Flowpaths: Compiling Stack-Based IR to Hardware

Darrin M. Hanna and Richard E. Haskell
CSE Dept., Oakland University, Rochester, MI 48309
dmhanna@oakland.edu, haskell@oakland.edu

Abstract

*The performance of software executed on a microprocessor is adversely affected by the basic fetch-execute cycle. A further performance penalty results from the load-execute-store paradigm associated with the use of local variables in most high-level languages. Implementing the software algorithm directly in hardware such as on an FPGA can alleviate these performance penalties. Such implementations are normally developed in a hardware description language such as VHDL or Verilog. More recently, several methods for using C as a hardware description language and for compiling C programs to hardware have been researched. Several software-programming languages compile to an intermediate representation (IR) that is stack based such as Java to Java bytecodes. Forth is a programming language that minimizes the use of local variables by exchanging the load-execute-store paradigm for stack manipulation instructions. This paper introduces a new systems architecture for FPGAs, called **flowpaths**, which can implement Java bytecodes or software programs written in Forth directly in an FPGA without the need for a microprocessor core. In the flowpath implementation of Forth programs all stack manipulation instructions are represented as simple wire connections that take zero time to execute. In the flowpath implementation of Java bytecodes the normal load-execute-store paradigm is represented as a single sequential operation and stack-manipulation operations become combinational thus executing faster. This paper compares the use of flowpaths in an FPGA generated from Java bytecodes and a high-level Forth program for the Sieve of Eratosthenes with C, Java, and Forth executed on microprocessors and microprocessor cores on FPGAs. The results show that flowpaths perform within a factor of 2 of a minimal hand-crafted direct hardware implementation of the Sieve and orders of magnitude better than compiling the program to a microprocessor.*

1. Introduction

FPGAs promise the flexibility of software running on a microprocessor together with increased performance in terms of throughput. However, this flexibility usually requires one to be a hardware designer using a hardware description language such as VHDL or Verilog. Several methods for using a language like C as a hardware description language or to generate hardware from software have been researched over the past decade and a half [1-7], but in most cases the designer is still thinking in hardware terms and solutions like these have seen little commercial interest [8, 9].

Several groups have explored developing unified hardware development processes and tools including formal specification languages such as LUSTRE, KRONOS, POLIS, SIGNAL, REACTIVE C, Synchronous Language (SL), LOTOS, Handel-C, SystemC and SDL [4, 10-13]. Some languages such as POLIS and LUSTRE are supplements to hardware description languages and some languages such as Handel-C are intended to develop hardware directly from software.

It would be desirable in terms of development time to take a traditional software program and compile it directly to hardware (FPGA or ASIC) in the same way as a software program is compiled to a particular microprocessor. One way to do this is to use a microprocessor core within the FPGA [14-16]. For the case of a microprocessor core the program itself must be stored in RAM or ROM. If this RAM or ROM is part of the FPGA itself, this will limit the size of programs that can be run on the microprocessor core. If the program is stored in external RAM or ROM, then the FPGA microprocessor core is no different than a normal microprocessor.

There is an inherent performance penalty of using any microprocessor. A microprocessor contains a general-purpose datapath and control unit that executes a sequence of instructions representing a software program that is stored in memory. A dedicated datapath and control unit that executes only a particular software program or algorithm can be more efficient in terms of both speed and area. The goal is therefore to generate this dedicated datapath and control unit automatically from a standard software program. This datapath and control unit can be described by a hardware description language such as VHDL or Verilog to synthesize to an FPGA. Note that this is not the same as writing the software program directly in Verilog or VHDL because although the program might be able to be simulated it would not,

in general, be synthesizable. For example, a while loop that depends on unknown inputs cannot be synthesized in VHDL or Verilog.

In this paper we present a new architecture called *flowpaths* [17] that has proved to be useful in automatically converting software programs to hardware. It does this by representing the software program as a stack-based algorithm that leads to an efficient datapath and control unit implementation that can then be synthesized using VHDL or Verilog.

Several software-programming languages compile to an intermediate representation (IR) that is stack based such as Java to Java bytecodes. For Java, the Java Virtual Machine (JVM) is stack-based. Each thread has a JVM stack that stores frames created each time a method is invoked. The frame contains an operand stack, an array of local variables, and a reference to the runtime constant pool of the current method's class [18]. To execute instructions, variables are loaded onto the stack, the instruction is executed, and finally the answer is stored back in the variable. In this fashion, a combination of stack and memory are used in this "stack-based" JVM.

This *load-execute-store* feature of Java bytecodes maintains the normal microprocessor paradigm. Indeed microprocessors have been developed that execute these Java bytecodes directly [21]. However, the performance of these microprocessors still suffers from the excessive use of local variables in the original software program. A programming language that inherently minimizes the use of local variables is Forth. A Forth program consists of a sequence of words, each of which is a sequence of other Forth words. Each Forth word expects input data on a parameter stack and leaves the result of executing the word on the same parameter stack. This result then becomes the input to the next Forth word in the program. In this way Forth programs typically use far fewer local variables than more traditional languages. The tradeoff is that Forth requires stack manipulation words such as DUP and SWAP to manipulate elements on the stack.

In this paper we will show how Java bytecodes and high-level Forth programs can be converted to *flowpaths*, a datapath and corresponding control unit that can be synthesized directly to hardware using VHDL or Verilog. The Forth stack manipulation words such as DUP, SWAP, and ROT are represented in flowpaths as simple wire connections that take zero time to execute. In this sense the stack has completely disappeared once the software program is converted to flowpaths, which therefore have the advantage of avoiding the load-execute-store penalty without incurring the stack manipulation penalty of traditional Forth programs. In a similar way the load-execute-store construct in Java bytecodes is represented in flowpaths as a single sequential operation. More details for converting all Forth words to flowpaths, implementations, and results can be found in Hanna (2003) [17]. We have implemented a basic compiler using most of these rules.

Flowpaths have a number of other potential advantages. Once the software program is compiled to flowpaths the program itself is no longer needed and therefore it is not necessary to store any machine code in RAM or ROM. The resulting flowpath typically takes up less FPGA resources than the corresponding microprocessor core and program ROM. In fact, in one example we have implemented, the size of the flowpath is less than that of only the corresponding program ROM [17]. A final advantage of flowpaths is that they lend themselves to reconfigurable computing. Flowpaths will be shown to be state machines and executable structures in which only a subset of all states and structures need to be implemented at one time. This could allow large software programs to be executed on a relatively small FPGA regardless of how unique or repetitive the instructions are in the program.

The input to the flowpath generator is either a standard Forth program or standard Java bytecodes. This means that high-level software programs written in either Java or Forth can be directly compiled to hardware. The hardware can be implemented in an FPGA resulting in significant improvement in performance compared with executing the same software using a microprocessor.

Section 2 describes flowpath basics. Section 3 uses a GCD algorithm to show how flowpaths can be created from both Java bytecodes and Forth. In Section 4 the example of the Sieve of Eratosthenes algorithm will be used to compare the performance of compiling the same software program in six different environments: 1) a Java program compiled to the Ajile AJ-80 processor that executes Java bytecodes directly; 2) a C program compiled to tight 68HC12 assembly code; 3) a Forth program compiled to 68HC12 assembly code; 4) a Forth program compiled to the FC16 Forth core in a Xilinx Spartan IIE FPGA; 5) a Forth program and Java bytecodes compiled to VHDL flowpaths in a Xilinx Spartan IIE FPGA; and 6) a hand-crafted direct VHDL implementation of the Sieve of Eratosthenes algorithm. The results will show that a straight conversion of a Forth program or Java bytecodes to flowpaths in an FPGA can produce performance within a factor of 2 of a hand-crafted direct hardware implementation (considered to be minimal) and orders of magnitude better than compiling the program to a microprocessor. The details of the flowpath implementation of the Sieve of Eratosthenes algorithm are given in Section 5.

Current methods for compiling high-level software programs to hardware yield large circuits for large programs. These large circuits require large, expensive FPGAs that inhibit their use in mass production forcing designers to create an ASIC or use a traditional microprocessor with an FPGA as a coprocessor. In order to design ASICs, engineers must use a language that is compatible with verification and foundry tools such as Verilog or VHDL to describe hardware instead of taking advantage of a high-level language such as C, Java, or Forth. Flowpath subsets can reuse FPGA resources using partial, dynamic reconfiguration while a subset of the algorithm is executing as discussed in Section 6. This would allow for larger programs to be compiled to large flowpaths that can be executed on smaller, less expensive FPGA targets. This potential is demonstrated by experiments with a similar implementation of Euclid's GCD algorithm using partial, dynamic reconfiguration on a Xilinx Virtex 1000 FPGA.

2. Flowpaths

Listing 1 Forth code and Java code for *squared* and *cubed*

```
: squared ( n -- n^2 )
     DUP * ;
: cubed ( n -- n^3 )
     DUP squared * ;

public int squared(int a)
{ return(a*a); }
public int cubed(int a)
{ return(a*squared(a)); }
```

Generating a flowpath from Forth code can be demonstrated with a simple example. Consider the Forth code and equivalent Java methods in Listing 1. The *cubed* function in both Forth and Java was written in terms of the squared method to illustrate function calls. The *squared* function requires a word of data on top of the data stack (of a stack-based processor). The word on the top of the stack is duplicated and the top of the stack and next element in the stack are multiplied (using the * instruction). After this function has executed, the resulting square of the number is left on the top of the stack. Similarly, the *cubed* function duplicates the value on the top of the stack, squares it using the *squared* function and multiplies the square of the number times the number itself leaving the resulting cube on the top of the data stack. The Java bytecodes for the *squared* and *cubed* methods are shown in Listing 2.

Listing 2 Java bytecodes for *squared* and *cubed*

```
public static int square(int);
Code:
0:   iload_0
1:   iload_0
2:   imul
3:   ireturn
public static int cubed(int);
Code:
0:   iload_0
1:   iload_0
2:   invokestatic#38;   //Method square(I)I
5:   imul
6:   ireturn
```

This stack-based IR, Java bytecodes, behaves almost identically to the Forth example except that there are local variables. These local variables are pushed onto the operand stack where instructions are executed leaving the result on top of the stack. After the instruction has been executed the method returns since the return value is already on top of the stack. More often, the result is stored back to a variable or successive operations are performed.

The flowpath generated for this code contains no data (or operand) stack. In fact, the data are passed through and generated at each OP accordingly. The data are ordered, that is, the concepts of the top-most data, second-most data, and so forth, are conserved. Figure 1 shows the top-level block diagram of the flowpath that will square the input data or "top of the stack".

The *squared* OP shown in Figure 1 is a single clock-cycle OP that is no more than a simple registered multiplication, the same as the Forth and Java code for *squared*. The squared OP requires one input and will produce one output. In addition to the required input, any data that exist during program execution at the point when this OP is executed are registered through the OP. The *DUP* instruction is compiled to a "copy" of the data bus and does not require any additional clock cycles.

Incidentally, the flowpath for *squared* will execute in less clock cycles than the Forth or Java function. Figure 2 shows the flowpath inside of the *squared* OP. The junction labeled with an "A" in Figure 2 implements the two *iload_0* bytecode operations and equivalently the *DUP* instruction in Forth. This is necessary since the standard-OP *multiply/imul* requires two inputs and yields one output. The *squared* OP is labeled with a "B".

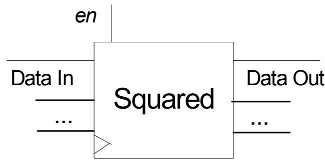

Figure 1 The top-level flowpath for the Forth/Java function *squared*

As shown in Figure 2, the constructs are the same for Java bytecodes as they are for Forth. Similarly, the cubed OP implements the Forth cubed function using the standard multiplication OP and employs the same technique for compiling the *DUP* instruction. Figure 3 shows the inside of the cubed OP. Instead of showing two pictures, one for Forth and one for Java bytecodes, a single generic picture is shown. The *multiply* OP could be either the Forth * OP or Java's *imul* OP, interchangeably.

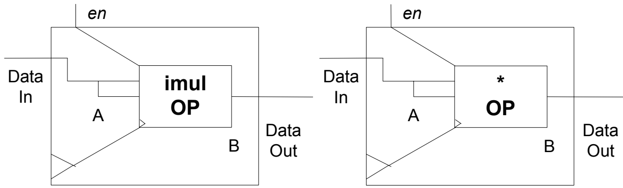

Figure 2 Inside the *squared* OP using the Java *imul* on the left and Forth * on the right

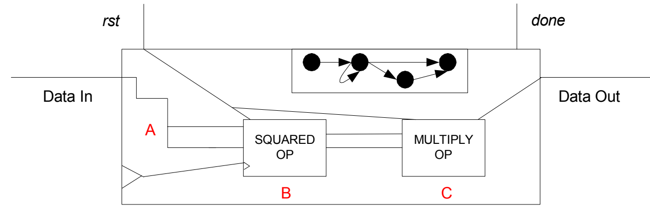

Figure 3 Inside the *cubed* OP

The junction labeled with an "A" depicts the implementation of the *DUP* instruction. "B" is the *squared* OP and "C" is the *multiply* OP. Instead of an *en* enable input, the *cubed* OP has *rst* and *done* signals. The *rst* signal is an input that when asserted low enables the OP. *Done* is low until the OP has finished execution, then it is high. Since this is a multi-cycle instruction, one for the *squared* and one for the *multiply*, a control state machine synchronizes the execution of the appropriate OP and would synchronize appropriate multiplexers to control inputs and outputs if it were necessary. Also notice that there are two inputs entering the *squared* OP. Since the *DUP* instruction duplicates the "top of the stack" creating two data elements, the "top" element is sent into the *squared* OP as the single input required. The second element is passed through the OP so that the flowpath carries all of the active data all the time as illustrated in Figure 1. Figure 4 shows the *cubed* OP and Figure 5 shows the state diagram of the state machine that controls the cubed OP. This datapath and control state machine together makes a flowpath.

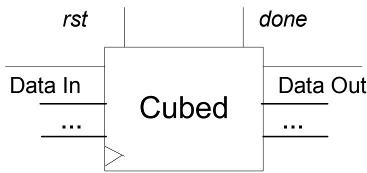

Figure 4 The *cubed* OP

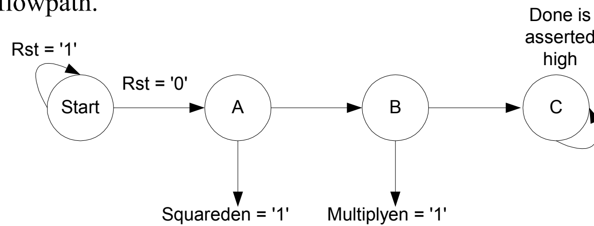

Figure 5 The state-transition diagram for δ for the *cubed* OP

Both standard and custom OPs can be used in functions. An instance of a function or standard OP is produced to execute a particular portion of code. Additionally, the OPs are autonomous. Data are sent to them and through them, the OP is enabled (not reset) and the parent OP waits until the OP signals that it has finished executing via the *done* signal.

3. Stack-based IRs and Flowpaths

Consider a simple example of computing the greatest common divisor of two numbers. There are several Euclidean algorithms for computing two number's greatest common divisor. An iterative version of Euclid's GCD algorithm is shown in Listing 3 as a method in Java. Listing 4 shows the Java bytecodes for this method.

Since this method would be implemented in a frame with an array of local variables (in this case x being 1 and y being 2), *iload_1* and *iload_2* are used to push these variables onto the operand stack. In general, algorithms can be considered as a set of operations, data, and a controller to determine execution order. Others have described methods to convert such programs into special-purpose processors where each variable is implemented as a register [19]. Along these lines, consider the hardware that would be generated from these bytecodes. This hardware is shown in Figure 6. Each block is a digital component that performs the function as labeled. The inputs and outputs are data busses. Each of the components is executed according to the state diagram shown in Figure 7.

Listing 3 Euclid's GCD algorithm in Java

```
1:  int gcd (int x, int y)
2:  {
3:    //Euclid's GCD
4:    while (x != y)
5:    {
6:      if (x<y)
7:         y -= x;
8:      else
9:         x -= y;
10:   }
11:   return x;  }
```

Listing 4 Java Bytecodes for Euclid's GCD

```
0:  goto 19

3:  iload_1       //Push x
4:  iload_2       //Push y
5:  if_icmpge 15  //if x>=y goto 15

8:  iload_2       //Push y
9:  iload_1       //Push x
10: isub            //y-x
11: istore_2        //Store new y
12: goto 19

15: iload_1       //Push x
16: iload_2       //Push y
17: isub            //x-y
18: istore_1        //Store new x

19: iload_1  //Push x
20: iload_2  //Push y
21: if_icmpne 3     //if x!=y goto 3

24: iload_1  //Push x
25: ireturn
```

State B enables the *if_icmpge* (annotated as line 5 which matches Listing 4), the result of which will transition the controller to either state C or F. In the case of state C, the *isub (10)* will be enabled on the next clock cycle. Execution down this path continues with state D which enables the y-register's load completing the *y-=x* computation. Otherwise, *isub (17)* will execute according to F and similarly proceeds to G, which stores the result of *x-y* to *x* by enabling the x-register's load. In either case, execution continues to state H. State H executes the *if_icmpne* (21) and execution continues back to state B if the outcome is true or state I if false. State I loads the return value on the stack and state J returns.

A simple optimization can be made by substituting the two *isub* components with a single *isub* component and a multiplexer as shown in Figure 6.

Although this is a possible approach to generating hardware directly from software applied to an intermediate representation that is stack-based with a local variable array, it is important to note a general problem which will become very large, quickly. Since the Java bytecode representation uses local variables that are implemented as registers, the inputs to the operations must be routed from the registers or memory storing the variables and the results must be stored back into the register or memory. This presents a significant routing problem.

Using a stack-based intermediate representation such as Forth, where local variables are stored on a stack and operations are done to the top elements of the stack, with results left on top of the stack, data flow can be streamlined and the problem eliminated. By placing local variables on a stack, stack manipulation words will frequently be used to move elements around before and after executing operations, but result-loading instructions can be eliminated. This presents a significant advantage while using flowpaths since stack manipulation words translate into zero-clock-cycle routing.

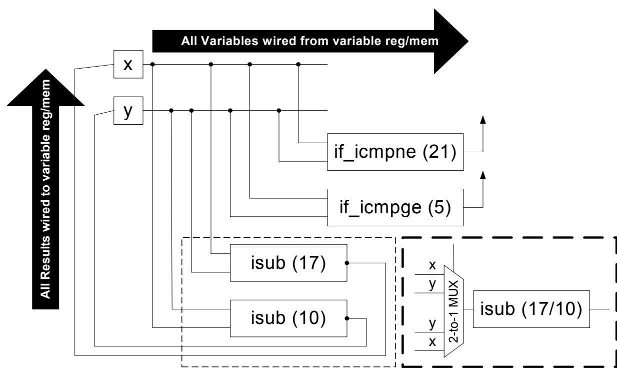

Figure 6 Hardware to implement the GCD Java bytecodes

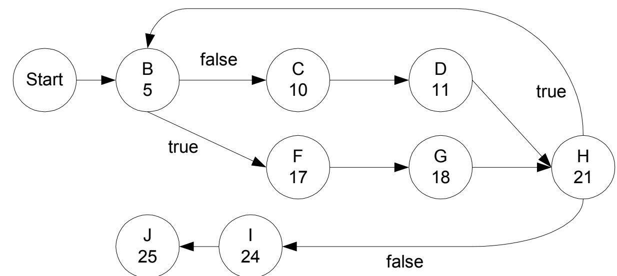

Figure 7 A state diagram to control the execution path of the hardware in Figure 6

In Java, local variables are kept in a local variable array resulting in numbered variables. These variables are loaded into the operand stack and loaded with successive results. Since all local variables are 'passed through' registers in each OP, successive *iload_*x_i instructions are translated into copies of the wires corresponding to the x_i's routed in the proper order for input into the next OP.

One can implement a stack-based IR with a local variable array in the same manner and achieve several orders of magnitude faster performance without the routing problem described when using the variable-to-register methods for special-purpose processors. Flowpaths significantly reduce the space required and increase the maximum execution speed in many cases. Standard OPs that are pre-designed with generic bus-width and with a variable number of 'pass-through' registers form a flowpath standard library. These standard OPs perform basic instructions, such as *AND*, *PLUS*, and *GT* (greater than) for Forth and *imul, iadd,* and *if_icmpne* for Java. Additionally, custom OPs can be designed at a hardware or software level and stored in this library where they will be considered standard words by the compiler. This allows one to invoke methods that aren't developed in Java but are OPs that follow the standard *reset-done* flowpath paradigm. Interfaces must capture specific timing information and are often more easily designed using a hardware description language such as VHDL. Therefore, Custom OPs can be OPs written in VHDL, Forth, or Java.

The flowpath to compute the greatest common divisor requires three OPs: An equality detector (OPEq), a magnitude comparator (OPLt), and a subtraction OP (OPMinus). As usual, a path using multiplexers and demultiplexers connects these OPs and a simple state-machine controller controls the entire circuit. Each of these OPs requires only one clock-cycle to execute. The GCD algorithm in Listing 3 can easily be implemented in Forth as shown in Listing 5.

Figure 8 shows the GCD algorithm in Listing 5 implemented as an optimized flowpath. The formal algorithm for compiling Forth code to flowpaths, an initial unoptimized GCD flowpath, and optimization techniques are detailed in Hanna (2003) [17]. The data are 32-bit integers. Characteristic of a multi-cycle OP, the GCD OP itself has a *reset* signal and *done* signal.

Listing 5 A Forth implementation of a GCD algorithm

```
: GCD  ( X  Y  --  GCD)
BEGIN                   \ X Y
   2DUP = NOT           \ X Y FLAG
WHILE                   \ X Y
   2DUP U<
   IF                   \ X    Y
      OVER -            \ X    Y-X
   ELSE                   \ X     Y
      TUCK - SWAP       \ X-Y  Y
   THEN
REPEAT
DROP ;                      \ X (GCD)
```

When the *reset* signal is brought low, the OP begins execution and when execution has completed, the *done* signal is asserted high for one clock cycle. Initially, the entering data, *x* and *y*, are used by OPEq. The output of this OP is the same as most of the comparison operators, the data are registered through the operation and a *flag* is outputted for the Boolean answer. On the clock cycle following the OPEq, the OPLt compares the data. After OPLt has executed (a single clock cycle later), the data operations *OVER* and *TUCK* are performed in zero clock cycles. Data manipulation functions from Forth words like *OVER*, *TUCK*, and *DROP* are performed inline while routing data into the next sequential OP. Although these stack manipulation words are not explicit in Java bytecodes, they are inferred from successive *iload_x* instructions. The data from OPLt are mapped into a multiplexer. This multiplexer switches data that will be input to OPMinus between data resulting from manipulations conducted based on the results of OPLt. If *x* is less than *y*, the multiplexer will switch data after being rearranged according to *OVER* into OPMinus. If *y* is less than *x*, the multiplexer switches the data after being rearranged according to *TUCK*. Following OPMinus, the results are demultiplexed based on whether the results from OPMinus were from the *minus* that came from the call following the *OVER* or the *TUCK*. Remember, OPs are reused. The demultiplexed data are rearranged as necessary and multiplexed back into OPEq for the next iteration.

The demultiplexer at the output end of OPEq is used to switch the data to either OPLt for comparison or directly to the output of the GCD function when it is finished. This construct is a result of the *BEGIN..WHILE..REPEAT* statement shown in Listing 5, or equivalently it is a result of the *if_icmpne goto 3*, *iload_1, ireturn* statements in lines 21, 24, and 25 of the Java code in Listing 4.

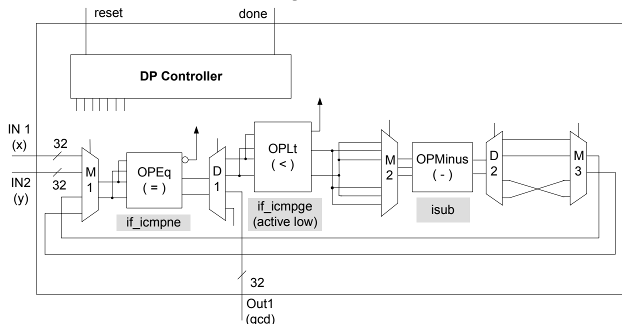

Figure 8 Flowpath implementation of GCD

Figure 9 shows the state transition diagram for the flowpath controller. Since the GCD function requires more than one clock cycle to execute, the *rst* signal must be brought low to begin execution. Execution begins in the *Start* state. From the *Start* state, the control signals are properly set to activate OPEq and multiplex the proper data to OPEq for the next rising edge of the clock and the state machine transitions to state *B*. In state *B*, depending on the output of the OPEq, either OPLt is enabled and a transition is made to state *C* or the data are multiplexed to the main output of the GCD

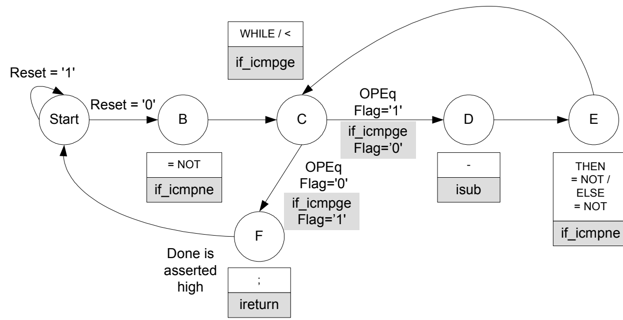

Figure 9 Flowpath controller for GCD

function, *done* is asserted high since the computation has finished ($x = y$), and the state machine transitions to the final state, *F*. State *C* activates OPLt for the comparison and the next state is set to state *D*. State *D* sets up the control signals for the multiplexer entering OPMinus and enables the OP for the next rising edge of the clock. Finally, state *E* enables OPEq resulting from the *BEGIN…WHILE…REPEAT* loop for Forth and the *WHILE* loop in Java implemented as *if_icmpne goto 3* in the bytecode stack-based IR. After the final state, *F*, the state machine returns to the *Start* state and will begin again unless the *reset* signal is returned to high. Flowpaths in general support delays, loops, branches, memory access, basic recursion, and other programmatic constructs. A similar implementation of the GCD algorithm can be found using Handel-C [20]. Table 1 shows a comparison between the Handel-C and flowpath implementations on Xilinx 4010 and Spartan 2 FPGAs.

Table 1 A synthesized 32-bit GCD comparison

Hardware Platform	Hardware Utilization		Clock (MHz)	
	Handel-C	Java Flowpath	Handel-C	Java Flowpath
Xilinx XC4010E-1	69 %	47 %	14	80
Xilinx XC2S200-5	8 %	6 %	26	195

Using flowpaths requires less space and has a maximum clock frequency much higher than the FPGA implementation of the GCD implemented using Handel-C. The Forth code used to implement the GCD algorithm didn't use any special symbols or constructions; it is exactly the same code that would be written for execution on a PC.

4. A Performance Benchmark: Sieve of Eratosthenes

As an example of compiling a software program directly to hardware using flowpaths the Sieve of Eratosthenes algorithm for finding all of the prime numbers less than 2048 was executed on the JStamp that uses the Ajile AJ-80 processor that executes Java bytecodes directly [21]. The program took 933,900 clock cycles to execute at 73.728 MHz as shown in the first row of Table 2. This is an indication of the performance penalty resulting from the load-execute-store paradigm and other overhead of Java bytecodes. The identical algorithm was also written in C and compiled to tight HC12 assembly language code using the ImageCraft HC12 ANSI C Tools V6.15A [22]. This program took 399,031 clock cycles to execute on a Motorola 68HC12 microcontroller at 8 MHz as shown in the second row of Table 2. While this performance is more than twice as good as the Java implementation the local variables associated with the C program still result in a load-execute-store performance penalty.

Listing 6 shows this same algorithm written in Java and Listing 7 in Forth. When implemented on the Motorola 68HC12 using the version of Forth described in [23] this program took 1,763,369 clock cycles at 8 MHz as shown in the third row of Table 2. The reason that this takes nearly twice the clock cycles of the Java bytecode example is that the Forth used is a subroutine-threaded Forth in which the stack manipulation words are implemented as 68HC12 subroutines

Table 2 Relative performance of the Sieve algorithm

Experiment	Hardware Description	#clock cycles	Time (ms) @8MHz	Time (ms) @25MHz	Time (ms) @73.7MHz
Forth – HC12	Stack-based processor emulated on a register-based processor	1,763,369	220.42	----	
JStamp	Processor that executes bytecode directly	933,900	116.74	37.36	12.67
C – HC12	Register-based processor with assembly compiled from C	399,031	49.88	----	
FC16 - FPGA	Stack-based processor optimal for executing each Forth instruction in a single clock cycle	60,299	7.54	2.41	
Flowpath	Processor-less architecture generated by compiling using Flowpath rules	12,222	1.53	0.489	
FPGA-VHDL	Processor-less custom architecture generated 'by hand' for the specific algorithm	6,828	0.85	0.273	

Listing 6 Sieve of Eratosthenes in Java

```
public class Sieve2 {
  private static int j, a, b;
  private static int primeCount;
  private static int flags[] = new int[1024];
  private static int size;
  public static void main(String[] args) {
    int count2;
    size = 1024;
    // first create an array of flags, where each member of the
    // array stands for a odd number starting with 3.
    initFj();
    primeCount = 0;
    for (j = 0; j < size; j++)
    {
        if (flags[j] == 1)  {
        // found a prime, count it
            primeCount++;
       // now set all multiples of the current prime number
        // to false because they couldn't possibly be prime
          a = 2*j + 3;  // odd numbers starting with 3
          b = a + j;
          while(b < size)  {
             flags[b] = 0;
             b = b + a;
          }
        }
     }
}
```

Listing 7 Sieve of Eratosthenes in Forth

```
\\ Forth program for Sieve of Eratosthenes
: sieve ( -- n )         \ n = no. of prime #s
  0 1024 1 FILL          \ fill flags array
  0                      \ cnt
  1024 FOR               \ cnt
       1024 R@ -         \ cnt j
       DUP C@            \ cnt j flags[j]
       IF                \ cnt j
          DUP 2* 3 +     \ cnt j a = (2*j + 3)
          TUCK  +        \ cnt a b = (a + j)
          BEGIN          \ cnt a b
            DUP 1024 <   \ cnt a b f
          WHILE          \ cnt a b
            0 OVER C!    \ store 0 at flags[b]
            OVER +       \ cnt a b = (a + b)
          REPEAT
          2DROP 1+       \ cnt = cnt+1
       ELSE
          DROP           \ cnt
       THEN
  NEXT ;
```

involving register-based load-execute-store instructions.

This same Forth program was compiled to the FC16, a Forth core that has been implemented on a Xilinx Spartan IIE (x2s200e) FPGA [14]. In this case the program ran in 60,299 clock cycles at 25 MHz as shown in the fourth row of Table 2. This improved performance is due to the fact that all stack manipulation instructions are executed in a single clock cycle.

Stack-based programs can be directly implemented in hardware without the need for a core by using flowpaths [17]. In this case the number of clock cycles is reduced to 12,222 as shown in the fifth row of Table 2. This is about five times better performance than the Forth core because all the stack manipulation words have become simple wiring that take zero clock cycles. The details of this implementation are given in Section 5.

Finally, for comparison purposes, a direct VHDL hardware implementation of this same algorithm was carried out. This optimal datapath and control unit implementation executed in 6,828 clock cycles at 25 MHz as shown in the last row of Table 2. This is an optimal design specifically hand-crafted in VHDL for the Sieve. Using Xilinx ISE 6.2i targeting a Spartan IIE, this custom architecture synthesized to 233 Slices (9% of the chip) and the Sieve flowpath required 295 Slices (12% of the chip). Approximately 40% of the area used in the flowpath was used for combinational logic and 80% in the custom architecture. Thus the custom architecture uses more combinational logic and fewer flip-flops whereas the flowpath used less combinational logic and more flip-flops.

These results for the Sieve experiment were consistent with general expectations of the performance gain using flowpaths. While stack-manipulation operations require clock cycles in a processor, they translate into routing in flowpaths and do not require any type of clocked execution. It follows that flowpaths would require at most the same number of clock cycles as a processor optimized for performing stack-based instructions in a single clock cycle such as the FC16 Forth Core [14].

5. The Sieve Compiled to a Flowpath

The Forth program shown in Listing 7 yields the datapath shown in Figure 10 and the state machine controller shown in Figure 11. The datapath together with the controller form the flowpath that implements the Sieve program. The flowpath presented in this paper is a raw flowpath generated using compilation rules found in [17] without any special optimization since space is not the concern of this work.

Execution begins when the *reset* signal is brought low with state *A*. State *A* pushes the loop maximum to the "return stack". In flowpaths the stack is virtual; the data in the "data stack" are stored in registers that are transferred from one OP to the next and the data on the "return stack" are also stored in registers. The return stack is used in Forth for *FOR..NEXT* loops, return addresses for returning from subroutines, and temporary storage. When considering flowpaths,

return addresses don't exist. In Java, there is no return stack. Therefore, the construct described for the Forth-generated flowpath that implements the *FOR..NEXT* loop is different from that which is generated from the Java bytecodes. This bytecode translation is straightforward.

Since *FOR..NEXT* loops are implemented as a down-counter in Forth while the loop counter counts up in Java, an array indexed by a loop counter such as *flags[j]* would be considered *flags[size-RS]* in Forth where *size* is the maximum loop count given in *for(j=0; j<size; j++)*. This hardware, for accessing an array using a for-loop counting variable is circled in Figure 10 and is different from the equivalent Java-generated flowpath.

Execution continues to state *B*, which enables the *MINUS* OP. The next instruction, *C@*, reads data from memory. The FPGA's block memory was used with a global initial value of '1'. Since the block memory requires more than one clock cycle for memory reads, two states were necessary to implement this operation. OPs for memory instructions, *C@* and *C!*, which are used for reading data from and writing data to memory, respectively, are created specifically for a given memory target. These OPs are implemented when these instructions are compiled for the specific memory target. The *C@* and *C!* OPs equivalently perform the Java bytecodes *iaload* and *iastore*, respectively. State *C* latches the address into the block memory's address register while state *D* latches the data into the *C@* OP, which are available on the clock-edge after the address write. C@ is followed by the *IF* statement. The rest of the controller implements the algorithm as a direct mapping from the stack-based programs.

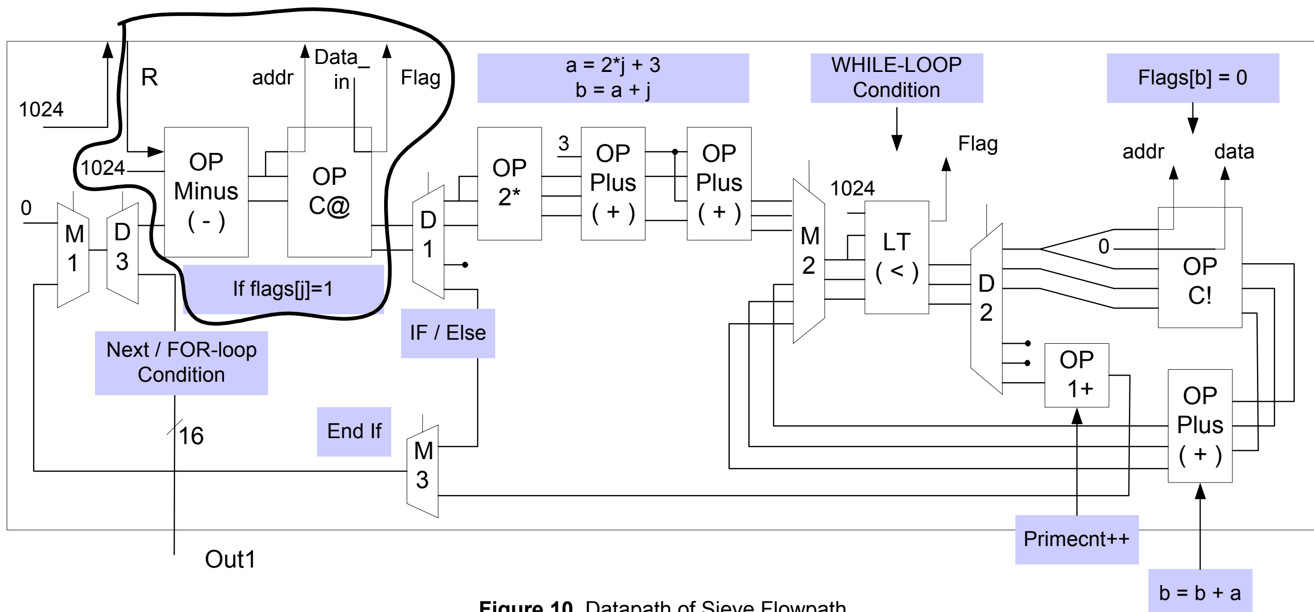

Figure 10 Datapath of Sieve Flowpath

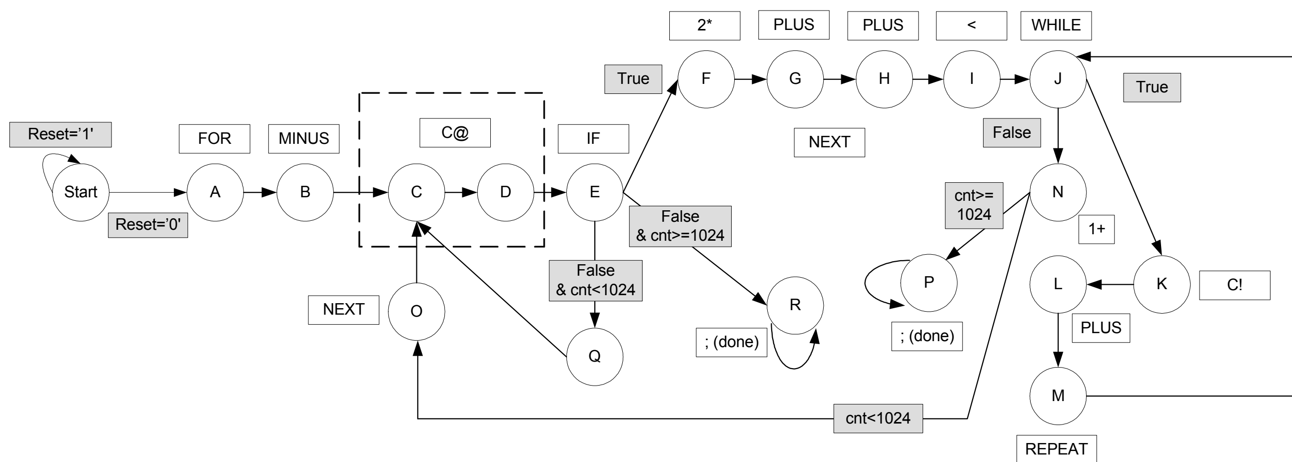

Figure 11 Controller of Sieve Flowpath

The up-arrows in Figure 10 indicate signals that are output from the datapath and input to the controller. *OUT1* is the 16-bit output indicating the number of primes that were found and flagged in memory for debugging purposes. A high-level program calling the *Sieve* flowpath would use another flowpath generated from a Java, Forth, or similarly stack-based IR for displaying the prime numbers with the proper delays. This implementation performed almost two-orders of magnitude better than the same Java code executed on the JStamp's native bytecode microprocessor.

6. Flowpaths and Dynamic Reconfiguration

Several researchers have considered using a form of C as a hardware description language and some have developed methods for compiling C-programs directly to hardware [1-7]. On the one hand, a programmer must consider hardware constructs when programming while on the other the programmer develops a software program with minimal consideration of the hardware target. These approaches have targeted reducing time-to-market by generating hardware directly from familiar languages. In both cases, large algorithms require significant hardware resources and depend on increasing FPGA densities to implement more realistic, larger designs. Often, the cost of the FPGAs with enough equivalent gates to support a circuit generated in this fashion prohibits them from being applied to a mass-production product.

Few people however, have considered dynamically reprogramming the FPGA to reuse resources on-the-fly [1, 24, 25] to make it feasible to implement larger, practical algorithms on affordable, reasonable FPGAs. This would allow more applications to be cost-effectively implemented on FPGAs instead of on an ASIC. In general, the architectures generated by the aforementioned methods are not necessarily designed to exploit the benefit of executing in subsets and hence, are more traditional circuits that must be downloaded to a large enough chip on which to execute.

Since flowpaths are comprised of a series of registered OPs that forward the local variables from one OP to the next, valid data are moved throughout the flowpath as execution proceeds. Therefore flowpaths can be partitioned into subsets of which only particular subsets are required for a given number of clock cycles of execution. The FPGA can be partitioned into sections. Part of the flowpath can execute on one section while another section of the FPGA is dynamically reconfigured with the next logical flowpath subset. Using this technique, flowpaths generated by large algorithms can be split into smaller pieces allowing the large algorithm to be executed on an FPGA on which the full algorithm would not fit. Likewise, multiple large algorithms can be executed on a single FPGA concurrently.

Experiments have been successfully conducted implementing a flowpath for Euclid's Greatest Common Divisor generated from a Forth program on a Xilinx Virtex 1000 as proof of concept. The Xilinx Virtex 1000 supports dynamic partial reconfiguration. A basic flowpath splitting algorithm and design considerations including speed limitations caused by waiting for the FPGA-half to reconfigure are being developed.

7. Conclusion and Future Work

This paper has shown how standard stack-based programs, such as Java bytecodes and Forth, can be represented in hardware by a new technique called flowpaths. These flowpaths are made up of sequential modules called *OPs* that represent instructions that perform operations on data. The OPs are connected together through multiplexers and are enabled by a controller module that implements a state machine that describes a particular stack-based program.

This paper has also shown that compiling high-level Forth code directly to hardware using flowpaths can produce performance close to that of a hand-crafted direct circuit implementation. By contrast, the same program run on traditional microprocessors and in FPGA microprocessor cores can have much longer execution times. Flowpaths can outperform microprocessors at lower clock frequencies and therefore consume less power than microprocessors or microprocessor cores. Flowpaths can be portioned into subsets that can take advantage of dynamic partial reconfiguration.

To develop a complete compiler, capable of generating flowpaths from a typical Java program there are three areas that must be considered: 1) general Java methods and syntax, 2) event handling, and 3) threads. This paper has shown how to address the first of these three considerations. Current work is exploring and implementing numbers 2) and 3). Another major impact of generating flowpaths from multi-threaded Java code is that each thread can execute in parallel unlike in a microprocessor where the threads share clock resources.

Darrin M. Hanna received his B.S. from Oakland University in 1999 with top honors in both Computer Engineering and Mathematics and Ph.D. in Systems Engineering in 2003 and is an Assistant Professor of Engineering in the Department of Computer Science and Engineering at Oakland University, Rochester, Michigan. As a sophomore, Dr. Hanna started a company, Technology Integration Group Services, Incorporated, specializing in technical infrastructure, intelligent application development, and wireless systems. The company has continued to grow internationally, opening additional offices in London in 2002. Dr. Hanna's research interests include BioMEMS, microprocessor-less architectures for implementing hardware directly from high-level source code, and pattern recognition techniques for embedded systems.

Richard E. Haskell is a professor of engineering in the Department of Computer Science and Engineering in Oakland University's School of Engineering and Computer Science. He received his Ph.D. from Rensselaer Polytechnic Institute in 1963, and has been on the faculty at Oakland since 1966. He has also worked in government laboratories at the Air Force Cambridge Research Laboratories and at the Johnson Spacecraft Center. His research interests have included plasma physics, holography and coherent optics, pattern recognition and image processing, computer learning, and microprocessor applications and embedded systems. In addition to his published research work, Dr. Haskell is the author of fifteen books and holds six patents. He has taught numerous undergraduate and graduate courses including courses in microprocessors, embedded systems and digital design using VHDL.

8. References

[1] J. M. P. Cardoso and M. Weinhardt, "From C Programs to the Configure-Execute Model," presented at IEEE Design, Automation and Test in Europe (DATE) Conference and Exhibition, Munich, Germany, 2003.

[2] D. Soderman, "Implementing C Designs in Hardware," presented at DesignCon98, 1998 International Verilog Conference & 1998 FPGA Configurable Computing Machine Conference, 1998.

[3] M. Aubury, I. Page, G. Randall, J. Saul, and R. Watts, "Handel-C Language Reference Guide," Oxford University Computing Laboratory 1996.

[4] F. Boussinot, "Reactive C: An Extension of C to Program Reactive Systems," *Software Practice and Experience*, vol. 21, pp. 401-428, 1991.

[5] Celoxica, "The Technology Behind DK1," Milton Park, Abingdon, Oxfordshire, United Kingdom Application Note, AN 18 v1.0.

[6] A. Ye Zhi, N. Shenoy, and P. Banerjee, "A C compiler for a processor with a reconfigurable functional unit," *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 2000.

[7] L. Semeria, "Applying Pointer Analysis to the Synthesis of Hardware from C," Stanford University, 2001.

[8] S. A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," presented at International Workshop of Logic and Synthesis (IWLS), Temecula, CA, 2004.

[9] S. A. Edwards, "The Challenges of Hardware Synthesis from C-like Languages," *Proceedings of Design Automation and Test in Europe (DATE)*, Munich, Germany, 2005.

[10] F. Balarin, P. Giusto, A. Jurecska, C. Passerone, E. Sentovich, B. Tabbara, M. Chiodo, H. Hsieh, L. Lavagno, A. Snagiovanni-Vincentelli, and K. Suzuki, *Hardware-Software Co-Design of Embedded Systems, The POLIS Approach*. Boston, Massachusetts: Kluwer Academic Publishers, 1997.

[11] C. Daws, A. Olivero, S. Tripakis, and S. Yovine, "The Tool Kronos.," *Proceedings of Hybrid Systems III, Verification and Control*, 1996.

[12] F. Boussinot and R. De Simone, "The SL Synchronous Language," *IEEE Transactions on Software Engineering*, vol. 22, pp. 256-266, 1996.

[13] P. H. J. Van Eijk, C. A. Vissers, and M. Diaz, *The Formal Description Technique LOTOS*: Elsevier Science Publishers B. V., 1989.

[14] R. E. Haskell and D. M. Hanna, "A VHDL Forth Core for FPGAs," *Microprocessors and Microsystems*, vol. 28, pp. 115-125, 2004.

[15] R. Wilson, "Actel, ARM put soft CPU in FPGA," in *EE Times*, 2005.

[16] I. Xilinx, "Microblaze - The Low-Cost and Flexible Processing Solution," 2005.

[17] D. M. Hanna, "A Novel Method for Generation Microprocessor-less Systems with Applications in Bioengineering," Ph.D. Dissertation in the *Department of Computer Science and Engineering*. Rochester, MI: Oakland University, 2003.

[18] P. Haggar, *Practical Java Programming Language Guide*, 1 ed: Addison-Wesley Professional, 2000.

[19] F. Vahid and T. Givargis, *Embedded System Design: A Unified Hardware/Software Introduction*: John Wiley & Sons, Inc., 2002.

[20] Celoxica, "FPGA Implementation of Software GCD Algorithm," Campbell, California App. Note, AN 06 v1.0.

[21] Systronix, "JStamp: Real-time Native Java Module," 2003.
[22] J. Valvano, *Embedded Microcomputer Systems: Real-time Interfacing*: Brooks Cole, 2000.
[23] R. E. Haskell, *Design of Embedded Systems using 68HC12/11 Microcontrollers*: Prentice Hall, 1999.
[24] V. Sklyarov, "Reconfigurable models of finite state machines and their implementation in FPGAs," *Journal of Systems Architecture*, vol. 47, pp. 1043 - 1064, 2002.
[25] I. Ouaiss, et al., "An Integrated Partitioning and Synthesis System for Dynamically Reconfigurable Multi-FPGA Architectures," presented at Fifth Reconfigurable Architectures Workshop (RAW), Orlando, FL, 1998.