Delft University of Technology
Parallel and Distributed Systems Report Series

Crystal: a simple programming language for Custom Computing Machines

C. van Reeuwijk

report number PDS-2004-001

ISSN 1387-2109

Published and produced by:
Parallel and Distributed Systems Section
Faculty of Information Technology and Systems Department of Technical Mathematics and Informatics
Delft University of Technology
Zuidplantsoen 4
2628 BZ Delft
The Netherlands

Information about Parallel and Distributed Systems Report Series:
reports@pds.twi.tudelft.nl

Information about Parallel and Distributed Systems Section:
http://pds.twi.tudelft.nl/

Abstract

With the growing capacity of Field-Programmable Gate Arrays (FPGAs), they have become an attractive platform for solving computationally intensive problems. For a wide spectrum of problems, implementations in reconfigurable logic show very significant speedups compared to standard implementations.

Despite the attractiveness, reconfigurable computing is still rarely used, mainly because there is no suitable programming environment. Although many attempts have been made to implement such an environment, none have become popular. A common flaw is that an inappropriate programming model—imperative programming—is used. Since translation from an imperative program to reconfigurable logic is difficult, it results in software that is too large and/or too slow.

To address this fundamental problem, we propose to use the *synchronous* programming model. Although this results in a very different programming language, we believe that such a language is as easy to understand as a language such as C or Java.

To demonstrate our point, we will describe a synchronous programming language, called Crystal. Although the language is fairly simple, We will use it to implement a number of realistic computations.

1 Introduction

Now that the growth of standard processor performance is nearing its end, it is worthwhile to look at the potential of other architectures. One interesting architecture is the Field-Programmable Gate Arrays (FPGAs). An FPGA can be configured to implement arbitrary digital circuits. Thus, a computational problem can be solved by designing a custom computing machine (CCM) to solve a specific problem, and implementing this CCM on an FPGA.

Modern FPGAs can accommodate very large circuits. The largest FPGAs available today have nearly $100,000$ logic cells [2, 35], and can implement digital circuits with up to $5,000,000$ logic gates. In addition, these devices provide dedicated memory blocks to implement register sets, stacks, FIFO, etc., and often other dedicated blocks such as multipliers. As Lignon [27] et al. report, such an FPGA can accommodate, for example, roughly 150 single precision floating point adders or multipliers, 500 16-bit integer multipliers, or several thousand 16-bit integer adders.

Since the configuration bits are stored in RAM cells, the FPGA can be reconfigured at any moment. This means that the FPGA can be treated as a processor, and a FPGA configuration as a program for that processor. The processor is rather unusual because it deviates from traditional processors with separate memory banks and a central processing unit: it is not a 'von Neumann' architecture.

Using a FPGA CCM is called *reconfigurable computing*. It can lead to spectacular speedups: compared to an implementation on a standard processor, speedup of a factor 10 to 1000 or more is quite feasible for some algorithms, particularly in signal processing, cryptography, and some classes of routing and planning problems. In Section 3 we describe some existing applications of reconfigurable computing.

An important obstacle to mainstream acceptance of FPGAs is the inaccessibility of their programming languages. However, we are convinced that there is no fundamental

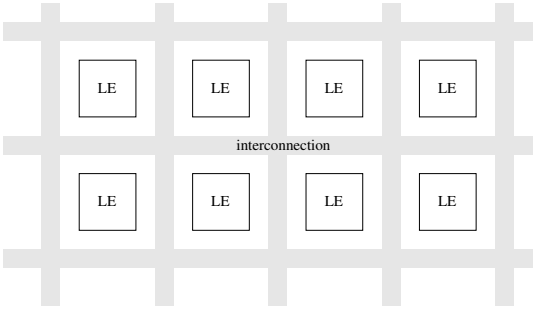

Figure 1: A simple field-programmable gate array, consisting of a grid of logical elements and interconnection hardware.

reason that FPGA programming should be harder.

A common flaw is that an inappropriate programming model—imperative programming—is used. Since translation from an imperative program to reconfigurable logic is difficult, it often results in software that is too large and/or too slow, or cannot be translated at all.

To address the problem, we base our programming environment on a different programming model: synchronous computing. Although this results in a very different programming language, we believe that such a language is as easy to understand as a language such as C or Java.

The purpose of the paper is to show that it is possible to provide a programming language for FPGAs that is as accessible and convenient as standard programming languages. To this end, we have designed and implemented a simple programming language, called Crystal. We will present the language, and we will use it to solve a number of programming tasks.

2 FPGA architecture

In its simplest form, a Field Programmable Gate Array (FPGA) consists of a grid of *logical elements* (LEs) surrounded by interconnection hardware (Fig. 1). The interconnection hardware provides connections between the inputs and outputs of the logical elements. It contains switches that allow the interconnections to be configured to implement a specific wiring of the logical elements.

A simple logical element (Fig. 2) consists of a lookup table to implements a simple logical function, and a register to store a single bit.

The combination of programmable LEs and programmable interconnection allows a FPGA to implement arbitrary digital circuits. Moreover, since the logical elements and interconnection hardware is programmable with configuration bits that are stored in RAM cells, the implemented digital circuit can be rapidly replaced. The basic FPGA architecture is usually augmented with hardware to provide more efficient implementation for frequently required functionality.

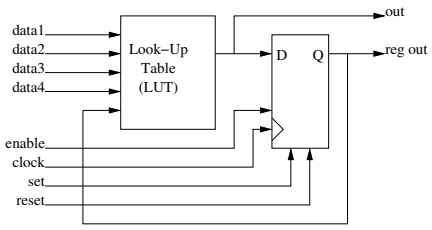

Figure 2: A simple FPGA logical element.

Application	Speedup	Reference
Sub-graph isomorphism (NP-hard)	10	[20]
Nesting problem (NP-hard)	13	[3]
Encryption using IDEA algorithm	14	[26]
Infrared automatic target recognition	20	[23]
VLSI design rule checking	25	[29]
Lempel-Ziv compression	30	[18]
Image processing operations	30	[11]
Parsing (speech recognition)	10 to 70	[7]
DNA matching	10 . . . 100	[14]
Travelling Salesman problem (NP-hard)	70	[13]
Fast Fourier Transform	102	[25]
Enigma cipher machine emulation	400	[30]
Satisfiability problem (NP-hard)	1 to 7000	[1]

Table 1: Example results of using FPGAs for reconfigurable computing.

3 Example applications of FPGA CCMs

Before we consider new approaches to programming FPGAs, it is useful to investigate what has been achieved with existing tools.

A good illustration of potential achievements is a study done by Graham and Nelson [13]. They report a speedup of roughly a factor 70 for an FPGA implementation of a genetic algorithm for the travelling salesman problem. As is typical for FPGA implementations, there are two reasons for the success. The first is the sheer speed of the CCM they implement: in this case it is 9 times faster than the software implementation they compare to. Moreover, multiple engines can be used in parallel: in this case they were able to put 8 engines on the FPGA board they used.

Table 1 lists this result together with a number of other examples of successful applications of FPGA CCMs. The shown speedups are approximate, since the different projects use different software platforms as reference, and often we have summarised a range of results in one characteristic speedup figure. Nevertheless the results indicate that for all these problems highly significant speedups can be achieved by using an FPGA CCM.

Algorithms that benefit most from a CCM implementation tend to be integer operations with much fine-grain parallelism. Good examples can be found in image, signal, and string processing, and in encryption and decryption. NP-hard problems also tend to be good candidates, since next to subtle algorithms they often require sheer processing power.

FPGAs have recently become large enough [27] to accommodate substantial numbers of floating point operations, which makes reconfigurable computing feasible for a new large range of computationally intensive problems.

Since currently FPGA programming is hard, applications tend to be algorithms for which the benefits are overwhelmingly clear. Once FPGA programming is more accessible, we expect that FPGA implementations for which the benefits are not so clear will also be tried, thereby widening the field of applications considerably.

Another important advantage of an FPGA implementation is that it typically consumes only a fraction of the power of an implementation on a standard processor system. This makes reconfigurable computing very attractive for embedded systems.

4 The synchronous computation model

The *synchronous computation* model describes a system as a set of memory elements that store the state of the system, and a set of specifications for the next state of the system. Computation proceeds by computing the next state of all memory elements, and then updating all memory elements in parallel. These two steps are repeated indefinitely.

In a *clocked* synchronous system, the memory updates occur at regular moments in time. Other state update schemes are also possible, e.g. upon a change to the input. However, in this paper we will only deal with clocked synchronous systems.

There is a natural mapping of synchronous systems to reconfigurable hardware: the memory elements map to registers, and the computations for the next state map to combinatorial logic. This natural mapping allows a programmer to work 'close to the metal', in the same way that C allows this for standard processors.

The synchronous model has been used for both hardware and software description. For software, synchronous computation is popular in the area of *reactive computing*, where guaranteed response times are required. For this purpose languages such as Lustre [16], Signal [15], and Esterel [4] have been developed. They have become popular in the french aerospace industry, where they are used for the development of mission-critical embedded software.

For hardware, the synchronous model is part of standard design practice. Also, synchronous languages like Lustre and Esterel have been translated to hardware [31].

The language shown in this paper, Crystal, is similar to Lustre, but as a more C-like syntax, and has a number of additional language constructs. We have implemented a translation from Crystal to VHDL, and hence to hardware.

5 The Crystal programming language

In this section we describe a simple but effective programming language for FPGAs based on the model of synchronous logic. We call this language 'Crystal'. We will use it in later sections for a number of example projects.

Crystal describes *synchronous* digital circuits. A synchronous digital circuit consists of a number of registers that contain the state of the circuit, and a number of logical gates that compute the next value for each of the registers. The state of the registers is updated at every pulse of a global clock that is implicitly clocks all registers. In Crystal synchronous circuits are described by declaring the registers and the equations to compute the next state.

At the lowest level, Crystal supports constants and variables of boolean and integer types, and a set of operations on these expressions. The variables represent *connections*, and the operations represent digital circuits. For example, the Crystal definition

```
a = b + c + 1;
```

defines a circuit that has `b` and `c` as inputs, and `a` as output, representing the sum of the two inputs and the constant 1. Next to the + operator shown here, Crystal supports most other C operators (e.g. `*` and `&`). The lefthand side of a connection definition is called a *connection*, the righthand side is an *expression*.

Optionally, the type of a connection or expression can be indicated with the `:` operator. For example:

```
a:int = b + c;
```

The integer type is represented by `int`, the boolean type is represented by `bool`.

Although any connection or expression can be annotated with a type, in most cases Crystal is able to infer the type itself. Only for register variables and formal parameters of components (both described later on) a type specification is mandatory.

Next to standard variables, Crystal supports *register* variables. Register variables must be defined with a *register definition*. For example:

```
reg count:int = 1;
```

defines a register variable `count` that has the initial value `1`. If no initial value is specified, Crystal uses a default (0 or `false`). All registers in an Crystal circuit are regularly updated. Conceptually, there is a global clock that produces regular 'ticks'. At every tick, all registers are updated. The new value of a register is represented as the register name followed by a single quote (`'`), For example, the connection definition

```
count' = count+1;
```

defines a circuit that increments `count` at every clock tick.

As stated, Crystal supports most C operators. In particular, it supports the `?:` operator. For example,

```
mx = (a>b)?a:b;
```

variable `mx` represents the maximum of `a` and `b`. This operator is also frequently used to define conditional updates to registers. For example, the connection definition

```
count' = enable?count+1:count;
```

defines a circuit that increments `count` at every clock tick at which `enable` is `true`.

To improve readability, Crystal also supports *conditional* definitions. These definitions are equivalent with one or more connection expressions with a `?:` operator. For example, the definition

```
if( enable ){
    count' = count+1;
}
else {
    count' = count;
}
```

is equivalent to the previous definition. Moreover, for register variables conditional connection definitions that propagate the current value of the register can be omitted. Thus, the definition

```
if( enable ){
    count' = count+1;
}
```

Is equivalent to the previous two definitions. Finally, if the then or else part of a conditional definition contains exactly one definition, the brackets can be omitted. Thus,

```
if( enable )
    count' = count+1;
```

is also allowed.

Next to the simple types `int` and `bool`, Crystal supports *tuples*. A tuple groups a list of connections or expressions into a single entity. For example,

```
result = [valid:bool,value:int];
```

defines variable `result` to be a tuple of two elements. Generally, tuples can be constructed by surrounding a comma-separated list of expressions with square brackets. Similarly, a tuple type is written as a comma-separated list of types surrounded by square brackets. For example, since in the above example both `valid` and `value` have a type annotation, we can infer that the type of `result` is `[bool,int]`.

As a refinement, for a type `T` and an integer constant `n`, a type expression of the form `T^n` is equivalent to `n` repetitions of type `T`. Thus, `[bool^3,int]` is equivalent to `[bool,bool,bool,int]`.

Tuples connection expressions are also allowed: they can be interpreted as a pattern that matches elements of the tuple expression they are connected to. For example, the `result` variable defined above can be 'picked apart' by the following connection definition:

```
[ok,val] = result;
```

A set of definitions can be grouped into a *component declaration*. Such a component consists of a list of definitions, and an input and an output. Both the input and the output are a formal parameter pattern, where a formal parameter pattern is either a variable name with a type annotation, or a tuple of formal parameter patterns.

For example, the following defines a component `counter`:

```
def counter [enable:bool,reset:bool] => val:int
    reg val:int = 0;

    if( reset )
        val' = 0;
    else if( enable )
        val'  = val+1;
```

The component has two inputs, `enable` and `reset`, and one output, `val`. The output is connected to a register that contains the current value of the counter. Initially it is 0. If `reset` is true, at the next clock tick `val` be set to 0. If `reset` is false, but `enable` is true, `val` will at the next clock tick be incremented, else it will retain its current value.

Note that the input consists of a tuple of formal parameter patterns, whereas the output has a single variable name as formal parameter pattern.

Once we have defined a component, we can use it. For example:

```
gnomes = counter [sawGnome,daylight];
```

A component may not use itself directly or indirectly.

Next to the component declaration shown above, Crystal also supports constant declarations, for example

```
const GnomeAlarmCount:int = 42;
```

Such a constant can then be used in expressions. For example:

```
def gnomeAlarm [sawGnome:bool,daylight:bool] => panic:bool
    gnomes = counter [sawGnome,daylight];
    panic = (gnomes>=GnomeAlarmCount);
```

Crystal also allows symbolic names for type expressions by providing a `typedef` declaration. For example:

```
typedef pattern:[int^8];
```

Finally, Crystal supports *enumerated types*. Such a type is defined with a `enum` declaration. For example:

```
enum Answer { Yes, No, Maybe };
```

6 A LRU cache

As the first of a series of larger examples, we now show a circuit to implement a Least Recently Used (LRU) cache. In particular, we implement a circuit that returns true when the current value at the input is equal to one of the values in the cache. When a value is stored that is not in the cache, the least recently used value is removed to make room. If the newly stored value is already in the cache, only the recentness adminstration is updated.

The basic idea behind the LRU cache is that the elements are stored in arrival order. Thus, whenever a new value is stored, all values in the cache are moved to the next older cache slot. However, if a cache slot contains the same value as the new one, that slot will be overwritten, and all older cache slots are left unchanged.

A single slot in this cache can be implemented with the following component:

```
def CacheSlot [val:int,holdin:bool,shiftin:int]
  => [match:bool,holdout:bool,shiftout:int]
    reg v:int = 0;

    match = (val == v);
    shiftout = v;
    holdout = match | holdin;
    if( !holdin )
        v' = shiftin;
```

Input `val` is the new value to be stored in the cache. Output `match` is true when the current value in this element is equal to `val`. Input `shiftin` is the value from the previous cache element (i.e. the one storing the next more recent value). Input `holdin` indicates whether the previous cache element should be loaded. Also, the cell has a `shiftout` output that is connected to the next `shiftin`, and an output `holdout` that is connected to the next `holdin`.

The `Cache` component chains a number of `CacheSlot` components, and calculates an overall `match` result from all individual match results:

```
def Cache [val:int] => [match:bool]
    [match0,h0,sv0] = CacheSlot [val,h1,sv1];
    [match1,h1,sv1] = CacheSlot [val,h2,sv2];
    [match2,h2,sv2] = CacheSlot [val,h3,sv3];
    [match3,h3,sv3] = CacheSlot [val,false,val];
    match = match0 | match1 | match2 | match3;
```

Input `val` is the value to be stored, output `match` is true when the current value is in the cache.

7 A Collatz conjecture tester

The Collatz conjecture states that the function

$$f \in \mathcal{N} \rightarrow \mathcal{N}$$
$$f(x) = \begin{cases} 1, & \text{x=1} \\ x/2, & x \text{ is even} \\ 3 \cdot x + 1, & \text{otherwise} \end{cases}$$

has 1 as fixed point for its entire domain. Thus, given any number in $\mathcal{N}$, by repeatedly applying function f to it, eventually 1 will be reached.

No proof is known for this conjecture, but it has been checked by computer for a large range of numbers. In this section we will show a simple Crystal circuit that checks the conjecture for a limited range of numbers.

The following component implements the computational process of the Collatz conjecture:

```
def collatz [load:bool,val:int] => done:bool
    reg x:int = 1;

    done = (x == 1);
    if( load )
```

```
        x' = val;
    else
        if( x != 1 )
            if( (x%2) == 0 )
                // x is even.
                x' = x/2;
            else
                // x is odd.
                x' = x+x+x+1;
```

When `load` is true, the value `val` is loaded in the internal register `x`. When `load` is false, a step in the Collatz computation is done on `x`. The output `done` is true when `x` is 1.

Since we don't know for sure that the Collatz computation will reach 1 (after all, this is the conjecture we're testing), we use the `collatz` component shown above to construct a component that tries at most `MAXSTEPS` steps in the computation, and then indicates failure.

```
const MAXSTEPS:int = 100;

def limitedCollatz [load:bool,val:int] => [done:bool,ok:bool]
    reg steps:int = 0;

    ok = collatz [load,val];
    if( load )
        steps' = 0;
    else
        steps' = steps+1;
    timeout = steps>=MAXSTEPS;
    done = ok | timeout;
```

When `load` is true, the value `val` is loaded. When `load` is false, a step in the Collatz computation is done on `x`. When the `collatz` engine reaches 1, output `ok` is true. When output `ok` is true, or when `MAXSTEPS` steps in the computation have been done, the output `done` is true.

Note that as long as no new value is loaded, the `collatz` engine continues, even after `MAXSTEPS` step.

Using the `limitedCollatz` component shown above, we can now construct a driver component that tests the Collatz conjecture for the numbers between 1 and `MAXVAL`. The component runs a number of `limitedCollatz` components in parallel. Whenever an engine is empty, it is given the next value to run the computation on.

```
const MAXVAL:int = 8000;

def collatzFarmer [] => [resval:int,verdict:bool,valid:bool,alldone:bool]
    reg val:int = 1;
    reg verdict:bool = false;
    reg resval:int = 0;
    reg valid:bool = false;
    reg val0:int = 0;
    reg val1:int = 0;
    reg val2:int = 0;
    reg val3:int = 0;
```

```
[done0,ok0] = limitedCollatz[load0,val];
[done1,ok1] = limitedCollatz[load1,val];
[done2,ok2] = limitedCollatz[load2,val];
[done3,ok3] = limitedCollatz[load3,val];
if( done0 ){
    verdict' = ok0;
    resval' = val0;
    val0' = val;
    val' = val+1;
    valid' = true;
    load0 = true; load1 = false; load2 = false; load3 = false;
}
else if( done1 ){
    verdict' = ok1;
    resval' = val1;
    val1' = val;
    val' = val+1;
    valid' = true;
    load0 = false; load1 = true; load2 = false; load3 = false;
}
else if( done2 ){
    verdict' = ok2;
    resval' = val2;
    val2' = val;
    val' = val+1;
    valid' = true;
    load0 = false; load1 = false; load2 = true; load3 = false;
}
else if( done3 ){
    verdict' = ok3;
    resval' = val3;
    val3' = val;
    val' = val+1;
    valid' = true;
    load0 = false; load1 = false; load2 = false; load3 = true;
}
else {
    valid' = false;
    load0 = false; load1 = false; load2 = false; load3 = false;
}
alldone = val>=MAXVAL;
```

Register `val` stores the next value to run the process on. Output `alldone` is true when `val` has reached `MAXVAL`. Whenever one of the `limitedCollatz` engines produces a result, as indicated by the variables `done0` to `done3`, that result is loaded into the output registers `resval`, `verdict`, and `valid`. Output `valid` indicates that there is a valid result, `resval` is the number for which there is a result, and `verdict` indicates whether the engine has reached 1[1]. If there is an idle engine, it is loaded with the next value to test.

The registers `val0` to `val3` store the starting value of each `limitedCollatz` engine, so that this value can be loaded into `resval` upon completion of the compu-

[1]Since we have put a limit on the number of computational steps, a false value on `verdict` does not mean the machine has found a counter-example of the Collatz conjecture.

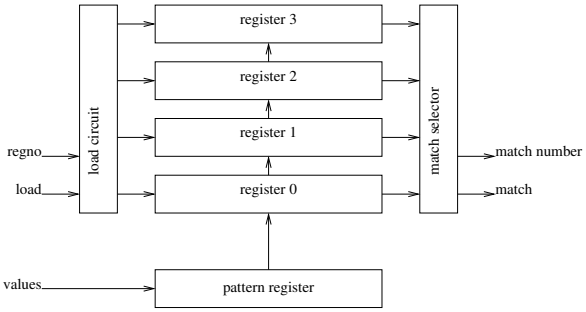

Figure 3: An example CCM: a DNA sequence matcher.

tation.

8 A DNA matcher

DNA (*deoxyribonucleic acid*) macromolecules consist of long linear chains of components, called *nucleotides*. There are four nucleotides, denoted by the letters A, C, G and T. Using gel electrophoresis, the sequence of DNA strands of up to 700 nucleotides can be determined directly. For longer strands, of say 10,000 to 100,000 nucleotides, an indirect approach is necessary. In the *shotgun sequencing* method, multiple copies of the long strands are broken into fragments that are sequenced directly. The sequence of a large strand is then determined by 'fusing' overlapping fragments by computer.

One of the most time-consuming steps in this reassembly process is finding overlapping sequences, because a large number of comparisons must be made. In this section we show a custom computing machine that allows a large number of these comparisons to be done in parallel, see Fig. 3.

To use the CCM, DNA sequences are streamed into it one nucleotide at a time. The machine stores the most recent 8 nucleotides, and compares this sequence with a number of reference sequences. When one of the reference sequences matches, the position of the matching reference sequence is output. The registers that store there reference sequences are loaded by external command.

The following component stores a reference sequence:

```
typedef dnaPat:[int^4];

def dnaregister [load:bool,in:dnaPat] => match:bool
    reg pat:dnaPat;

    if( load ){
        pat' = in;
    }
    [r0,r1,r2,r3] = pat;
    [i0,i1,i2,i3] = in;
```

```
    match = (r0==i0) & (r1==i1) & (r2==i2) & (r3==i3);
```

The component is loaded when the `load` signal is true. The value of `in` is then stored in the internal register `reg`. The input sequence `in` is compared against the reference sequence in `reg`. If it matches, the `match` value is asserted.

The `dnamatcher` component uses four `dnaregisters`. (This is to keep the listing small; a realistic engine would use many more registers.)

```
def dnamatcher [load:bool,val:int,regno:int] => [match:bool,regno:int]
    reg v0:int;
    reg v1:int;
    reg v2:int;
    reg v3:int;

    v = [v0,v1,v2,v3];
    l0 = load & regno == 0;
    l1 = load & regno == 1;
    l2 = load & regno == 2;
    l3 = load & regno == 3;
    m0 = dnaregister[l0,v];
    m1 = dnaregister[l1,v];
    m2 = dnaregister[l2,v];
    m3 = dnaregister[l3,v];
    v0' = val;
    v1' = v0;
    v2' = v1;
    v3' = v2;
    match = m0 | m1 | m2 | m3;
    regno = m0?0 : m1?1 : m2?2 : m3?3 : 0;
```

Every clock cycle, `dnamatcher` receives a new DNA base that is put in the shift register formed by `v0` to `v3`. The value in the shift register is compared by all `dnaregister` components. If one of the sequences matches, the output `match` is asserted, and the output `reg` is set to the number of the register that contains the match.

9 A SAT solver

Boolean satisfiability (SAT) is a well-known NP-complete problem. Given a symbolic Boolean expression, the problem requires that a set of assignments to the variables in the expression are found for which the expression evaluates to true, or that is established that no such set of assignments exists.

In this section we will develop a simple engine that solves the SAT problem for a Boolean expression that can be downloaded by the user. Our SAT solver is inspired by [1], but in our implementation the expression to be solved is downloaded, whereas in [1] the Boolean expression is directly implemented in hardware. Therefore, their solver engine is specific for a certain Boolean expression, and must be compiled again for a different expression.

Traditionally, SAT solvers represent Boolean expressions in Conjunctive Normal Form (CNF). Expressions in CNF consist of a list of *clauses* that all must be satisfied. Each clause consists of a list of literals, one of which must be satisfied. A literal is

a variable or its negation. For example, a and $\neg a$ are literals, $a \vee \neg b$ is a clause, and $(a \vee \neg b) \wedge (b \vee c \vee \neg d)$ is a CNF expression.

A brute force solution to this problem is to try all possible assignments, and see if one of the assignments yields true. However, in the worst case (the expression cannot be satisfied), for a Boolean expression with n variables this approach requires 2^n assignments to be tested.

A better strategy is to try to eliminate large parts of the solution space at once by evaluating partial assignments. For example, the expression $(a \vee \neg b) \wedge (b \vee c \vee \neg d)$ can never be satisfied with the partial assignment $a = \mathsf{F}, b = \mathsf{T}$, since the first clause cannot be satisfied, no matter what we assign to c and d. However, the partial assignment $a = \mathsf{T}, c = \mathsf{T}$ satisfies the entire Boolean expression, no matter what we assign to b and d.

Based on this principle, we implement a solver that evaluates partial assignments, and that only tries a more specific assignment if the more general one does not yield a definite answer.

In our circuit we represent literals in the Boolean expression, and variables in the assignment, with the following `enum`:

```
enum V = { V_DONTCARE, V_TRUE, V_FALSE }
```

In our descriptions of the engine we will write these as $\star$, T, and F respectively. A CNF clause can be represented by a vector of `V` values. For example, consider the following Boolean expression:

$$(a \vee \neg b \vee d) \wedge (\neg a \vee \neg b \vee c \vee e) \wedge (c \vee d) \tag{1}$$

This expression consists of three clauses, and can be represented by the following list of `V` vectors:

$$\langle \mathsf{T}, \mathsf{F}, \star, \mathsf{T}, \star \rangle, \langle \mathsf{F}, \mathsf{F}, \mathsf{T}, \star, \mathsf{T} \rangle, \langle \star, \star, \mathsf{T}, \mathsf{T}, \star \rangle \tag{2}$$

In a similar way (partial) assignments can be represented. For example, the assignment

$$b = \mathsf{F}, c = \mathsf{T} \tag{3}$$

satisfies Eqn. 1. This assignment can be represented as the `V` vector $\langle \star, \mathsf{F}, \mathsf{T}, \star, \star \rangle$.

We will now describe the components of the SAT solver.

A literal is stored using the following component:

```
def literal [load:bool,loadval:V,val:V] => [sat:bool,possible:bool]
    reg l:V = V_DONTCARE;

    if ( load )
        l' = loadval;
    sat = (val != V_DONTCARE) & (val == l);
    possible = (val == V_DONTCARE) & (l != V_DONTCARE);
```

Register `l` stores the literal. When `load` is true, a new literal is loaded from `loadval` into `l`. Also, the component gives a verdict on the value on the `val` port: `sat` is true when the `val` satisfies the stored value. The `possible` output is true when the literal assignment could be satisfied with a more specific value than the current value of `val`.

Since a stored V_DONTCARE cannot be satisfied, for this stored value possible is always false.

Using the literal component, we can construct a component to store an entire clause. As shown here, a clause may contain at most 5 literals. This is for the sake of brevity; a realistic implementation would use a much larger capacity.

```
def clause [load:bool,loadval:[V^5],val:[V^5]]
  => [sat:bool,conflict:bool]
    reg inactive:bool = true;
    [sat0,pos0] = literal[load,loadval[0],val[0]];
    [sat1,pos1] = literal[load,loadval[1],val[1]];
    [sat2,pos2] = literal[load,loadval[2],val[2]];
    [sat3,pos3] = literal[load,loadval[3],val[3]];
    [sat4,pos4] = literal[load,loadval[4],val[4]];
    sat = inactive | sat0 | sat1 | sat2 | sat3 | sat4;
    possible = inactive | pos0 | pos1 | pos2 | pos3 | pos4;
    conflict = !possible;
    if( load )
        inactive' = false;
```

The clause is loaded from the loadval input when load is true. The component gives a verdict on the vector of assignments on the val port: sat is true when the displayed assignment satisfies the stored clause. The conflict output is true when the displayed assignment conflicts with the stored clause. That is, if none of the variables that are currently at V_DONTCARE can be set to a value that will a satisfy the clause. Note that a separate register, inactive, is used to disable clauses that have never been loaded, because otherwise they would interfere with the overall evaluation of the Boolean expression.

Using the clause component, we can construct a component to store the entire Boolean expression:

```
def clauses [load:bool,loadval:[V^5],val:[V^5]]
  => [sat:bool,conflict:bool,full:bool]
    reg clauseno:int = 0;
    [sat0,conf0] = clause [load & (clauseno==0),loadval,val];
    [sat1,conf1] = clause [load & (clauseno==1),loadval,val];
    [sat2,conf2] = clause [load & (clauseno==2),loadval,val];
    [sat3,conf3] = clause [load & (clauseno==3),loadval,val];
    [sat4,conf4] = clause [load & (clauseno==4),loadval,val];
    [sat5,conf5] = clause [load & (clauseno==5),loadval,val];
    [sat6,conf6] = clause [load & (clauseno==6),loadval,val];
    [sat7,conf7] = clause [load & (clauseno==7),loadval,val];
    sat = sat0 & sat1 & sat2 & sat3 & sat4 & sat5 & sat6 & sat7;
    conflict = conf0|conf1|conf2|conf3|conf4|conf5|conf6|conf7;
    full = (clauseno>=8);
    if( load & !full )
        clauseno' = clauseno+1;
```

The register clauseno stores the number of the next clause to be loaded. When all 8 clauses are filled, the full output is true. A new clause is loaded from the loadval input when load is true. The component gives a verdict on the list of values on the val input: sat is true when all clauses are satisfied, conflict is true when the displayed assignment conflicts with the Boolean expression. The shown circuit can

store a Boolean expression with at most 8 clauses. Again, a realistic implementation would have a larger capacity.

To enumerate the partial assignments, we define a chain of enumeration cells, one for each variable. Each cell remains in an idle state, and outputs `V_DONTCARE` for its variable, until it is activated by the previous cell in the chain. Once activated, it outputs `V_FALSE` for its variable. If that causes the `prune` input to be true, in the next step it outputs `V_TRUE`. However, if for the value `V_FALSE` `prune` is false, the cell activates the next cell in the chain, waits until that cell is finished, and then outputs `V_TRUE`. Once the `V_TRUE` value is displayed the Boolean expression is again evaluated, and perhaps the next cell is activated again. After that the cell signals to the previous one that it has finished, by setting `done` to true, and becomes idle again.

The state of a cell is represented by the following `enum`:

```
enum cellstate = { CS_IDLE, CS_ACTIVE, CS_WAITING }
```

An enumeration cell is implemented with the following component:

```
def valueCell [reset:bool,try:bool,nextdone:bool,prune:bool]
  => [val:V,nexttry:bool,done:bool]
    reg val:V = V_DONTCARE;
    reg state:cellstate = CS_IDLE;

    done = state == CS_IDLE;
    nexttry = state == CS_ACTIVE & !prune;
    if( reset ){
        val' = V_DONTCARE;
        state' = CS_IDLE;
    }
    else {
        if( state == CS_IDLE & try ){
            val' = V_FALSE;
            state' = CS_ACTIVE;
        }
        else {
            if(
                (state == CS_ACTIVE & prune) |
                (state == CS_WAITING & nextdone)
            )
            {
                // Advance to next state.
                if( val == V_FALSE ){
                    val' = V_TRUE;
                    state' = CS_ACTIVE;
                }
                else {
                    val' = V_DONTCARE;
                    state' = CS_IDLE;
                }
            }
            else {
                if( state == CS_ACTIVE ){
                    state' = CS_WAITING;
                }
            }
        }
    }
```

Finally, using the previously declared components, we can now declare a component to implement the entire SAT solver. It contains the chain of value cells, connected to the `clauses` component, and the circuitry to start a search each time a new clause is loaded.

```
def satSolver [load:bool,val:[V^5]]
  => [valid:bool,done:bool,res:[V^5],overflow:bool]
    reg go:bool = true;
    reset = load;
    [v0,try1,done] = valueCell [reset,go,done1,prune];
    [v1,try2,done1] = valueCell [reset,try1,done2,prune];
    [v2,try3,done2] = valueCell [reset,try2,done3,prune];
    [v3,try4,done3] = valueCell [reset,try3,done4,prune];
    [v4,try5,done4] = valueCell [reset,try4,true,prune];
    prune = sat | conflict;
    [sat,conflict,overflow] = clauses [load,val,res];
    if( reset )
        go' = true;
    else if( done )
        go' = false;
    valid = (!reset) & sat;
    res = [v0,v1,v2,v3,v4];
```

The search process is started upon loading a clause into the solver machine. When `valid` is true, the `res` output is an assignment that satisfies the loaded Boolean expression. Once the search for a solution has been completed, `done` is set to true.

10 Related work

10.1 Mainstream hardware description languages

Nowadays most hardware design, including design for FPGAs, is done in either Verilog [21] or VHDL [22]. Both these HDLs are tools for expert digital circuit designers to exploit FPGAs as efficiently as possible. Considerable knowledge of digital circuit design is required to use these languages, as is a thorough training in the particular circuit synthesis tool for the target FPGA.

Also, the model of computation of both VHDL and Verilog is somewhat unclear: the simulator and circuit generator part of the language use an imperative programming model, but even the hardware description language is influenced by the imperative model of computation.

All in all, the low-level nature of these languages, their unfocused model of computation, their complexity, and the lack of portability between implementations make these programming languages unsuitable as accessible languages for FPGA programming.

10.2 Imperative programming languages

Considering the overwhelming dominance of imperative programming language like C, C++, and Java, it is very tempting to try and reuse them for hardware description. That way, at least part of the knowledge of using the programming language can be used

for hardware development. However, practical considerations require that hardware compilation is confined to a subset of the language. Moreover, usually a number of new language constructs are added, if only as library calls that are recognized treated in a special way by the compileer.

This approach is used in languages such as Handel-C [6], SpecC [9, 33], and SystemC [12, 34].

The obvious advantage of this approach is that an existing and familiar language can be used. However, the programmer must be aware that the program will be used for circuit synthesis: it is not possible to simply compile existing software to hardware. Consequently, the user is still required to learn an essentially new language.

Moreover, since the imperative programming model requires a complex translation process, the quality of the resulting circuits is very dependent on particulars in the compiler. This makes the compiler difficult to use.

A similar approach has been proposed for functional programming languages such as Haskell [17], and FP [32]. However, many of the same objections apply.

10.3 FPGA-specific and other HDL languages

A few researchers have implemented programming languages specifically designed for FPGA programming. A mature example is SA-C [10, 11] ("sassy"). In the words of the developers, it is "a single-assignment variant of the C programming language designed to exploit both coarse-grain and fine-grain parallelism in computer vision and image processing applications".

SA-C addresses a number of problems in the other approaches: its single-assignment model of computation is more abstract than that of Verilog and VHDL, and SA-C is simpler to synthesise than standard programming languages. However, it was explicitly designed for image processing algorithms only, and is limited to applying 'window' operations on two-dimensional arrays.

Similarly, RaPiD-C [8] was designed for the RaPiD reconfigurable pipelined datapath architecture, but compilation to general FPGAs has also been implemented. RaPiD-C is an imperative language that extends a subset of C with domain-specific constructs such as `Ram` and `Pipe` data types and `Seq`, `Par`, `Signal` and `Wait` control structures. Its imperative basis is a disadvantage for compilation to standard FPGAs.

Pebble [28] was explicitly designed to allow programming of FPGAs with a simpler language than Verilog or VHDL. They have achieved this goal admirably: their simple language is indeed able to describe gate-level circuits clearly and concisely, and their generator language supports repetition and conditional circuit generation. Moreover, the generator language is clearly separated from the structural language. The main drawback of Pebble is that it is still a very low-level language. The user must specify the circuit down to the gate level, only augmented by a target-dependent library of primitive components. The clock must also be specified explicitly.

A related class of languages (e.g. NSL [19]) is designed to program systolic arrays. However, since they target simpler hardware, they are usually not general enough for our purposes.

Ruby [24] is a hardware description language based on the functional language FP. Like FP, the user constructs circuits from a set of atomic components and a set of

circuit combinators (cascade, parallel, repeat, etc.). The advantage of this approach is that it allows formal reasoning about circuits and about transformations on the circuits. The disadvantage is that the interconnections between components must be described as combinator functions; a very painful process compared to using named connections.

10.4 Synchronous languages

Crystal is squarely based on the synchronous programming model. There are a number of other languages that also use that model. In particular, the language LUSTRE [16] is very similar to our approach, since it also uses a declarative notation to specify the next state of a system.

but we have specifically designed the language for easy compilation to reconfigurable hardware. In particular, we require that equations specify the next value of a signal, whereas in LUSTRE equations refer to previous values of a signal.

Data-flow languages such as Sisal [5] are related to synchronous languages. However, these languages have been developed for data-flow machines, and are not directly useful for FPGA programming, since they allow the data-flow graph to be changed dynamically.

11 Future work

Although we have been able to address real problems, the examples in this paper suggest a number of changes and additions to the language that could help a great deal in allowing large-scale programming.

- Repetitive structures, such as the cache elements in the LRU cache (Sec. 6), the pattern registers in the DNA matcher (Sec. 8), etc., should be described more consisely.
- For efficiency reasons, the language should have a much more precise type system, so that e.g. 5-bit integers could be declared.
- The language should provide a library of useful components.
- The language should provide a module system, for example derived from that of Modula 2 or Java.
- The language should support assertions and design by contract.

More speculatively, it is useful to consider adopting other features from other programming languages, such as dynamic memory allocation, dynamic circuit reconfiguration, object orientation, and exceptions. All of these concepts will require some adaptation before they can be integrated in a language such as Crystal. However, the these concepts address a need in imperative programming, and it is important to consider how the same needs can be addressed in the context of Crystal.

12 Conclusions

In this paper we have described a programming language, Crystal, that is as simple to learn as standard languages like C and Java, but that is efficiently translated to reconfigurable hardware. In its current form it lacks a lot of the features of mainstream programming languages, but even without them we were able to effectively describe circuits that solve real problems: both the DNA matcher and the SAT solver address real problems and promise significant speedups compared to normal approaches.

The Crystal language requires a very different way to look at things, but there is no reason that people could not learn to use it as effectively as C or Java.

By adding more features, we expect to arrive at a language that is as effective a tool for programming reconfigurable hardware as C and Java are for standard processors.

Acknowledgements

The author wishes to thank Arjan van Gemund for his encouragement and helpful discussions.

References

[1] M. Abramovici, J.T. de Sousa, and D. Saab. A massively-parallel easily-scalable satisfiability solver using reconfigurable hardware. In K.L. Pocek and J.M. Arnold, editors, *Proc. of the Design Automation Conference*, pages 684–690, June 1999.

[2] Altera. *Altera device information*, October 2001. http://www.altera.com/products/devices/dev-index.jsp.

[3] J.C. Alves, J.C. Ferreira, C. Albuquerque, J.F. Oliveira, J.S. Ferreira, and J.S. Matos. FAFNER – accelerating nesting problems with FPGAs. In K.L. Pocek and J.M. Arnold, editors, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 168–176, April 1999.

[4] Gérard Berry. *The Esterel v5 Language Primer*. INRIA, April 1999.

[5] A.P.W. Böhm, D.C. Cann, R.R. Oldehoeft, and J.T. Feo. SISAL Reference Manual Language Version 2.0. CS 91-118, Colorado State University, Fort Collins, Colorado, USA, 1991.

[6] Celoxia. *Handel-C Language Reference Manual*. available from www.celoxia.com/home.htm.

[7] C. Ciressan, E. Sanchez, M. Rajman, and J.-C. Chappelier. An FPGA-based coprocessor for the parsing of context-free grammars. In B.L. Hutchings, editor, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 236–245, April 2000.

[8] Darren C. Cronquist, Paul Franklin, Stefan G. Berg, and Carl Ebeling. Specifying and compiling applications for RaPiD. In Kenneth L. Pocek and Jeffrey Arnold, editors, *IEEE Symposium on FPGAs for Custom Computing Machines*, pages 116–125, Los Alamitos, CA, 1998. IEEE Comp. Soc. Press.

[9] Rainer Dömer, Andreas Gerstlauer, and Daniel Gajski. SpecC language reference manual. Technical report, SpecC Technology Open Consortium, March 2001.

[10] Bruce Draper, Walid Najjar, Wim Böhm, Jeff Hammes, Bob Rinker, Charlie Ross, and Monica Chawathe José Bins. Compiling and optimizing image processing algorithms for FPGAs. In *Proc. of the IEEE International Workshop on Computer Architecture for Machine Perception (CAMP)*, September 2000.

[11] Bruce A. Draper, A.P. Willem Böhm, Jeff Hammes, Walid Najjar, J. Ross Beveridge, Charlie Ross, Monica Chawathe, Mitesh Desai, and José Bins. Compiling SA-C programs to FPGAs: Performance results. In *Proc. of the International Conference on Vision Systems*, pages 220–235, June 2001.

[12] Joachim Gerlach and Wolfgang Rosenstiel. System level design using the SystemC modeling platform. In *SNUG Europe*, 2000.

[13] Paul Graham and Brent Nelson. A hardware genetic algorithm for the travelling salesman problem on SPLASH 2. In Will Moore and Wayne Luk, editors, *Field-Programmable Logic and Applications*, pages 352–361. Springer-Verlag, Berlin, 1995.

[14] P. Guerdoux-Jamet and D. Lavenier. Systolic filter for fast DNA similarity search. In *Proc. of The International Conference on Application Specific Array Processors (ASAP'95)*, pages 145–156, July 1995.

[15] P. Le Guernic, M. Le Borgne, T. Gauthier, and C. Le Maire. Programming real time applications with signal. *Proceedings of the IEEE, Special Issue*, September 1991.

[16] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. The synchronous data flow programming language LUSTRE. *Proc. of the IEEE*, 79(9):1305–1320, September 1991.

[17] John Hawkins and Ali E. Abdallah. An overview of systematic development of parallel systems for reconfigurable hardware. In Burkhard Monien and Rainer Feldmann, editors, *Euro-Par*, pages 615–619. Springer, August 2002. LNCS 2400.

[18] W.-J. Huang, N. Saxena, and E.J. McCluskey. A reliable LZ data compressor on reconfigurable coprocessors. In B.L. Hutchings, editor, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 249–258, April 2000.

[19] Richard Hugley. Programming systolic arrays. In *Proc. Int. Conf. Application-Specific Array Processors*, pages 604–618. IEEE Com. Soc., August 1992.

[20] S. Ichikawa, L. Udorn, and K. Konishi. Hardware accelerator for subgraph isomorphism problems. In B.L. Hutchings, editor, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 283–284, April 2000.

[21] IEEE Com. Soc. *IEEE Standard Verilog Hardware Description Language*, September 2001. IEEE Std 1364-2001.

[22] IEEE Comp. Soc. *IEEE Standard VHDL Language Reference Manual*, May 2002. IEEE Std 1076-2002.

[23] J. Jean, Xiejun Liang, B. Drozd, and K. Tomko. Accelerating an IR automatic target recognition application with FPGAs. In K.L. Pocek and J.M. Arnold, editors, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 290–291, April 1999.

[24] Geraint Jones and Mary Sheeran. Designing arithmetic circuits by refinement in Ruby. In R. S. Bird, C. C. Morgan, and J. C. P. Woodcock, editors, *Mathematics of Program Construction*, pages 388–395. Springer Verlag, 1993. LNCS 669.

[25] D. Lau, A. Schneider, M.D. Ercegovac, and J. Villasenor. FPGA-based structures for on-line FFT and DCT. In K.L. Pocek and J.M. Arnold, editors, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 310–311, April 1999.

[26] M.P. Leong, O.Y.H. Cheung, K.H. Tsoi, and P.H.W. Leong. A bit-serial implementation of the international data encryption algorithm idea. In B.L. Hutchings, editor, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 122–131, April 2000.

[27] Walter B. Ligon, Scott McMillan, Greg Monn, Kevin Schoonover, Fred Stivers, and Keith D. Underwood. A re-evaluation of the practicality of floating-point operations on FPGAs. In *Proc. of the IEEE Symposium on FPGAs for Custom Computing Machines*, pages 206–215, April 1998.

[28] W. Luk and S. McKeever. Pebble: a language for parametrised and reconfigurable hardware design. In R.W. Hartenstein and A. Keevallik, editors, *Field-Programmable Logic and Applications*, pages 9–18. Springer, 1998. LNCS 1482.

[29] Zhen Luo, M. Martonosi, and P. Ashar. An edge-endpoint-based configurable hardware architecture for VLSI CAD layout design rule checking. In K.L. Pocek and J.M. Arnold, editors, *Proc. of the IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 158–167, April 1999.

[30] C. van Reeuwijk. Enigma code breaking using a field programmable gate array. PDS Technical Report PDS-2002-001, Delft University of Technology, January 2002. www.pds.ewi.tudelft.nl/reports/2002/PDS-2002-001.

[31] F. Rocheteau and N. Halbwachs. Pollux, a lustre-based hardware design environment. In P. Quinton and Y. Robert, editors, *Conference on Algorithms and Parallel VLSI Architectures II*, Chateau de Bonas, June 1991.

[32] M. Sheeran. *muFP, an Algebraic VLSI Design Language*. PhD thesis, University of Oxford, 1983.

[33] SpecC website. URL: www.specc.org.

[34] SystemC website. URL: www.systemc.org.

[35] Xilinx. *Xilinx data sheets*, October 2001. available on www.xilinx.com/partinfo/databook.htm.