Delivering Acceleration: The Potential for Increased HPC Application Performance Using Reconfigurable Logic

David Caliga
SRC Computers, Inc
4240 N. Nevada Ave
Colorado Springs, CO 80907
719-262-0213

david.caliga@srccomp.com

David Peter Barker
SUPERsmith
PO Box 2226
Salinas, CA 93902-2226
831-442-8343

dbarker@supersmith.com

ABSTRACT

SRC Computers, Inc. has integrated adaptive computing into its SRC-6 high-end server, incorporating reconfigurable processors as peers to the microprocessors. Performance improvements resulting from reconfigurable computing can provide orders of magnitude speedups for a wide variety of algorithms. Reconfigurable logic in Field Programmable Gate Arrays (FPGAs) has shown great advantage to date in special purpose applications and specialty hardware. SRC Computers is working to bring this technology into the general purpose HPC world via an advanced system interconnect and enhanced compiler technology.

Categories and Subject Descriptors

Compiler technology

General Terms

Algorithms, Performance, Design, Experimentation

Keywords

Reconfigurable computing, FPGA

SC2001 November 2001, Denver (c) 2001 ACM 1-58113-293-X/01/0011 $5.00

1. INTRODUCTION

The SRC-6 system is a unique architecture capable of supporting a combination of up to 512 Intel® microprocessors and 256 Multi-Adaptive Processors (MAP™) on a common shared memory. A patented SRC crossbar switch design supports the connection of these processors with up to 256 memory ports with each port containing 16 banks of SDRAM (see Figure 1). The architectural design of the memory sub-system provides flat access latency from any microprocessor or reconfigurable processor in the SRC-6 system. This combination will offer in excess of 3 TFlops of theoretical peak [256 Pentium 4 processors and 128 MAPs on 32b floating-point data]. SRC Computers is pushing forward to harness this into sustained performance.

This paper explains how SRC Computers, Inc. has made advances in the reconfigurable computing field by incorporating FPGA technology at many levels in the SRC-6 system architecture.

Also described is SRC's patented FPGA-based MAP that user applications can utilize to deliver algorithm-specific computational acceleration.

2. RECONFIGURABLE COMPUTING DEFINED

In simplest terms, reconfigurable computing, based on FPGA technology, could be defined as the capability of reprogramming hardware to execute logic that is designed and optimized for a specific user's algorithms. Associated compiling technology can provide a transparent method of integrating the computational capability of FPGA technology and microprocessors into a single application executable code. The use of such integrated compiling technologies enables reconfigurable architectures to extend beyond the FPGAs and the "glueware" that attaches them to the host computer.

Automatic compilation of applications onto reconfigurable architectures generates the logic for both the specific hardware configuration and also the execution management of the FPGA resources. The compilation environment must also be extensible to a wide range of user applications on a single system.

SRC Computers has taken into account all of these factors in developing its unique reconfigurable computing capability for the SRC-6 system.

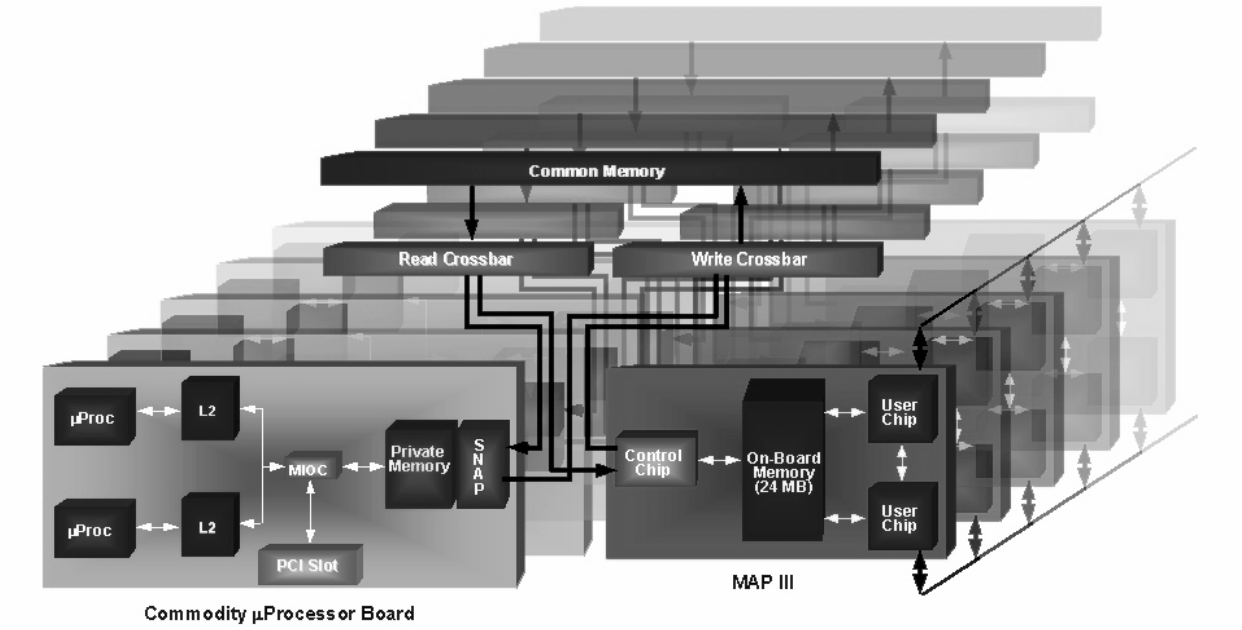


Figure 1. Overview of SRC-6 System Architecture

3. MULTI-ADAPTIVE PROCESSOR (MAP)

The heart of the SRC-6 system is its Multi-Adaptive Processing feature provided by MAP units (see Figure 2). These units utilize hardware-implemented functions, which can greatly accelerate application algorithms over compiler implemented instruction sets for microprocessors.

Major architectural characteristics of MAP include:

- Each MAP executes independently of the general-purpose processors, including loading and storing needed data, after being provided with a list of commands to execute.
- The control of the MAP is done by the application via a Command List (COMLIST). The COMLIST contains a list of controlling instructions for the Control Chip. Examples of functions performed by these instructions are Direct Memory Access (DMA) reads and writes and the execution synchronization with the User Array.
- MAP units have access to Common Memory (CM).
- Common Memory addresses specified by commands or generated by user applications are virtual addresses. Addresses are translated, virtual to physical, by Translation Look-aside Buffer (TLB) entries with the DMA logic of each MAP.
- The User Array portion of a MAP unit is configured for algorithmic requirements. This logic can read and write on-board memory through multiple ports and can interact with the control logic and DMA Engine.
- By means of chain ports, MAP units can communicate between themselves without using any memory bandwidth. Thus a particular MAP can send partial results to another unit, or similarly, can receive such partial results from another MAP.
- Full system interrupt and semaphore capability is available between MAP units or the microprocessors.

Multi-processor applications can easily utilize the MAP by identifying the relevant portions of the parallel computational algorithms. The application can utilize N microprocessors and M MAPs.

4. RECONFIGURATION LOGIC PERFORMANCE POTENTIAL

The attraction of using FPGAs in SRC's MAP is the ability to generate algorithm specific logic, which has the potential of orders of magnitude speedups for computationally intensive algorithms. There have been many demonstrations showing this magnitude of speedup in algorithms for genetic sequencing, encryption/de-encryption, string searches and integer forms of image and signal processing.

The performance improvement of these algorithms can come from any or all of the following:

- Memory bandwidth improvements
  - Six ports to memory
- Data flow parallelism
  - Bit-sized data allows for multiple parallel processing streams per 64 bits of data read from memory or algorithms that may need multiple 32 or 64-bit input values
- Computational block level re-scheduling
  - Re-schedule independent computations to be concurrent in time
- Instruction Set Architecture (ISA) effectiveness
  - Create operations that are "right-sized" in bits relative to the type of data, i.e. 6-bit or 256-bit integer operations

The following examples show potential performance improvement contributors of an FPGA over that of a microprocessor.

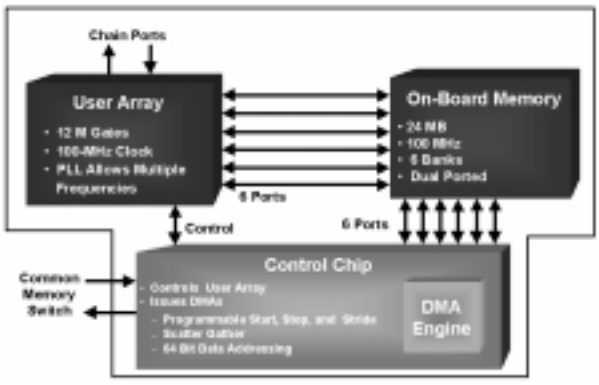


Figure 2. MAP Block Diagram

Example 1:

This example shows a performance comparison between an FPGA and a 900-MHz microprocessor.

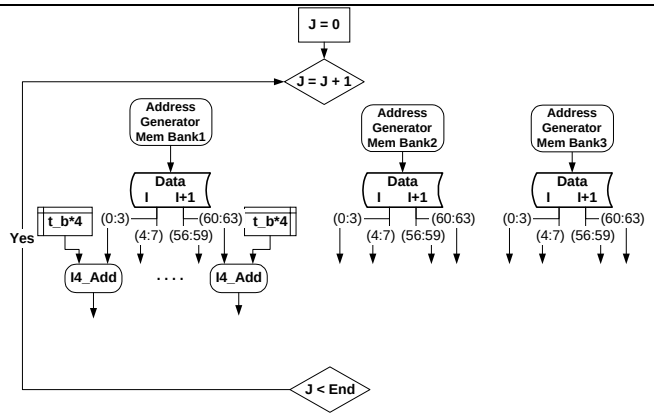
Perform an integer add operation on each 4 bits of a data stream. The MAP reads in 64 bits of data from each memory bank. The algorithm will use 3 memory banks to read data from the input stream and the logic will create 16 parallel computation streams.

Feature	Speedup over Microprocessor	Derivation
Clock rate	100/900 or 1/9	100 for the FPGA and 900 for the microprocessor
Memory bandwidth	3	3 memory banks to read data from input stream
Data flow parallelism	16	16 parallel computation streams
Block level re-scheduling	1	None
ISA effectiveness	10	Number of instructions on the microprocessor required to perform the equivalent integer operation on the 4-bit data value
Total	**53.3**	**1/9 *3 * 16* 1 * 10 = 53.3**

Algorithm Pseudo Code

```
Loop over 4-bit values in 32-bit data value (isrc) and add a 4-bit value to input value. Store resulting 4-bit value (ires)

     len = 4    //4-bit value
     ipos = 29  //start at position 29

    Do 100 j = 1, 16
      ibgn = (j-1)* 4 + 1

      mvbits (isrc,ibgn,len,itemp,ipos)

      ires  = idest + t_b4(j)
           //4-bit value stored in ires

      mvbits (ires,ipos,len,idest,ibgn)
100 Continue
```

Segment of Logic Flow



Example 2:

The following example shows a comparison between an FPGA and the Alpha EV6.

Perform a 32-bit floating-point convolution filter on a set of vector data. The convolution filter will be 64 points. The filter will be stored in a set of registers in the FPGA chip. Two values of the input data will be read every clock. The output computation rate will generate two output values every clock.

Operation	FPGA	DEC Alpha EV6
Read input data values	2 values every clock	1 value every clock
Operations every clock	128 Multiply-Adds	1 Mult and 1 Add

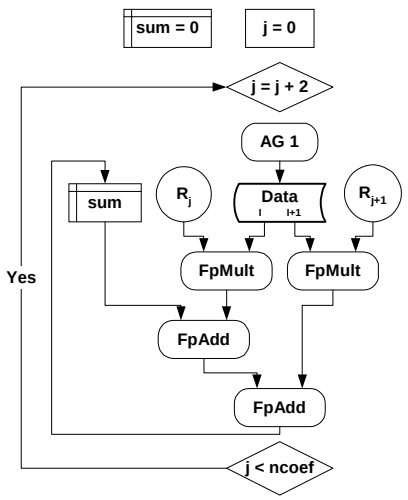
Feature	Speedup over Alpha EV6	Derivation
Clock rate	100/800 or 1/8	100 for FPGA and 800 for Alpha EV6
Memory bandwidth	2	2 values read every clock
Data flow parallelism	1	N/A
Block level re-scheduling	64	Convolution filter is 64 points
ISA effectiveness	3/2	Number of instructions on the FPGA relative to the microprocessor per clock
Total Speedup	**24**	**1/8 * 2 * 1 * 64 * 3/2 = 24**

A segment of the filter code is shown here:

Filter Pseudo Code

```
C Loop over output points

   Do 100 n = 1, nout
     sum = 0.

C Loop over filter coefficients
     Do 200 j = 1, ncoef
       sum = data_in(n+j-1) * filter(j)
100   continue

     data_out(n) = sum

100 continue
```

Logic Flow



Example 3:

This example also shows a comparison between an FPGA and the Pentium 4, using integer data.

Integer arithmetic function units take much less space within an FPGA chip than the floating-point unit. Let's change the previous example to 32-bit integer data. The convolution filter will be 256 points. The filter will be stored in a set of registers in the FPGA chip. The input/output data will each be striped across two memory banks. Four values of the input data will be read every clock. The output computation rate will generate two output values every clock.

Operation	FPGA	Pentium 4
Read input data values	4 values every clock	1 value every clock
Operations every clock	512 Multiply-Adds	1 Mult or 1 Add

Feature	Speedup over Pentium 4	Derivation
Clock rate	100/1700 or 1/17	100 for the FPGA and 1700 for the Pentium 4
Memory bandwidth	4	4 values read every clock
Data flow parallelism	1	N/A
Block level re-scheduling	256	Convolution filter is 256 points
ISA effectiveness	2	Number of instructions on the FPGA relative to the microprocessor per clock
Total Speedup	**120.5**	**1/17 * 4 * 1 * 256 * 2 = 120.5**

4.1 Amdahl's Law

A generally used measure of performance or speedup of an algorithm for various architectures is Amdahl's Law. It has been used for vector and parallel systems to show the benefit of optimizing portions of code. Amdahl's Law points out that given the percent of time spent in a portion of code the overall application may get only marginal overall speedup even though the algorithm was made to execute much faster. We can apply the same principal to algorithms moved into MAP. Let's examine the previous examples and look at the application speedup up given various percentages of time spent in the algorithm.

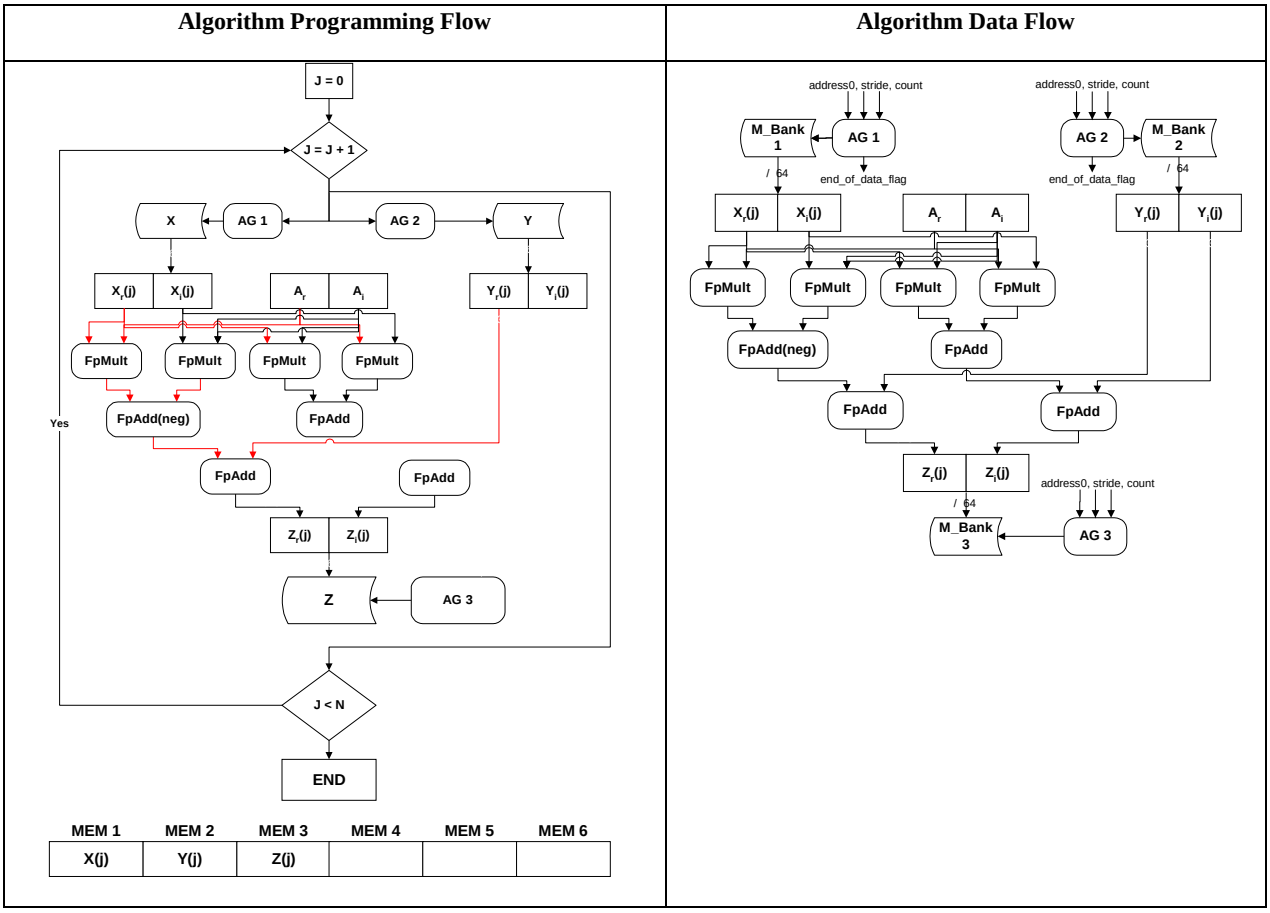
These examples have shown application speeds for several types of integer and floating-point types of problems. The benefits of using FPGAs are obviously for algorithms that have high performance speedups and high percentages of computation time.

	Ex. 1	Ex. 2	Ex. 3
MAP Algorithm Speedup	53x	24x	120x
Percent of time spent in algorithm	**Application Speedup**		
80	4.7	4.3	4.8
90	8.5	7.3	9.30
95	14.7	11.2	17.3
99	34.9	19.5	54.8

5. PROGRAMMING FLOW VS. DATA FLOW

Algorithms that will be moved into the MAP should be thought of as data flow problems from the hardware logic perspective. The following example will look at the CAXPY algorithm [def: A*X(j) + Y(j) = Z(j)]. The traditional way for a programmer to think of CAXPY is as a loop over the number of elements in arrays X and Y. However, in hardware, it is thought of as a data flow through logic. The logic will read values for X and Y every clock and send the values through the logic definition to the set of FpMults and FpAdds and create a value for Z every clock.

The compiler, as part of its analysis of code segments, creates a data flow graph and performs dependency analysis. This information can then be used to create an algorithm data flow that will be put into hardware logic for the FPGAs.



The arithmetic function units instantiated in the FPGAs have a pipeline design. This means that a new data value can be input into the function unit every clock. There is a latency associated with each function unit before a final result is produced from the input data value. After the first data value comes out, subsequent output values will come out every clock. Assume that the latency for an FpMult and FpAdd are both 10 clocks.

Figure 3 shows how data flows through a pipelined function unit. In addition, the figure shows a function unit that has multiple phases. Floating-point units often have pre- and post-processing for format conversion, i.e., conversion from IEEE Floating-Point representation into an internal representation.

Going back to the CAXPY example, Table 1 shows how many clocks it will take for data samples to pass through stages of the hardware logic.

The processing time of hardware logic is similar to that of very long vectors on vector processor systems. The time in clocks to process a set of data, of length **Nelem**, through CAXPY would be:

Time (clocks) = Nelems + Latency
= Nelems + 35 clocks

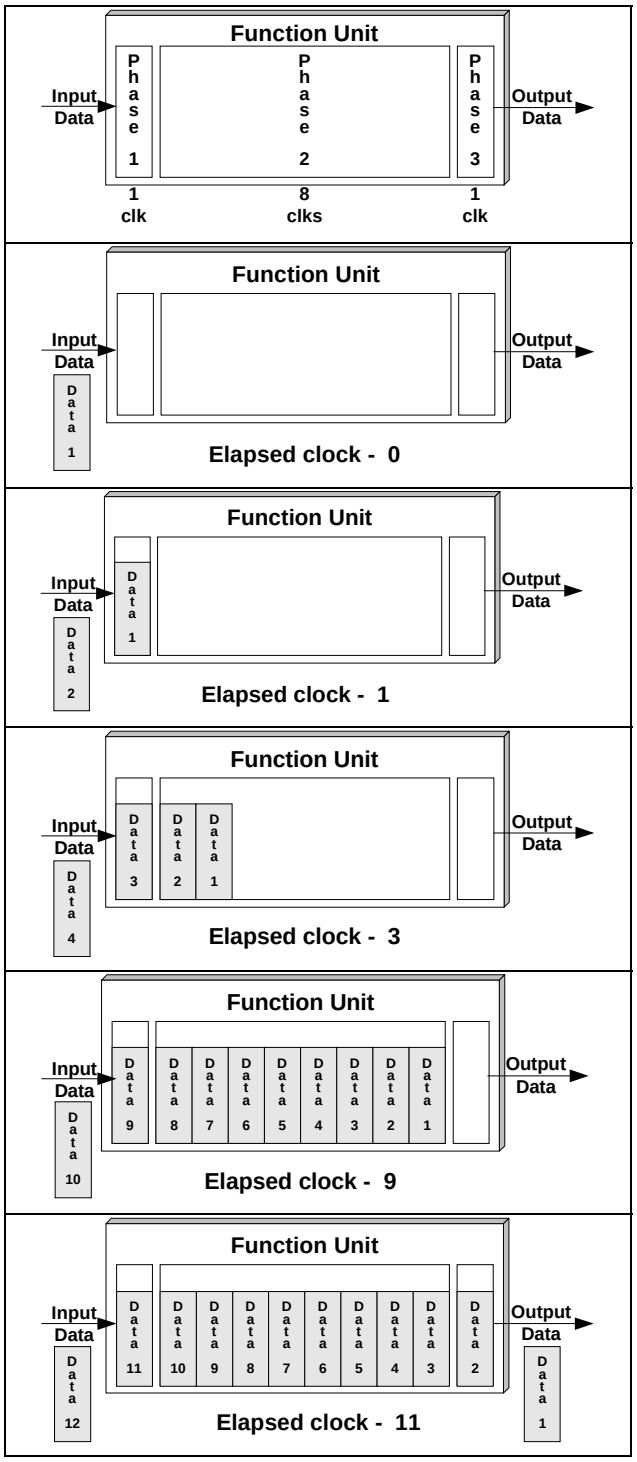
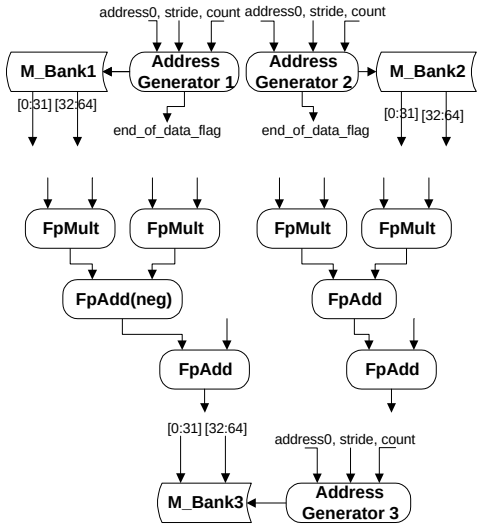


Figure 3. Data Flow through Pipelined Function Unit

Table 1. Data Sample Processing Latency

Data Flow Logic	Clock								
	1	2	…	13	…	24	…	35	36
	D 1	D 2		D 3		D 4		D 5	D 6
		D 1		D 2		D 3		D 4	D 5
				D 1		D 2		D 3	D 4
						D 1		D 2	D 3
								D 1	D 2

6. DESIGNING ALGORITHMS FOR MAP

One of the major issues that have hindered the use of FPGAs to date for general-purpose scientific algorithms has been the lack of an ability to use floating-point arithmetic. This impediment has come from two related factors.

The first has been the actual size, gate count, of FPGAs. Until recently, the size of FPGAs has been under 1M gates. The advent of multi-million gate FPGAs has dramatically changed the possibility of the type of logic than can be loaded into a single FPGA chip.

The second limitation has been the size of floating-point (32- or 64-bit) functional units for FPGAs. The number of gates required to define IEEE-754 compliant function units is much greater than those for integer function units.

Table 2 shows the approximate number of function units that can be defined in an SRC MAP III.

Another consideration is the amount of the algorithm logic that will fit in a single FPGA chip. The MAP environment has the capability to extend the algorithm across multiple FPGA chips and multiple MAPs. The MAP User Array interconnect provides the ability to transfer 24 bytes per clock. A transfer can be configured dependent upon the data type being transferred. A transfer could send three 8-byte elements, six 4-byte elements, twenty-four 1-byte elements, etc. The transfer between MAPs via the Chain Ports can send 12 bytes per clock. This MAP interconnect does not use any switch bandwidth.

Table 2. Function Count for MAP III

Function Unit Type		Function Count
Integer, 32b	Multiply	140
	Add	3000
64b	Multiply	36
	Add	1570
Floating Point, 32b	Multiply	48
	Add	108
64b	Multiply	24
	Add	48

7. SRC COMPILER TECHNOLOGY AND TOOLS

SRC is extending a well-known vendor's FORTRAN and C compilers to target compilation of computationally intensive portions of applications for FPGAs. Part of the compilation process is the generation of a data flow graph (DFG) and the dependency analysis of the application code. The DFG and dependency analysis are used to define the layout of programmable logic in the FPGAs and data layout in on-board memory. Two execution-time components will be generated. The first is the hardware logic defined from the data and control flow analysis. The second is the definition of a "Command List", or COMLIST, for the execution control of the User Array environment and issuance of direct memory access (DMA) instructions.

Current hardware design can be accomplished through the use of Hardware Definition Languages (HDL). An HDL is a high level language similar in concept to a software high level language such as C. The process of converting the HDL into a hardware design consisting of wires and transistors etc. is called "synthesis". Synthesis is somewhat analogous to normal software compilation.

There are a great number of tools and products available that work with various HDL. The compilation strategy for the MAP is to leverage the ready availability of synthesis tools etc. available for HDL (see Figure 4). The basic premise is that compilation for the MAP shall consist of translating the software language of the user,(i.e. C or FORTRAN) into an HDL representation. This HDL representation can readily be processed by existing hardware design toolsets into the bitstream used to configure the User Logic (U_Logic).

The chosen HDL target language is VERILOG. The VERILOG language allows representation of hardware design at various levels of abstraction. At the lowest level, it is possible to write VERILOG code that represents individual gates and wires. At the opposite extreme, one can write so-called "behavioral" code, which has a high degree of abstraction. In fact, it is possible to write behavioral code that cannot be synthesized into hardware. Such code is useful for the purposes of simulation.

It is not the intent of the compiling strategy that the synthesis of VERILOG shall include the synthesis of floating-point operations etc. One of the primary constraints for the compiler is compile time. The user of the compiler will have expectations that the compilation should be done quickly. This presents a significant challenge in the "place and route" process. Since the compiler will be using existing commercial tools, the compiling system must function in such a way as to make the job of these tools as easy as possible.

To facilitate the synthesis process, the compiler-generated VERILOG code shall consist solely of instantiation and hookup of an existing set of pre-defined VERILOG modules. A VERILOG module is similar in concept to a software subprogram, or subroutine. Importantly, experienced hardware designers at a very low design level can create these pre-defined modules, in a pre-placed manner. This makes them efficient in terms of speed and resource usage, and, since they are "relationally placed", the place and route portion of the final synthesis can proceed faster.

The number and type of pre-defined modules is relatively small. Just as normal software compilation builds the behavior of the user's program from a small set of basic op-codes, so too can the U_Logic configuration be built up from a finite set of pre-defined modules, such as integer/floating-point add, multiply, divide, etc..

The development of domain-specific modules (e.g. Convolve, FFT) that can be used as building blocks in the design of FPGA-based programs is a necessity. These modules provide a bridge between the algorithm developer and the hardware logic "designer". The module will have a counterpart in an FPGA macro library. These macros will be incorporated into the DFG and used in the hardware definition language (HDL). Examples of potential libraries for the generation of macros are signal and image processing and linear algebra routines. In addition, application or user-specific modules can be defined and added to an FPGA macro library set.

The ability of the user to optionally manipulate the compiler-generated logic is extremely important. This manipulation step must be available for those customers that want to get the last "drop" of algorithmic performance out of the hardware. The DFG generated by the high-level language (HLL) compiler

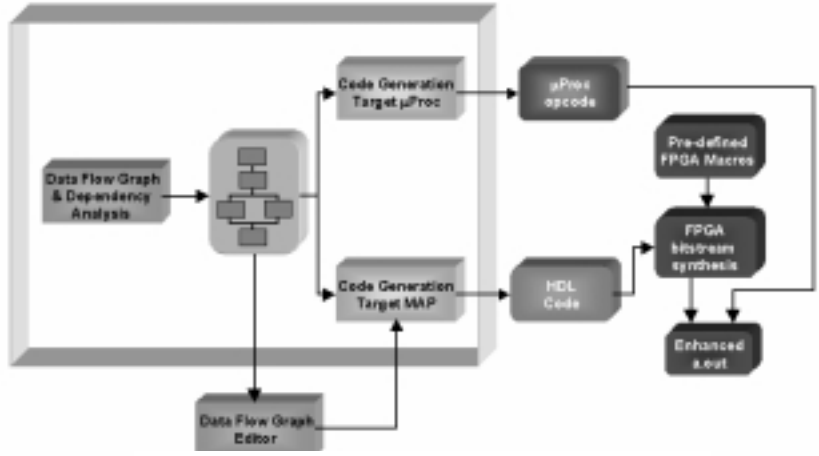


Figure 4 . Diagram of the SRC Compiler Process

will be an optional output. This graph will be modifiable in a DFG Editor to provide potentially higher levels of optimization.

The on-board memory of the MAP will be used like a software-controlled cache for a more conventional microprocessor. An added bonus is that this "cache" can have different user-defined caching strategies. The data access and compute strategies will maximize temporal locality with respect to the use of resident, on-board data and will overlap computation on this data with streaming in the "working set" of data for the next phase of the computation. We will generate both the Common Memory DMA instructions and the on-board data layout from the data-flow/dependence analysis.

Furthermore, the provision for rapid prototyping of the algorithm in MAP is a desirable tool for the optimization process. We are investigating potential tools for DFG translation for input to commercial packages; e.g., MATLAB/Simulink could provide easy access to prototyping and optimizations that a wide variety of people use today.

The available resources of the U_logic FPGA chips in the MAP are a finite resource. It will be a simple matter to compile user code into a final U_Logic configuration that exceeds the physical resources available in the chips. There will obviously have to be some feedback from various portions of the synthesis tools to the compilation system. The compilation system will also have to apply various heuristics in regards to precisely what portions of the user code should be targeted for the MAP in the first place, and what sort of transformations might be applied to the user code to allow for optimal performance. The feedback from the synthesis tools and the compilation heuristics can only be determined from direct experience. Initial versions of the compiler will necessarily omit this functionality. Adding this functionality based on the learned experience represents a significant portion of the overall effort required to create a mature compiler.

8. ALGORITHM STUDIES

The benefit of using reconfigurable computing has been shown in many domain areas[1][2][5]. Several algorithms will be reviewed to show the performance improvements over RISC and microprocessor-based systems. The definition of the hardware logic for these algorithms started with DFGs from our HLL compiler. The MAP compiler generates an equivalent DFG that it uses for the generation of HDL. Algorithm performance on MAP is from hardware simulation of the logic. In addition, optimization techniques and experiences will be discussed relative to compiler generated HDL.

The SRC-developed compiler is in a prototype stage of HDL generation. The compiler has demonstrated the ability to import pre-defined function unit modules and generate logic with correct results. Development on the compiler is proceeding in a directed, methodical manner. It does not yet fully compile the discussed cases. It can be expected that compilation of the following cases will be achieved by the time of the presentation of this paper at the SC01 conference.

8.1 Case 1: Convolution – Zero Phase Filter

This algorithm has been discussed briefly earlier in the paper. Let's examine the 32b floating-point convolution problem with a 64-point zero phase filter. The critical challenge for the compiler is to know how to take a simple convolution code and define a method for the scheduling of operations to take advantage of the opportunity to use a large number of multiply/adds relative to traditional processor implementation. The MAP can read up to 384b of data from on-board memory every clock.

The existing compiler's HDL implementation would utilize only a single input data element every clock. The performance of this level of optimization would be 2 Flops/clock. The MAP logic can consume the two 32b input data values every clock. If the logic can process only one element every clock, then the logic has to stall for a clock in order to consume the second input data value. Therefore, the processing rate can only create an output data element every other clock.

In order to optimize performance relative to the reading of the input data, hardware logic was developed that loaded the data appropriately into two sets of shift registers for the even and odd data elements. Two processing streams were defined to process the even and odd input data elements.

Figure 5 shows how the input data is loaded into the shift registers for the even/odd input data values. The shift registers are loaded up from the read of the first thirty-two 64b data values from memory.

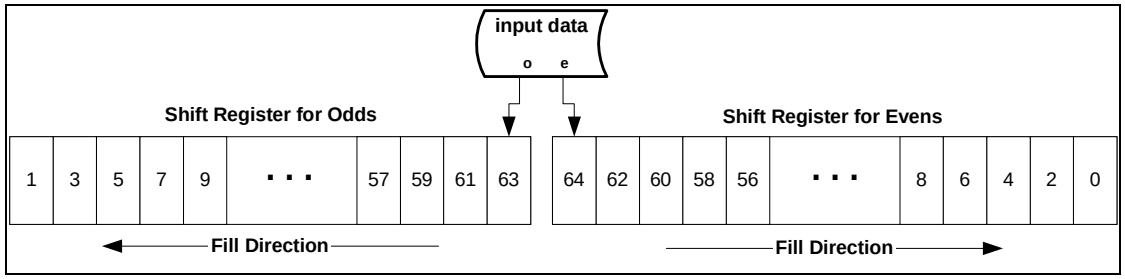


Figure 5. Data Streams Going into Shift Registers

After the shift registers are full, the multiplication of the data values with the filter coefficients can start. The process is pipelined so that the next input values will go into the shift registers. The generation of the output points is shown in Figure 6 as the contribution of the two shift registers.

This strategy is similar to loop unrolling that the compiler can often generate. We are in the process of developing heuristics for the compiler so that it can automatically generate this level of optimization strategy.

The computational performance on MAP for this convolution algorithm will be 256 flops/clock. This compares to 2 flops/clock on the Alpha EV6.

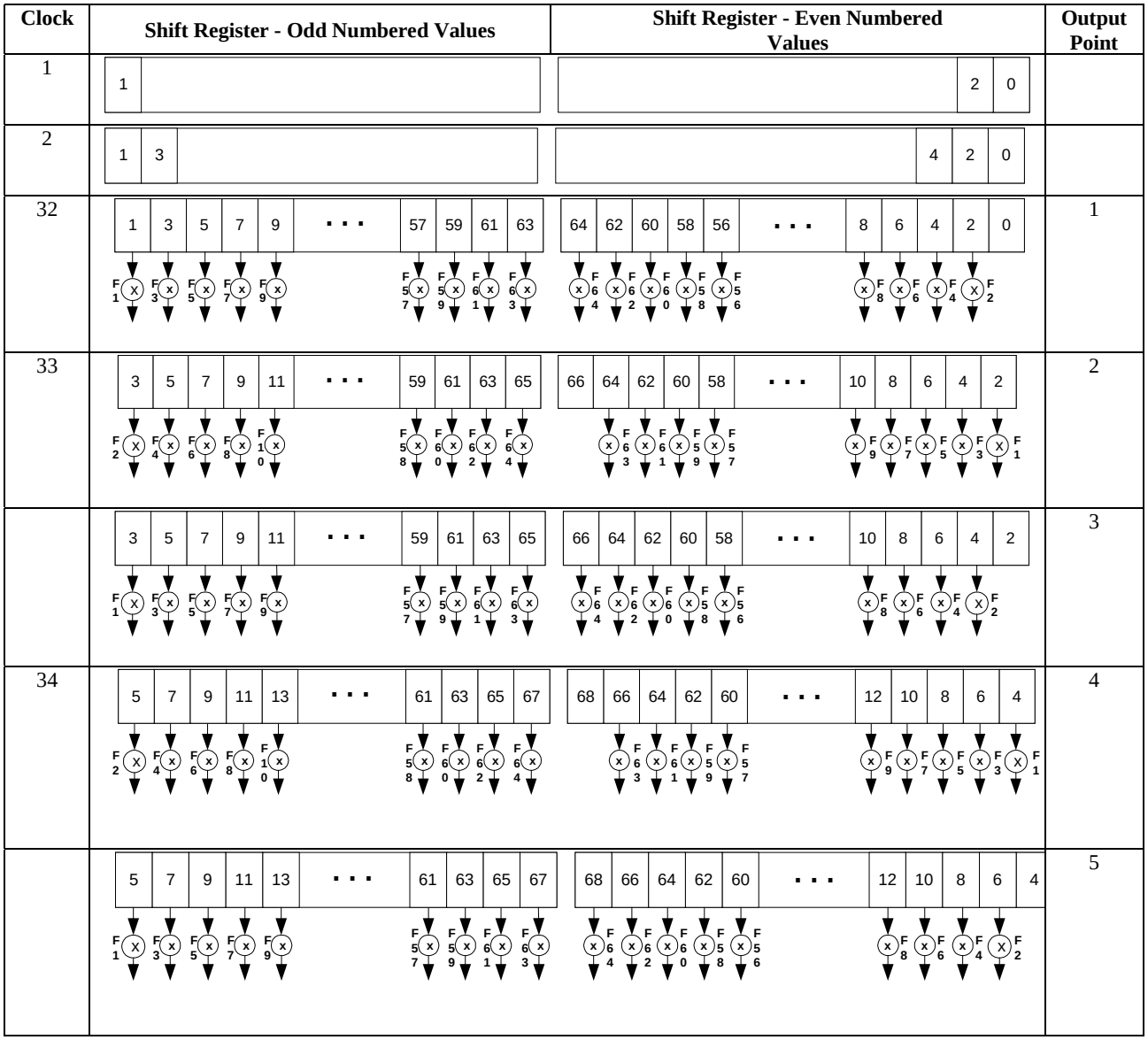


Figure 6. Processing Streams for Data

8.2 Case 2: Routine SCORE from MAXSEGS

MAXSEGS is a Smith-Waterman implementation of a genetic sequence alignment algorithm. The focal point of this analysis has been on the routine, SCORE. The routine, developed by Alex Ropelewski[4] of Pittsburgh Supercomputing Center, is a fully vectorized version of the Waterman-Eggert dynamic program algorithm. The computationally intensive portion of SCORE is shown in the following code segment.

```
do 200 k=1,ls1+ls2-1,1
  i=min(k,ls2)
  j=max(1,(k+1-ls2))

  do 210 index = start(k),end(k),ls2
     left = index-(ls2+1)
     diag = (index-(ls2+1))-1
     up   = index-1

     extgvi(i)= max(0,extgvi(i)+gap,
                    simils(left)+gap+newgap)
     extgvj(j)= max(0,extgvj(j)+gap,
                    simils(up)+gap+newgap)

     simils(index)=max(simils(diag)+
                       wt(s1(j),s2(i)),
                       extgvi(i),extgvj(j),0)
     i=i-1
     j=j+1
210  continue
200 continue
```

The goal for any algorithm in the MAP is to create a data flow that will maximize the use of input data and computational results within a single pass of the logic flow. The challenge with this code is to create a data flow that would maximize the use of 32-bit data coming from the input arrays EXTGVI, EXTGVJ and SIMILS. As explained in Ropelewski et al., the key lies in the diagonal access patterns of the algorithm.

Arrays are accessed across diagonals of the matrix in this fashion:

Modified version of Ropelewski schematic representation[4]

and values are computed in the following order:

(1,1)

then (1,2), (2,1)

then (1,3), (2,2), (3,1)

and continuing on until the value in (4,4) is computed.

Values of start and end shown in the code segment are the starting and ending diagonal values – (1,1) and (4,4), respectively.

The nested loops are unrolled by a factor of two to utilize the two 32-bit values read from on-board memory in the MAP. A macro can be made for the inner loop computation of EXTGVI, EXTGVJ and SIMILS. The logic flow for the computational macro is shown in Figure 7.

The hardware implementation of SCORE is pipelined (see Figure 8). The execution of the SIMILS macro can take a new value every clock and generate the values for SIMILS, EXTGVI and EXTGVJ every clock. The latency for a value in the macro is three clocks.

The layout of the data in on-board memory can dramatically affect the performance of the logic in MAP. The algorithm logic needs to read and write data to SIMILS with each computation of the inner loop. The first optimization approach was to replicate the data in SIMILS into four memory banks. This allows for reads and writes of four values with each read or write operation every clock. The approach is very effective in getting the necessary values for the computation. Because the overall algorithm logic is pipelined, there is a conflict of reading and writing SIMILS values to the memory banks at the same clock. In order to sustain the maximum processing rate, we had to alter the logic so that the read and write operations would occur at alternating clock cycles. The logic will process four updated values of SIMILS every two clocks.

A second approach for memory allocation for SIMILS was to use the Block RAM available within the FPGAs of MAP. There is 2.7 Mb available in 18b units. The logic used the feature that the RAM can be defined as dual-ported. This feature is exactly what we needed for the reading and writing of the SIMILS values. The computation can now generate four updated values of SIMILS every clock.

The performance of the original code on a microprocessor takes 55 clocks to process a single updated value for SIMILS. The MAP has a performance improvement of 13x over a 1700-MHz Pentium 4 microprocessor.

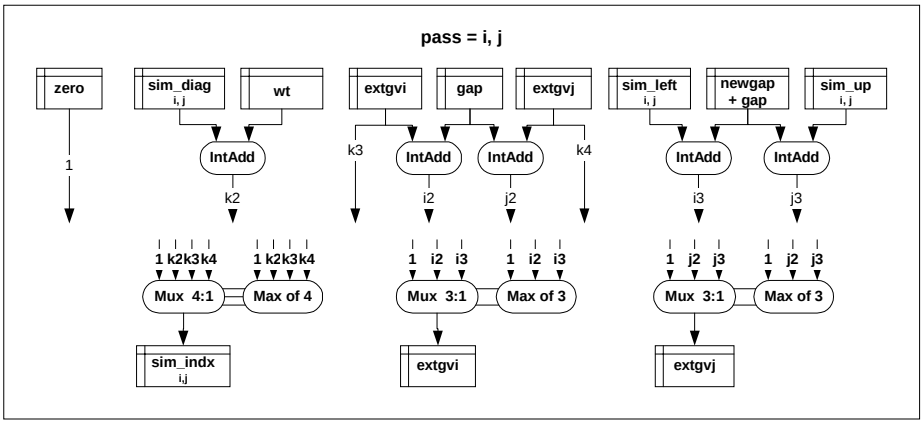


Figure 7. Compute Macro Logic

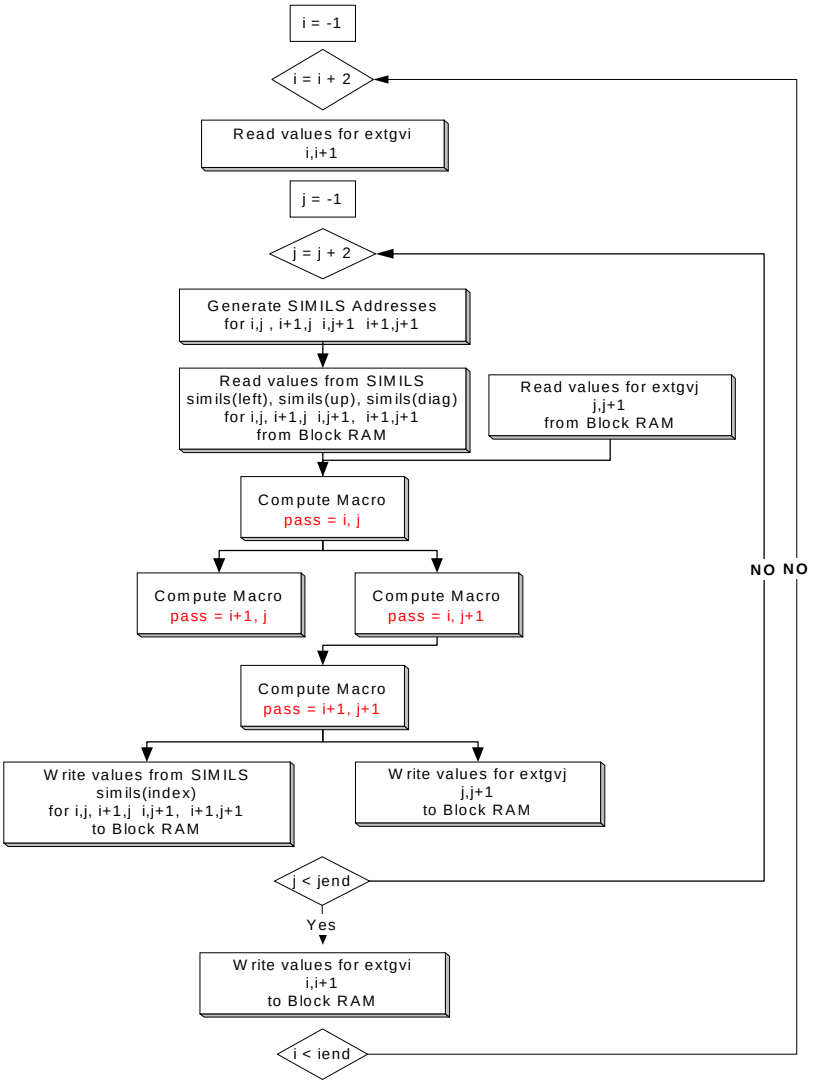


Figure 8. Data Flow Representation of SCORE

8.3 Case 3: Routine P7Viterbi from application HMMER

HMMER is a popular application for performing protein sequence analysis. The application profiles hidden Markov models (HMMs). There are several companies that have developed specialty ASICs that perform key computational algorithms in HMMER. We profiled several executions of HMMER (hmmcalibrate) and they pointed out that over 99.5% of the time was spent in the routine P7Viterbi. This case will focus on the optimization steps taken to move P7Viterbi into the MAP.

Figure 9 shows the sequence of tests made to determine the score for the alignment matching process. [3] The letters in the diagram correspond to the various "states" in the algorithm definition that follows shown in Figure 10.

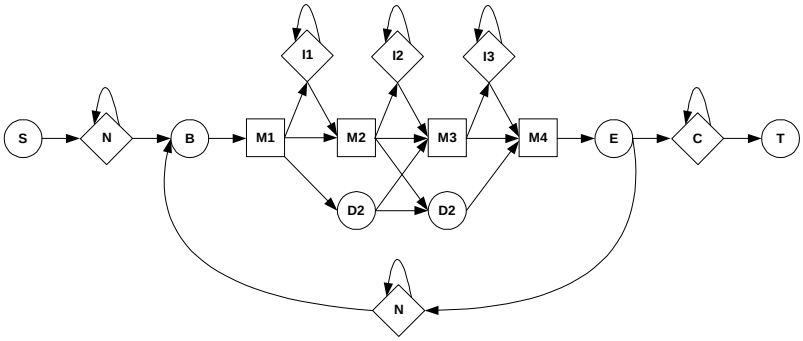


Figure 9. Diagram of the Algorithm State Comparisons

Core of Algorithm Code

```
for (i = 1; i <= L; i++) {
 mmx[i][0] = imx[i][0] = dmx[i][0] = -INFTY;

 for (k = 1; k <= M; k++) {
                                                        /* match state */
     mmx[i][k]  = -INFTY;
     if  ((sc = mmx[i-1][k-1] + tsc[k-1][TMM]) > mmx[i][k])     mmx[i][k] = sc;
     if  ((sc = imx[i-1][k-1] + tsc[k-1][TIM]) > mmx[i][k])     mmx[i][k] = sc;
     if  ((sc = xmx[i-1][XMB] + bsc[k]) > mmx[i][k])            mmx[i][k] = sc;
     if  ((sc = dmx[i-1][k-1] + tsc[k-1][TDM]) > mmx[i][k])     mmx[i][k] = sc;
     if  (msc[idsq][k] != -INFTY)                               mmx[i][k] += msc[idsq][k];
                                                  else          mmx[i][k] = -INFTY;

                                                        /* delete state */
     dmx[i][k] = -INFTY;
     if  ((sc = mmx[i][k-1] + tsc[k-1][TMD]) > dmx[i][k])       dmx[i][k] = sc;
     if  ((sc = dmx[i][k-1] + tsc[k-1][TDD]) > dmx[i][k])       dmx[i][k] = sc;

                                                        /* insert state */
     if (k < hmm->M) {
    imx[i][k] = -INFTY;
    if ((sc = mmx[i-1][k] + tsc[k][TMI]) > imx[i][k])           imx[i][k] = sc;
    if ((sc = imx[i-1][k] + tsc[k][TII]) > imx[i][k])           imx[i][k] = sc;
    if (isc[idsq][k] != -INFTY)                                 imx[i][k] += isc[idsq][k];
                                                  else          imx[i][k] = -INFTY;
     }
   }

                                                        /* N state */
     xmx[i][XMN] = -INFTY;
     if  ((sc = xmx[i-1][XMN] + xsc[XTN][LOOP]) > -INFTY)       xmx[i][XMN] = sc;

                                                        /* E state */
     xmx[i][XME] = -INFTY;
     for (k = 1; k <= hmm->M; k++)
        if  ((sc =  mmx[i][k] + esc[k]) > xmx[i][XME])          xmx[i][XME] = sc;

                                                        /* J state */
     xmx[i][XMJ] = -INFTY;
     if  ((sc = xmx[i-1][XMJ] + xsc[XTJ][LP]) > -INFTY)         xmx[i][XMJ] = sc;
     if ((sc = xmx[i][XME] + xsc[XTE][LP]) > xmx[i][XMJ])       xmx[i][XMJ] = sc;

                                                        /* B state */
     xmx[i][XMB] = -INFTY;
     if  ((sc = xmx[i][XMN] + xsc[XTN][MV]) > -INFTY)           xmx[i][XMB] = sc;
     if  ((sc = xmx[i][XMJ] + xsc[XTJ][MV]) > xmx[i][XMB]       xmx[i][XMB] = sc;

                                                        /* C state */
     xmx[i][XMC] = -INFTY;
     if  ((sc = xmx[i-1][XMC] + xsc[XTC][LP]) > -INFTY)         xmx[i][XMC] = sc;
     if  ((sc = xmx[i][XME] + xsc[XTE][MV]) > xmx[i][XMC])      xmx[i][XMC] = sc;
 }
                                                        /* T state  */
 sc = xmx[L][XMC] + xsc[XTC][MV];

}
```

Generalized Flow Graph

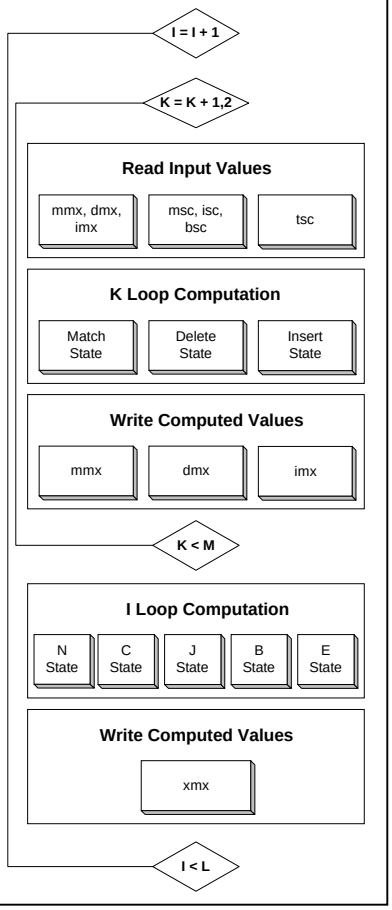


Figure 10. HMMER Algorithm Definition

The DFG produced by our standard compiler was the starting point for the logic definition. Unfortunately, this starting point did not take advantage of the MAP's ability to shift the logic for the Insert and Delete States to be concurrent with the Match State. Therefore, we manually performed this level of optimization. These are optimizations that we see the compiler eventually being able to do.

The data type for all variables in the algorithm is 32b integers. The first approach to the definition of the hardware logic was to take advantage of data parallelism through reading in two 32b values from on-board memory every clock for the computational inputs xmx, imx, dmx, msc, bsc, isc, and tsc. The algorithm uses previously computed points in vectors mmx and imx. The inner loop will be unrolled by two. The logic has the potential of easily using the two values read in per clock. The dependency upon previous computed values meant that we could not pipeline the algorithm. Therefore, the performance of the algorithm is gated by the performance of the logic for the inner loop. Upon further investigation into the options for the logic, we determined that we could take the cascaded sets of greater-than tests and put them into a single MAX and MUX construct. This allows us to reduce the latency through this portion logic by a factor of two.

Figure 11 shows the two sets of logic.

The Delete and Insert States also use a similar form of logic. The performance of the algorithm on the MAP was 10 clocks for each pass through the inner loop and 4 clocks through each pass the logic at the end of the outer loop.

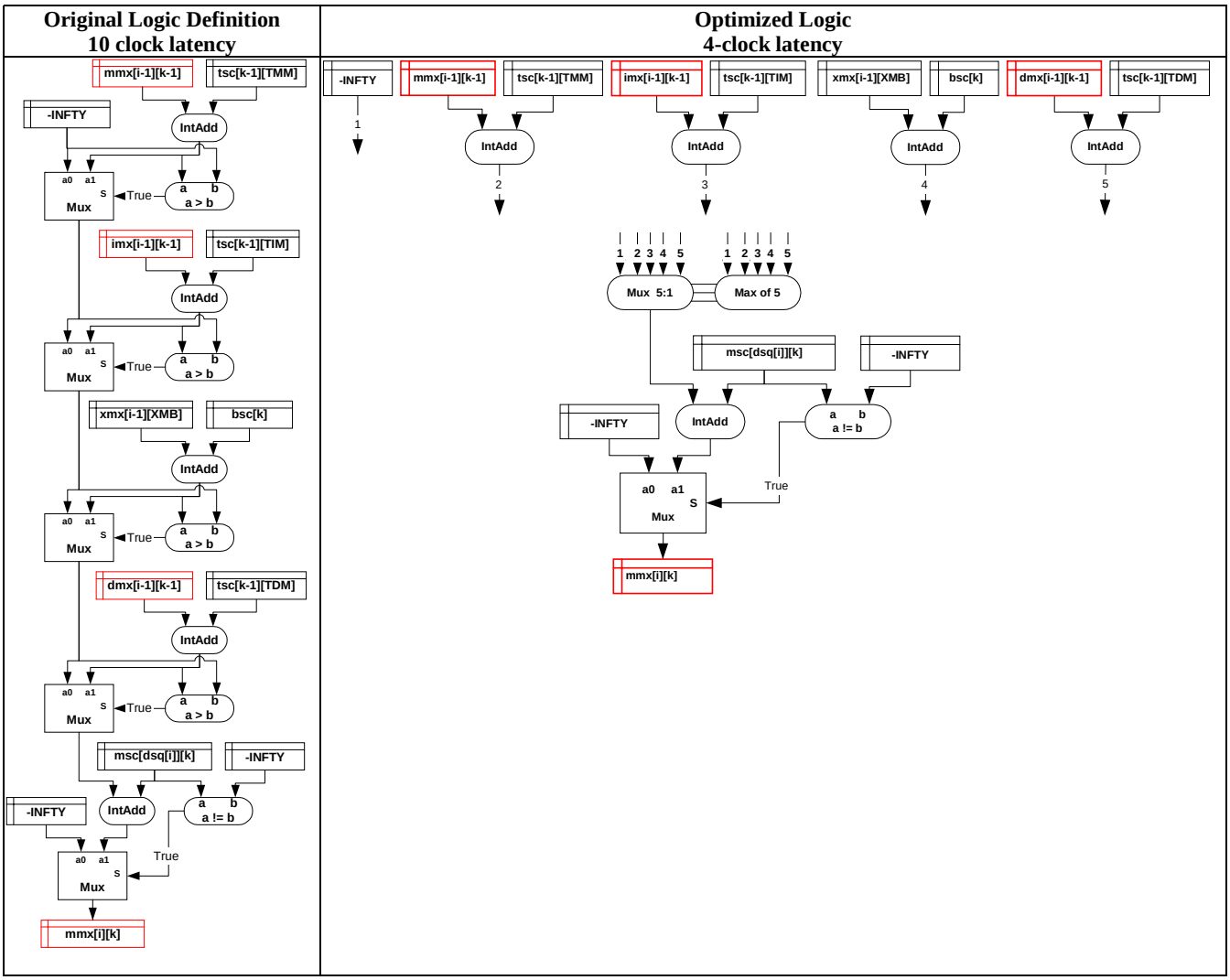


Figure 11. Logic Definition for Match State for the kth Value

Now that we have the basic logic definition of the algorithm, we need to investigate the "memory" access of the mmx, dmx, imx variables. A simple analysis shows that the backward looking aspect of the algorithm can use values in the unrolled inner loop and the unrolled outer loop. This pattern is shown in Figure 12. Note that the I+1, I+2 Loops are shifted in time in order to consume values computed previously.

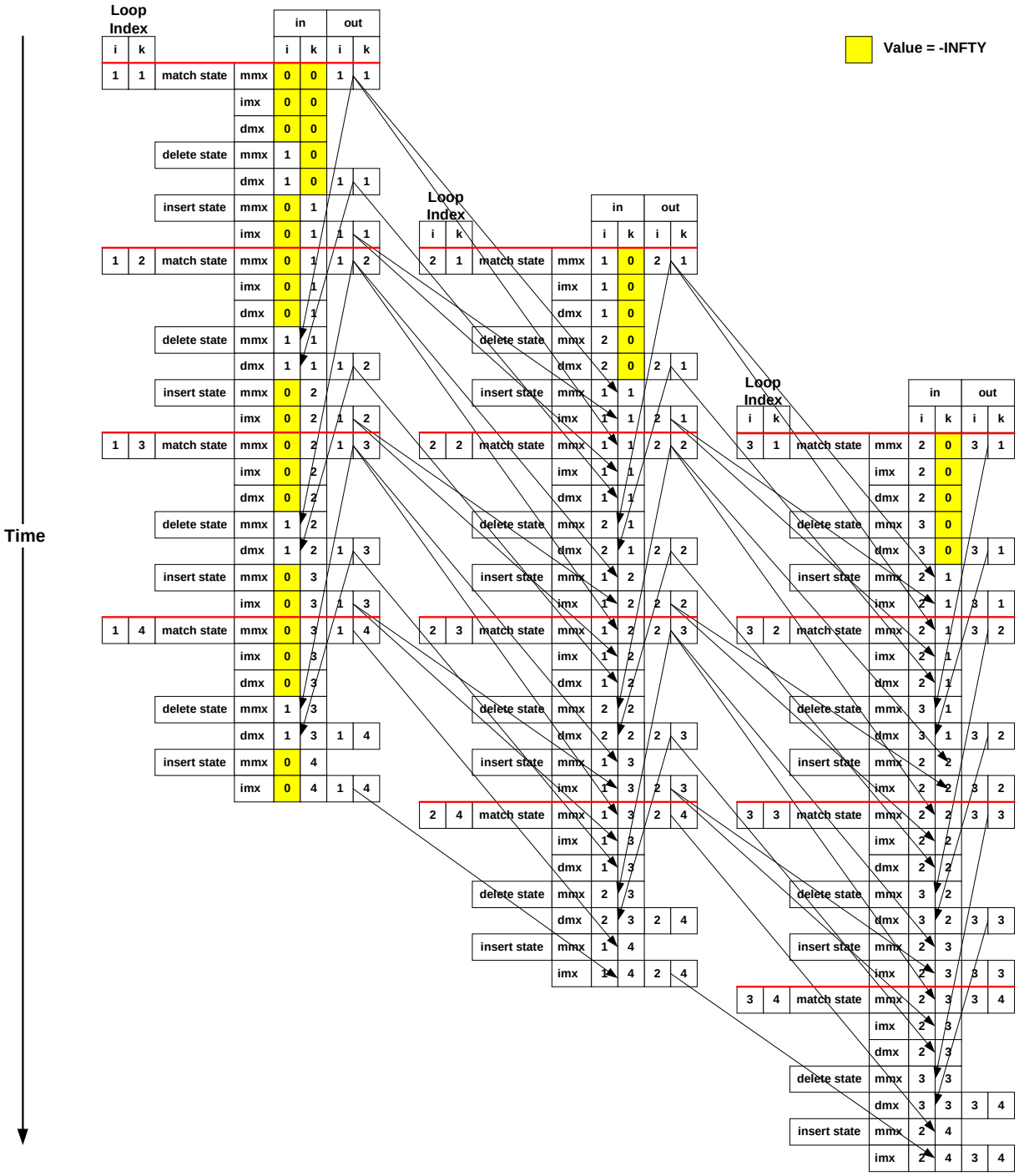


Figure 12. Memory Access Patterns in the Inner and Outer Loops

The memory access pattern analysis shows that additional execution parallelism can be obtained through unrolling of the outer loop. The memory access pattern is similar to the inner loop problem. The algorithm uses previous computed vectors in I-1. The implementation used a set of temporary values stored in Registers for the values of xmx[I-1], dmx[I-1] and imx[I-1]. Analysis of the problem showed that we need to delay loop I+1 by at least 6 clocks. This would provide the [I][K]th value required in the compute of the I+1 loop for the Match, Delete and Insert States. In addition, the mmx, dmx and imx values do not have to be stored in memory because the values are completely consumed by the logic. By not needing to write these values to memory the latency of the inner loop decreases to 9 clocks and removes any need to consider potential bank conflicts to on-board memory.

We have the ability to completely unroll the outer loop. However, for large counts of L, outer loop count, we could easily exhaust the logic building blocks, CLBs or Slices, in the FPGA. The final definition of the logic and place and route of the FPGA determined that we could unroll the outer loop by twenty. We use a common guideline to not use more that 80% of the available resources in order to get an efficient place-and-routed algorithm.

The performance of the algorithm was compared with that of a 1700-MHz Pentium 4 microprocessor. The MAP functions units and logic were operating at 100 MHz. A measurement of the average amount of time spent in the inner loop was made as the number of clocks to do the nested loops divided by the product of the loop counts (L * M). The average was taken over 5000 runs of the routine. The following table shows three data cases and the type of algorithm speedups that were achieved. Another feature of the MAP is that it has two FPGAs available for computation. The algorithm was unrolled across both FPGAs and the additional speedup is shown in Table 3.

Given the percentage of time spent in the routine, 99.5%, the application speedup achieved was over 50x. The price-performance of the MAP exceeded that of the microprocessor on this problem by a factor greater than 12.5x.

Table 3. Speedup of Algorithm with MAP

Problem Size (L, M)	**Avg. Time spent in inner loop per iteration (Clocks)**		**Parallelism (#Loops, # FGPAs)**	**Algorithm Speed-up**
	Microprocessor	**MAP**		
355, 162	175	4.55	1 / 1	2.26
			20/ 1	35
			40/ 2	58
348, 251	185	4.54	1 / 1	2.39
			20 / 1	39
			40 / 2	65
344, 462	192	4.54	1 / 1	2.49
			20 / 1	42
			40 / 2	72

9. EXTENDING COMPILER HEURISTICS FOR FPGAS

Memory access promises to be the most challenging area for optimal expression of an algorithm on an FPGA. Fortunately there is a rich history of processor and memory architectures that can be gleaned for various approaches. In this regard, the ability to reconfigure is the best aspect of the FPGA, since completely different schemes can be used by the compiler in different contexts. There is also a well-understood history of compiler transformations that can be applied to user code in order to take advantage of various hardware specifics. The key to optimal compiler-generated performance for the FPGA would seem to be a balanced set of applied heuristics at the memory interface and code transformation levels, all the while operating within the bounds of the FPGA resource limits. The FPGA brings an interesting slant to compilation, as constraints that are fixed in a traditional processor are now merely another set of variables.

10. CONCLUSIONS

The potential of getting orders of magnitude speedups from reconfigurable computing has been shown in a variety of application domain areas at the component or single board level. The question of applicability to a generalized set of computationally intensive scientific algorithms is currently being addressed by SRC Computers.

The requirement to get broad-based acceptance of reconfigurable computing will be based on the ability to demonstrate the following:

- Price/Performance improvement over existing processors
- Ease of achieving this improvement

SRC recognizes that it is critical to have compiler technology and tools available to programmers that will minimize the effort to get the desired performance from MAP.

11. SUMMARY

This paper has shown that there are major steps taking place to make the first level compiler optimizations for FPGA technology and to provide adequate performance gains. There will still be a place for handcrafted hardware logic for time critical algorithms. However, the optimization capabilities of the compiler will evolve just as it has for vector and cache-based computing systems.

The promise of increased performance with MAP has been established; it will only improve with new generations of FPGA chips.

12. REFERENCES

[1] Buell, Duncan; Arnold, Jeffrey; and Kleinfelder, Walter. Splash2: FPGAs in a Custom Computing Machine. IEEE Computer Society Press, 10662 Los Vasqueros Circle, PO Box 3014, Los Alamitos, CA 90720-1264, (1996).

[2] DeHon, Andre. Comparing Computing Machines. In Configurable Computing: Technology and Applications, volume 3526 of Proceedings of SPIE. SPIE, (November 1998).

[3] Eddy, Sean. HMMER User's Guide http://www.psc.edu/general/software/packages/hmmer/manual/main.html. (April, 1995) 18.

[4] Ropelewski, Alexander J.; Nicholas Jr., Hugh B.; and Deerfield II, David W. The Journal of Supercomputing, 11:3, (1997) 237-253.

[5] Scott, Stephen D.; Seth, Sharad; and Samal, Ashok.. A synthesizable VHDL coding of a genetic algorithm. Technical Report UNL-CSE-97-009, University of Nebraska-Lincoln, (1997). ga97aSDScott.