Fly – A Modifiable Hardware Compiler

C.H. Ho[1], P.H.W. Leong[1], K.H. Tsoi[1], R. Ludewig[2], P. Zipf[2], A.G. Ortiz[2], and M. Glesner[2]

[1] Department of Computer Science and Engineering
The Chinese University of Hong Kong, Shatin NT HK.
{chho2,phwl,khtsoi}@cse.cuhk.edu.hk
[2] Institute of Microelectronic Systems
Darmstadt University of Technology, Germany.
{ludewig,zipf,agarcia,glesner}@mes.tu-darmstadt.de

Abstract. In this paper we present the "fly" hardware compiler for rapid system prototyping research. *Fly* takes a C-like program as input and produces a synthesizable VHDL description of a one-hot state machine and the associated data path elements as output. Furthermore, it is tightly integrated with the hardware design environment and implementation platform, and is able to hide issues associated with these tools from the user. Unlike previous tools, *fly* encourages modification of the compiler for research in rapid system prototyping and code generation, and the full source code to the compiler is presented in the paper. Case studies involving the implementation of an FPGA based greatest common divisor (GCD) coprocessor as well as the extension of the basic *fly* compiler to solve a differential equation using floating point arithmetic are presented.

1 Introduction

With the rapid advancements in FPGA technology and the constant need to improve designer productivity, increasingly higher levels of abstraction are desired. We have found that using a RTL based design methodology results in low productivity compared, for example, with software development in C. It is believed that this is due to the following issues:

- Hardware designs are parallel in nature while most of the people think in von-Neumann patterns.
- The standard technique of decomposing a hardware design into datapath and control adds complexity to the task.
- Designers must develop a hardware interface for the FPGA board as well as a software/hardware interface between a host system and the FPGA.

The above issues serve to significantly increase the design complexity, with an associated increase in design time and debugging. Furthermore, the time spent in the above process restricts the amount of time which can be spent on dealing with higher level issues such as evaluating different algorithms and architectures for the system.

M. Glesner, P. Zipf, and M. Renovell (Eds.): FPL 2002, LNCS 2438, pp. 381–390, 2002.

Hardware description languages (HDL) have been proposed to address some of the issues above, notable examples being VHDL, SystemC, Handel-C [1], Pebble [2] and Lola [3]. For asynchronous circuits, Brunvand [4] has detailed a similar methodology. All of the HDLs above allow the user to describe a circuit using a behavioural model. Tools are used to translate from this higher level of abstraction into either a netlist or a register transfer language (RTL) code in a HDL such as VHDL or Verilog. With the exception of Lola which is written in Oberon, a programming language which is not in widespread usage, none of the above are available in source code form.

In order to facilitate research in high level synthesis and backend code generation, a compiler for the translation of a new language called *fly*, a small subset of Perl, was developed. The full source code to the compiler is included in Appendix A of this paper. Key differences between our approach and that of previous work are that:

- Fly's implementation has an open source license and can be easily understood, and modified by other users. We hope that other researchers will use this tool for research in high level synthesis, language design and circuit generation architectures and techniques.
- *Fly* supports a simple memory mapped interface between a host processor and the FPGA, serving to hide the details of the host interface from the designer.

The rest of the paper is organized as follows, in Section 2, the *fly* programming language, its implementation and development environment is presented. In Section 3, the application of *fly* in developing a greatest common divisor coprocessor is described. In Section 4, the extension of *fly* to operate on floating point numbers, and the application of this new system to the solution of differential equations is given. A discussion and conclusions are given in Sections 5 and 6.

2 The Fly Programming Language

The syntax of the *fly* programming language is modelled on Perl and C, with extensions for parallel statements and the host/FPGA interface. The language is minimal in nature and supports while loops, if-else branches, integer arithmetic, parallel statements and register assignment. Table 1 shows the main elements of the *fly* language with simple examples. The formal grammar definition can be found in the parser (see Appendix A).

2.1 Compilation Technique

Programs in the *fly* language are automatically mapped to hardware using the technique described by Page [1, 3]. In order to facilitate the support of control structures, Page's technique assigns a `start` and `end` signal to each statement which specifies when its execution begins and ends. By connecting the `start`

Construct	Elements	Example
assignment	var = expr;	$var1 = tempvar;$
parallel statement	[{ ... } { ... } ...]	[$\{a = b;\}$ { $b = a * c;\}$]
expression	val op expr; valid ops: $*$,/,+,−	$a = b * c;$
loop	while (condition) { ... }	while $(x < y)$ { $a = a + b; y = y + 1;\}$

if-else	if (condition) { ... } else { ... }
	
	if (condition) { ... }
		if $(i <= j)$ { $a = b;\}$
		else $\{a = c;\}$
		if $(i > j)$ $\{i = i + 1;\}$

condition	expr rel expr valid rels: >,<,<=,>=,==,! =	$i >= c$

Table 1. Main elements of the *fly* language.

and `end` signals of adjacent statements together, a one-hot state machine is constructed that serves as the control flow of the hardware.

The *fly* compiler generates synthesizable VHDL code instead of a netlist, simplifying code generation and making the output portable to many different FPGA and ASIC design tools. Furthermore, using VHDL as an intermediate language enables the logic optimization of the synthesis tool to be included in the design flow.

2.2 Implementation Details

Fly is written in the Perl programming language [5]. Perl is a language with very good portability, string handling facilities and libraries. We feel that the *fly* system's source code is made simpler and concise as a result of using Perl. Development of the *fly* compiler was also facilitated using a parser generator called `Parse::RecDescent` [6] which generates a Perl based recursive descent parser from a description of the grammar of the target language.

2.3 Host Interface

Although the interface is easily adaptable to any reconfigurable computing card, the *fly* system currently only supports the Pilchard reconfigurable computing platform [7]. Pilchard uses a DIMM memory bus interface instead of a conventional PCI bus. The advantage of the memory bus is that it achieves much improved latency and bandwidth over the standard PCI bus.

The translated output of a *fly* program is interfaced with a generic Pilchard core written in VHDL. A shell script includes all of the required libraries and invokes all of the programs required to compile the VHDL representation of the user's program to a bitstream. The bitstream is then downloaded to the FPGA and the host interface program invoked. By automating the process of synthesis, implementation and downloading using shell scripts, the specifics of the compilation and execution process are hidden from the user.

Registers are used to transfer data between the FPGA and host. In normal operation, the host processor would initialize values in $din[1] to $din[x], and then start execution of the FPGA based coprocessor by performing a write cycle to the $din[0] register. The write cycle causes the start signal of the first statement in the FPGA to be asserted. The software then polls the least significant bit of $din[0] which is connected to the end signal of the last statement. When execution on the FPGA finishes, the least significant bit of $din[0] is set and the program can read values returned by the hardware by reading the appropriate registers.

3 A GCD Processor

The *fly* program for a greatest common divisor (GCD) coprocessor is given below:

```
{
    $s = $din[1]; $l = $din[2];
    while ($s != $l) {
        $a = $l - $s;
        if ($a > 0) {
            $l = $a;
        }
        else {
            [ {$s = $l;} {$l = $s;} ]
        }
    }
    $dout[1] = $l;
}
```

The GCD coprocessor design was synthesized for a Xilinx XCV300E-8 and the design tools reported a maximum frequency of 126 MHz. The design, including interfacing circuitry, occupied 135 out of 3,072 slices.

The following perl subroutine tests the GCD coprocessor using randomly generated 15-bit inputs.

```
for (my $i = 0; $i < $cnt ; $i++) {
    $a = rand(0x7fff) & 0x7fff;
    $b = rand(0x7fff) & 0x7fff;

    &pilchard_write64(0, $a, 1);    # write a
    &pilchard_write64(0, $b, 2);    # write b
    &pilchard_write64(0, 0, 0);     # start coprocessor

    do {
        &pilchard_read64($data_hi, $data_lo, 0);
    } while ($data_lo  == 0);       # poll for finish
    &pilchard_read64($data_hi, $data_lo, 1);

    print ("gcd $a, $b = $data_lo\n");
}
```

The GCD coprocessor was successfully tested at 100 MHz by calling the FPGA-based GCD implementation with random numbers and checking the result against a software version. For randomized input test program above, the resulting system had an average execution time of $1.63\mu s$ per GCD iteration, which includes all interfacing overheads but excludes random number generation, checking and Perl looping overheads (Perl overheads were avoided by using inlined C in critical sections).

4 Floating Point Extension

As an example of how the *fly* system can be extended, floating point operators were added. Firstly, a parameterised module library which implemented floating point adders and multipliers, similar to that of Jaenicke and Luk[8] was developed [9]. In the library, numbers are represented in IEEE 754 format with arbitrary sized mantissa and exponent [10]. Rounding modes and denormalized numbers were not supported. The performance of the library is summarized in Table 2. The adder is not yet fully optimized and the maximum frequency was 58 MHz.

Table 2. Area and speed of the floating point library (a Virtex XCV1000E-6 device was used). One sign bit and an 8-bit exponent was used in all cases.

Fraction Size (bits)	Circuit Size (slices)	Frequency (MHz)	Latency (cycles)
Multiplication			
7	178	103	8
15	375	102	8
23	598	100	8
31	694	100	8
Addition			
7	120	58	4
15	225	46	4
23	336	41	4
31	455	40	4

The following modifications were then made to the floating point module library and *fly* in order to utilize this library:

- `start` and `end` signals were added to the floating point operators.
- A dual-ported block RAM interface to the host processor via `read_host()` and `write_host()` was added. This interface works in a manner analogous to the register based host interface described in Section 2.3 and allows data between the host and FPGA to be buffered.

- Three new floating point operators ".+", ".-" and ".*" were added to the parser to invoke floating point addition, subtraction and multiplication respectively.
- The parser was changed to enforce operator precedence and to instantiate the floating point operators appropriately.

4.1 Application to Solving Differential Equations

The modified *fly* compiler was used to solve the ordinary differential equation $\frac{dy}{dt} = \frac{(t-y)}{2}$ over $t \in [0, 3]$ with $y(0) = 1$ [11]. The Euler method was used so the evolution of y is computed by $y_{k+1} = y_k + h\frac{(t_k - y_k)}{2}$ and $t_{k+1} = t_k + h$ where h is the step size.

The following *fly* program implements the scheme, where h is a parameter sent by the host:

```
{
    $h = &read_host(1);
    [
      {$t = 0.0;} {$y = 1.0;} {$dy = 0.0;}
      {$onehalf = 0.5;} {$index = 0;}
    ]
    while ($t < 3.0) {
        [ {$t1 = $h .* $onehalf;} {$t2 = $t .- $y;} ]
        [ {$dy = $t1 .* $t2;} {$t = $t .+ $h;} ]
        [
          {$y = $y .+ $dy;}
          {$index = $index + 1;}
        ]

        $void = &write_host($y, $index);
    }
}
```

In each iteration of the program, the evolution of y is written to the block RAM via a `write_host()` function call and a floating point format with 1 sign bit, 8-bit exponent and 23-bit fraction was used throughout. The floating point format can, of course, be easily changed. Parallel statements in the main loop achieve a 1.43 speedup over a straightforward serial description.

The differential equation solver was synthesized for a Xilinx XCV300E-8 and the design tools reported a maximum frequency of 53.9 MHz. The design, including interfacing circuitry, occupied 2,439 out of 3,072 slices. The outputs shown in Table 4.1 were obtained from the hardware implementation at 50 MHz using different h values. The resulting system ($h = \frac{1}{16}$) took 28.7μs for an execution including all interfacing overheads.

t_k	$h = 1$	$h = \frac{1}{2}$	$h = \frac{1}{4}$	$h = \frac{1}{8}$	$h = \frac{1}{16}$	$y(t_k)$ Exact
0	1.0	1.0	1.0	1.0	1.0	1.0
0.125				0.9375	0.940430	0.943239
0.25			0.875	0.886719	0.892215	0.897491
0.375				0.846924	0.854657	0.862087
0.50		0.75	0.796875	0.817429	0.827100	0.836402
0.75			0.759766	0.786802	0.799566	0.811868
1.00	0.5	0.6875	0.758545	0.790158	0.805131	0.819592
1.5		0.765625	0.846386	0.882855	0.900240	0.917100
2.00	0.75	0.949219	1.030827	1.068222	1.086166	1.103638
2.50		1.211914	1.289227	1.325176	1.342538	1.359514
3.00	1.375	1.533936	1.604252	1.637429	1.653556	1.669390

Table 3. Results generated by the differential equation solver for different values of h.

5 Discussion

There are certain limitations associated with the compilation method used in this paper. The compiler produces only one-hot state machines which may be inefficient in certain cases. In addition, the language only supports simple constructs and may be awkward for describing certain types of parallel programs. Finally, unless the designer fully understands the translation process and can explicitly describe the parallelism, the resulting hardware is mostly sequential in nature and would not be very efficient. Despite these limitations, we feel that the benefits in productivity and flexibility that are gained from this approach would outweigh the cons for many applications.

The compiler in Appendix A generates a fixed point bit parallel implementation, and it was shown how this could be extended to a floating point implementation. If, for example, a digit serial operator library were available, it could be easily modified to use digit serial arithmetic. Similarly, both fixed point and floating point implementations of the same algorithm could be generated from the same *fly* description. In the future, we will experiment with more code generation strategies. Many designs could be developed from the same program, and different *fly* based code generators could serve to decouple the algorithmic descriptions from the back-end implementation. In the case of using a digit serial library, users could select the digit size, or produce a number of implementations and choose the one which best meets their area/time requirements.

It is also possible to modify the compiler to produce code for different HDLs, program proving tools, and programming languages. Having an easily understandable and easily modifiable compiler allows the easy integration of the *fly* language to many other tools.

6 Conclusions

In this paper, the Perl programming language was used to develop a powerful yet simple hardware compiler for FPGA design. Unlike previous compilers, *fly* was designed to be easily modifiable to facilitate research in hardware languages and code generation. Since *fly* is tighly integrated with the hardware design tools and implementation platform, designers can operate with a higher level of abstraction than they might be accustomed to if they used VHDL. Examples involving a GCD coprocessor and the solution of differential equations in floating point were given.

Acknowledgements

The work described in this paper was supported by a direct grant from the Chinese University of Hong Kong (Project code 2050240), the German Academic Exchange Service DAAD (Projekt-Nr.: D/0008347) and the Research Grants Council of Hong Kong Joint Research Scheme (Project no. G_HK010/00).

References

[1] Page, I.: Constructing hardware-software systems from a single description. Journal of VLSI Signal Processing **12** (1996) 87–107
[2] Luk, W., McKeever, S.: Pebble: a language for parametrised and reconfigurable hardware design. In: Field-Programmable Logic and Applications. Volume LNCS 1482., Springer (1998) 9–18
[3] Wirth, N.: Hardware compilation: Translating programs into circuits. IEEE Computer **31** (1998) 25–31
[4] Brunvand, E.: Translating Concurrent Communicating Programs into Asynchronous Circuits. Carnegie Mellon University, Ph.D thesis (http://www.cs.utah.edu/∼elb/diss.html) (1991)
[5] Wall, L., Christianson, T., Orwant, J.: Programming Perl. 3rd edn. O'Reilly (2000)
[6] Conway, D.: Parse::RecDescent Perl module. In: http://www.cpan.org/modules/by-module/Parse/DCONWAY/Parse-RecDescent-1.80.tar.gz. (2001)
[7] Leong, P., Leong, M., Cheung, O., Tung, T., Kwok, C., Wong, M., Lee, K.: Pilchard - a reconfigurable computing platform with memory slot interface. In: Proceedings of the IEEE Symposium on FCCM. (2001)
[8] Jaenicke, A., Luk, W.: Parameterised floating-point arithmetic on FPGAs. In: Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing. (2001) 897–900
[9] Ho, C., Leong, M., Leong, P., Becker, J., Glesner, M.: Rapid prototyping of fpga based floating point dsp systems. In: Proceedings of the 13th IEEE Workshop on Rapid System Prototyping (to appear). (2002)
[10] Hennessy, J.L., Patterson, D.A.: Computer Architecture: A Quantitative Approach 2nd Edition. Morgan Kaufmann (1999)
[11] Mathews, J., Fink, K.: Numerical Methods Using MATLAB. 3rd edn. Prentice Hall (1999)

A Fly Source Code

In this appendix, the entire source code for the *fly* system used to compile the GCD coprocessor is given. Updates to this program and the modified *fly* compiler which supports floating point operations are available from: http://www.cse.cuhk.edu.hk/∼phwl/fly/fly.html.

```
package main;
use Parse::RecDescent;

my $grammar = q {
{ my ($seq, $comb, $aux, $paux, $s, %sigs) =
         ("", "", 0, 0, "signal"); }

prog: stmtlist /^$/  {
  print "library ieee;\n";
  print "use ieee.std_logic_1164.all;\n";
  print "use ieee.std_logic_arith.all;\n\n";
  print "package hc_pack is \n";
  print "  subtype word is integer;\n";
  print "  type   words is array(integer ";
  print "range <>) of word;\nend hc_pack;\n\n";
  print "library ieee;\n";
  print "use ieee.std_logic_1164.all;\n";
  print "use ieee.std_logic_arith.all;\n";
  print "use work.hc_pack.all;\n\n";


  print "--type words is array (integer range <>) ";
  print "of word;\nentity arith_core is\nport(\n";
  print "\tclk: in std_logic;\n";
  print "\trst: in std_logic;\n";
  print "\tstart: in std_logic;\n";
  print "\tdin : in words( $sigs{din} ";
  print "downto 1);\n\tfinish: out std_logic;\n";
  print "\tdout: out words( $sigs{dout} ";
  print "downto 1));\nend arith_core;\n";
  print "architecture rtl of arith_core is\n";

  foreach my $k (keys %sigs) {
    if ($sigs{$k}) {
      print "$s  $k :\t words($sigs{$k}  " .
        "downto 0);\n"
      if !($k eq "din")
        and !($k eq "dout") ;
    }
    else {
      print "$s  $k :\t word; \n";
    }
  }
  for (my $i=1; $i<$aux; $i++) {
    print "$s s$i, f$i :\t boolean; ";
    print "--std_logic;\n";
  };
  for (my $i=1; $i<=$paux; $i++) {
    print "$s p$i, q$i :\t boolean; ";
    print "--std_logic;\n";
  };

  print "$s s$item[1], f$item[1] :\t boolean; ";
  print "--std_logic;\nbegin --architecture\n";
  print " s$item[1] <= TRUE when start='1' ";
  print "else FALSE ;--start;\n finish <= '1' ";
  print "when f$item[1] else '0'; --f$item[1];\n";
  print "process(clk)\nbegin\n";
  print "if rising_edge(clk) then\n";
    print $seq;
    print "end if;\nend process;\n";
    print "--combinational part\n$comb";
    print "end rtl;\n";
  }

  stmtlist: stmt | '{' stmt(s) '}' {
    my $fst_in = shift(@{$item[2]});
    my $int_in = $fst_in;
    $aux += 1 ;
    $comb .= "s$int_in <= s$aux; \n";
    foreach $int_in (@{$item[2]}) {
      $comb .= "s$int_in <= f$fst_in;\n";
      $fst_in = $int_in;
    }
    $comb .= "f$aux <= f$fst_in;\n";
    $aux;
  }

  stmt: asgn | ifelse | if  | while |
        pstmtlist | <error>

  pstmtlist:  '[' stmtlist(s) ']' {
    $aux += 1;
    my $int_in;
    my @plist = ();
    foreach $int_in (@{$item[2]}) {
      $comb .=sprintf("s%d<=s%d;--pstmtlist\n",
                      $int_in, $aux);
      $paux += 1;
      push (@plist, $paux);

      $seq .= "if f$aux then --pstmtlist\n\t";
      $seq .= "q$paux <= false;\n";
      $seq .= "else\n\t";
      $seq .= "q$paux <= p$paux; \n";
      $seq .= "end if; \n";

      $comb .= "p$paux <= f$int_in or q$paux;";
      $comb .= " --pstmtlist\n";
    }
    my $pend ="f$aux <= p".join("and p",@plist)
      . "; --pstmt end\n";
    $comb .= $pend;
    $aux;
  }

  asgn: var '=' expr ';' {
    $aux = $aux + 1;
    $seq .= "if s$aux then\n\t";
    $seq .= "$item[1] <= $item[3];\n";
    $seq .= "end if;\n";
    $seq .= "f$aux <= s$aux;\n\n";
    $aux;
  }

  expr: val op expr{"$item[1]$item[2]$item[3]"}
        |  val

  op: '*' | '/' | '+' | '-'

  val: /\d+/ | var

  var: /\$[a-z][\w\[\]]*/  {
    $item[1] =~ s/^\$//;
    my $sig = $item[1];
    $sig =~ s/\[(\d+)\]//;
    $sigs{"$sig"} = ($sigs{"$sig"} &&
```

```
    ($sigs{"$sig"} > $1) ) ? $sigs{"$sig"} : $1;
  $item[1] =~ tr/\[\]/\(\)/;
  $item[1];
}

while: 'while' '(' cond ')' stmtlist {
  $aux += 1;
  $comb .= "s$item[5] <= ($item[3]) and " .
           "(s$aux or f$item[5]);\n";
  $comb .= "f$aux <= (not ($item[3])) and " .
           "(s$aux or f$item[5]);\n";
  $aux;
}

ifelse: 'if' '(' cond ')' stmtlist 'else' stmtlist {
  $aux += 1;
  $comb .= "s$item[5] <= ($item[3]) and s$aux;\n";
  $comb .= "s$item[7] <= (not ($item[3])) and s$aux;\n";
  $comb .= "f$aux <= f$item[5] or f$item[7];\n";
  $aux;
}

if: 'if' '(' cond ')' stmtlist {
  $aux += 1;
  $comb .= "s$item[5] <= ($item[3]) and s$aux;\n";
  $comb .= "f$aux <= (not ($item[3]) and s$aux) or f$item[5];\n";
  $aux;
}

cond: expr rel expr  { "$item[1] $item[2] $item[3]" }

rel: '>' | '<' | '<=' | '>=' | '!=' { "/=" } | '==' { "=" }

varlist: var ',' varlist  { "$item[1] $item[3]" } | var
};

$::RD_HINT = 0;
$::RD_AUTOACTION = q { $item[1] };
my $parser = Parse::RecDescent->new($grammar)
  or die "Bad grammar";

local $/;
my $script = <>;
my $tree = $parser->prog($script) or die "Bad script";
```