To Appear in Proc. SPIE Configurable Computing: Technology and Applications, Nov. 1998, Boston MA.

Rapid Implementation of Mathematical and DSP Algorithms in Configurable Computing Devices

Paul D. Fiore, Cory S. Myers, John M. Smith, Eric K. Pauer

Sanders, a Lockheed-Martin Company, Nashua NH

ABSTRACT

Many real-time DSP applications can benefit from the use of FPGA or adaptive computing devices because these devices provide very high throughput for a small hardware cost. This efficiency comes with a price: it is very time consuming to actually program complicated algorithms in these devices. We have developed a new method of performing this implementation process, providing for an order of magnitude reduction in design time.

Our approach consists of an interactive algorithm development tool closely coupled to a set of FPGA devices. From an algorithm script, the tool derives a hardware design, which includes the data path, host interface, custom sequencer, and address generators. The design is loaded and executed in the FPGA whenever a call for the function is encountered in the algorithm script. Pieces of the algorithm may be ported incrementally; the algorithm will always execute properly, regardless of the state of porting from software to hardware.

This approach is optimized to allow mathematicians, generally unskilled in efficient hardware algorithm design, to directly implement algorithms in FPGAs. This enables designers to quickly see the effect of algorithmic changes and approximations on hardware efficiency, reducing the number and time of design iterations. We are currently porting our approach to the publicly available Ptolemy environment, which will facilitate the transfer of this methodology to the adaptive computing community.

Keywords: FPGA, DSP, adaptive computing, configurable computing, rapid development, Ptolemy

1. INTRODUCTION

There is much effort devoted today in the development of real-time, custom computing machines. Many military and commercial electronic systems that utilize digital signal processing (DSP) for processing sensor data, enhancing signal quality, or automated decision-making, can benefit from the use of custom computing devices. This is because these devices provide very high-speed processing for a small hardware cost.[1–7]

Today, Field-Programmable Gate Arrays (FPGAs) are a popular device family[8]; in our work, we use the Xilinx XC4000 FPGA family.[9] These devices contain enough onboard resources to execute very complicated algorithms at rates that cannot be attained via software. Typically, FPGAs are loaded with the algorithm at power-up, and execute that algorithm until power-down. A recent class of devices, known as adaptive or configurable computing devices, are designed to be either partially or fully reloaded while other portions of the device continue to process data. Architectures for these devices are abundant,[10–13] although to date there is no effective way to program these devices at an algorithmic level.[14]

While these devices potentially offer a dramatic speed improvement for algorithm execution, it is very time consuming to actually program them. Traditionally, the development of custom DSP hardware solutions progresses in stages. The first stage is the theoretical exploration of algorithmic concepts. High-level software packages such as Matlab, Maple, Ptolemy, or Mathematica are typically used to aid in this process. Next, a hardware implementation of the algorithm must somehow be realized. Mathematical operators that implement the algorithm functionality must be found or be created for the specific target FPGA. The operators must then be hooked together and must be either simulated or actually fabricated for testing. Errors in either the design of the operators or in the algorithm itself must be discovered and fixed. All of these tasks may take a large amount of labor and calendar time. It is not unusual for several man-months or years to be expended in mapping an algorithm into custom hardware.

Often, there are approximations that can be made in order to improve the algorithm efficiency. These could include variable wordlengths and data representations, and approximations to functions in order to reduce system

Send correspondence to pfiore@sanders.com.

complexity. Many times, the effect of these approximations on algorithm performance is not clear-cut, resulting in several iterations of algorithm and hardware design.

For many algorithms, there are specialized arithmetic structures that have been developed. These structures enable algorithm execution to be further improved, assuming the circumstances are correct for their application. Algorithm designers and mathematicians typically are not cognizant of these structures, placing the burden of recognizing their applicability on the hardware designer. This can be problematic because it may take a rearrangement of the algorithm to use a specialized structure, a task which the algorithm designer would ideally perform.

2. DESCRIPTION OF OUR APPROACH

We have developed a new method of performing this implementation process. Our approach will allow for a substantial reduction in the time it takes to design, implement, and test custom DSP algorithms in FPGAs and configurable computers. This design time reduction will enable mathematicians to quickly see the effect of algorithmic changes on hardware efficiency, reducing the number and time of design iterations.

Our method consists of a combination of high-level mathematical algorithm development software linked to more traditional hardware design CAD tools, and a set of FPGA or configurable computing devices. The mathematician develops an algorithm as usual using the high-level tool, producing a software-executable script. When the designer targets portions of the algorithm to hardware, the tool derives a hardware netlist from the specified operations. A library of designs is checked for implementations of the functions that are called out. These functions are then automatically generated by a novel tool called a Smart Generator. Next, the design is placed and routed, and the final FPGA programming file is available for use. When the algorithm script is run again, the tool recognizes that a hardware design is available. It loads and executes the design in the FPGA whenever it encounters a call for this function in the algorithm script. Multiple custom hardware designs are supported; a design is loaded and executed whenever it is encountered in the user's script.

With our system, the user's algorithm script will always execute properly, regardless of the state of porting of the algorithm from software to hardware. This allows for simple and rapid testing of the hardware implementation for the effects of approximations. It also automatically builds a test environment, a task which usually requires some expenditure of labor.

The seamless interface between custom hardware and the algorithm script allows for a new style of custom hardware design. An inexperienced user can start with a purely software script, incrementally port sections of the script into a custom hardware implementation, and immediately test the resulting hardware/software combination. When the combined system is executing the algorithm fast enough, the user can stop porting, locking down the hardware designs, the sequencing of these designs onto the hardware resources, and the communication of the hardware with the remaining software. Painstaking architecture trades, partitioning studies, scheduling, layout, interfacing, and test generation may be reduced or eliminated because it is a simple matter to just try an implementation.

The Smart Generator tools provide a systematic way to capture design information about specialized arithmetic structures. Smart Generators allow for rapid population of libraries with new components by capturing algorithm-specific implementations, from the bit-level up to the coarse-grain function level, and by making information about the implementation available to the Run-Time Workspace Manager and layout tools. An algorithm designer specifying a mathematical operation may not be cognizant of many of the ingenious implementations that have been developed. Even simple structures such as adders, multipliers, and filters have a bewildering array of implementation alternatives, many of which have been finely tuned by computer architects and VLSI designers. A Smart Generator can choose the correct implementation considering the user's specification and the target technology. This approach is different (and better) than a simple parameterized building block.

Figure 1 shows a conceptual diagram of our approach. The main blocks consist of Custom Computing Hardware closely linked with a software environment and user interface. Portions of the software environment may reside in either a workstation or an embedded processor.

The Algorithm Script or Interactive User Command Window is the user's primary interface to the system. The user's commands are executed by the system, in hardware or software as appropriate. This style of algorithm development has proven successful in purely software implementations (for example, MATLAB). Experimentation is fostered by allowing interactive interpretation of commands typed directly into the Command Window, allowing the user to debug and fine tune the algorithm. Our basic variable data item is a matrix (vectors and scalars are

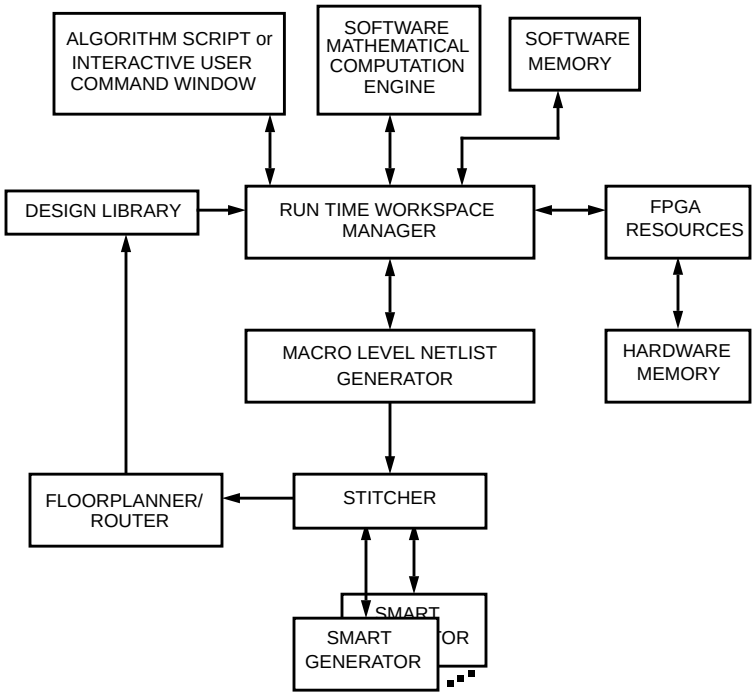


Figure 1. Conceptual diagram of our environment.

considered to be matrices). This eases hardware design because there is a direct correspondence between software data vectors and hardware custom processing pipelines.

The Software Mathematical Computation Engine provides processing results to the user for operations and variables that are not mapped to hardware. For our experiments we used the MATLAB Engine environment. Including this function in our environment allowed the user to incrementally move portions of the Algorithm Script to hardware. Any functions not ported will run in software, interacting with hardware functions as necessary under command of the Run Time Workspace Manager. When the desired operating speed for the Algorithm Script is reached, the user may stop the porting process.

Software Memory is traditional memory that software programmers refer to as "heap space". When the user declares a new variable that has not been previously assigned to Hardware Memory, space is allocated in Software Memory and the variable is stored in the allocated space. The Software Memory also contains a symbol table, which is used to keep track of all the variables in the algorithm.

Hardware Memory is critical to our approach because the FPGA resources require high-speed, low-overhead, data sources and sinks. Workspace management overhead is eliminated during FPGA processing of data because the data is directly streamed out of the Hardware Memory into the FPGAs and back again (if necessary). Because FPGAs typically have hundreds of usable I/O pins, a processing speedup over conventional software approaches is possible by streaming data over all these pins, rather than the limited 32 or 64-bit data buses of conventional processors.

The Run-Time Workspace Manager interprets the user's commands to perform the appropriate actions, depending on the state of hardware designs and degree of porting from software to hardware. It does this by recognizing basic MATLAB binary operations and function calls. Functions not mapped to hardware are evaluated by the Software Mathematical Computation Engine. When the Run-Time Workspace Manager encounters a function that is mapped to hardware, the actions taken vary depending on where the data is located and what design is loaded into the FPGA. If the data is located in Software Memory, then the Run-Time Workspace Manager assumes that a pure software evaluation is to be performed. Any variables that are located in Hardware Memory are automatically uploaded to Software Memory, and then all operations are performed in software.

If the variable is in Hardware Memory and the correct FPGA design is in the Design Library, then the Run-Time Workspace Manager assumes that the user wants to evaluate the expression using the FPGA resources. It checks to see if the correct design is loaded into the FPGA. If not loaded, the it commands a download of the correct design. Once it is determined that the correct design is loaded into the FPGA, the Run-Time Workspace Manager commands the hardware to process the data and place the results into the correct location in Hardware Memory.

If the variable is in Hardware Memory and the correct FPGA design is not in the Design Library, then the Run-Time Workspace Manager assumes that the user wants to evaluate the expression purely in software, and then download the results to the Hardware Memory. The expression is parsed, all operations are performed in software, and the results are downloaded to the Hardware Memory.

2.1. Macro Level Netlist Generator

This function translates Algorithm Script code lines into a high-level netlist. This netlist is different from HDL approaches because the netlist at this level is more abstract than a wire-accurate low-level netlist. In this mode, where the variable is in Hardware Memory and the correct FPGA design not in the Design Library, the Macro Level Netlist Generator compiles information about the hardware design. Each binary operation and function call generates a set of Netlist records; one record for each input and one record for each output. Critical information, such as the instance number, node number, and precision data, are contained in each Netlist record.

The example shown in Figure 2 demonstrates the generation of a Macro Level Netlist from an Algorithm Script. In fact, there are three examples; all of them are complex multiplication operations. The first example is a purely software implementation. Note that no netlist or FPGA hardware is generated. The second example is a combined hardware/software implementation, and the third example is a purely hardware implementation. To move incrementally from software to hardware, one merely needs to map more of the computation by a hardware allocate (HALLOC) command.

Several explicit hardware-oriented commands must be added to the user's algorithm to enable porting to hardware. The HALLOC command tells the system that a particular variable is to reside in Hardware Memory, rather than Software Memory. If the variable is previously defined and contains data in Software Memory, the data is transferred to Hardware Memory. The user may also specify a starting address for the variable vector or matrix, or may allow the system to autoassign an address. Also entered with the command is the arithmetic format of the variable. When "halloc(identifier,paramvec)" is encountered by the Run-Time Workspace Manager, it adds a record to the Netlist containing the "identifier" name, arithmetic format, source/destination flag, and the Macro name "HWMEM". The CHAIN keyword declares a variable to be an intermediate result, used in other hardware computations, but not of interest to the user by itself. This allows the user to optimize memory accesses by not storing unneeded data. When "chain identifier" is encountered, the Netlist is modified by switching the macro name from "HWMEM" to "CHAIN". The Hardware Design Available (AVAIL) keyword declares that the hardware configuration is available in the design library. An optional switch on the command will actually configure the FPGA with the design.

2.2. Smart Generators

Smart Generators allow for rapid population of libraries for new adaptive computing components by capturing either algorithm-specific implementations or a rule for creating an implementation. These generators can exist from the bit-level up to the coarse-grain function level, and their information about implementations are made available to analysis and mapping functions through a common library.

An algorithm designer specifying a mathematical operation may not be cognizant of many of the ingenious implementations that are available. The problem is compounded by the fact that with an adaptive computing system, the available hardware may not be a static quantity, the variations being caused by possible dynamic run-time scheduling. Even if compile-time scheduling is assumed, the scheduling software generally must have at its disposal several implementations for the same function in order to efficiently use available resources. New implementations must be able to be generated on the fly, under software control, with the ability to change the operand size, execution speed, layout shape, output accuracy, and even target technology.

A novel feature of the Smart Generator is the ability to utilize other Smart Generators in constructing its hardware implementation. Clearly, very useful higher-order functions can be constructed in this manner, with a minimum of effort. For example, filters, Fast Fourier Transforms, and linear algebraic operations such as matrix multiplication can be constructed out of simpler building blocks such as additions, subtractions, multiplications, and divisions. A

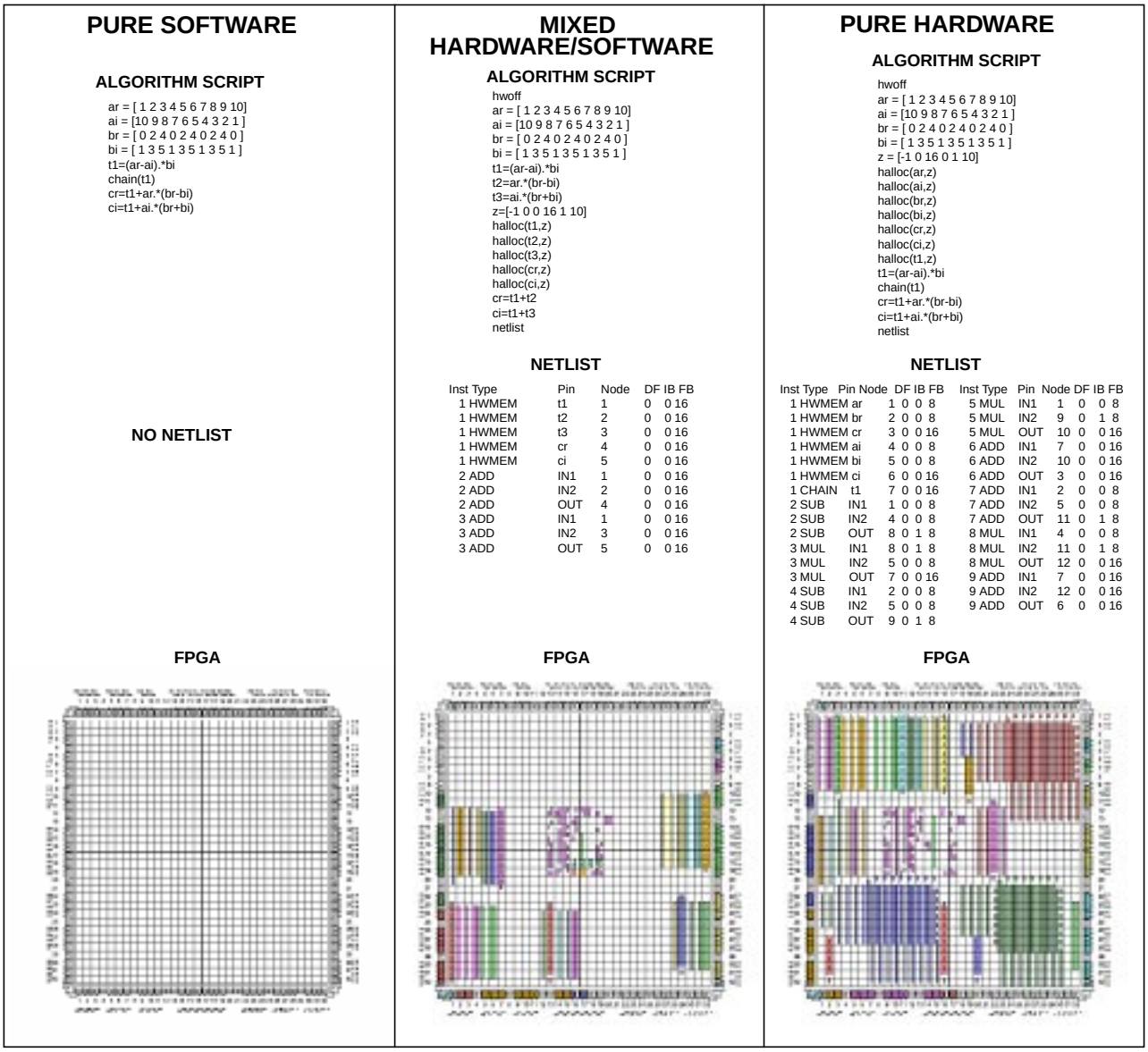


Figure 2. Incremental porting of a complex multiplication operation.

Smart Generator may actually choose among multiple implementation strategies that are known to the designer, and may even call itself with different design parameters. Table 1 summarizes the benefits of the Smart Generator approach over a conventional parameterized building block approach.

To actually write a Smart Generator, a designer must invoke basic element building blocks which are appropriate to the FPGA technology. Relative positioning information can be supplied, capturing the designer's knowledge about local dataflow requirements. Our basic Xilinx 4000 FPGA logic family building blocks are shown in Table 2. This list represents what is typically needed to describe in detail the design of a higher level building block.

2.2.1. Signed Multiplier Smart Generator Example

For clarity, an example of how Smart Generators are used by specialized algorithm designers to produce efficient, parameterized, macros is given. Here we demonstrate a Smart Generator that implements a signed Pezaris array

Table 1. Comparison of standard Parameterized Building Block (PBB) and Smart Generator.

Feature	PBB	Smart Generator
Performance Analysis	No direct support	Utilizes mathematical tools to optimize performance/cost
Architecture	Fixed	Can choose among several architectures, can utilize recursion, new methods easily linked
High-Level functions	Rare	Easy to populate using hierarchy of low-level Smart Generators.
I/O Formats	Param. is # of bits	Formats can be different – Smart Generator customizes HW to formats
Pipelining	Fixed	Variable – Smart Generator negotiates with Stitcher
Technology	Fixed	Multiple technologies (VHDL, vendor FPGA formats, C) easily supported
Layout	Fixed	Can change layout rules to during global placement
Path to dynamic run-time mapping	None	Capabilities are extensible to support dynamic HW/SW mapping

multiplier.[15] Figure 3 shows a 4×4 multiplier, built from Full Adders, AND gates, and NAND gates.

Figure 4 shows a floorplan of the 4×4 signed multiplier. Each block labeled "CLB" is a configurable logic block, capable of containing two generic 4-input, 1-output combinatorial logic elements (labeled "F" and "G"). The grey shaded areas in the figure indicate how basic building blocks and interconnect are to be replicated for different sized multipliers. Figure 5 shows this in more detail. In this figure, the numbers inside the grey shaded areas are the row and column sizes in CLBs needed for the region. The text accompanying each area is the required functionality inside each CLB within the area. Figures 3, 4, and 5 show how to floorplan the multiplier, and what logic functions belong in each CLB, as a function of operand size. With this knowledge, the Smart Generator can be written.

Figure 6 shows a MATLAB code fragment of this Smart Generator. The function "read_message" gets the design data from the Stitcher. The string variables "in1", "in2", and "out" are the basenames of the input and output buses. The matrices "i_sizes" and "o_sizes" contain the numeric formats of the input and output buses. The variables "clbrows" and "clbcols" contain the number of rows and columns of CLBs that the design will occupy. The vectors "inds1" through "inds4" are expanded versions of the input and output numeric formats, used for alignment purposes. The code at the top right of Figure 6 is concerned with grounding desired output bits that are below the LSB of the multiplier array and sign extending the multiplier output to desired output bits above the MSB of the array.

Next in Figure 6 is the double nested loop that builds the grey shaded region of Figure 5 labeled "AND Gate and Full Adder". This is the only region that grows in both directions as a function of N and M. The strings "valstrsum" and "valstrcar" contain the truth table for the two 16ROM function generators inside each CLB. The variables "row" and "col" contain the row and column numbers for each 16ROM block. Additionally, to specify the position of a 16ROM block, we need to distinguish between the "F" and "G" locations within a CLB. The string matrix "innames" contains the names of each of the four inputs to the CLB. Notice that two of the inputs are from the primary inputs "in1" and "in2". The other two input name strings start with "fac" and "fas" and are the carry and sum outputs from the previous column in the array. The rest of the input name strings are based on the loop indices and are included as part of the name so that the correct bits from the previous column are hooked up to the CLB. The string matrices "outnamesum" and "outnamecar" contain the names of the two outputs of the CLB. The name convention here is the same as that for the "innames" string matrix, ensuring that the current column will

Table 2. Basic Smart Generator building blocks.

Function Name(s)	Description
AND, NAND, OR, NOR	Simple combinatorial logic
NOT, XOR, XNOR	Simple combinatorial logic
16ROM	Generic 4-input 1-output combinatorial logic element
FD	Positive edge triggered flip-flop
FDCE	Positive edge triggered flip-flop with clock enable and clear
VCC,GND	Used to tie off an input to a fixed high or low voltage
FA	Full Adder
FA2	Two bits of a fast ripple carry adder string
CONNECT	Connect two signals together

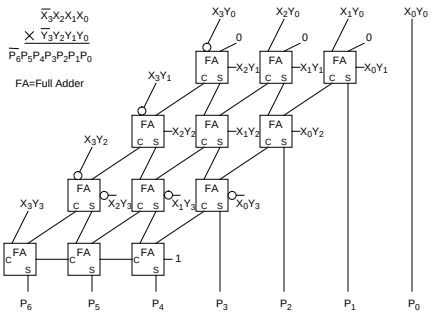


Figure 3. Signed, nonpipelined Pezaris array multiplier.

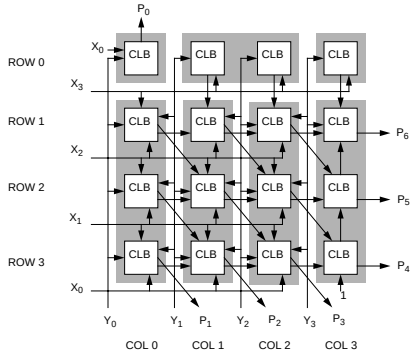


Figure 4. Floorplan of 4×4 signed multiplier in XC4000 technology.

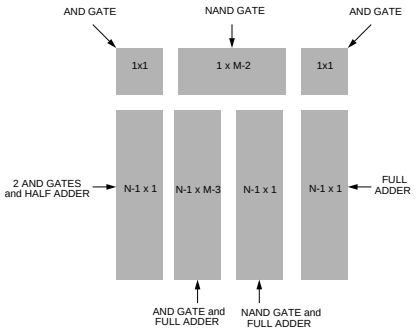


Figure 5. Replication pattern for an $N \times M$ signed multiplier.

get hooked up to the next column. Finally, all the input and output names and truth table values for both 16ROM blocks are known, as well as the relative position within the CLB array. The 16ROM blocks are next invoked with this information at the bottom of the double nested loop.

In Figure 6 the actual code next implements all the other grey shaded regions in a similar manner. This code has been omitted for brevity. The remainder of the code performs some housekeeping functions and generates a response for the Stitcher.

2.3. Stitcher

This function uses the data produced by the Macro Level Netlist Generator to coordinate the production of an actual low-level netlist and hardware design. This is done by properly accessing information contained in the Smart Generators, augmenting the netlist with this information, commanding the Smart Generators to actually create their associated hardware, and then linking all the hardware designs together.

The Stitcher negotiates with the Smart Generators to produce an optimal design; information about word formats, pipelining options, and relative macro sizes are exchanged. The Stitcher then schedules the algorithm onto the FPGA hardware. This involves pipeline realignment and data port conflict resolution. To perform these functions, the Stitcher will add elements into the netlist. These include delays, multiplexers, address generators, tristates, and latches, each of which has its own Smart Generator. At this point, the Stitcher creates the detailed low-level hardware design.

The Stitcher scheduling algorithm is involved and is best illustrated with an example. Figure 7a shows a conceptual algorithm data flowgraph with three inputs, two outputs, and five operations, or "instances". The three input vector streams A, B, and C all reside in one bank of RAM (also called a Port). The two output vectors D and E reside in a second port. All data vectors are assumed to be the same size. All the instances in the design will be created with a pipeline delay of one, except I2, which requires a pipeline delay of two. Because of the unequal amount of pipeline delays, extra delays must be appropriately added to the netlist to correctly align the pipelines.

In addition to this pipeline alignment problem, the usage of the data ports must be correctly scheduled. For example, we cannot schedule D and E to be written at the same time. Our approach is to add multiplexers to the netlist for cases where multiple outputs go to the same port. We also add tristate buffers for cases where both reads and writes to the same port can occur. Latches for input data can also be provided. These are not strictly necessary, but can provide for improved timing.

Figures 7b and 7c shows the results of this scheduling operation. Note the addition of the input latches, the internal delay, and the output multiplexer into the netlist. The Stitcher also connects all logical ports to the appropriate physical port. The timing chart in the figure shows the activation sequence of the internal nodes of the algorithm as well as the activity of the multiplexer control signal, latch load enable signal, and port activity. Several consecutive launches are shown in the chart. Note that the maximum launch rate of three cycles is attained.

To show the flexibility of this scheduling approach, the same algorithm example is used, but with a different data distribution. Figure 7d shows that input C now resides in Port 2, along with the outputs D and E. Figures 7e and

```
function gen_mul

% Generate an arbitrary sized multiplier
% pezaris array approach

stitcher_globals; %get global definitions

% Determine what the stitcher has provided for us to create
[mode,i_names,i_sizes,techno,output_file,o_names,o_sizes,opt_mode, ...
    pipe_d,aspect]=read_message;

in1=unpak_str(i_names(1,:)); %name of x bus
in2=unpak_str(i_names(2,:)); %name of y bus
out=unpak_str(o_names(1,:)); %name of p bus

% Determine if gen_mul needs to determine output sizes
if (isempty(o_sizes))
  %determine output size
  o_sizes=sum(i_sizes); %ointbits=sum of i intbits, etc
end

t=sum(i_sizes'); %no. bits each word
s1=t(1);
s2=t(2);

%figure out the size of the mult
clbrows=s1;
clbcols=s2;

% If only a query, then provide a response and return
if (mode==QUERY)
  % Determine size and pipe delay estimates
  create_response(mode,[clbrows clbcols],0,o_sizes,[],[]);
  return;
end

output_file=[output_file '.xnf'];
fid=fopen(output_file,'w');
xnf_4000_hdr(fid,[0 0],[],[],techno(CHIP,:));

inds1=[1-i_sizes(1,2):i_sizes(1,1)]-1; %indices lsb to msb first input
inds2=[1-i_sizes(2,2):i_sizes(2,1)]-1; %indices lsb to msb 2nd input
inds3=[ inds1(1)+inds2(1):max(inds1)+max(inds2)+1]; %mult output indices
inds4=[1-o_sizes(1,2):o_sizes(1,1)]-1; %desired output indices

%first take care of any desired outputs that would not get hooked up
%to multiplier outputs. any mult outputs that arent used will just dangle

%look for desired outputs below lsb of multiplier outputs and ground them
t=find(inds4 < min(inds3));
for k=1:length(t)
   xnf_4000_gnd(fid,[],[],[out num2str(inds4(t(k))) ]);
end

%now look for desired outputs above msb of mult outputs and sign extend
t=find(inds4 > max(inds3));
c1=[out num2str(max(inds3))]; %the sign bit for sign extension
for k=1:length(t)
  c2=[out num2str(inds4(t(k))) ];
  xnf_4000_connect(fid,[],c1,c2);
end

%build the inner "AND Gate and Full Adder" array
valstrsum='8778'; %lookup table for FA-AND sum output
valstrcar='F880'; %lookup table for FA-AND carry output
for k=2:s2-1
  for l=1:s1-1
    row=s1-l;
    col=k-1;
    innames=strvcat( ...
    [in2 num2str(inds2(k+1))], ...
    [in1 num2str(inds1(l))], ...
    ['fac_' num2str(k-1) '_' num2str(l) ], ...
    ['fas_' num2str(k-1) '_' num2str(l+1) ]);
    outnamesum=['fas_'num2str(k) '_' num2str(l) ];
    outnamecar=['fac_'num2str(k) '_' num2str(l) ];
    xnf_4000_16rom(fid,[row col F],innames,outnamesum,valstrsum);
    xnf_4000_16rom(fid,[row col G],innames,outnamecar,valstrcar);
  end %l
end %k

   <... code omitted for brevity ...>

xnf_4000_end(fid);  %write trailer to file
fclose(fid); %close the file
%generate response to stitcher
create_response(mode,[clbrows clbcols],[0],o_sizes,[],[]);
```

Figure 6. Code fragment for signed Pezaris array multiplier Smart Generator.

7f shows the results of the scheduling algorithm. Note that the internal pipeline realignment delays were added in different places than in Figure 7b. Also, a tristate element was added for the Port 2 read and write operations. The timing chart in the figure again shows that the maximum launch rate of three cycles is attained.

2.4. Address Generators

Each bank of Hardware Memory requires an Address Generator to provide proper sequencing of the data vectors. In our approach, the parameters needed by the Address Generators associated with a computation are automatically derived by the Macro Level Netlist Generator. This information is used by the Stitcher to calculate needed address offsets.

During the scheduling, the Stitcher derives an Address Generator for each data port. This approach frees the algorithm designer from the task of writing code or developing hardware for this purpose. Also, we have found that generating custom address generators for each algorithm substantially reduces the required amount of logic from what would be required for general purpose address generators.

To demonstrate the hardware savings that can accrue using custom designed Address Generators, we first designed a typical fixed general purpose address generator and sequencer, shown in Figure 8. Note that this "overhead" logic occupies almost half of the chip. Also note that this was a very restricted sequencer in that it could only handle one Port, with a very fixed sequencing pattern (two reads, then a write). In contrast, our custom designed Address Generators are substantially smaller than this, as well as much more capable.

Our Address Generators consist of an accumulator, a multiplexer, and several hardwired constants. The Address Generator operates by adding a precalculated offset to the current address whenever activity occurs on the

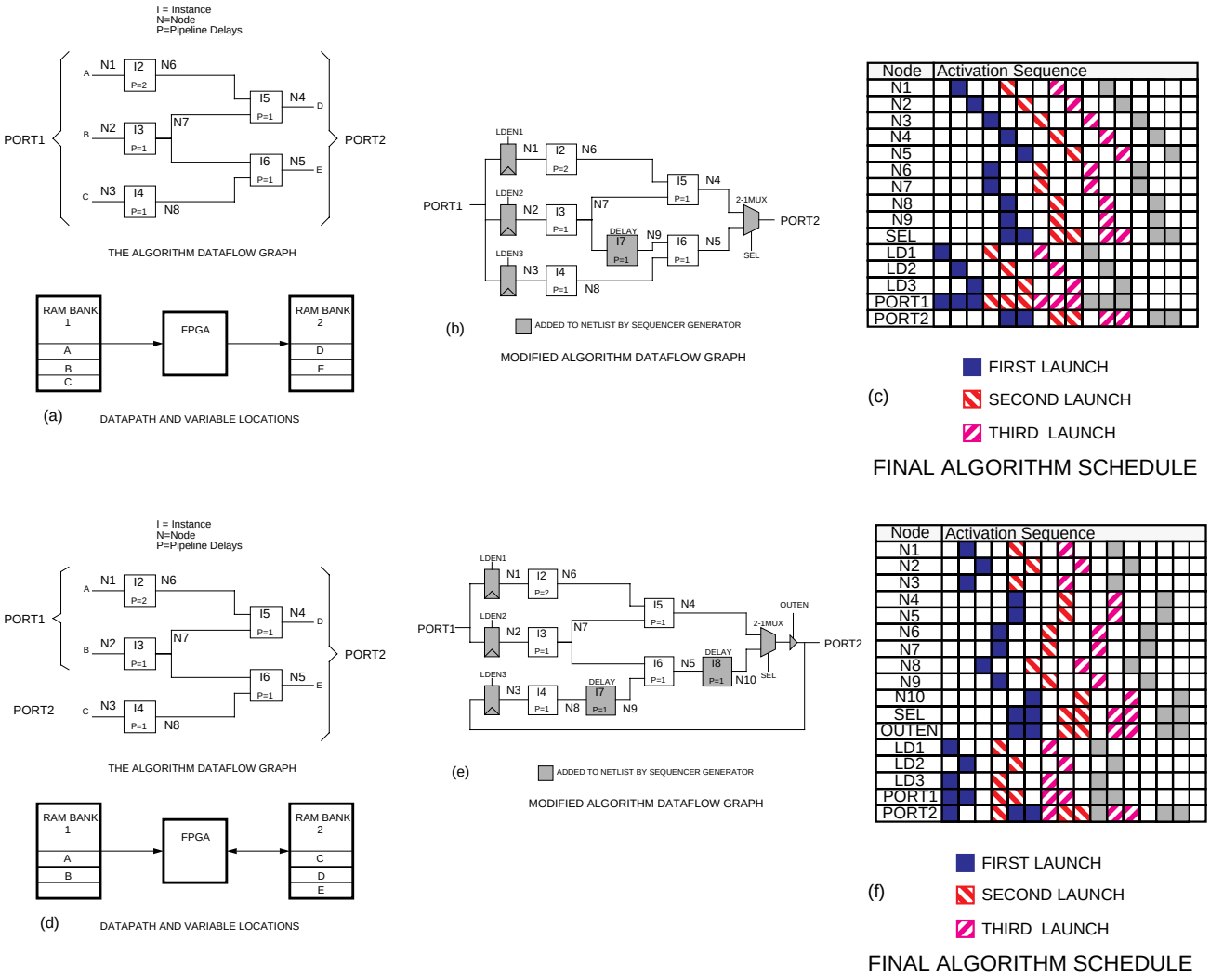


Figure 7. (a)Scheduling example. (b)Modified flowgraph for pipeline alignment and data multiplexing. (c)Resulting schedule. (d)Same algorithm as in (a), but different data distribution. (e)Modified flowgraph. (f) Scheduling results.

corresponding data port. The offsets are easily calculated by examining the schedule. The offsets are then stored as constants which drive the multiplexer input.

Figure 9 shows the FPGA hardware that was designed for the example problem of Figure 7a. Contrasting this customed designed Address Generator and Sequencer approach to that of the fixed, general purpose approach of Figure 8 shows considerable hardware savings and increased functionality for the new approach.

3. CURRENT EFFORTS

We are currently extending our design approach under DARPA/ITO sponsorship.[16,17] New capabilities are being developed using the Ptolemy signal processing algorithm and implementation environment from the University of California (UC), Berkeley.[18] Ptolemy supports heterogeneous system simulation and design using several different models of computation, each implemented in a separate domain. The class of application problems addressed by our work falls into Ptolemy's synchronous dataflow (SDF) domain, where the flow of data is predictable and does not change. In addition to its well-defined models of computation, Ptolemy provides an open software architecture with standard interfaces among its functional areas. Our choice of Ptolemy will allow both rapid insertions of our new

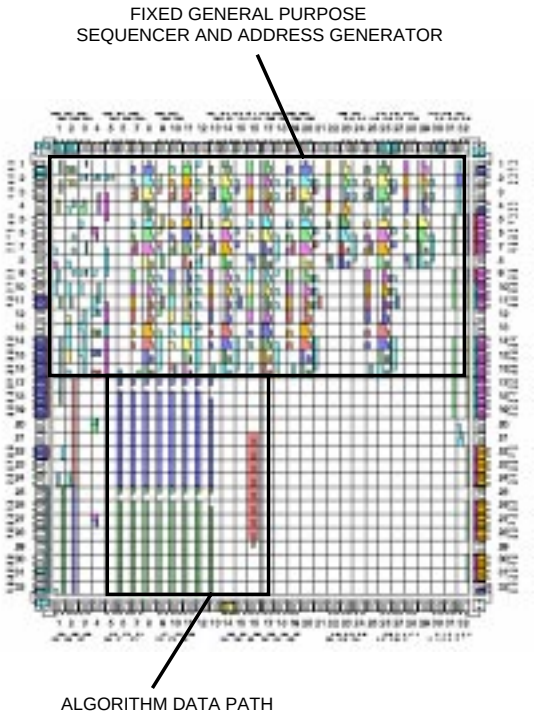


Figure 8. Typical fixed general purpose address generator and sequencer.

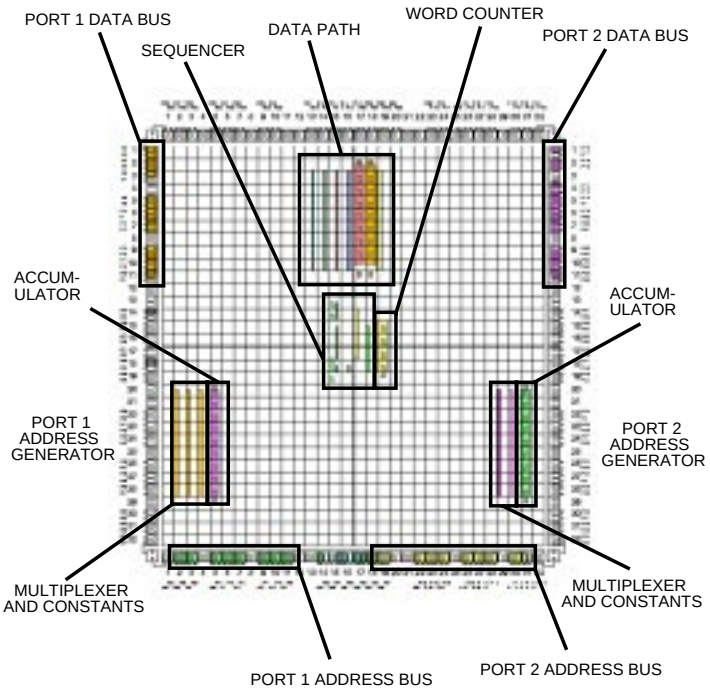


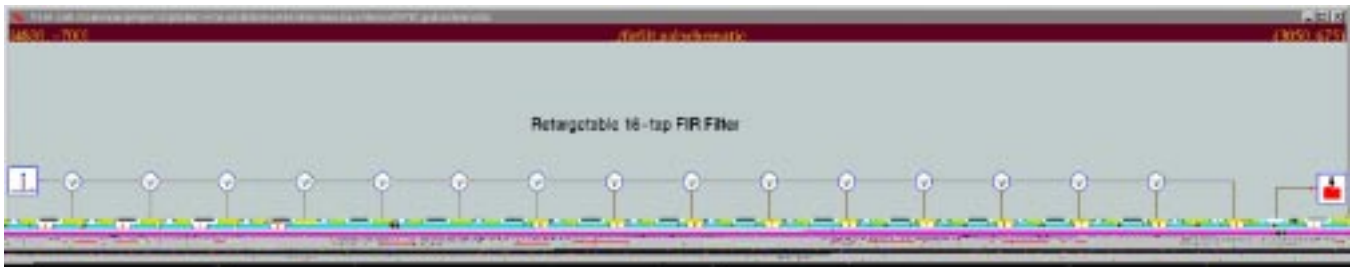
Figure 9. FPGA design for example of Figure 7a. Note the dramatic reduction in control and address generation hardware over the fixed general purpose design of Figure 8.

capabilities as well as the ability for additional researchers to build their own extensions and libraries, an approach that has been used successfully in the past.

Within Ptolemy, there has traditionally been a distinction between simulation and implementation (code generation), and separate domains were created for each purpose. No implementations (code or hardware designs) could be generated from the simulation domains and simulation was not possible from the code generation domains. Also, it has not been possible to move among different types of simulations (e.g. floating-point to fixed-point) or among code generation (e.g. C code, assembly code, hardware designs) without recreating graphs with different models.

Together with UC Berkeley, we are extending Ptolemy by developing a new Adaptive Computing System (ACS) domain that separates the interface specification from the implementation for each signal processing functional block, allowing simulation and design/code generation to exist in the same domain. In the ACS domain, algorithms are still represented by SDF graphs. We are using a "Corona/Core"[18] architecture, where each block has a common interface known as the Corona, and one or more implementations, known as the Cores. A retargeting mechanism allows the users to change Cores and hence implementations. The implementation of a data flowgraph between various simulation models (floating-point, fixed-point) and implementations (C code, VHDL code, etc.) by selecting a different target. Using our current capability, Figure 10 shows an FIR filter that can be retargeted between floating-point simulation, fixed-point simulation, and code generation in C. Figure 11 shows a comparision of the floating-point and fixed-point frequency response.

Our approach is to focus on three areas of capability critical to the success of Adaptive Computing (Algorithm Analysis, Algorithm Mapping, and Smart Generators) and to integrate these capabilities into an open, extensible software framework provided by Ptolemy. These capabilities will provide algorithm designers with tools to determine the proper wordlengths and with estimators that give feedback about the hardware implications of the design decisions. Other capabilities will include the analysis of algorithms at the bit-level to determine appropriate finite precision implementations, statistical analysis, symmetry detection, and error propagation, performance analysis, partitioning assistance, and automatic scheduling. Automatic scheduling will eliminate one of the most time-consuming hardware tasks, the design of a controller to sequence operations. The scheduling and partitioning functions will be built on top


Retargetable 16-tap FIR Filter