 2000. He received his B. S. degree in Electrical Engineering from the University of Illinois at Chicago in 1986, and his M.S. degree in Electrical Engineering from Stanford University in 1988.