International Journal of Reconfigurable Computing

Selected Papers from SPL 2008: Programmable Logic and Applications

Guest Editors: Gustavo Sutter and Richard Katz

Selected Papers from SPL 2008: Programmable Logic and Applications

Selected Papers from SPL 2008: Programmable Logic and Applications

Guest Editors: Gustavo Sutter and Richard Katz

Contents

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 921921, 2 pages
doi:10.1155/2008/921921

Editorial

Selected Papers from SPL 2008: Programmable Logic and Applications

Gustavo Sutter[1] and Richard Katz[2]

[1] *School of Computer Engineering, Autonomous University of Madrid, Carretera de Colmenar Km. 15, 28049 Madrid, Spain*
[2] *Microelectronics and Signal Processing Branch, NASA Goddard Space Flight Center, Greenbelt, MD 20771, USA*

Correspondence should be addressed to Gustavo Sutter, gustavo.sutter@ieee.org

Received 20 November 2008; Accepted 20 November 2008

Field programmable logic devices have traditionally been used as vehicles for prototyping and implementing digital circuits; but beyond this use, continued improvements in device density and functionality have made the technology a mainstream one for implementing large systems and accelerators for specific applications.

Field-programmable gate array (FPGA) is one of the most well-known commercial names of programmable logic. The FPGA technology, marketed in 1984 by a startup company called Xilinx, allowed designers to build complex circuits with virtually zero setup costs, enabling the development of the small-scale products common to most Latin American technological companies.

Nowadays, after more than two decades of progress, programmable logic has become the key technology in digital systems design. Not only FPGAs are now capable of implementing multimillion gate systems working at hundreds of megahertz but also the design costs of custom ASICs have soared to levels where only million-unit projects are profitable, keeping them out of the reach of most companies.

Applications of FPGAs include almost every application that needs a fast electronic system including digital signal processing (DSP) applications, ASIC prototyping, medical imaging, computer vision, speech recognition, cryptography, bioinformatics, aerospace and defence systems, computer hardware emulation, as well as a growing range of other areas.

FPGAs market and applications growing are reflected also in the variety of programmable logic conferences around the world and the number of papers published by the research community.

The Southern Conference on Programmable Logic (SPL, www.splconf.org) is the austral meeting point for research interested in FPGA technology. It started in 2005 as a cooperation project between Spain and Latin America called SURLABS. The project "SURLABs: Joint Latin American FPGA Laboratories" was financed by the Banco Santander Central Hispano. The interest and rapid growth of the conference, since SPL 2007, has been giving the IEEE Circuit and Systems Society (CAS) a technical cosponsorship.

More than 95 papers were submitted to the last IV SPL conference. The 29 selected papers and the 23 short papers presented at the IV SPL were authored by researchers from Argentina, Australia, Belgium, Brazil, Canada, Colombia, France, Germany, Hong Kong, Iran, Italy, México, Peru, Romania, Spain, Taiwan, UK, Uruguay, and USA. These high technical quality works cover aspects of FPGA-based system design like custom DSPs, computer arithmetic, cryptography, control systems, instrumentation, video processing, embedded processors, test, fault tolerance, low-power design, high-level languages, and education. More than 100 researches and student attend this conference.

The selection of articles presented in this special issue is from the SPL2008 (IV Southern Conference on Programmable Logic) held in Bariloche, Argentina, during March 26–28, 2008. Thirty relevant researches help us in the review process to select the final 6 contributions covering topics of high-level languages, wireless sensor network, configurable architectures, signal processing, and arithmetic units.

The editors of this special issue on programmable logic and applications hope that this edition constitutes a contribution of FPGA design, being valuable for electronic engineers and designers.

Gustavo Sutter
Richard Katz

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 674340, 14 pages
doi:10.1155/2008/674340

Research Article

Area Optimisation for Field-Programmable Gate Arrays in SystemC Hardware Compilation

Johan Ditmar,[1] Steve McKeever,[2] and Alex Wilson[3]

[1] *Kellogg College, University of Oxford, 62 Banbury Road, Oxford OX2 6PN, UK*
[2] *Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford OX1 3QD, UK*
[3] *Celoxica Ltd., 66 Milton Park, Abingdon, Oxfordshire OX14 4RX, UK*

Correspondence should be addressed to Johan Ditmar, johan.ditmar@kellogg.ox.ac.uk

Received 21 June 2008; Accepted 18 August 2008

Recommended by Gustavo Sutter

This paper discusses a pair of synthesis algorithms that optimise a SystemC design to minimise area when targeting FPGAs. Each can significantly improve the synthesis of a high-level language construct, thus allowing a designer to concentrate more on an algorithm description and less on hardware-specific implementation details. The first algorithm is a source-level transformation implementing function exlining—where a separate block of hardware implements a function and is shared between multiple calls to the function. The second is a novel algorithm for mapping arrays to memories which involves assigning array accesses to memory ports such that no port is ever accessed more than once in a clock cycle. This algorithm assigns accesses to read/write only ports and read-write ports concurrently, solving the assignment problem more efficiently for a wider range of memories compared to existing methods. Both optimisations operate on a high-level program representation and have been implemented in a commercial SystemC compiler. Experiments show that in suitable circumstances these techniques result in significant reductions in logic utilisation for FPGAs.

1. Introduction

Hardware compilation translates a program written in a high-level language into a description of a hardware circuit. The ultimate aim is to take software code and produce an efficient digital system design. SystemC [1] is a language designed for this purpose, allowing modelling of hardware in C++ syntax. This capability allows a designer to work at a higher level of abstraction compared to RTL design. Furthermore, SystemC offers faster simulation, enabling rapid prototyping, and effective design exploration [2]. These benefits can result in a significant boost in productivity. SystemC was originally designed as a modelling language but there are now several hardware compilers for this language, one of which being the agility compiler [3].

This paper focusses on methods for area optimisation in hardware compilation. For ASICs, this can significantly reduce the chip area and thus the production costs involved. For FPGAs, improving logic usage may be a necessity, given that these devices have limited resources. There are a variety of ways to improve the logic usage of a design. Most of these are optimisation techniques that are known for a long time, well understood, and described in, for example, [4]. These techniques are part of the domain of logic synthesis and are performed on a gate-level description. At this level, they can be applied to both RTL synthesis and hardware compilation. This paper investigates two area optimisation methods that are specific to hardware compilation and are performed on a high-level program representation such as an abstract syntax tree or a control and data flow graph, rather than on a gate-level description. The first method implements function exlining, which is the task of mapping a function to a dedicated piece of hardware that is shared between calls. Our method implements exlining as a source-level transformation that can be supported in existing compiler frameworks with relatively little effort. The second optimisation technique automatically maps arrays in SystemC to multiport memories in hardware. This involves a novel procedure for automatically assigning concurrent array accesses to memory ports whilst avoiding resource conflicts.

Both optimisations have been implemented in the agility compiler, which is discussed in Section 2. Sections 3 and 4 describe the two optimisation methods and their implementations and demonstrate their benefits in minimising logic utilisation for FPGAs. Section 5 summarises the main findings of this study. Finally, Section 6 discusses the current limitations of the two implementations and explores possible avenues for future research.

2. Agility SystemC Compiler

The agility compiler [3] is a commercial hardware compiler intended for the compilation of a SystemC program to a hardware description. It provides facilities for creation, compilation and synthesis of a large subset of the SystemC language. In addition to support for an extensive range of input language constructs, the compiler back end can target a wide range of architecture-specific functionality for a variety of technologies, enabling efficient synthesis. Agility is a timed synthesis tool, accepting designs composed either of SystemC threads punctuated by wait statements, or fully synchronous or fully asynchronous SystemC methods. As a result, the cycle timing of synthesised output exactly matches that of an input design, significantly aiding functional verification.

2.1. Language Support

The agility compiler language support is extensive, including all of the synthesisable subset defined by the Open SystemC Initiative (OSCI) Synthesis Working Group [5]. This includes most C++ constructs, such as

(i) conditional statements — **if**, **switch**;

(ii) loop statements — **while**, **do** . . . **while**, **for**;

(iii) control flow — **break**, **continue**, **return**.

In addition, agility supports C++ templates for generic programming in SystemC as well as object-oriented constructs such as (abstract) classes, inheritance, and polymorphism. Exceptions, dynamic (run-time) recursion, and dynamic pointer synthesis (including dynamic dispatch of virtual functions) are not supported, as their synthesis is either impossible on many devices (dynamic recursion) or would result in very inefficient hardware.

2.2. Synthesis

The agility compiler allows a designer to compile SystemC source code and produce different output formats: EDIF, VHDL, and Verilog. Figure 1 shows this design flow. When targeting FPGAs, agility can directly produce an EDIF netlist for Xilinx and Altera architectures. The EDIF is optimised and technology mapped and can be passed directly to the vendor's place and route tools. Alternatively, agility can produce RTL VHDL or Verilog for use with a third-party RTL synthesis or simulation tool.

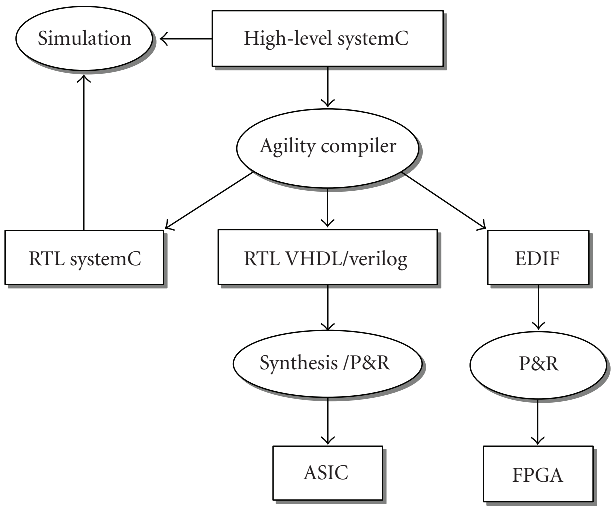


Figure 1: Agility design flow.

2.3. Verification Support

In addition to the aforementioned synthesis outputs, the compiler also supports the output of RTL SystemC for verification purposes. This output has exactly the same external interface as the synthesised input design, allowing the input design's test bench to be reused for functional verification. In addition, by design, this SystemC output is structurally identical to the RTL VHDL and Verilog output, allowing functional verification of the HDL output without requiring the use of an HDL simulator.

After synthesis through agility and then through target-specific place and route tools, final timing-level verification of fully synthesised designs can be achieved. This can be accomplished using the original SystemC test bench, a timing back-annotation of the HDL output and one of the several available mixed-language cycle-accurate simulators such as Aldec's Active-HDL [6] or Mentor Graphics's ModelSim [7].

3. Function Call Optimisation

Functions are commonly used in SystemC to divide a system up into tasks. Traditionally there are two methods for handling function calls in hardware compilation. One method, *inlining*, replaces each function call with the body of the function. Another method builds a single-hardware module for the function, which is subsequently shared between calls. This is called function *exlining*. Function exlining can potentially improve logic usage by reuse of the hardware associated with the function.

Function exlining has been implemented in several hardware compilers, such as [8, 9]. For these tools however, there is no description of how this optimisation is performed. This paper investigates the benefits of function exlining in hardware and describes a method for implementing this optimisation in SystemC compilation. It is shown that exlining can be adequately described in SystemC with the addition of asynchronous channels. This approach makes it possible to implement the method as a source transformation in existing compiler frameworks with relatively little effort. A further benefit of this method is that it allows arguments to be passed by value as well as by reference without relying on run-time pointer resolution, a feature not supported by many hardware compilers.

The effects of function exlining on the efficiency of the produced hardware are discussed in the next section. Then in Section 3.2, the mentioned method for function exlining in SystemC is explained. Finally, Section 3.3 presents results that demonstrate function exlining in the agility compiler for various SystemC programs.

3.1. Function Calls in Hardware

This section describes function inlining and exlining in hardware compilation and the effect that these methods have on the size and speed of hardware designs. The benefits of function exlining are discussed, as well as the design restrictions that apply when using this function call method.

3.1.1. Inlining Versus Exlining

In software, a call to a function causes execution to jump to a new part of the code. Assuming the typical execution environment for a C++ program with registers and a stack, the registers and parameters get written to the stack just before the function call, then the parameters get read from the stack inside the function and read again to restore the registers when the function returns. These operations can add a significant time overhead, in particular for functions that take little time to execute. An inline function call is expanded without causing a function call. That is, the compiler inserts the complete body of the function in every context where that function is used. Inline expansion is typically used to eliminate the transfer of control overhead that occurs in calling a function. However, because inline calls are replaced with a copy of the function body, they can result in a significant increase in code size.

The notion of inline and exline functions applies similarly to hardware compilation. Here, exline functions are synthesised to separate modules that are shared between calls. Alternatively, inlining replaces function calls with the bodies of the called functions. Figure 2 shows a SystemC thread calling two functions `f` and `g` that have been defined elsewhere. Each wait statement represents the end of a clock cycle, except for the first wait, which marks the end of the reset cycle.

Function `f` has a single adder and a single multiplier in its datapath and has two states. States essentially correspond to wait statements in SystemC and their number largely determines how large the control logic for this function is in hardware. Function g is larger, both in terms of datapath logic and number of states.

Figure 3(a) describes the structure of the hardware synthesised from this program by exlining `f` and `g`. In this case, one hardware module is synthesised from one function. Therefore, only a single module is synthesised from function `f` despite having been called twice. After inlining, however, the function accessor will contain multiple instances of the function and the resulting hardware is larger. This is illustrated in Figure 3(b), which shows the hardware structure that is generated by inlining calls to `f` and `g`. From this example, it follows that function exlining results in smaller logic compared to inlining as hardware is being shared. However, this view is not the whole picture and there are other factors involved that affect the results when exlining.

(1) Exlined Functions Require Additional Multiplexers

If arguments are passed to an exlined function, and the function is called multiple times, multiplexers must be created in hardware to switch between arguments from different calls. The logic depth of these multiplexers and thus the delay through them increase with the number of function calls and so do their sizes. Function exlining can therefore potentially decrease the maximum frequency $f_{\max}$ of a design, if these multiplexers are in the critical path. Furthermore, the size of multiplexers can be significant, in particular, for FPGAs where they are implemented in general-purpose lookup tables [10]. If the function that is exlined is small, this means that the overhead of multiplexers could outweigh the benefits of exlining the function.

(2) Function Exlining May Hinder Resource Sharing

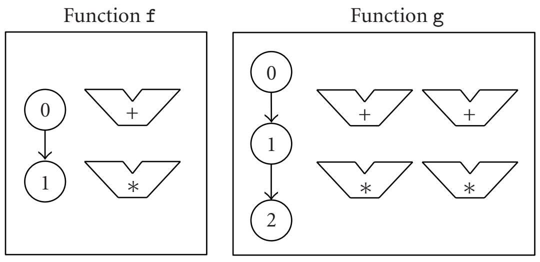
Resource sharing is an optimisation that automatically shares hardware resources between arithmetic operations in a program and is performed by many hardware compilers. Resource sharing is generally only performed on resources within the same module and those in different modules cannot be shared due to the difficulty of determining exclusive access to a resource from multiple threads of execution. This means that the hardware produced by function inlining, in which all hardware resources associated with functions become part of the same module, is more suited to resource sharing than exlining, in which a separate module is produced for each function. As a result, the size of the data path after exlining functions may be larger than the size after inlining [11]. Similarly, memory port sharing, as described in the second part of this paper, may also be hindered by function exlining.

When making the decision on whether to inline or exline calls to a function, it is therefore necessary to balance the circuit area saved by exlining against the added overhead associated with exlining.

3.1.2. Restrictions of Exlined Functions

The SystemC standard [1] does not specify when function calls should be inlined or exlined. In C++, functions are shared or exlined by default. By analogy, one could assume that exlining is a suitable default implementation of functions in SystemC. However, exlined functions are more restrictive in their use than inlined functions.

```
void thread()
{
  wait ();
  f ();
  wait ();
  g ();
  wait ();
  f ();
  wait ();
}
```



(a) (b)

FIGURE 2: SystemC program calling two functions f and g.

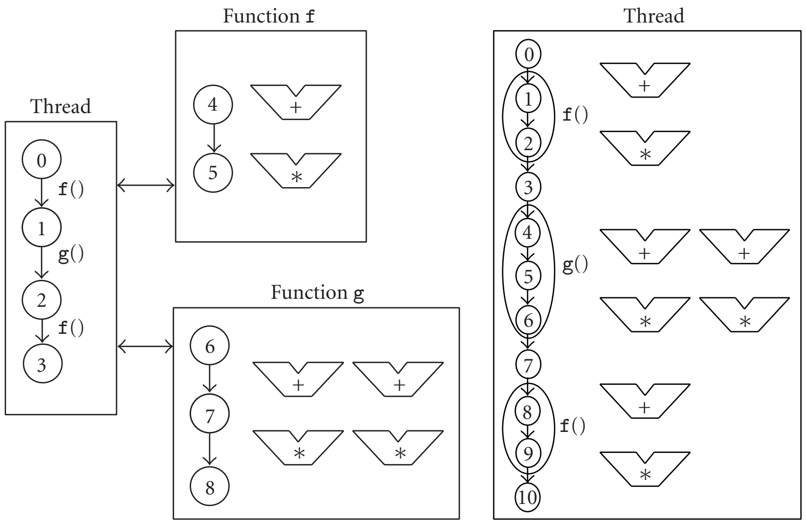


FIGURE 3: Function call methods in hardware. (a) Function exlining. (b) Function inlining.

(1) A Single Instance of an Exlined Functions Cannot Be Called in Parallel

Exlined functions cannot be called simultaneously from different threads as there is only one instance of each function to perform the task. Inlined functions do not have this restriction, as function instances are not shared in this case.

(2) Exlined Functions Cannot Be Called Recursively

Exlined functions cannot be called recursively as there is no stack in hardware. Functions labelled as inline can be called recursively without the use of a stack by means of recursive instantiation of the function, provided that the maximum recursion depth can be determined at compile time.

(3) Calls to a Particular Exlined Function Must Be in the Same Clock Domain

An exlined function must be in the same clock domain as its callers to avoid cross-clock domain synchronisation issues. Resynchronisation logic that is commonly used to resolve such issues would break SystemC timing semantics. Exlined functions can not therefore be called from multiple clock domains.

Despite these restrictions, exline functions are useful in hardware. They can greatly reduce the hardware size by sharing resources between calls. Unlike automatic resource sharing, exline functions allow sharing resources between threads and allow sharing of control path as well as data path logic. A hardware compiler will therefore benefit from supporting exlining as a method for synthesising function calls.

```
int f( int x )
{
  // function body
}

class Module : public sc_module
{
  sc_in < bool > clock;
  sc_in < int > input1, input2;
  sc_out < int > result;

  void thread()
  {
    wait ();   // end of reset cycle

    result = f( input1 );   // inlined call 1
    wait ();
    result = f( input2 );   // inlined call 2
    wait ();
  }

  Module(sc_module_name name) : sc_module(name)
  {
    SC_CTHREAD( thread, clock.pos() );
  }
};
```

Listing 1: SystemC program with function calls.

3.2. Synthesising Function Calls

This section describes a method for exlining function calls in SystemC synthesis. It explains how function exlining can be modelled in the SystemC language and how it has been implemented in the agility compiler.

3.2.1. Exlining in SystemC

In order to synthesise exline functions, it is useful to manually describe exlining in SystemC. This way, it can be evaluated and tested before implementation in a compiler. Furthermore, if function exlining can be modelled in SystemC, it can be conveniently implemented as a source transformation in a hardware compiler, rather than treating function exlining as a special case requiring significant additional functionality. To manually exline a function in SystemC, the following steps can be taken.

(1) The body of the function is moved to a newly created thread, inside an infinite loop.

(2) Handshaking is added between the function calls and the new thread to signal the start and end of function execution.

(3) Communication channels are added between the function calls and the new thread to transfer arguments and return value.

Step (1) is straightforward and involves creating a new thread that runs on the same clock as the calling thread or accessor. Steps (2) and (3) require asynchronous communication between two clocked threads for which SystemC has no facilities. To communicate between threads, SystemC uses synchronous `sc_signal` channels that introduce a clock cycle latency. This means that if they were used for exlined function calls, there would be an overhead of several clock cycles in calling a function. While this is perhaps

```
class Module : public sc_module
{
  sc_in < bool > clock;
  sc_in < int > input1, input2;
  sc_out < int > result;

  // handshaking
  sc_async_signal < bool > start, done;
  // argument and return value
  sc_async_signal < int > arg_chan, rtn_chan;
  void thread()
  {
    wait ();   // end of reset cycle

    // exlined call 1 :
    arg_chan.write( input1 );   // send argument
    start.write( true );        // start execution
    while( !done.read() )   { wait(); }
    result = rtn_chan.read(); // recv return value

    wait ();   // end of clock cycle

    // exlined call 2 :
    arg_chan.write( input2 );   // send argument
    start.write( true );        // start execution
    while( !done.read() )   { wait(); }
    result = rtn_chan.read(); // recv return value

    wait ();   // end of clock cycle
  }

  // newly created thread for function f
  void f_thread()
  {
    wait ();   // end of reset cycle

    while(1)
    {
      while( !start.read() )   { wait(); }
      int arg = arg_chan.read(); // recv argument
      int rtn = f( arg );       // inlined call to f
      rtn_chan.write( rtn );    // send return value
      done.write( true );       // end execution
    }
  }

  Module(sc_module_name name) : sc_module(name)
  {
    SC_CTHREAD( thread, clock.pos() );
    SC_CTHREAD( f_thread, clock.pos() );
  }
};
```

Listing 2: SystemC program modelling function exlining.

```
void f( int * x, int * y )
{
  (*x)++;
  (*y)--;
}

void thread()
{
  int x = 1;    //initialise x

  wait ();      // end of reset cycle

  f( &x, &x );   // x remains unchanged
  wait ();

  output = x;   // output = 1
  wait ();
}
```

Listing 3: SystemC program illustrating pointer aliasing.

```
int f( int arg )
{
  // function body
}

ag_share_routine( f ); // exline all calls to f
```

Listing 4: Agility directive for exlining a function.

acceptable in untimed synthesis, it would break the timing semantics of SystemC in which only `wait` statements take clock cycles. For the purpose of exlining, a new channel type is therefore introduced, called `sc_async_signal`. A channel of this type has the same interface as `sc_signal`, but implements asynchronous communication between synchronous threads. This channel type is used both for handshaking and transferring arguments.

3.2.2. Example

Listing 1 shows a SystemC program with two (inlined) calls to a function `f`.

In order to exline calls to `f`, a new thread `f_thread` is created, as shown in Listing 2. Two asynchronous channels, `start` and `done`, are introduced to signal the start and end of function execution. Two additional channels, `arg_chan` and `rtn_chan`, transfer argument and return value between the callers and the function.

The result is that only one instance of function `f` is created, rather than the two instances in the original program.

3.2.3. Passing Arguments by Reference

Function arguments in SystemC can be passed either by value or by reference. If an argument is passed by reference, a pointer to the argument is passed to the function. This pointer may then be derefenced inside the function which allows the argument to be modified. If the function is exlined, it can be accessed by multiple callers and pointers to arguments that need to be resolved during execution inside the function. This feature relies on a hardware compiler being able to synthesise pointers. Although pointer synthesis is possible, it tends towards producing inefficient hardware in terms of area and speed and at the same time offers little modelling benefit in the absence of dynamic memory allocation [12]. For this reason, not many hardware compilers support this feature, including agility. As a consequence, it would not be possible to modify arguments within a function.

Fortunately it turns out that if the value of a pointer argument is known at compile time for every caller of an exline function, then the call-by-reference can be replaced by a call-by-value without the need for pointers in hardware. This is achieved by dereferencing the pointer at the point of call rather than inside the function, and passing the result over an asynchronous channel to the function. The function receives this value and may modify it. After the function finishes execution, the modified value is sent back to the caller on a second, different channel and the call finishes.

Although this method allows arguments to be modified inside an exlined function, it has some limitations as well. By sending arguments over a channel rather than passing them by reference, the function will operate on a copy of the argument rather than the original. This requires that the argument must be of a copyable type that can be sent over a channel. Furthermore, the copy requires extra storage in the function and potentially increases sequential logic. Fortunately, this usually does not lead to an overall increase in logic area in FPGAs, except for register-rich designs such as those containing large register files.

Another potential issue arises when several arguments are passed by reference to a SystemC function where two or more pointers refer to the same object. In this case, changing one of the arguments inside the function may have an indirect effect on another argument. This effect is called *pointer aliasing* and is illustrated in Listing 3.

In this example, two pointers are passed to `f` that both point at the same integer `x`. The result is that `x` will first be incremented and then decremented and the effect is that `x` remains unchanged. When function `f` is exlined using the method described in this section, then `x` is sent on two different channels and the function will operate on two distinct copies of `x`. This would remove any pointer aliasing and cause a mismatch in behaviour between SystemC and hardware implementation: depending on the order in which channel communication happens, `x` will either be incremented or decremented. Fortunately, given the restriction that pointers must be resolved at compile time, pointer aliasing can be detected by the compiler.

```
int f( int arg )
{
  // function body
}

int f1( int arg )
{
    return f( arg ); // create instance of f
}

int f2( int arg )
{
    return f( arg ); // create instance of f
}

ag_share_routine( f1 );  // exline all calls to f1
ag_share_routine( f2 );  // exline all calls to f2
```

Listing 5: Creating multiple shared instances of a function.

3.2.4. Function Calls in Agility

Early versions of Agility only supported inlining as a method for synthesising function calls. In order to achieve better synthesis results, function exlining was added based on the method described in this section. Together with the addition of asynchronous channels to Agility, the method was implemented as a source-to-source transformation on the abstract syntax tree (AST), the compiler's internal representation of a SystemC program.

The obvious way to control function call expansion in Agility is to use the `inline` keyword and other C++ rules set out in [13]. This approach however has several disadvantages. Firstly, it would change the behaviour of existing designs that rely on functions being inlined rather than exlined. Exlining function calls that were previously inlined would not only affect the hardware that is produced, but would potentially break the design due to the restrictions of exline functions that were mentioned in Section 3.1.2. Furthermore, the rules for inlining in C++ are not strict and merely hint to the compiler that inlining is preferred. This would not provide many users the control they desire. For these reasons, a new synthesis directive, `ag_share_routine`, was added to Agility to exline a function and automatically perform the described source-to-source transformation. This directive takes the function to be exlined, which is illustrated in Listing 4.

In this example, all calls to function `f` are exlined and a single instance is created in hardware. In order to decrease multiplexer depth and improve clock frequency, it is sometimes beneficial to map a function to multiple shared instances instead. This can be achieved in Agility by creating several exline functions for each call `f` as is illustrated in Listing 5.

In this example, if `f` is always called via `f1` or `f2`, no more than two shared instances of `f` are created in hardware.

3.3. Results

Experiments were performed to demonstrate the effect of function exlining and inlining on the efficiency of hardware produced by Agility. For this purpose, three designs in SystemC were used: an inverse discrete cosine transform (IDCT), calculating the determinant of a 3×3 matrix (DET), and multiplying two 3×3 matrices (MULT). These designs all contain a function that is called multiple times and can be exlined. Each design was compiled to EDIF and implemented on an Xilinx Virtex-4 device in two versions: one inlining and another exlining the function. Post-implementation simulations were performed to verify that both versions are equivalent. Table 1 shows the number of slices and maximum clock frequency $f_{\max}$ for each design for the two function call methods as reported by the Xilinx tools. For each design, the table also lists the size of the function that is exlined as well as the number of calls to this function and the number of arguments. All arguments are 32-bit wide.

From these results, it follows that function exlining reduces the size of all designs, in particular those containing a large function such as MULT. For smaller functions and those with many arguments, the overhead of multiplexers that are created to switch between arguments from different calls becomes noticable. This is true for the IDCT example, where exlining only has marginal effect. The same multiplexers also add to the logic delay and can reduce $f_{\max}$ if they become part of the critical path. This is the case for DET and MULT, where the the maximum frequency is significantly reduced by exlining.

4. Array Optimisation

The array is a commonly used data structure in SystemC and can be mapped in different ways to hardware. Normally, arrays are mapped to register files. This implementation matches the behaviour of arrays in SystemC, but is not very efficient in terms of performance and logic area. ASIC libraries generally include efficient RAM components and modern FPGAs typically contain a large number of RAM blocks which can be used to implement arrays instead. Memories have a limited number of ports, and part of the process of mapping arrays to memories is assigning each memory access to a port such that contention is prevented. Many RTL synthesis tools can infer RAMs from arrays, but they require that the designer assigns access to ports manually. High-level languages do not offer this kind of control and a hardware compiler must therefore be able to automatically assign each array access to a memory port such that no port is accessed multiple times in parallel.

The problem of automatically assigning memory accesses to ports has received little attention by itself. The reason is that this problem has traditionally been solved using general resource sharing methods such as described in [14]. As we shall show, these methods cannot be used for all types of memories and thus a different approach must be taken. This

Table 1: Comparison between function call methods for various designs targeting an XC4VLX40 FPGA.

Example	Func size (slices)	Calls	Args	Method	Size (slices)	$f_{\max}$ (MHz)
IDCT	1,008	2	9	inline	6,213	69
				exline	6,165	69
DET	541	3	4	inline	2,418	74
				exline	1,715	61
MULT	2,424	3	3	inline	7,218	78
				exline	2,747	64

paper proposes an algorithm to solve this problem. The algorithm has been implemented in the agility compiler.

The effects of mapping arrays in SystemC to memories in hardware are discussed in the next section. Then in Section 4.2, existing research in this field is examined. Our proposed method for assigning array accesses to memory ports is discussed in Section 4.3. Finally, Section 4.4 presents results that show the benefits of using this method in minimising logic utilisation for FPGAs.

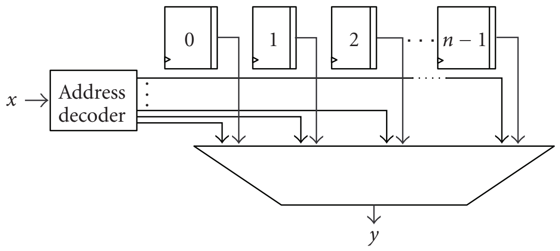


Figure 4: Array read access in hardware.

4.1. Arrays in Hardware

In C++, an array is represented by a continuous memory segment containing all array elements in a representation corresponding to their type. By analogy, one could assume that a memory is a suitable hardware implementation of an array in SystemC, both being multi-dimensional representations of bits. However, arrays in SystemC have different semantics from memories.

(1) SystemC Arrays Offer Parallel Access to Elements

In SystemC, a design can access multiple array elements in the same clock cycle and there are no restrictions to the number of parallel accesses. A memory on the other hand has a limited number of ports which means that only a limited number of simultaneous accesses is allowed.

(2) SystemC Arrays are Accessed in One Cycle

In timed SystemC threads, only wait statements take clock cycles and nothing else. An array access must therefore finish within one cycle. Many architectures however support synchronous memories, where a read operation is controlled by the system clock and takes two cycles: one to setup the address and one to read the data. To match the SystemC timing semantics, memory accesses could be pipelined in an attempt to establish the address one cycle ahead. However, this is only possible in certain program contexts.

(3) SystemC Arrays Have Write-Before-Read Semantics

In SystemC, when an array write is followed by an array read in the same cycle from the same address, the value that is read is the value that has just been written in the same cycle. In hardware, many multi-port memories have read-before-write behaviour, which means that a value that is written does not become available until the next clock cycle. Consequently, any value that is read has always been written in an earlier cycle. This behaviour can cause a mismatch between SystemC model and implementation.

Because of the difference in behaviour between arrays in SystemC and memories in hardware, not all arrays can be implemented in memory. For this reason, Agility implements an array as a register file by default, consisting of registers and combinational logic. This implementation however may use considerable logic resources. This is illustrated in Figure 4, showing the hardware that is built for a read operation `y = Array[x]` from a register array with `n` elements. Each array element requires a register in hardware. An address decoder translates address `x` into a bit vector which controls the output multiplexer. This multiplexer selects the output of the particular element that is indexed by `x`. If an array is read several times, several address decoders and multiplexers are required.

Figure 5 shows the hardware that is built for a write operation `Array[x] = y` to a register array with `n` elements. In this case, the output lines of the address decoder are connected to the write enables of the registers to select which element to write to. If an array is written to multiple times, multiplexers are required on the inputs of the registers to select which data to write.

With more complex systems being developed onFPGA platforms, the need for storage in these devices is increasing. Thus modern FPGAs contain a large amount of on-chip memory. This memory can be targeted automatically from arrays in a SystemC program. Mapping arrays to memory rather than general purpose logic can significantly reduce the logic usage of a design.

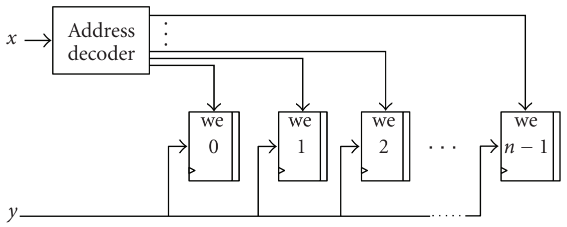


Figure 5: Array write access in hardware.

4.2. Related Research

The problem of synthesising memories from arrays has received attention in the past, though most of this research has focussed on efficient mapping of arrays to physical memory blocks [15, 16]. The problem of assigning memory accesses to ports has not been investigated by itself. The reason is that memory accesses are treated as normal data path operations and are covered by general sharing methods such as described in [14, 17]. In these methods, a compatibility graph is built where two operations are compatible when they can be assigned to the same resource, independent of the type of resource. This is not always the case for memory accesses. For example, suppose a particular memory has one read only port and one read-write port. Whilst read operations can use either port, write operations can only be assigned to the read-write port. The compatibility between a read access and a write access thus depends on which port the read access will be assigned to. Consequently, a global clique partitioning algorithm operating on a compatibility graph cannot be applied to solve this problem.

Assigning array accesses to memory ports can also be performed using a constructive approach, in which operations are assigned to functional units in a step-by-step fashion [18]. For each memory access, such an algorithm attempts to find a memory port that is capable of executing the read or write operation and that has not been assigned yet in the current clock cycle. In the case where there are two or more memory ports that meet these conditions, the one which results in minimum multiplexer depth is chosen. Whilst this method is simple, it is based on local information only and therefore often leads to suboptimal results. This is true particularly in the presence of exclusive branches, where an efficient assignment of accesses in one branch depends on the accesses present in the other branches.

Another paper that addresses the problem of assigning operations to functional units is [19]. This method builds, for each clock cycle, a bipartite graph containing the operations that are executed in this cycle together with functional units that the operations can be assigned to. All edges in the graph run between operations and functional units and specify whether an operation can be performed on a certain unit. As with clique partitioning, weights can be associated with these edges, representing the costs associated with particular assignments. The problem of assigning each operation to a unique functional unit, such that the sum of all edge weights is minimal, is called *weighted bipartite matching*.

The bipartite graph does not contain information regarding compatibility between operations and it is therefore not possible to assign operations that are executed in mutually exclusive branches to the same functional unit. To overcome this limitation, the method proposed in this paper uses a transformed bipartite graph which, instead of nodes representing operations, contains nodes representing sets of operations that are executed in mutually exclusive branches. Each of these sets can then be assigned to functional units using weighted bipartite matching. To build this type of bipartite graph for assigning memory accesses, the algorithm must analyse the program to gather all memory accesses that are executed in a particular cycle and merge those that occur in mutually exclusive branches. An algorithm for performing this analysis is presented in the next section.

4.3. Proposed Method

To assign memory accesses to ports, the algorithm needs to determine which accesses may occur simultaneously and which are independent. If two accesses are erroneously determined to be independent, incorrect hardware will be produced that suffers from memory port contention. On the other hand, it is acceptable if the algorithm is conservative and determines that two accesses can occur at the same time, when in fact they cannot. The proposed method is divided into two parts: access analysis and port assignment. The first part analyses the semantic structure of the program to determine which memory accesses are independent. This is the case if they are separated by a wait statement or are in different branches of an if/switch statement. The information that is gathered by access analysis is then used by the port assignment algorithm in order to assign accesses to ports.

4.3.1. Control Flow Representation

In order to describe the algorithm, a SystemC program is represented in a control flow graph (CFG). A CFG is a representation, using graph notation, of all paths that might be traversed through a program during its execution. The nodes represent operations and directed edges are used to represent jumps in the control flow. For the purpose of assigning memory accesses, a CFG is presented in which there are four node types: *conditional forks*, *conditional joins*, *waits*, and *basic blocks*. A basic block is a sequence of operations that is always entered at the beginning and exited at the end. Without loss of generality, it is assumed here that a basic block contains at most a single-memory access.

Figure 6(a) shows a SystemC program with conditional constructs in which an array is accessed. Figure 6(b) shows the corresponding CFG, containing three basic blocks.

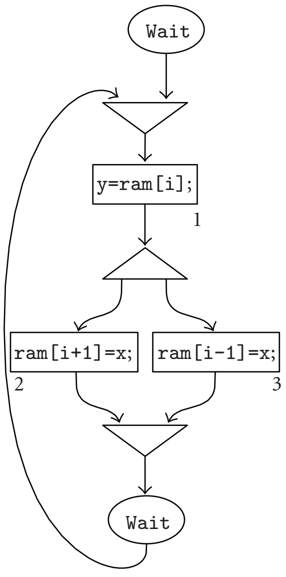
Cycles in the CFG are created by loops in the SystemC program. It is assumed that all combinational loops in SystemC will have been unrolled by the compiler at this stage. Consequently, cycles in the CFG always contain at least one wait node, as they cannot otherwise be implemented in synchronous hardware.

```
void thread()
{
  sc_uint ⟨8⟩ i = 0;
  wait ();
  while(1)
  {
   y = ram[i];
   if (i < 128}
   {
    ram[i + 1] = x;
   }
   else
   {
    ram[i − 1] = x;
   }

   i ++;

   wait ();
   }
}
```

(a)



(b)

FIGURE 6: Control flow representation. (a) SystemC description, (b) control flow graph.

```
Assign(CFG)
{
  result := $\varnothing$
  for each g $\in$ subDAGs(CFG)
  {
    accesses := $\varnothing$
    for each blk $\in$ topsort (basic blocks in g)
    {
      // merge control flows into blk
      merged := merge {accesses [b] | b $\in$ pred[blk]}

      // add the effect of a memory access in blk
      accesses [blk] := effect blk merged
    }
    result[g] := merge {accesses [b] | succ[b] = $\varnothing$}
  }
  return result
}
```

LISTING 6: Memory access analysis algorithm.

4.3.2. Access Analysis

The access analysis algorithm gathers, for each clock cycle, sets of independent memory accesses that are assigned to different memory ports. It performs this process independent of the number of memory ports available and the access types of these ports. It attempts to combine memory accesses in such a way that the final number of sets, and thus the number of required memory ports, is minimal. Some sets may contain both read and write accesses and must be mapped to read-write ports. As not all memories have these ports, these sets may later have to be split into sets that can be assigned to simple ports. This increases the number of required memory ports and the algorithm therefore attempts to minimise the number of sets with mixed accesses.

The algorithm takes the CFG as the input. It then splits it into directed acyclic subgraphs (sub-DAGs) corresponding to individual clock cycles and processes them separately. One way of doing this is to remove all wait nodes and edges incident upon them from the CFG. Then the graph can be split into subgraphs by temporarily regarding all edges as undirected, and finding all connected nodes (e.g., using depth-first search). As all cycles in the CFG contain at least one wait node, all directed subgraphs thus obtained will be acyclic. To assign accesses to memory ports, the basic blocks in each sub-DAG corresponding to a clock cycle are traversed in topological order, starting with accesses early in the cycle. During traversal, the algorithm gathers sets of independent memory accesses that can be mapped to the same memory port. This information is represented as a triple list of sets (r, w, rw) of accesses as follows:

(i) r contains sets of independent read accesses;

(ii) w contains sets of independent write accesses; and

(iii) rw contains sets of independent read and write accesses.

Accesses in the same set can be mapped to the same memory port and accesses in different sets must be mapped to different ports to prevent resource conflicts. Consequently, the minimum number of memory ports required to assign all accesses in the triple to ports is equal to the total number of sets in the triple. For example, suppose the triple is equal to

$$(r, w, rw) = ([\{r_1, r_2\}], [\{w_1\}, \{w_2\}], [\{r_3, w_3\}]). \tag{1}$$

In this case, at least four memory ports are required to assign all accesses: one port capable of reading, two ports capable of writing, and one port capable of both reading and writing.

Pseudocode for the algorithm that gathers all accesses to a particular memory in a program is shown in Listing 6.

The map `accesses` store the access triple at each basic block and are used for temporary storage. `pred` and `succ`, respectively, return the parents and children of a basic block. When a particular basic block is encountered, the access triples of its predecessors are merged to combine accesses from mutually exclusive branches. Then, the memory access in the current basic block is added to the triple. After all basic blocks in a clock cycle have been visited, the final access triple is calculated by merging those basic blocks without successors. Function `effect` models a memory access in a basic block and appends the access as a singleton set to the end of the appropriate list in the access triple

```
mergetwo : triple − > triple − > triple
mergetwo (r1,w1,rw1) (r2,w2,rw2) = (fr, fw, frw)
 where (r1r2, r1′, r2′)      = combine r1 r2
       (w1w2, w1′, w2′)      = combine w1 w2
       (rw1rw2, rw1′, rw2′)  = combine rw1 rw2
       (r1rw2, r1″, rw2″)    = combine r1′, rw2′
       (w1rw2, w1″, frw2)    = combine w1′, rw2″
       (rw1r2, rw1″, r2″)    = combine rw1′, r1″
       (rw1w2, frw1, w2″)    = combine rw1″, w1″
       (r1w2, fr1, fw2)      = combine r1″, w2″
       (w1r2, fw1, fr2)      = combine w1″, r2″

       −− take sets of accesses from two lists
       −− xs and ys and combine them into zw
       combine xs ys = (zw,drop n xs,drop n ys)
        where zw = zipwith ( ∪ ) xs ys
              n  = length zw

       −− the merged triple is created from the
       −− combined sets plus any remaining sets
       fr  = r1r2 ++ fr1 ++ fr2
       fw  = w1w2 ++ fw1 ++ fw2
       frw = rw1rw2 ++ r1rw2 ++ w1rw2 ++
             rw1r2 ++ rw1w2 ++ r1w2 ++
             w1r2 ++ frw1 ++ frw2
```

Listing 7: Function for merging two control flows.

(r, w, rw). Read accesses are added to r and write accesses to w. Function `merge` combines the memory acesses in a number of mutually exlusive branches, for example at the end of an if statement. The aim of `merge` is to combine accesses in the most efficient way as to minimise the number of memory ports required. The algorithm for merging two triples is shown in Listing 7 in functional programming notation.

The merging of two triples P and Q consists of repeatedly picking a set of accesses from P and a set from Q and taking the union until either P or Q is empty. If any sets remain in P or Q, these sets are just inserted into the merged triple. There are many ways in which sets in P or Q can be combined, where some combinations are more favourable than others. For example, suppose two access triples P and Q are defined as

$$P = ([\{r_1\}], [\{w_1\}], []),$$
$$Q = ([\{r_2\}], [\{w_2\}], []). \tag{2}$$

One possible way to merge P and Q is to combine read accesses with write accesses: $([], [], [\{r_1, w_2\}, \{r_2, w_1\}])$. Although this requires two memory ports, not all memories have read-write ports. A better way to merge P and Q is $([\{r_1, r_2\}], [\{w_1, w_2\}], [])$, which requires one read port and one write port. The merge function therefore favours combinations between accesses of the same type over accesses of different types. In addition, merge attempts to avoid combining sets that merely contain read accesses with those that merely contain write accesses. For example, if P and Q are defined as

$$P = ([\{r_1\}], [], []),$$
$$Q = ([], [\{w_1\}], [\{r_2, w_2\}]), \tag{3}$$

then P and Q can be merged by combining $\{r_1\}$ and $\{w_1\}$: $([], [], [\{r_2, w_2\}, \{r_1, w_1\}])$, or $\{r_1\}$ and $\{r_2, w_2\}$: $([\{r_1\}], [], [\{r_2, w_2, r_1\}])$. Both solutions require two memory ports. However, when read access $\{r_1\}$ and write access $\{w_1\}$ are combined, a set with mixed access types is created which requires an additional read-write port.

Figure 7 shows a fragment of a SystemC program, together with the control flow subgraph for the particular clock cycle.

Table 2: Steps showing progress of access analysis.

Operation	Accesses (r, w, rw)
`a = ram[p];`	$([\{1\}], [], [])$
`ram[q] = b;`	$([\{1\}], [\{2\}], [])$
`c = ram[r];`	$([\{3\}], [], [])$
`ram[s] = d;`	$([], [\{4\}], [])$
`merge (3,4);`	$([], [], [\{3, 4\}])$
`ram[t] = e;`	$([], [\{5\}], [\{3, 4\}])$
`merge (2,5);`	$([], [\{2, 5\}], [\{3, 4, 1\}])$

The code contains two read accesses (1 and 3) and three write accesses (2, 4, and 5). Table 2 shows the progress of access analysis whilst traversing the CFG. The final access triple for this clock cycle is $([], [\{2, 5\}], [\{3, 4, 1\}])$, which can be assigned to a memory with one write port and one read-write port.

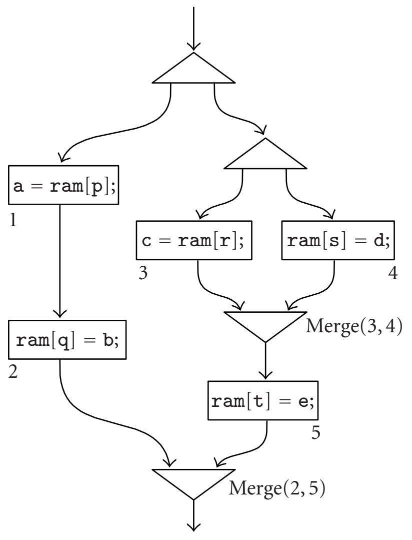
4.3.3. Assigning Accesses to Ports

To assign accesses to ports, a bipartite graph is constructed for each clock cycle from the access triple (r, w, rw). The sets of accesses in r and w can be mapped to read/write only ports as well as read-write ports whilst the sets in rw can be mapped to read-write ports only. The latter therefore gives the algorithm less freedom in assigning accesses to ports optimally. Furthermore, not all memories have read-write ports. For this reason, the access triples are transformed as to minimise the size of rw. This transformation involves repeatedly splitting sets in rw into two sets: one containing the read and one containing the write accesses. These are then added to r and w respectively. This process increases the number of required memory ports and is repeated until the number of required ports is equal to the number of ports on the memory or until rw is empty. For example, the final access triple in the example of Figure 7 was $([], [\{2, 5\}], [\{3, 4, 1\}])$, requiring one write port and one read-write port. If these accesses are mapped to a memory with one read-only and two write only ports instead, the triple is transformed as $([\{3, 1\}], [\{2, 5\}, \{4\}], [])$.

After the transformation, the bipartite graph is built. Each edge in the graph has a weight associated with it. This is a measure of the cost of binding the accesses in a set to

```
...
wait ();
if (c1)
{
  a = ram[p]; // 1
  ram[q] = b; // 2
}
else
{
  if (c2)
  {
   c = ram[r]; // 3
  }
  else
  {
   ram[s] = d; // 4
  }
  ram[t] = e; // 5
}
wait ();
...
```

(a)



(b)

Figure 7: Memory access analysis. (a) SystemC fragment, (b) control flow subgraph corresponding to clock cycle.

a particular port. In the proposed algorithm, the weight on an edge (a_i, p_j) is based on the total number of accesses bound to port p_j if the set of accesses a_i is assigned to it. This weight is a global cost which takes into account accesses that were assigned in other cycles as well. This way, memory accesses will be balanced between ports and the depths of multiplexers on the input of ports is minimised. After building the graph, bipartite matching is performed to assign all accesses to ports.

4.4. Results

Experiments were performed to demonstrate the effect of mapping an array to memory. For this purpose, an inverse discrete cosine transform (IDCT) was used. An implementation of this algorithm was written in C by the MPEG Software Simulation Group (MSSG) [20] which contains an array requiring 1 kb of storage. This design was ported to SystemC and compiled to EDIF with and without mapping the array to memory. The generated EDIF for both array implementations was passed to the Xilinx design tools and implemented on a Virtex-4 device. Post-implementation simulations were performed to verify that both implementations are functionally correct. Table 3 shows the number of flip-flops and slices for the IDCT design for the two array implementations as reported by the Xilinx design tools.

From these results it follows that mapping the array to memory significantly reduces the size of the IDCT design. The proportion of used slices on the particular FPGA device is reduced from 34 percent to 15 percent, whilst only 1 out of 96 available RAM blocks is needed for implementing the array. As a result, the design can be implemented on a smaller, and thus cheaper, FPGA device.

Table 3: Comparison between array implementations for IDCT design targeting an XC4VLX40 FPGA.

Array impl.	Flip-flops	Slices	Logic usage (%)
registers	1,687	6,213	34
memory	663	2,724	15

5. Conclusion

This paper has presented two area optimisation procedures for FPGAs in SystemC hardware compilation. The first is function exlining, which aims to reduce the logic size of a design by mapping a function to a separate piece of hardware that is shared between calls. It has been shown that function exlining can be described in SystemC with the addition of an asynchronous channel to the language. This method can be easily implemented in a hardware compiler as a source transformation and performed automatically. The second optimisation algorithm deals with mapping arrays in SystemC to memories in hardware. This method analyses the program and gathers sets of independent accesses that can be mapped to the same memory port whilst avoiding resource conflicts. Compared to previous methods, it solves the assignment problem more efficiently for a wider range of memories. Both optimisations can help to transform a behavioural specification into an efficient implementation in hardware. This is in the spirit of hardware compilation, where designers should focus on the algorithm itself rather than on manually optimising code.

The proposed methods were implemented in the agility compiler and experiments were performed that showed the benefits of these methods in reducing logic utilisation in FPGAs. It was found that function exlining can greatly improve the logic usage of a design. However, sharing a function between different callers also introduces multiplexers to switch between arguments. The overhead of these multiplexers in terms of logic size and delay is potentially large for FPGAs, where they are implemented in general-purpose lookup tables. It is therefore necessary to balance the circuit area saved by exlining against the added overhead associated with exlining. Whilst function exlining saves resources by sharing them between tasks, mapping arrays to memories is based on choosing a different hardware implementation for a given task. Modern FPGAs contain a large amount of on-chip memory and this method allows a designer to target this abundant resource without significantly changing a design's specification. As experiments showed, this can significantly reduce logic area such that the design can be implemented on a smaller, and thus cheaper, FPGA device.

6. Limitations

Although the optimisation techniques described in this paper have shown promising results, there are several opportunities for improvement as well as for further research. Function exlining in agility is currently controlled by the user via the `ag_share_routine` directive. If a function is specified as shared, all calls to this function are exlined and care must be taken that no resource conflicts arise due to simultaneous calls to the function. The decision to exline a function could be made by the compiler instead. A method to achieve this is described in [21].

In the current implementation, all calls to an exline function share the same hardware module. In order to optimise multiplexer usage and avoid resource conflicts, a SystemC function could be mapped to multiple hardware modules instead. In this approach, function calls with common arguments could be detected through static analysis and combined in order to reduce multiplexer depth and thus improve clock frequency.

The current implementation supports arguments passed by reference without the need to resolve pointers during execution. As discussed, this not only poses restrictions on the type of arguments passed, but also increases sequential logic. A solution using run-time pointer resolution would avoid these issues, a method for which is presented in [22].

In the proposed algorithm for synthesising arrays, each array is mapped to a separate logical memory. This can be inefficient if a program contains several arrays that are smaller in size than the available memory components. In theory, those arrays could be implemented in a single memory thereby reducing memory cost. Schmit and Thomas [16] propose a method for grouping arrays of different sizes and dimensions and packing them into memories, which is suited to this purpose.

The bipartite matching algorithm that is used to assign memory accesses to ports is based on a cost function. In the current implementation, this function only takes multiplexer depth into account and attempts to balance accesses between ports. The algorithm could be improved by using true delay information instead.

Finally, this paper discussed the interaction between function exlining and other optimisation techniques. It was mentioned that function exlining may hinder other optimisations, such as resource sharing and memory port sharing, which are typically performed on each module individually rather than globally. More research is required to investigate how the proposed optimisation techniques can be extended to operate optimally together and in synergy with other optimisations.

References

[1] Open SystemC Initiative, "*SystemC 2.1 Reference Manual*," 2005, http://www.systemc.org/.

[2] T. Rissa, A. Donlin, and W. Luk, "Evaluation of systemC modelling of reconfigurable embedded systems," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '05)*, vol. 3, pp. 253–258, Munich, Germany, March 2005.

[3] Celoxica, "*Software Product Description for Agility Compiler v1.2*," 2006.

[4] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, New York, NY, USA, 1994.

[5] OSCI Synthesis Working Group, "*SystemC Synthesizable Subset, Draft 1.1.18*," December 2004, http://www.systemc.org/.

[6] Aldec, *Active-HDL*, http://www.aldec.com/.

[7] Mentor Graphics, *ModelSim*, http://www.model.com.

[8] S. Gupta, R. K. Gupta, N. D. Dutt, and A. Nicolau, "Coordinated parallelizing compiler optimizations and high-level synthesis," *ACM Transactions on Design Automation of Electronic Systems*, vol. 9, no. 4, pp. 441–470, 2004.

[9] Agility Design Solutions, "*Handel-C Language Reference Manual*," 2008, http://www.agilityds.com/literature/HandelC_Language_Reference_Manual.pdf.

[10] J. Ditmar, *Area optimisation in systemC hardware compilation*, M.S. thesis, University of Oxford, Oxford, UK, 2007.

[11] Y. Hara, H. Tomiyama, S. Honda, and H. Takada, "Function call optimization in behavioral synthesis," in *Proceedings of the 9th EUROMICRO Conference on Digital System Design: Architectures, Methods and Tools (DSD '06)*, pp. 522–529, Dubrovnik, Yugoslavia, August-September 2006.

[12] E. Grimpe and F. Oppenheimer, "Extending the systemC synthesis subset by object-oriented features," in *Proceedings of the 1st IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '03)*, pp. 25–30, Newport Beach, Calif, USA, October 2003.

[13] ISO/IEC 14882:1998, "*Programming languages—C++*," ISO/IEC, Geneva, Switzerland, 1998.

[14] S. Raje and R. A. Bergamaschi, "Generalized resource sharing," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '97)*, pp. 326–332, San Jose, Calif, USA, November 1997.

[15] D. Karchmer and J. Rose, "Definition and solution of the memory packing problem for field-programmable systems," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '94)*, pp. 20–26, San Jose, Calif, USA, November 1994.

[16] H. Schmit and D. E. Thomas, "Array mapping in behavioral synthesis," in *Proceedings of the 8th International Symposium on System Synthesis (ISSS '95)*, pp. 90–95, Cannes, France, September 1995.

[17] C.-J. Tseng and D. P. Siewiorek, "Automated synthesis of data paths in digital systems," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 5, no. 3, pp. 379–395, 1986.

[18] K. Kucukcakar and A. C. Parker, "Data path tradeoffs using MABAL," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pp. 511–516, Orlando, Fla, USA, June 1990.

[19] C.-Y. Huang, Y.-S. Chen, Y.-L. Lin, and Y.-C. Hsu, "Data path allocation based on bipartite weighted matching," in *Proceedings of the 27th ACM/IEEE Design Automation Conference (DAC '90)*, pp. 499–504, Orlando, Fla, USA, June 1990.

[20] MPEG Software Simulation Group, http://www.mpeg.org/.

[21] F. J. Kurdahi and A. C. Parker, "REAL: a program for REgister allocation," in *Proceedings of the 24th ACM/IEEE Design Automation Conference (DAC '87)*, pp. 210–215, Miami Beach, Fla, USA, June-July 1987.

[22] L. Semeria and G. De Micheli, "SpC: synthesis of pointers in C application of pointer analysis to the behavioral synthesis from C," in *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '98)*, pp. 340–346, San Jose, Calif, USA, November 1998.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 428265, 9 pages
doi:10.1155/2008/428265

Research Article

Neuromorphic Configurable Architecture for Robust Motion Estimation

Guillermo Botella,[1] Manuel Rodríguez,[2] Antonio García,[3] and Eduardo Ros[2]

[1] *Department of Computer Architecture and Automation, Complutense University of Madrid, 28040 Madrid, Spain*
[2] *Department of Computer Architecture and Technology, University of Granada, 18071 Granada, Spain*
[3] *Department of Electronics and Computer Technology, University of Granada, 18071 Granada, Spain*

Correspondence should be addressed to Guillermo Botella, gbotella@fdi.ucm.es

Received 1 July 2008; Accepted 8 October 2008

Recommended by Gustavo Sutter

The robustness of the human visual system recovering motion estimation in almost any visual situation is enviable, performing enormous calculation tasks continuously, robustly, efficiently, and effortlessly. There is obviously a great deal we can learn from our own visual system. Currently, there are several optical flow algorithms, although none of them deals efficiently with noise, illumination changes, second-order motion, occlusions, and so on. The main contribution of this work is the efficient implementation of a biologically inspired motion algorithm that borrows nature templates as inspiration in the design of architectures and makes use of a specific model of human visual motion perception: Multichannel Gradient Model (McGM). This novel customizable architecture of a neuromorphic robust optical flow can be constructed with FPGA or ASIC device using properties of the cortical motion pathway, constituting a useful framework for building future complex bioinspired systems running in real time with high computational complexity. This work includes the resource usage and performance data, and the comparison with actual systems. This hardware has many application fields like object recognition, navigation, or tracking in difficult environments due to its bioinspired and robustness properties.

1. Introduction

Bioinspired systems emulate the behavior of biological ones. Neuromorphic approximations [1] are based on the way how the nervous systems create physical architectures and computations, attending to the morphology, information coding, robustness against damage, and so on. Neuromorphic systems usually deliver good primitives for the building of more complex systems, being the output of each system simpler than its input. This data reduction helps in the task of integrating every response associated with all information channels [2].

Attending to the estimation of a pixel motion inside the image sequence, there are many models and algorithms that could be classified as belonging to the matching domain approximations [3], energy models [4], and gradient models [5]. Related to this last family, different studies [6–8] show that this represents an admissible choice for keeping a tolerable tradeoff between accuracy and computing resources. For designing systems operating efficiently, it is required to deal with many challenges, such as robustness, static patterns, illumination changes, different kinds of noise, contrast invariance, and so on. If bioinspirational behavior is required, that is, ability to detect correct motion related to optical illusions or avoiding operations like matrix inverse or iterative methods that are not biologically justified, we have to select carefully a model that carries out this kind of requirements. This is the Multichannel Gradient Model (McGM) [9–12].

Motivated by these previous results and analysis, we present the architecture and implementation of a customizable optical flow embedded processing core running in real time. This system works in the framework of a codesign scheme that is able to manage complex situations in real environments [13] better than other algorithms [14] and mimic some behavior of the mammalians [15].

This paper is organized as follows. First, the stages of McGM model are explained very briefly; after that, we tackle

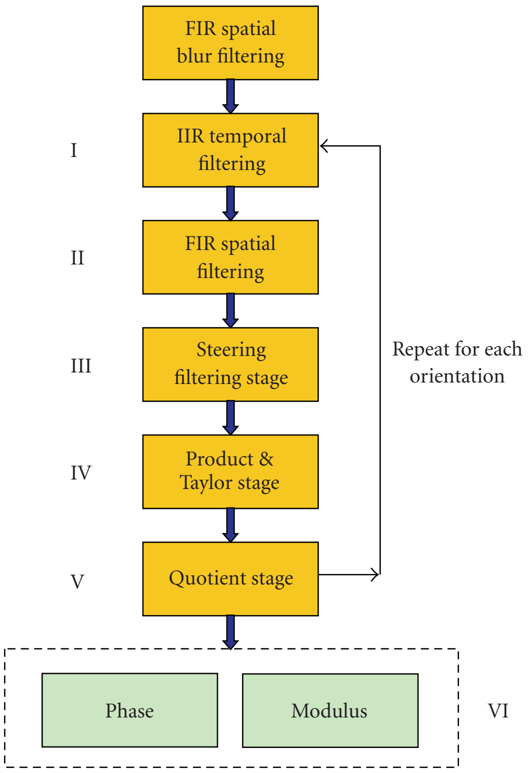


Figure 1: General scheme of the McGM algorithm.

the precision study of every conceptual stage, obtaining a set of bit width values which models the filters and the bit width stage required to obtain results that match with the statistical error metric requirements. From this previous study, we design the customizable architecture implementation attending to the original model plus several hardware modifications in order to improve the feasibility of the system. An example of this is the design of IIR filters replacing the original FIR filters due to the memory limitation of the prototyping platform, or the use of several information channels with a few bit width, replicating the nature of the brain (large number of neurons with very little precision for a few channels with huge information capacity) [14]. After that, we explain the coarse pipeline processing architecture and the platform and language used in our systems. Finally, quality results, hardware associated cost, and comparisons to other implementations are shown.

2. Multichannel Gradient Model (McGM)

The original algorithm was proposed by Johnston and Clifford, and we have applied Johnston's description of the McGM model [9], while adding several variations to improve the viability of the hardware implementation. Figure 1 shows a simplified scheme of the algorithm.

2.1. IIR Filtering

A temporal IIR filter is modeled from its original FIR description due to the limitation of available memory in our prototyping platform [15, 16]. The result is a recursive filter with only two frames of latency, being o and i the output and input, respectively, of the filter and a_i , b_i the coefficients from our previous work [14, 15]:

$$O(n) = a_1 i(n-1) - b_1 o(n-1) - b_2 o(n-2), \qquad (1)$$

where $a_1 = e^{-1/\alpha}/\alpha^2$; $b_1 = 2e^{-2/\alpha}$, $b_2 = e^{-2/\alpha}$, and α drives the peak in the temporal impulse response function. It is calibrated with a frame peak value equals to 10 following a critical flicker fusion limit of 60 Hz, according to the human visual system evidences [11]:

$$R(t) = \frac{1}{\sqrt{\pi}\tau\alpha} e^{-(\ln(t/\alpha)/\tau)_2}. \qquad (2)$$

Attending to the original algorithm, we need to perform the order zero, one and two derivatives, which represent our first triplet of information to be processed, as shown in Figure 1. The derivatives are obtained applying a gradient operator of minimal length (+1,−1) to (1):

$$\begin{aligned} T_0(n) &= o(n-1), \\ T_1(n) &= o(n) - o(n-2), \\ T_2(n) &= o(n) - 2o(n-1) + o(n-2). \end{aligned} \qquad (3)$$

2.2. FIR Spatial Filtering

A set of spatial FIR filters is modeled by the next impulse response corresponding to bidimensional Gaussians and their separable derivatives:

$$\begin{aligned} \frac{d^n}{dx^n}(G_O) &= \frac{d^n}{dx^n}\left(\frac{e^{-(x^2+y^2)/2\sigma^2}}{\sigma\sqrt{2\pi}}\right) \\ &= Hn\left(\frac{x}{\sqrt{2}\sigma}\right)Hn\left(\frac{y}{\sqrt{2}\sigma}\right)\left(\frac{-1}{\sqrt{2}\sigma}\right)^{2n} \\ &\quad \cdot \left(\frac{e^{-(x^2+y^2)/2\sigma^2}}{\sigma\sqrt{2\pi}}\right), \end{aligned} \qquad (4)$$

where σ represents the spread of the Gaussian and H_n is the Hermite polynomial of order n . The convolution is done in a separable way, taking derivatives in x and y directions up to sixth and second order, respectively, due to bioinspired and robustness reasons [11–13]. The aim of this stage is to cover enough spread area of information channels that allow us to contribute to the calculus when any of them are null due to many reasons, such as noise. Therefore, we have three spatial structures, each one containing a pyramidal set of several filters corresponding to Gaussians and their different derivatives.

2.3. Steering Stage

The steering stage represents the approach to projecting the space-temporal filters calculated in previous stages, under the different orientations. Being n and m the order in x and y directions, respectively, θ the angle projected, D the derivative operator, and G_O the Gaussian expression, we obtain the general expression of the filter rotated in the space as a linear combination of filters belonging to the same order basis [14]. Thus, we have to apply this transformation to each value:

$$G^{\theta}_{n,m}(x,y) = \left[\sum_{k=0}^{n}\binom{n}{k}(D_x\cos\theta)^k(D_y\sin\theta)^{n-k}\right] \cdot \left[\sum_{i=0}^{n}\binom{m}{i}(-D_x\sin\theta)^i(D_y\cos\theta)^{m-i}\right]G_0. \tag{5}$$

2.4. Taylor Expansion Stage

In this stage a truncated Taylor expansion is done, substituting it for the point on the space-time image in order to further enhance the algorithm. To perform this, it is necessary to use each oriented filter previously calculated. This expansion is highly versatile and represents a robust information structure of the sequence in space and time:

$$I^{\theta}(x+p,y+q,t+r) = \sum_{i=0}^{l}\sum_{j=0}^{m}\sum_{k=0}^{n}\frac{p^i q^i r^k}{i!\,j!\,k!}\frac{\partial^n}{\partial x^i\partial y^j\partial t^k} \times I^{\theta}(x,y,t). \tag{6}$$

With this, it is necessary to differentiate each Taylor expansion respect to x, y, t, calling these derivatives X, Y, T, and forming the following sextet of quotient as shown in the quotient stage:

$$\begin{bmatrix} X^{\theta} = \partial I^{\theta}/\partial x \\ Y^{\theta} = \partial I^{\theta}/\partial y \\ T^{\theta} = \partial I^{\theta}/\partial t \end{bmatrix}_{3\times1} \longrightarrow \begin{bmatrix} X^{\theta}X^{\theta} & X^{\theta}Y^{\theta} & X^{\theta}T^{\theta} \\ Y^{\theta}Y^{\theta} & Y^{\theta}T^{\theta} & T^{\theta}T^{\theta} \end{bmatrix}_{2\times3}. \tag{7}$$

2.5. Quotient Stage (General Primitives) and Following Stages

This is the last stage belonging to the common path, where a quotient of every sextet's component is computed from every measurement of the product of steered Taylor expansion differentiates:

$$\begin{bmatrix} Y^{\theta}Y^{\theta}/T^{\theta}T^{\theta} & X^{\theta}Y^{\theta}/X^{\theta}X^{\theta} & X^{\theta}T^{\theta}/X^{\theta}X^{\theta} \\ Y^{\theta}Y^{\theta}/X^{\theta}X^{\theta} & X^{\theta}Y^{\theta}/Y^{\theta}Y^{\theta} & X^{\theta}T^{\theta}/T^{\theta}T^{\theta} \end{bmatrix}_{2\times3}. \tag{8}$$

The architecture of the core is branched in two separated ways, modulus and phase, with different bit operations working independently, containing products, several quotients, and even trigonometric operations as arctangent, which are performed in software. The details of the software stages can be found in previous works [14, 15] being the final aim to recover a dense representation of motion. Therefore, we have two values for each input pixel corresponding to modulus and phase of the velocity, *that is*, velocity projection in x and y directions, following the next expressions

$$\text{Phase} = \tan^{-1}\left(\frac{\left(\frac{X\cdot T}{T\cdot T}+\frac{X\cdot T}{X\cdot X}\left(1+\left(\frac{X\cdot Y}{X\cdot X}\right)^2\right)^{-1}\right)\sin(\theta)+\left(\frac{Y\cdot T}{T\cdot T}+\frac{Y\cdot T}{Y\cdot Y}\left(1+\left(\frac{X\cdot Y}{Y\cdot Y}\right)^2\right)^{-1}\right)\cos(\theta)}{\left(\frac{X\cdot T}{T\cdot T}+\frac{X\cdot T}{X\cdot X}\left(1+\left(\frac{X\cdot Y}{X\cdot X}\right)^2\right)^{-1}\right)\cos(\theta)-\left(\frac{Y\cdot T}{T\cdot T}+\frac{Y\cdot T}{Y\cdot Y}\left(1+\left(\frac{X\cdot Y}{Y\cdot Y}\right)^2\right)^{-1}\right)\sin(\theta)}\right), \tag{9}$$

$$\text{Modulus}^2 = \det\frac{\begin{vmatrix} \frac{X\cdot T}{X\cdot X}\left(1+\left(\frac{X\cdot Y}{X\cdot X}\right)^2\right)^{-1}\cos\theta & \frac{X\cdot T}{X\cdot X}\left(1+\left(\frac{X\cdot Y}{X\cdot X}\right)^2\right)^{-1}\sin\theta \\ \frac{Y\cdot T}{Y\cdot Y}\left(1+\left(\frac{X\cdot Y}{Y\cdot Y}\right)^2\right)^{-1}\cos\theta & \frac{Y\cdot T}{Y\cdot Y}\left(1+\left(\frac{X\cdot Y}{Y\cdot Y}\right)^2\right)^{-1}\sin\theta \end{vmatrix}}{\begin{vmatrix} \frac{X\cdot T}{X\cdot X}\left(1+\left(\frac{X\cdot Y}{X\cdot X}\right)^2\right)^{-1}\frac{X\cdot T}{T\cdot T} & \frac{X\cdot T}{X\cdot X}\left(1+\left(\frac{X\cdot Y}{X\cdot X}\right)^2\right)^{-1}\frac{Y\cdot T}{T\cdot T} \\ \frac{Y\cdot T}{Y\cdot Y}\left(1+\left(\frac{X\cdot Y}{Y\cdot Y}\right)^2\right)^{-1}\frac{Y\cdot T}{T\cdot T} & \frac{Y\cdot T}{Y\cdot Y}\left(1+\left(\frac{X\cdot Y}{Y\cdot Y}\right)^2\right)^{-1}\frac{Y\cdot T}{T\cdot T} \end{vmatrix}}. \tag{10}$$

3. Precission Study (Bit Width Analysis)

We have designed a specific strategy to define the bit width required in each conceptual stage following this previous algorithm. The basic idea is to transform every calculus in the model, applying a chained process of quantization. For the sake of clarity, if the parameters of the convolution are the bit width of the input I, the length of the filter L, the mask size M, we can compute the output bit width simply shifting the range the output bit width O:

$$\text{stage}_n = \frac{2^O}{2^{I+L+M}}\text{stage}_{n-1}. \tag{11}$$

Applying this method in each stage, we obtain a set of values that throw back the transformation between floating point domain and integer domain, getting a tradeoff between bit width and affordable error.

As the metric error value, we have proposed the most common ones used in the specialized literature, such as Barron's vector [8] and Galvin's couple of metrics [6], where *Vc* and *Ve* are the values of the correct and experimental velocities, respectively, and $g^{\perp}$ is the normal component to the Galvin vector difference:

$$\vec{v} = \frac{(v_x, v_y, 1)}{\sqrt{v_x^2 + v_y^2 + 1}} \longrightarrow \psi_{\text{BARRON}} = \arccos(\vec{v_c} \cdot \vec{v_e}),$$
$$\psi_{\text{GALVIN}} = \|\vec{v_c} - \vec{v_e}\|,$$
$$\psi_{\text{GALVIN}\perp} = \|(\vec{v_c} - \vec{v_e}) \cdot \hat{g}^{\perp}\|. \tag{12}$$

We have also taken into account the simple error measures (absolutes and relatives) relative to modulus and phase:

$$\psi_{\text{MOD}} = \left| \|\vec{v_e}\| - \|\vec{v_c}\| \right|,$$
$$\psi_{\text{FAS}} = \left| \arctan(\vec{v_{cy}}/\vec{v_{cx}}) - \arctan(\vec{v_{ey}}/\vec{v_{ex}}) \right|. \tag{13}$$

Regarding the stimuli, we have used synthetic compositions of sine waves of different spatial frequencies and the famous stimulus of diverging tree and translating tree [17], commonly used to evaluate optical flow. As a result, we obtain the set of precision parameters that are applied in the model attending to the range of affordable error. Figure 2 shows the bit width of the stages performed in hardware, and Table 1 contains the final values chosen, for an FIR Blur filter length of 5 pixels, FIR spatial filter of 23 pixels, and IIR temporal filter equivalent to an FIR length of 21 frames, with a more detailed analysis available in [14].

4. Codesign Process

The system has been designed as a codesign process working with an asynchronous pipeline (micropipeline). The PC feeds the FPGA with a stream of frames through a bank of memory connected to PCI bus. The board takes a continuous stream of pixels at its input (1 byte/pixel); however, we employ 32 bits at the output, coming back to the PC, where they are reordered and written to the hard disk. We have selected Handel C to implement this core, using DK tool [18]. Relating to the prototyping board, an AlphaData RC1000 board has been used, which includes a Virtex 2000E-BG560 chip and 4 SRAM banks of 2 MB each [19]. The memory banks can be accessed both from the FPGA and the PCI bus, Figure 3 showing the communication scheme of the codesign system between the external memory banks, FPGA, and the host platform.

We have implemented a bit width precision defined version of the model, that we called "semihardware" version or SmHW; furthermore the next step is to implement different hardware cores for examining the tradeoff between accuracy and efficiency. We have developed in the FPGA two kinds of platforms that are called "basic" (HWbas) and "extended" (HWext) architectures. The SW version is implemented using the temporal FIR filtering, 24 orientations (each 15°), the SmHW version keeps the same number of orientations, although the implementation of the IIR filters and the Taylor Expansion is not completed (only are used the 65% of the weights). The basic architecture has one less order of spatial differentiation than the versions commented above, and it has only 18 orientations (each 20°), remaining the rest of the parameters constant. The extended architecture has one additional order less than the basic and also decreases in the number of orientations, taking 8 orientations (each 45°). Table 2 summarizes the main differences between these versions attending to the nature of temporal filter, the final spatial derivative order, the number of orientations, and the density of the weights used in the expansion.

Table 1: Parameters of each stage (100% density).

Stage	Bit width	Error (%) phase	mod
I	$F_1 = 6$, $O_{1\_r} = 9$	3.64	4.95
II	$F_2 = 8$, $O_{2\_r\_I} = 9$, $O_{2\_r\_II} = 10$	4	4.95
III	$W_3 = 6$, $O_{3\_r} = 10$	4.73	6.24
IV	$W_4 = 11$, $O_{4\_r} = 17$	5.31	9.01
V	$O_{5\_r} = 12$	5.52	12.83

Being F_1 temporal IIR, F_2 spatial FIR, W_3 steering weight, W_4 Taylor expansion, O_{i_r} bit width output of stage *i*,.

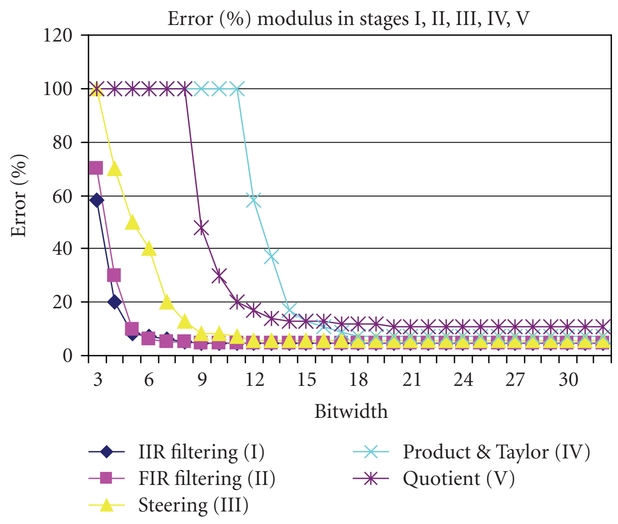
Table 2: Summary of the different implementations.

Main differences	SW	SmHW	HWbas	HWext
Temp filter	FIR	IIR	IIR	IIR
Esp. filter	6	6	5	4
Orientations	24	24	18	8
Taylor weights	100%	65%	65%	65%

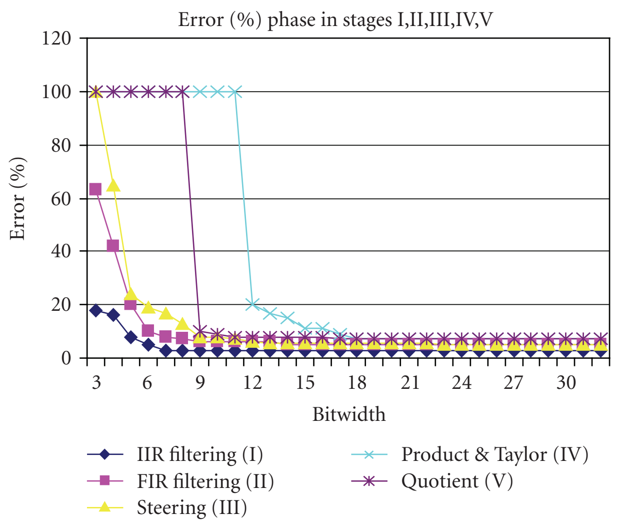
5. Results

We have analyzed the resources required by the platform and also the number of cycles (NCs) of each stage in Table 3. Every stage belonging to both architectures has been designed as customizable, scalable, and modular.

The basic architecture computes initial blur filter in order to remove aliasing components, IIR temporal filtering that performs the temporal derivatives, FIR spatial filtering, *that is,* spatial derivatives, and steering filtering that project the results onto the whole space (the SW prefix denotes that these stages are performed in software). This architecture contains the processing scheme belonging to most of gradient-based



(a)



(b)

FIGURE 2: Evaluation of the bit width needed in the modulus (a) and phase (b) converting the data to fixed point.

TABLE 3: Slices and memory requirements and number of cycles for basic and extended architectures.

Pipeline stage	Basic architectures			Extended architectures		
	Slices (%)	Block RAM (%)	MC	Slices (%)	Block RAM (%)	NC
Blur filter	289 (2%)	1%	4	289 (2%)	1%	4
IIR temporal filtering	190 (1%)	1%	9	190 (1%)	1%	9
FIR spatial filtering	1307 (7%)	36%	17	1307 (7%)	36%	17
Steering	5961 (31%)	2%	15	2012 (10%)	2%	29
Product and Taylor	SW	SW	SW	5952 (31%)	13%	24
Quotient	SW	SW	SW	8831 (46%)	19%	21

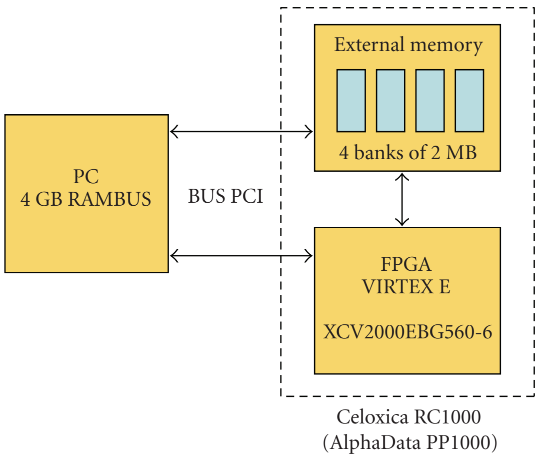


FIGURE 3: Scheme of the communication process.

optical flow models, thus it could be considered as a motion preprocessor [15, 16]. The extended architecture is able to cover more stages and is focused in the specific McGM algorithm, implementing all the stages commented previously, plus a Taylor expansion, Taylor product (their derivative products), and the quotient stage as shown in Figure 4.

5.1. Hardware Cost

The basic architecture consumes 41% of the board slices, with every stage being performed with parameter values very close to the original model (derivatives in x up to order 5, 18 orientations in the steering stage), implementing 4 stages. Nevertheless, the extended architecture requires 97% of the development board.

5.2. Performance

Related to the number of cycles, we have noted the Xilinx timing analyzer tool [20] to be very conservative; thus we can increase the throughput around 25%–35% if we clock the system manually from the values obtained. The slower stage in the basic architecture is the FIR filtering, while the last stages designed need the maximum number of Block RAMs and slices due to the computation being performed replicating the spatial convolution (FIR filter) concurrently

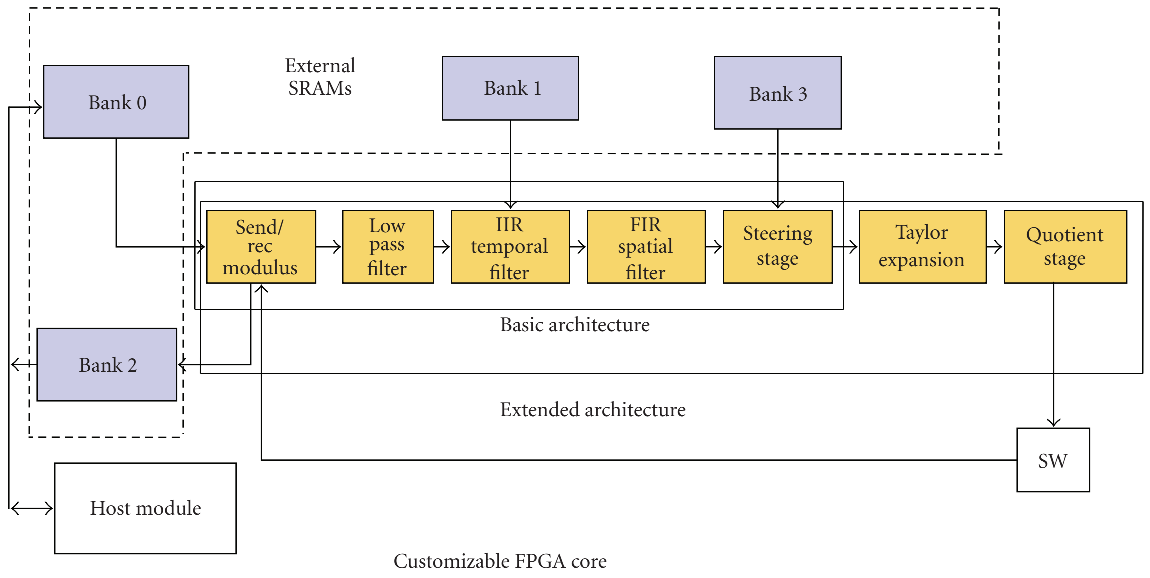


Figure 4: Scheme of the two architectures working with an asynchronous pipeline.

for n orientations until order m in x. Nevertheless, in the extended architecture we must keep resources for the next stages, removing some contributions and parallelizing the processing scheme in discrete groups, which replace the whole group entirely concurrently. For instance, the steering stage is performed with fewer terms and with reduced parallelization level, requiring almost the double of cycles. Applying this strategy of keeping enough resources in the prototyping board, we can extend the model to additional stages. We can see in Figure 4 the global codesign scheme and the two architectures involved, representing the transactions between external RAM (grey blocks) and the stages. The stage corresponding to IIR filter has to keep 3 frames using the bank number one, the steering stage reads the orientation weights from bank number three, and the send/receive modulus connects the input/output data between the FPGA and the host system via the PCI bus using DMA transfer. Figure 5 shows the performance for the whole systems using chained stages, attending to the pixel/seconds processed, concluding that it is possible to compute 177 frames/second with a resolution of 128×96 pixels in the basic architecture, and 37.9 frames/second for the extended architecture.

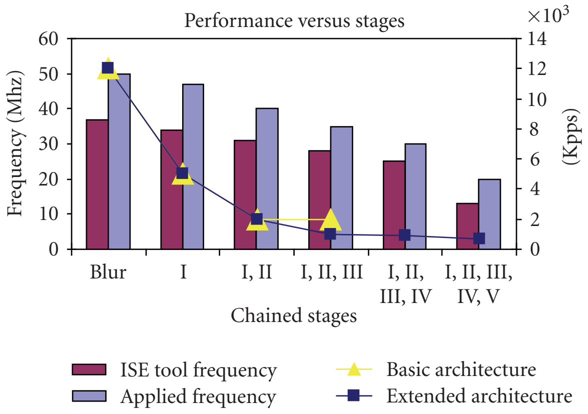


Figure 5: Throughput of the pipeline (Kpps) and frequency corresponding to basic and extended architectures.

5.3. Quality of the Results

An accuracy analysis has been carried out, being possible to examine the quality of the results under different transformations and metrics, as we can see in Figure 6. The phase and modulus metrics (difference between values) show a good behavior regarding the implementation changes, while Barron's metric seems to go well keeping the proportion accuracy under changes, but Galvin and Galvin perpendicular metrics suffer with the implementation change from SW to HW. It is due to the nature of the metric, which gives an idea about how the algorithm copes with the Aperture Problem [8], this topic being discussed in previous work [14]. Despite restricting every version in terms of precision parameters one step further until finally taking the extended architecture, in general the error values are delimited reasonably.

5.4. Some Visual Results

Figure 7 shows some visual results corresponding to different versions of our system, concretely SW versus HWbas. It can be noted that while the SW version keeps a calculus density close to 100% (middle row in Figure 7), HWbas loses some points due to precision bit width (bottom row in Figure 7), *that is*, the bit number of the parameters in each stage. The input sequence, called diverging tree (upper row in Figure 7) has a divergent structure where the modulus is supposed to vary poorly and the phase is changing regularly over 360°. Since we are working with synthetic sequences, we can estimate the error without any ambiguity. Also we have used

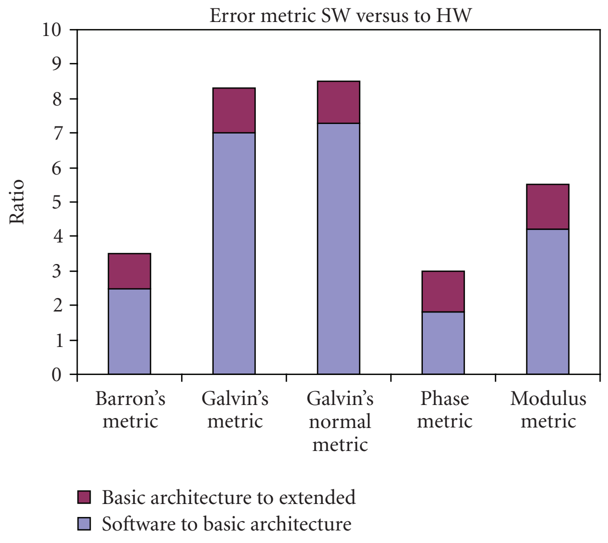


Figure 6: Quality of different implementations.

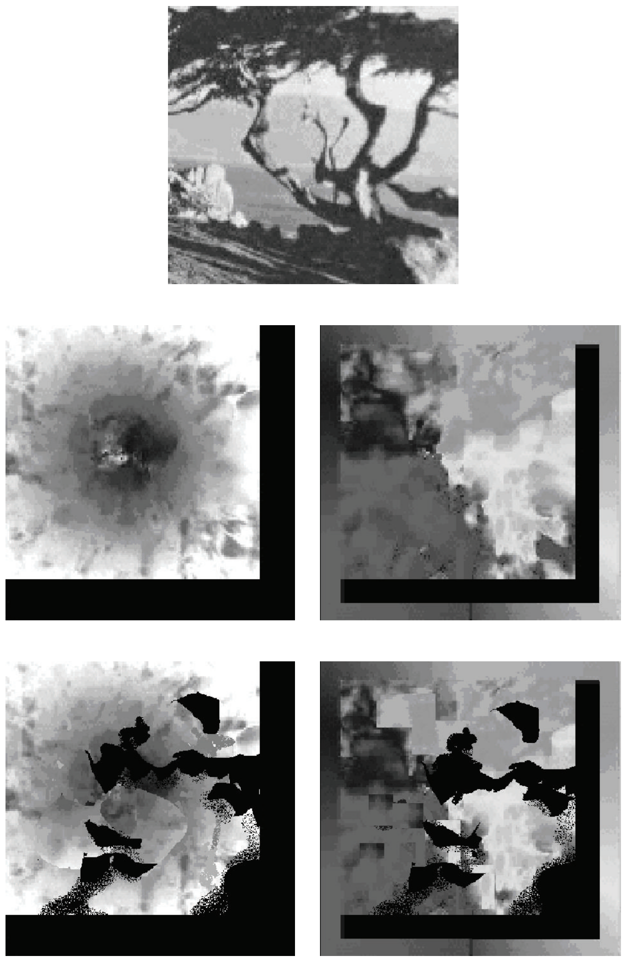


Figure 7: Some visual results corresponding to the software version versus the basic architecture (diverging tree sequence). Left hand indicates velocity modulus and right hand velocity phase.

Table 4: Summary of the different implementations for the Yosemite sequence. NP means not provided.

Models	Average error	Standard deviation	Density
Described here (HWbas)	5.5°	12.3	100%
Described here (HWext)	7.2°	11.1	100%
Described here (HWext)	6.1°	6.2	60%
Described here (HWext)	4.3°	3.1	20%
Díaz et al. [21]	18.30°	15.8°	100%
Díaz et al. [22]	7.6°	NP	<55%
Martín et al. [25]	NP	NP	<50%

the translating tree sequence, where modulus is changing from left to right and the phase is practically almost the same.

6. Comparison with other Approaches

There are other gradient optical flow models implemented in hardware [21, 22], belonging to the Lucas and Kanade algorithms [23] and to Horn and Schunk approximations [24, 25], while in Table 4 we can see the average error for different metrics, although only we compare the Barron's metric since the cited authors do not provide other measurements.

Attending to the errors, our implementation provides better results than the other approaches, even with calculation density 100%. Nevertheless, the final results are improved if the points where the scene structure changes, *that is,* points smaller than a determinate temporal derivative, are filtered. This is caused by a least squares process being performed at the end of the algorithm for calculating the modulus and the phase final values. The points filtered would force the slope of the linear regression to be very small, with the value of velocity is almost null.

Regarding throughput, we are able to calculate more than 2000 Kpixel/s in the basic Architecture and about 1000 Kpixel/s in the extended. It would locate our implementation between those in [23, 26], enough for real-time purposes, although it could be improved using a board with more resources that is used here and increasing the parallelism level.

The error using the diverging and translating tree sequences [17] is shown in Figure 7, and it is obtained with different metrics regarding the expressions (12)-(13).

7. Conclusion

We have developed an FPGA-based implementation of a bioinspired robust motion estimation system with an associated complexity higher than those found in other gradient-based models commonly used in the literature. The study of precision calibrates the model and adjusts the bit width needed for keeping a tradeoff solution between accuracy and efficiency, acting as a bridge between software and hardware and estimating the cost to convert every stage from floating to fixed point. Taking the results from this precision study, different hardware moduli have been designed, organizing this in two high parallelized architectures. The first one, referred to as basic architecture and common to optical flow gradient models, is a superconvolutive processor orientated along multiple angles. It could be used as a starting point for many computer vision algorithms, not necessarily restricted to the motion estimation field, like change detection, stereo, or even biometry techniques such as real-time signature recognition. The second architecture, called extended, is focused in the Mutichannel Gradient Model, and includes the truncated Taylor expansion representation of space temporal information of the scene, its three differentiates respect space and time, and the quotients of the products of these last functions. The rest of the stages, called velocity primitives, corresponding to the expressions (8)-(9) are performed in software in the framework of a codesign process, where the final modulus value is a quotient of determinants and the final phase is an arctangent. This extension can be implemented using a board with more resources than the VIRTEX 2000 E and, depending on the accuracy required, using a structure based on LUTs or implementing a CORDIC core. Both architectures are scalable and modular, and also extensible to one device with more resources that our prototyping platform.

Additionally, the resources consumed have been evaluated as well as the throughput and the accuracy of the designed coprocessors. All models come forward with asynchronous segmented architectures (micropipelines). Regarding quality, the average error has been compared using Barron's metric, since other authors do not provide results with other metrics; also the throughput of the design has been compared with other implementations. This work generates dense optical flow maps up to 80 frames/second and 185 frames/second for a resolution of 128×96 in the extended and basic architectures, respectively. The present contribution opens the door to embed complex bioinspired systems that require a huge quantity of computation. We are currently improving the system to extend the model to a fully stand alone platform also to deal with stereo vision. Several application fields are though to use it, such as motion illusion detection or video compression.

Acknowledgments

This work was partially supported by Projects TEC2007-68074-C02-01/MIC, TIN2005-05619-2004-07032 (Spain), EU Project DINAM-VISION (DPI2007-61683), and an EU "Marie Curie" Fellowship (QLK5-CT-1999-50523). The authors would like to thank the anonymous reviewers for their insightful suggestions and Professor Johnston and Dr. Dale, from the Vision Group at University College London, for their help and support during this research.

References

[1] C. Mead, "Neuromorphic electronic systems," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1629–1636, 1990.

[2] C. Mead, *Analog VLSI and Neural Systems*, Addison-Wesley, Reading, Mass, USA, 1989.

[3] H.-S. Oh and H.-K. Lee, "Block-matching algorithm based on an adaptive reduction of the search area for motion estimation," *Real-Time Imaging*, vol. 6, no. 5, pp. 407–414, 2000.

[4] C.-L. Huang and Y.-T. Chen, "Motion estimation method using a 3D steerable filter," *Image and Vision Computing*, vol. 13, no. 1, pp. 21–32, 1995.

[5] S. Baker and I. Matthews, "Lucas-Kanade 20 years on: a unifying framework," *International Journal of Computer Vision*, vol. 56, no. 3, pp. 221–255, 2004.

[6] B. McCane, K. Novins, D. Crannitch, and B. Galvin, "On benchmarking optical flow," *Computer Vision and Image Understanding*, vol. 84, no. 1, pp. 126–143, 2001.

[7] H. Liu, T.-H. Hong, M. Herman, T. Camus, and R. Chellappa, "Accuracy vs efficiency trade-offs in optical flow algorithms," *Computer Vision and Image Understanding*, vol. 72, no. 3, pp. 271–286, 1998.

[8] J. L. Barron, D. J. Fleet, and S. S. Beauchemin, "Performance of optical flow techniques," *International Journal of Computer Vision*, vol. 12, no. 1, pp. 43–77, 1994.

[9] A. Johnston and C. W. G. Clifford, "A unified account of three apparent motion illusions," *Vision Research*, vol. 35, no. 8, pp. 1109–1123, 1995.

[10] A. Johnston and C. W. G. Clifford, "Perceived motion of contrast-modulated gratings: predictions of the multi-channel gradient model and the role of full-wave rectification," *Vision Research*, vol. 35, no. 12, pp. 1771–1783, 1995.

[11] A. Johnston, P. W. McOwan, and C. P. Benton, "Robust velocity computation from a biologically motivated model of motion perception," *Proceedings of the Royal Society B*, vol. 266, no. 1418, pp. 509–518, 1999.

[12] P. W. McOwan, C. Benton, J. Dale, and A. Johnston, "A multi-differential neuromorphic approach to motion detection," *International Journal of Neural Systems*, vol. 9, no. 5, pp. 429–434, 1999.

[13] A. Johnston, P. W. McOwan, and C. P. Benton, "Biological computation of image motion from flows over boundaries," *Journal of Physiology-Paris*, vol. 97, no. 2-3, pp. 325–334, 2003.

[14] G. Botella, *Robust optical flow implementation in reconfigurable hardware*, Ph.D. thesis, University of Granada, Granada, Spain, 2007, ISBN 978-84-338-4381-4.

[15] G. Botella, E. Ros, M. Rodríguez, A. García, and S. Romero, "Pre-processor for bioinspired optical flow models: a customizable hardware implementation," in *Proceedings of the 13th IEEE Mediterranean Electrotechnical Conference (MELECON '06)*, pp. 93–96, Málaga, Spain, May 2006.

[16] G. Botella, E. Ros, M. Rodríguez, and A. García, "Bioinspired robust optical flow in a FPGA system," in *Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA '06)*, Dubrovnik, Croatia, August-September 2006.

[17] The input sequence was created by David Fleet at Toronto University and can be obtained from: ftp://ftp.csd.uwo.ca/pub/vision/TESTDATA.

[18] Handel-C Languaje reference manual and DK tool. Celoxica company, 2007.

[19] AlphaData RC1000 product, 2006, http://www.alpha-data.com/adc-rc1000.html.

[20] "Timing Analysis and Optmization of Handel-C designs for Xilinx chips," Celoxica application note AN 68 v1.1, 2005.

[21] J. Díaz, E. Ros, F. Pelayo, E. M. Ortigosa, and S. Mota, "FPGA-based real-time optical-flow system," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 16, no. 2, pp. 274–279, 2006.

[22] J. Díaz, E. Ros, R. Rodriguez-Gomez, and B. del Pino, "Real-time architecture for robust motion estimation under varying illumination conditions," *Journal of Universal Computer Science*, vol. 13, no. 3, pp. 363–376, 2007.

[23] B. D. Lucas and T. Kanade, "An iterative image registration technique with an application to stereo vision," in *Proceedings of DARPA Image Understanding Workshop*, pp. 121–130, Washington, DC, USA, April 1981.

[24] B. K. P. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1–3, pp. 185–203, 1981.

[25] J. L. Martín, A. Zuloaga, C. Cuadrado, J. Lázaro, and U. Bidarte, "Hardware implementation of optical flow constraint equation using FPGAs," *Computer Vision and Image Understanding*, vol. 98, no. 3, pp. 462–490, 2005.

[26] Z. Wei, D.-J. Lee, and B. E. Nelson, "FPGA-based real-time optical flow algorithm design and implementation," *Journal of Multimedia*, vol. 2, no. 5, pp. 38–45, 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 634306, 9 pages
doi:10.1155/2008/634306

Research Article

Design of a Mathematical Unit in FPGA for the Implementation of the Control of a Magnetic Levitation System

Juan José Raygoza-Panduro, Susana Ortega-Cisneros, Jorge Rivera, and Alberto de la Mora

Departamento de Electrónica, Centro Universitario de Ciencias Exactas e Ingenierías (CUCEI), Universidad de Guadalajara, Boulevard Marcelino García Barragan 1421, Guadalajara, Jal. 44430, Mexico

Correspondence should be addressed to Juan José Raygoza-Panduro, juan.raygoza@cucei.udg.mx

Received 2 July 2008; Revised 9 October 2008; Accepted 30 October 2008

Recommended by Gustavo Sutter

This paper presents the design and implementation of an automatically generated mathematical unit, from a program developed in Java that describes the VHDL circuit, ready to be synthesized with the Xilinx ISE tool. The core contains diverse complex operations such as mathematical functions including sine and cosine, among others. The proposed unit is used to synthesize a sliding mode controller for a magnetic levitation system. This kind of systems is used in industrial applications requiring high level of mathematical calculations in small time periods. The core is designed to calculate trigonometric and arithmetic operations in such a way that each function is performed in a clock cycle. In this paper, the results of the mathematical core are shown in terms of implementation, utilization, and application to control a magnetic levitation system.

1. Introduction

Mathematical control equations in an FPGA reconfigurable device is an important aspect in the design of arithmetic blocks when implementing control algorithms [1]. A well-known method utilized in the implementation of arithmetic operations in FPGAs is based upon the coordinate rotation digital computer (CORDIC) algorithm [2–6] which has become the standard solution for the implementation of complex operations in FPGAs.

This paper proposes the design of a mathematical unit dedicated to the implementation of control algorithms that involve several sequences of complex mathematical functions calculations.

Traditionally, the development of complex arithmetic functions in FPGA devices has resulted in difficulties to implement such operations. Therefore, the elaboration of mathematical operations in Xilinx FPGAs is proposed through the core generator [7]. The objective of this paper is to explain the development of a core capable of performing mathematical operations such as trigonometric functions in a clock cycle, using an alternative method of the core generator suggested by the manufacturer.

In order to construct such cores, the architecture of the mathematical unit is established by the user with Java software, in which the input and output parameters are defined as well as the functions needed to perform the desired control algorithm. This tool facilitates the users' implementation of mathematical blocks in FPGAs, simplifying the flow design to the adjustment of the interconnection of the required blocks in the main program described in VHDL. This reduces the designer's workload during the implementation stage of control algorithms. The tool is capable of implementing 16 different types of mathematical functions which may be described according to the required algorithm. The maximum number of functions that can be implemented depends on the available resources of the FPGA.

When the VHDL code generator is activated, a window initially appears, asking the characteristic of the input-output variables. The longitude of the input data indicating integer and decimal bits must be specified. At this point, selection of the functions to be implemented according to the control algorithm is made, and finally the code generator creates a file containing the description of each block in VHDL language ready to be synthesized

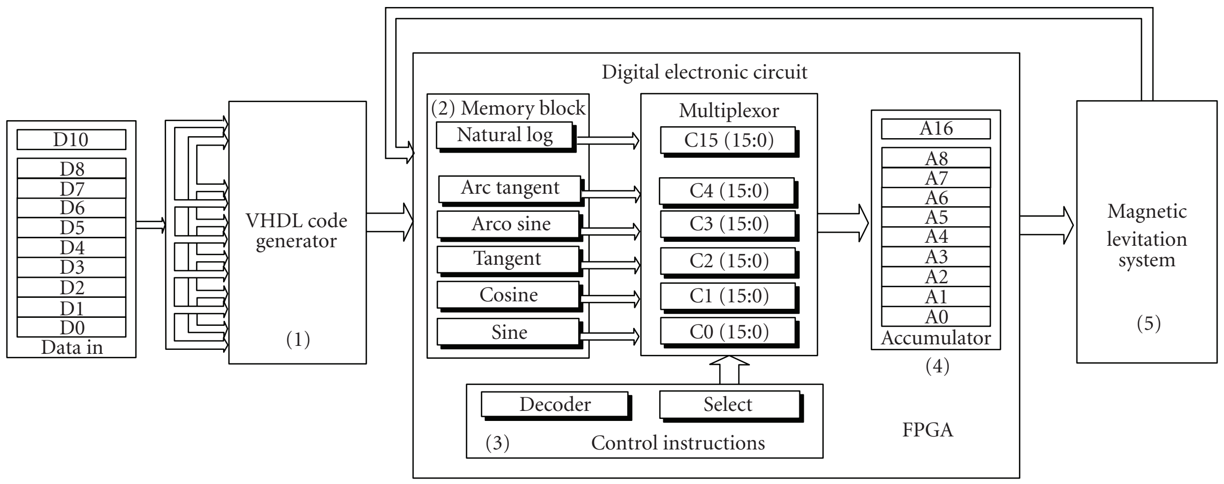


Figure 1: Block diagram of the FPGA control.

by the Xilinx ISE tool [8]. Each function might be an independent module that can be interconnected with the rest of the blocks in order to represent the equations that describe the desired algorithm. Trigonometric functions are implemented in the embedded memory of the FPGA. The advantage of solving complex functions with preloaded tables can be clearly seen in computing time, simplifying the execution of a mathematical function to the transfer of data from memory to the accumulator register.

The control algorithm of a magnetic elevation system is presented in order to provide an implementation study for the proposed mathematical unit. This system deals with the levitation of steel objects aided by a controlled electromagnetic force that is equal and opposed to the gravitational force acting on the steel object. This type of control is actually applied in commercial magnetic levitation (MAGLEV) trains [9].

2. Description of the Mathematical Unit

The mathematical unit has been developed with a Java program that generates blocks of mathematical functions in VHDL. The complete system is composed of 5 main modules, as shown in Figure 1, (1) VHDL code generator, (2) RAM or ROM memory block for mathematical operations, (3) control unit for instructions, (4) accumulator registers for results, and (5) magnetic levitation system.

The mathematical unit was functionally designed in VHDL code with instantiation of RAM or ROM memories that were created through the program generator functions, elaborated in Java language, especially for this job which is described in Section 2.1. The memories were programmed with input parameters assigned by the user, allowing the data input to have a suitable format according to the designers needs.

2.1. VHDL Code Generator

As previously mentioned, the proposed mathematical unit is capable of solving trigonometric functions in a clock cycle by using preestablished data tables. To accomplish this, a program was developed in Java language that calculates the values of the trigonometric or mathematical functions within the range of values defined by the user, followed by the creation of tables with the calculated values, and uses a RAM or ROM memory to store these data values and then to translate them into the description hardware language VHDL. The program defines the architecture, entity, and process which automatically adds to the libraries, reducing user time and the definition of each block to only the definition of ranges and precise values of input and output data in its integer and decimal parts.

The software function generator reduces the computational burden to the FPGA by using a standard computer to calculate the possible results of mathematical functions that require only one parameter in the instantiation of a RAM or ROM memory.

The program creates the desired function as an entity in VHDL with an input and an output of the selected size. The VHDL has syntax standards, which are contained in the libraries. The program generates the necessary lines for use by the corresponding libraries. The entity block is also created at the same time, along with the input data, ready to be synthesized by the Xilinx ISE simulator. The list of mathematical function values is calculated with the program code generator.

An example is shown in Table 1. This table corresponds to the calculation of the cosine function, which is implemented in a ROM memory of 16 bits × 1024 lines. The address bus is identified with the letters a9 to a0, where a0 is the least significant bit. Before executing the program generating code, the data format specifies the required bits for the integer part and the decimal part.

Table 1: Selection of preestablished data for a cosine function.

Address bus			Data bus				
a9-a8	a7-a4	a3-a0	d16	d15-d12	d11-d8	d7-d4	d3-d0
00	0000	0000	0	1000	0000	0000	0000
00	0000	0001	0	0111	1111	1111	1110
00	0000	0010	0	0111	1111	1111	1110
00	0000	0011	0	0111	1111	1111	1010
00	0000	0100	0	0111	1111	1111	1000
00	0000	0101	0	0111	1111	1111	0010
00	0000	0110	0	0111	1111	1110	1110
00	0000	0111	0	0111	1111	1110	0110
00	0000	1000	0	0111	1111	1110	0000
00	0000	1001	0	0111	1111	1101	0110
00	0000	1010	0	0111	1111	1100	1110
00	0000	1011	0	0111	1111	1100	0010

The value of the angle is defined in radians at an interval from 0 to 3.99. In the example in Table 2, this quantity may be defined by the user in the program generator. The calculation of the cosine function is made considering the bits from a6 to a0 as the decimals of the parameter and the bits from a9 to a7 as the integer part. The result of the function is located in the data bus where d0 is the sign bit, d1 to d13 is the decimal part, and from d14 to d16 is the integer part of the data.

The following program code fragment is an example of the result of the VHDL mathematical functions, where numbers 9 and 16 are the defining entrance parameters that were programmed:

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.std_logic_arith.all;
use IEEE.std_Logic_UNSIGNED.ALL;
entity block is
Port(angle:in std_logic_vector(9 downto 0);
    result:out std_logic_vector(16 downto 0));
end block;
architecture behavior of block is
type func is array (0 to 1024) of std_logic_vector
(16 downto 0);
constant Content: func:=(
B"00000000000100000,"
B"00000000000100000,"
B"00000000000100000,"
B"00000000000100000,"
```

The critical functions programmed in C language turned a floating chain of bits as well as the same operation in inverse form. An example of the code follows:

```
acadena(Number_to_turning, chain_of_exit,
    decimal_of_exit, size_of_exit)
adouble(Chain_to_turning,Number_of_decimal)
acadena(15.25,chain_of_exit,2,6) // chain_of_exit will have the value of 111101
acadena(15.5,chain_of_exit,2,6) // chain_of_exit will have the value of 111110
acadena(10.5,chain_of_exit,2,6) // chain_of_exit will have the value of 101010
acadena(8.75,chain_of_exit,2,10) // chain_of_exit will have the value of 0000100011
adouble("100011,"2);    // The result is 8.75
adouble("100011,"1);    // The result is 17.5
adouble("100011,"0);    // The result is 35
```

In the program, the "acadena" function transformed the floating value of bits and the "adouble" function converted a floating value of bits. A part of the second version which was generated in Java language follows:

```
import java.io.*;
#1  class seno
#2  {
#3     public static String acadena(double X,
       int enteros,int longitud){
#4           double Y=0.0;
#5     if (X<0)
#6     {
#7           Y=Math.abs(Math.ceil(X));
#8     }else{
#9           Y=Math.abs(Math.floor(X));
#10     }
```

Table 2: Results obtained from the mathematical unit.

Angle degrees	Angle radians	Math unit result	Matlab result	Error obtained
0.00000	0.00000	1.00000	1.00000	0.00000
9.84771	0.17188	0.98527	0.98510	0.00017
14.77157	0.25781	0.96695	0.96692	0.00003
19.69542	0.34375	0.94150	0.94141	0.00009
24.61928	0.42969	0.90910	0.90906	0.00004
29.99076	0.52344	0.86611	0.86609	0.00002
34.91462	0.60938	0.82001	0.81995	0.00006
39.83847	0.69531	0.76785	0.76782	0.00003
95.79138	1.67188	−.10091	−.10083	0.00008
210.38294	3.67188	−.86266	−.86255	0.00012

In order to complete the conversion of the floating value to chain of bits, we followed a 2-stage process; firstly, the whole part becomes a chain of bits, and after the part decimal is turned into a chain of bits. Later they are united in a single decimal number in binary code.

The conversion process starts with the whole part of the function; this requires rounding the smallest number (when positive) or rounding the largest number (when negative). Using the "ceil" function one can obtain the rounding of the number and using the "floor" function one can round the whole part. Since the conversion algorithm uses positive numbers, the "abs" function is used to take the absolute value from the rounded number. The variable "res" keeps the final result from the conversion.

The code generator program allows the usage of RAM or ROM memories and selection of these will depend on the application required. For example, when using ROM memories, these are implemented with the internal resources of the FPGA augmenting the utilization of the circuit; the flexibility of using these memories is their facility to adjust the size of the word and required address for the precise calculations that will be stored in them. When RAM memories are selected, as these are embedded, they do not impact the available resources in the FPGA, allowing a huge logic capacity for other circuit implementation, the disadvantage that it is limited to the implementation of variable arrays in the word longitude and address bus.

With the objective of observing the units behavior during the calculation of different trigonometric functions, a sequence of operations was established for the resolution of the functions with different angles. The obtained results are shown in Table 2. The first column corresponds to the evaluation angles; the second column is equivalent to the first column in radians; the third column shows the results of the cosine function obtained with the mathematical unit presented; the fourth column has the results obtained with Matlab; the last column presents the difference between the value calculated with Matlab and the value obtained with the mathematical unit.

It is important to emphasize that the mathematical function sequence can be carried out to form complete equations which are calculated and stored in a ROM or RAM memory, to be used later in the implementation of individual block control equations, that are capable of being calculated in a clock pulse, optimizing the calculation time.

Table 3: Utilization of the mathematical unit blocks.

Function	Utilization			
	Sel.	Slices	LUTs	TEGs
Cosine	0	11%	8%	6 121
Sine	1	11%	8%	6 282
Square root	2	1%	1%	50
Tangent	3	8%	6%	5 230
Arc cosine	4	3%	3%	2 799
Arc sine	5	3%	3%	2 807
Arc tangent	6	3%	3%	2 952
Exponential	7	5%	4%	3 956
Radians	8	4%	3%	3 303
Hyperbolic tangent	9	2%	1%	1 571
Hyperbolic cosine	10	5%	4%	4 375
Hyperbolic sine	11	5%	4%	4 391
Natural log	12	1%	1%	80
Inverse	13	1%	1%	1 345
Log base 10	14	1%	1%	671
Degrees	15	3%	3%	3 082

2.2. Description of the Mathematical Unit Operation

The mathematical unit was implemented in a FPGA Virtex II. The results of the utilization are shown in Table 3. The utilization of slices, LUTs, and total equivalent gate (TEG) is presented in independent columns. The column "sel" refers to the instruction code that mathematical unit executes. This makes 16 trigonometric and mathematical functions which may be selected through a control word of 4 bits.

The total circuit utilization is 95% of the available slices in the FPGA and 74% of LUTs, being equivalent in TEG to 58 157 out of 1 000 000 of the available total on the Xilinx Virtex II.

3. Application to the Control of a Magnetic Levitation System

In order to prove the capacities of the mathematical unit, a sliding mode controller [10] was used to regulate a magnetic levitation system. This type of system is used in several applications such as frictionless bearings [11], high-speed maglev passenger trains [12], wind tunnel levitation models [13], molten metal levitation [14], and the levitation of metal slabs during industrial manufacturing process [15]. These systems have natural unstable nonlinear dynamics requiring closed-loop control designs for stabilization. Several control techniques have been applied to the stabilization of MAGLEV systems, such as I/O linearization [16, 17], backstepping [18], and sliding mode control [19], among others. The sliding mode control [10] has been extensively used in electromechanical systems due to its robustness to unknown bounded perturbations. Another characteristic of sliding mode control is the discontinuous nature of its control signal which switches from two states. This is an advantage because it avoids using pulse width modulation (PWM). The drawback of sliding mode control is that the switching signal has an infinite frequency and when implemented with common switching power devices with a frequency around 20 KHz, produces an output phenomenon called chattering; small oscillations around the set-point. Nowadays, there are power devices available with a switching frequency of at least 150 KHz, which common digital signal processor boards cannot support. To take full advantage of such switching devices, one needs high speed digital media such as FPGAs that can support and match high switching frequencies. In this case, the chattering problem is considerably reduced.

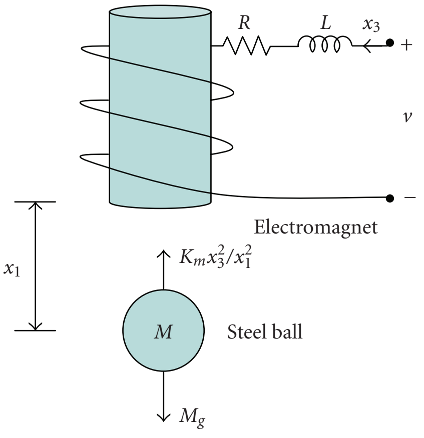


FIGURE 2: Block diagram of the FPGA control.

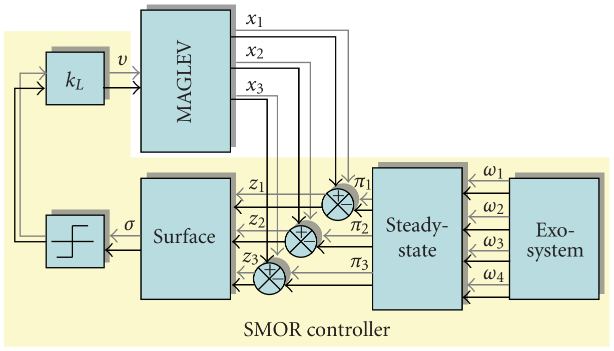


FIGURE 3: Schematic block diagram of SMOR.

3.1. Mathematical Model and Problem Formulation for the MAGLEV System

Figure 2 shows an schematic diagram of a maglev system.

The mathematical model of the MAGLEV system is given in the following equations [17]:

$$\begin{aligned} \dot{x}_1 &= x_2, \\ \dot{x}_2 &= g - \frac{k_m}{M}\frac{x_3^2}{x_1^2}, \\ \dot{x}_3 &= -\frac{R}{L}x_3 + \frac{1}{L}v, \\ y &= x_1 \end{aligned} \tag{1}$$

with state vector defined as $x = (x_1, x_2, x_3)^T$, where x_1 represents the position of the steel ball of mass M which is positively increasing in the downward position, x_2 is the velocity of the steel ball, x_3 is the current through the coil, v is the input voltage applied to the coil, and y is the output of the system. The constant parameters are the resistance of the coil denoted by R, the inductance denoted by L, g which is the gravitational constant and is considered as a known perturbation term, finally k_m which is the magnetic constant of the electromagnet.

The control problem is based upon forcing the output $y = x_1$ to track a reference signal $q(w)$. Therefore, one can consider the following output tracking error:

$$e = x_1 - q(w). \tag{2}$$

3.2. Sliding Mode Output Regulation for the MAGLEV System

The applied control design methodology is a combination of two important control techniques, output regulation theory (ORT) [20] and sliding mode control (SMC) [10]. The advantage of using ORT is that it plays an important role in trajectory output tracking and in the rejection of known disturbances. ORT deals with the problem of finding a control law such that the output of the controlled system can asymptotically track a signal generated by an exosystem and at the same time reject perturbations possible generated by the same exosystem. The nature of the control signal is continuous or smooth, and in this case PWM is required for implementation. When ORT is combined with SMC one obtains a control methodology commonly known as sliding mode output regulation (SMOR) [10] resulting in robust protection against unknown perturbations and avoids the use of PWM as just mentioned before.

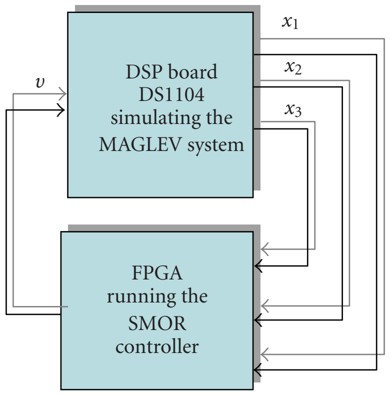

DSP board DS1104 simulating the MAGLEV system
FPGA running the SMOR controller
x_1
x_2
x_3
v

Figure 4: Schematic block diagram of HIL simulation.

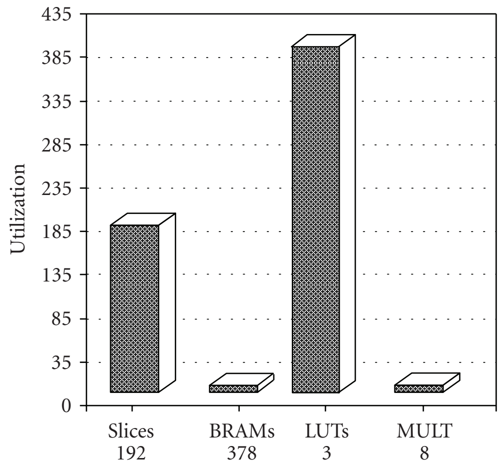


Figure 5: Utilization of the components in the FPGA Virtex II.

The exosystem is proposed as follows:

$$\begin{aligned}\dot{w}_1 &= -\alpha w_2,\\ \dot{w}_2 &= \alpha w_1,\\ \dot{w}_3 &= 0,\\ \dot{w}_4 &= 0\end{aligned} \tag{3}$$

with initial conditions $w_1(0) = w_2(0) = a$, $w_3(0) = b$, and $w_4(0) = c$, such that, the exosystem generates a reference output tracking signal for an MAGLEV system, which is chosen as $q(w) = w_1 + w_3$, that is, a sinusoidal shape signal with frequency α, peak value of $\sqrt{2}a$, and a dc bias value b. The reference signal is chosen in this way in order to test some trigonometric functions of the mathematical unit. In this case, the steel ball will move upward and downward as dictated by the amplitude and frequency of the reference signal.

What follows is the ideal steady state of operation for the MAGLEV system, that is, $\pi(w) = (\pi_1(w), \pi_2(w), \pi_3(w))^T$; this state is such that, if the original states of the MAGLEV, $x = (x_1, x_2, x_3)^T$, are driven to the ideal steady-state, then the output tracking error will asymptotically decay to zero, accomplishing the control objective. In order to find the steady state of operation one must solve the well-known Francis-Isidori-Byrnes [20] equations. In the case of the MAGLEV system results are as follows:

$$\frac{\partial \pi_1(w)}{\partial w} s(w) = \pi_2(w), \tag{4}$$

$$\frac{\partial \pi_2(w)}{\partial w} s(w) = d(w) - \frac{k_m}{M}\frac{\pi_3^2(w)}{\pi_1^2(w)}, \tag{5}$$

$$\begin{aligned}\frac{\partial \pi_3(w)}{\partial w} s(w) &= -\frac{R}{L}\pi_3(w) + \frac{1}{L}c(w),\\ 0 &= \pi_1(w) - q(w)\end{aligned} \tag{6}$$

with $s(w) = (-\alpha w_2, \alpha w_1, 0, 0)^T$. Note that the ideal steady-state value for e is obviously zero. Using this fact, one easily calculates from (6) $\pi_1(w) = w_1 + w_3$, replacing $\pi_1(w)$ in (4) one finds that $\pi_2(w) = -\alpha w_2$. Substituting $\pi_2(w)$ in (5), one reckons the expression for $\pi_3(w)$ as $\pi_3(w) = (w_1 + w_3)\sqrt{(M/k_m)(w_4 + \alpha^2 w_1)}$. The $c(w)$ variable represents the steady-state value for the control input v, but it is not neccesary to calculate such expression when using SMC actions. Let us define the steady-state error as

$$\begin{aligned} z &= (x - \pi(w)) = \begin{pmatrix} z_1 & z_2 & z_3 \end{pmatrix}^T \\ &= \begin{pmatrix} x_1 - \pi_1(w) & x_2 - \pi_2(w) & x_3 - \pi_3(w) \end{pmatrix}^T. \end{aligned} \tag{7}$$

The dynamic equation for (7) with tracking error e (2) can be obtained from (1) as

$$\dot{z}_1 = z_2 + \pi_2(w) - \frac{\partial \pi_1(w)}{\partial w} s(w), \tag{8}$$

$$\dot{z}_2 = d(w) - \frac{k_m}{M}\frac{(z_3 + \pi_3(w))^2}{(z_1 + \pi_1(w))^2} - \frac{\partial \pi_2(w)}{\partial w} s(w), \tag{9}$$

$$\begin{aligned}\dot{z}_3 &= -\frac{R}{L}(z_3 + \pi_3(w)) + \frac{1}{L}u - \frac{\partial \pi_3(w)}{\partial w} s(w),\\ e &= z_1 + \pi_1 - q(w).\end{aligned} \tag{10}$$

Now, one defines the sliding function and control as

$$u = -kL\,\text{sign}(\sigma), \qquad \sigma = z_3 + \Sigma_1 z^1,\ k > 0, \tag{11}$$

where sign is the typical signum function, with $\Sigma_1 = (\Sigma_{1,1}\Sigma_{1,2})$ and $z^1 = (z_1, z_2)^T$.

Making use of a rigorous stability analysis by means of a Lyapunov function [10], one finds a stability condition for gain k:

$$k > \left| \left(\frac{1}{L}\right) v_{eq}(z, w) \right|, \tag{12}$$

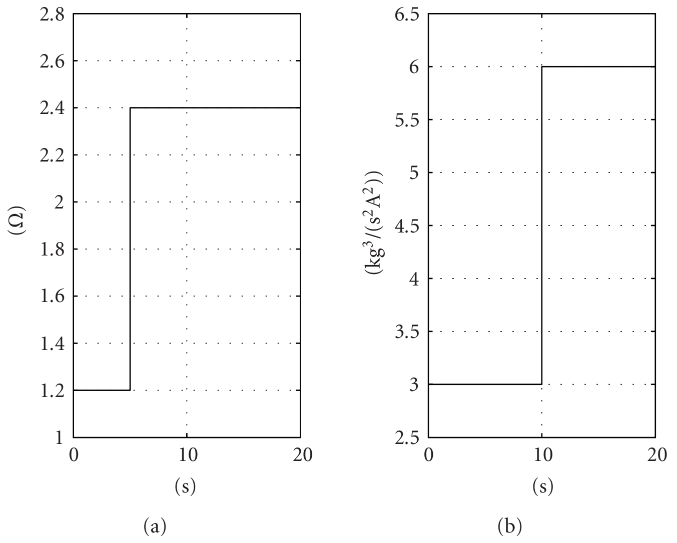


Figure 6: (a) Resistance variation, (b) magnetic constant variation.

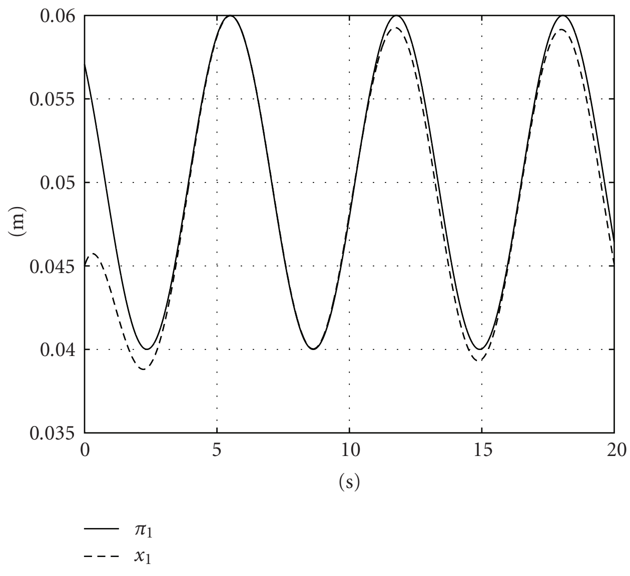


Figure 7: Output tracking signal.

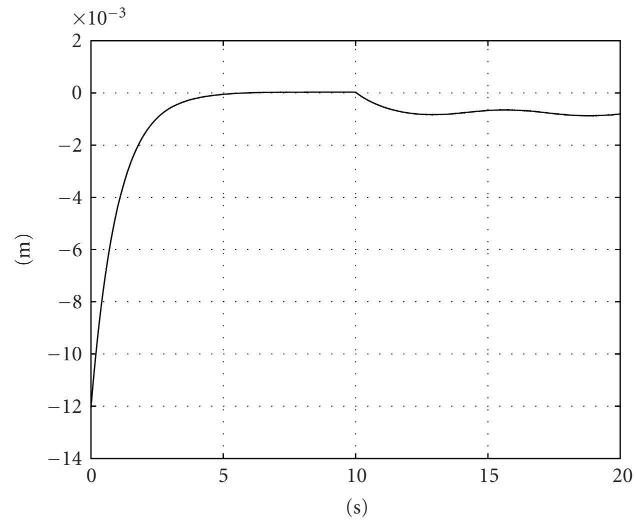


Figure 8: Output error signal.

where $v_{eq}(z, w)$ is a solution of $\dot{\sigma} = 0$, namely,

$$\begin{aligned} v_{eq} &= R(z_3 + \pi_3) + L\left(\frac{\partial \pi_3}{\partial \omega}\right)s(\omega) \\ &\quad - L\sum_{1,1}\left(z_2 + \pi_2 - \left(\frac{\partial \pi_1}{\partial \omega}\right)s(\omega)\right) \\ &\quad - L\sum_{1,2}\left(d(\omega) - \left(\frac{k_m}{M}\right)\frac{(z_3+\pi_3)^2}{(z_1+\pi_1)^2} - \left(\frac{\partial \pi_2}{\partial \omega}\right)s(\omega)\right). \end{aligned} \tag{13}$$

If condition (12) is satisfied then $\sigma = 0$ is guaranteed, implying that z_3 can be calculated from (11) as

$$z_3 = -\Sigma_1 z^1. \tag{14}$$

That is, the differential equation (10) is unnecessary as its solution (14) is now known. The remaining differential equations for z_1 and z_2 are obtained by replacing (14) in (8) and (9). This residual dynamic is known as the sliding mode dynamic. This dynamic is made stable by the proper choice of Σ_1. An easy way to stabilize the sliding mode dynamic is by using its linear approximation at the origin as shown here:

$$\begin{aligned} \dot{z}^1 &= (A_{11} - A_{12}\Sigma_1)z^1 + \text{H.O.T}, \\ \dot{w} &= Sw, \\ e &= z_1 + \pi_1(w) - q(w) \end{aligned} \tag{15}$$

with A_{11} and A_{12} being as proper dimensions matrices obtained from linear approximation, and where H.O.T. stands for higher-order terms, that vanish at the origin. Now, Σ_1 is chosen so that the matrix $(A_{11}-A_{12}\Sigma_1)$ is Hurwitz or has

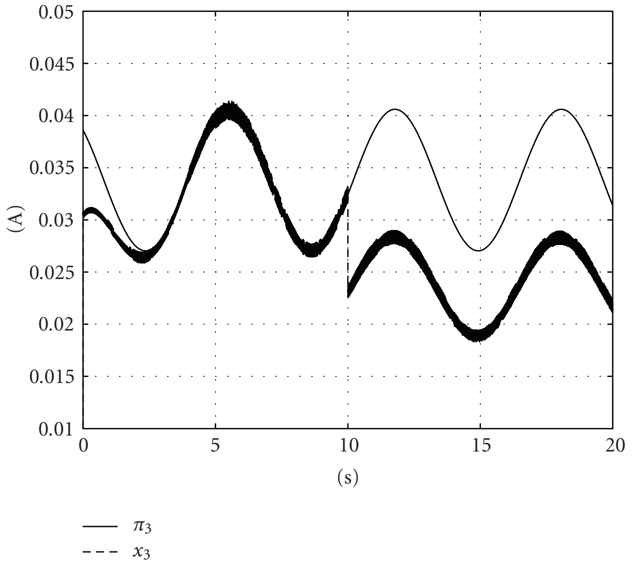


FIGURE 9: Current signals.

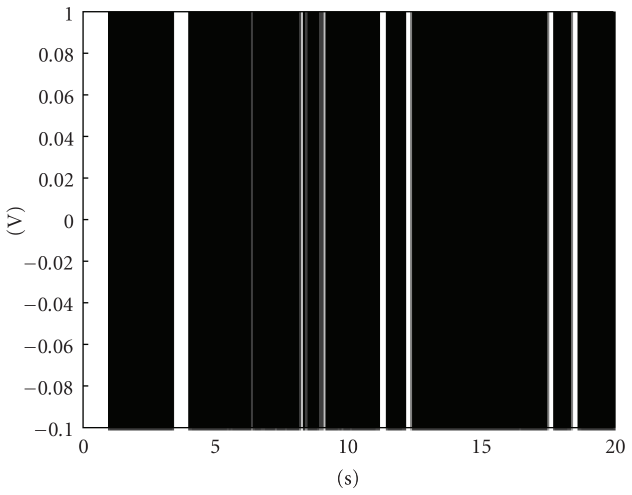


FIGURE 10: Voltage control signal.

negative real part poles. In this case $\lim_{t\to\infty} z^1(t) = 0$ and as a consequence by (14) z_3 tends to zero too. By continuity, using $\pi_1(w) = w_1 + w_3$ one finally finds that the output tracking error e asymptotically tends to zero, satisfying the control objective. Finally a closed-loop block diagram is presented in Figure 3.

4. Control Algorithm Implementation Results

The control algorithm was tested using an FPGA virtexII XE2V1000-4fg256, and the plant dynamic was simulated using the DSP board DS1104 from DSPACE. This type of simulation is known as hardware-in-the-Loop (HIL) simulation [21]. HIL simulation is a real-time simulation form. It differs from real-time simulation by the addition of a hardware component in the loop as an FPGA. This technique is increasingly being used in the development and testing of complex real-time embedded systems. Moreover, the complexity of the plant dynamic under control is commonly simulated in a graphical environment as SIMULINK from Matlab. In our case the plant dynamic was created in SIMULINK and then downloaded to the DSP board DS1104 in order to arrange the I/O ports. Figure 4 shows a simple diagram of the HIL simulation that was performed.

4.1. FPGA Implementation Results

The system is declared as an entity of three inputs that represents the position of the ball x_1, the velocity of the ball x_2, the current through the coil x_3, and the output voltage v closed loop with the MAGLEV system. The internal variables used for the calculation of the equations use a word of 32 longitude bits—15 bits to represent the integer part, 16 bits for the decimal part, and 1 bit to represent the sign. The variable v corresponds to the final calculation of the system and has a word longitude of 64 bits—4 for integers, 1 for signs, and 59 for decimals, providing the necessary accuracy for the stability of the system. The total processing time of the calculations of one cycle in the FPGA is 202 nanoseconds, representing maximum processing speed of up to 21 nanoseconds. Figure 5 shows the utilization of the components in the FPGA virtexII XE2V1000-4fg256, with 3% of slices, 3% of LUTs, 7% of RAMs, and 20% of multipliers. The device has sufficient resources available to implement additional circuits.

4.2. Closed-Loop System in Implementation Results

The nominal parameters of the MAGLEV system are $k_m = 3\,\mathrm{kgm^3/s^2A^2}$, $M = 0.14\,\mathrm{kg}$, $g = 9.8\,\mathrm{m/s^2}$, $R = 1.2\,\Omega$, $L = 1\times 10^{-3}\,\mathrm{H}$. The constant values of the exogenous signals (2) are $a = 0.0070716$, $b = 0.05\,\mathrm{m}$, and $c = 9.8$. Taking the nominal parameters of the MAGLEV system, the following pairs of matrices are calculated:

$$A_{11} = 013920, \qquad A_{12} = 0 - 579.6551. \tag{16}$$

The control parameters that appear in (11) are as follows:

$$k = 100, \qquad \Sigma_{1,1} = -12.9423, \qquad \Sigma_{1,2} = -12.2492. \tag{17}$$

The matrix Σ_1 in (11) is calculated using the LQR function provided in Matlab.

To verify the robustness properties, some plant parameter variations are introduced which can be seen in Figure 6, where R and k_m may change up to 100% from their nominal values. It is worth to mention that the perturbation term generated by the variation of R satisfies the matching condition [10], but not the variations on k_m.

Figure 7 shows the tracking of the output signal where can be appreciated a good performance for $0 \le t < 5$. But for $5 \le t < 10$ where the perturbation term due to the variation in R is present, and the output still performs well due to the matching conditions. Finally, the unmatched perturbation term due to the variation of k_m, appearing at $t \geq 10$ adversely affects the maglev system but the output still performs well.

Figure 8 shows the output tracking error where can be appreciated the transient and steady-state responses can be observed.

Figure 9 shows π_3, which represents the ideal steady-state behavior of the current. It can be seen that the current becomes different to π_3 for $t \geq 10$ due to the unmatched perturbation.

Finally, Figure 10 shows the voltage input signals where the discontinuous nature of the control signal can be appreciated. The main advantage of having discontinuous control signals is that it avoids the use of PWM as mentioned in [10], therefore, facilitating a straightforward implementation of the control action.

5. Conclusion

This work has presented the results of a program generator for VHDL code developed in Java language and designed to implement a mathematical unit prototyped and implemented in reconfigurable FPGA circuits from Xilinx. The mathematical unit was used to implement the control algorithm of a magnetic levitation system, accomplishing the requirements of speed and precision necessary to operate under nominal conditions. The code generator tool allows the implementation of blocks containing complex operations which may be grouped in the same memory, letting operations to run in a clock pulse, based on the calculation of functions through preestablished tables. Moreover, the HIL simulation test platformed has facilitated the verification of the results obtained when the physical plant is not available.

References

[1] S. Ortega-Cisneros, J. J. Raygoza-Panduro, J. Suardíaz Muro, and E. Boemo, "Rapid prototyping of a self-timed ALU with FPGAs," in *Proceedings of International Conference on Reconfigurable Computing and FPGAs (ReConFig '05)*, p. 8, Puebla City, Mexico, September 2005.

[2] J. S. Walther, "A unified algorithm for elemetary functions," in *Proceedings of the AFIP Spring Joint Computer Conference (SJCC '71)*, vol. 38, pp. 379–385, AFIPS Press, Montvale, NJ, USA, 1971.

[3] F. Cardells-Tormo and J. Valls-Coquillat, "Optimisation of direct digital frequency synthesisers based on CORDIC," *Electronics Letters*, vol. 37, no. 21, pp. 1278–1280, 2001.

[4] T. C. Chen, "Automatic computation of exponentials, logarithms, ratios and square roots," *IBM Journal of Research and Development*, vol. 16, no. 4, pp. 380–388, 1972.

[5] C.-S. Wu, A.-Y. Wu, and C.-H. Lin, "A high-performance/low-latency vector rotational CORDIC architecture based on extended elementary angle set and trellis-based searching schemes," *IEEE Transactions on Circuits and Systems II*, vol. 50, no. 9, pp. 589–601, 2003.

[6] H. Dawid and H. Meyr, "VLSI implementation of the CORDIC algorithm using redundant arithmetic," in *Proceedings of IEEE International Symposium on Circuits and Systems (ISCAS '92)*, vol. 3, pp. 1089–1092, San Diego, Calif, USA, May 1992.

[7] J. E. Volder, "The CORDIC trigonometric computing technique," *IRE Transactions on Electronic Computers*, vol. EC-8, no. 3, pp. 330–334, 1959.

[8] Xilinx, 2007, http://www.xilinx.com.

[9] M. Ono, Y. Sakuma, H. Adachi, et al., "Train control characteristic and the function of the position detecting system at the Yamanasi Maglev test line," in *Proceedings of the 15th International Conference on Magnetically Levitated Systems and Linear Drives (MAGLEV '98)*, pp. 184–189, Mt. Fuji, Yamanashi, Japan, April 1998.

[10] V. I. Utkin, A. G. Loukianov, B. Castillo-Toledo, and J. Rivera, "Sliding mode regulator design," in *Variable Structure Systems: From Principles to Implementation*, A. Sabanovic, L. Fridman, and S. Spurgeon, Eds., p. 1943, IEE, London, UK, 2004.

[11] P. Allaire and A. Sinha, "Robust sliding mode control of a planar rigid rotor system on magnetic bearings," in *Proceedings of the 6th International Symposium on Magnetic Bearings (ISMB '98)*, pp. 577–586, Boston, Mass, USA, August 1998.

[12] Hyung-Woo Lee, Ki-Chan Kim, and Ju Lee, "Review of maglev train technologies," *in IEEE Transactions on Magnetics*, vol. 42, no. 7, pp. 1917–1925, 2006.

[13] R. J. M. Muscroft, D. B. Sims-Williams, and D. A. Cardwell, "The development of a passive magnetic levitation system for wind tunnel models," *SAE Transactions: Journal of Passenger Cars: Mechanical Systems*, vol. 115, no. 6, pp. 415–419, 2006.

[14] K. Im and Y. Mochimaru, "Numerical analysis on magnetic levitation of liquid metals, using a spectral finite difference scheme," *Journal of Computational Physics*, vol. 203, no. 1, pp. 112–128, 2005.

[15] B. V. Jayawant and D. P. Rea, "New electromagnetic suspension and its stabilization," *Proceedings of the Institution of Electrical Engineers*, vol. 115, no. 4, pp. 549–554, 1965.

[16] A. El Hajjaji and M. Ouladsine, "Modeling and nonlinear control of magnetic levitation systems," *IEEE Transactions on Industrial Electronics*, vol. 48, no. 4, pp. 831–838, 2001.

[17] W. Barie and J. Chiasson, "Linear and nonlinear state-space controllers for magnetic levitation," *International Journal of Systems Science*, vol. 27, no. 11, pp. 1153–1163, 1996.

[18] Z.-J. Yang, K. Kunitoshi, S. Kanae, and K. Wada, "Adaptive robust output-feedback control of a magnetic levitation system by K-filter approach," *IEEE Transactions on Industrial Electronics*, vol. 55, no. 1, pp. 390–399, 2008.

[19] F.-J. Lin, L.-T. Teng, and P.-H. Shieh, "Intelligent sliding-mode control using RBFN for magnetic levitation system," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 3, pp. 1752–1762, 2007.

[20] A. Isidori and C. I. Byrnes, "Output regulation of nonlinear systems," *IEEE Transactions on Automatic Control*, vol. 35, no. 2, pp. 131–140, 1990.

[21] B. Lu, X. Wu, H. Figueroa, and A. Monti, "A low-cost real-time hardware-in-the-loop testing approach of power electronics controls," *IEEE Transactions on Industrial Electronics*, vol. 54, no. 2, pp. 919–931, 2007.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 926851, 10 pages
doi:10.1155/2008/926851

Research Article

Burst-Mode Asynchronous Controllers on FPGA

Duarte L. Oliveira,[1] Marius Strum,[2] and Sandro S. Sato[1]

[1] *Electronic Engineering Division, Aeronautics Institute of Technology, Praça Marechal Eduardo Gomes 50, 12228-900 São José dos Campos, SP, Brazil*
[2] *Microelectronic Laboratory, Polytechnic School, University of São Paulo, Avenida Prof. Luciano Gualberto, Trav 3, 158, 05508-900 São Paulo, SP, Brazil*

Correspondence should be addressed to Duarte L. Oliveira, duarte@ita.br

Received 5 July 2008; Revised 8 October 2008; Accepted 30 October 2008

Recommended by Gustavo Sutter

FPGAs have been mainly used to design synchronous circuits. Asynchronous design on FPGAs is difficult because the resulting circuit may suffer from hazard problems. We propose a method that implements a popular class of asynchronous circuits, known as burst mode, on FPGAs based on look-up table architectures. We present two conditions that, if satisfied, guarantee essential hazard-free implementation on any LUT-based FPGA. By doing that, besides all the intrinsic advantages of asynchronous over synchronous circuits, they also take advantage of the shorter design time and lower cost associated with FPGA designs.

1. Introduction

Due to the increasing complexity of digital systems combined with the market drive for higher performance, there has been an increased interest about asynchronous circuits [1, 2]. Asynchronous circuits do not present clock distribution related problems like clock skew. The circuits have low power consumption, better modularity, robustness toward variations in temperature, and low emission of electromagnetic radiation [3]. One known weakness of asynchronous circuits has been the difficulty to design hazard-free circuits and to solve the critical races [3]. Furthermore, asynchronous circuits frequently cannot benefit from the use of FPGAs due to the extra difficulty imposed by their fixed architecture to deal with hazards [4].

Asynchronous circuits can be classified according to different criteria like its function (controller—datapath); delay model (delay insensitive—quasi-delay insensitive—speed independent—generalized fundamental mode (GFM)) [2]; styles (global asynchronous local synchronous—self-timed systems—micropipeline—speed-independent controllers—burst-mode controllers) [5–9].

Burst-mode asynchronous controllers proposed by Nowick [9, 10] are a popular class of finite state machines. They allow multiple inputs changes. They operate according to the GFM, meaning that a new state transition may only start when the whole circuit (gates and lines) is stable. This paper addresses burst-mode asynchronous controllers. Their advantages are the use of basic gates, similarity with synchronous design. These controllers have been adopted in important industrial and academic designs [11–13].

FPGAs are popular components for prototyping and production of digital circuits due to their low cost and short design time. Their focus has been on synchronous digital circuits. There have been some recent efforts to prototype asynchronous circuits on both commercial [14–17] and academic FPGAs [4, 18–20].

Burst-mode controllers are usually designed using a logic-driven design methodology [21]. There are two reasons why off-the-shelf FPGAs are not fit for burst-mode asynchronous controllers [4, 14, 22].

(1) The *mapping process* of burst-mode Booleans functions (equations of next state—controllers) to logic blocks (macrocells) may introduce *logic hazards*.

(2) The *internal routing* among logic blocks may introduce significant delays that may result in *essential hazards*.

1.1. Avoiding Logic Hazards in Burst-Mode Controllers

The burst-mode specification proposed by Nowick is *functional-hazard free* [23]. Nowick also proposed a method to produce *logic-hazard free* burst-mode Boolean functions [24]. Furthermore, Siegel et al. [25] proposed a technique to decompose large fan-in burst-mode Boolean functions without introducing *logic-hazards*. Finally, Maheswaran and Akella [15] and Hauck et al. [4] showed that if Booleans functions are *functional-hazard free* then they can be mapped on ordinary LUT-based FPGAs without presenting *logic hazards* [26].

1.2. Avoiding Essential Hazards in Burst-Mode Controllers

Yun and Dill [27] and Nowick and Coates [10] proposed the insertion of delay elements on the feedback wires to avoid essential hazards in burst-mode controllers. However, this solution is not adequate for FPGAs because these components are not designed to ease the insertion of delay elements. Furthermore, delay elements degrade the circuit cycle time, area, and reliability.

In this paper, we demonstrated a *sufficient condition* that guarantees *essential hazard-free operation* of *any type of burst-mode controller* when mapped on *any type of LUT-based FPGA* component without the need of extra delay elements. The proof is based on two new concepts: (1) *essential signals*; (2) *essential super states*. The essential hazard-free operation is guaranteed if the following conditions are satisfied:

(1) essential hazard-free specification: for all state transitions in a burst-mode specification, if the label contains a nonempty output burst, it must also contain at least one *essential* input signal;

(2) essential hazard-free implementation: starting from an essential hazard-free specification, while building the burst-mode flow map, all *single states* whose incident state transitions are labeled with nonempty output bursts must be transformed into *essential super states*.

Furthermore, whenever a burst-mode specification does not satisfy the first condition, we present two *functional transformations* that *create essential input signals* without altering the original functionality:

(1) reduction of input concurrency: transforms concurrent transitions into sequential transitions whenever acceptable (but there is a latency penalty);

(2) addition of dummy input signals (but there is an area penalty).

2. Hazard-Free BM Conditions

This paper is divided in four sections. Section 3 briefly explains the burst-mode specifications. Section 2 presents the essential signal and essential super-state concepts and explains the two functional transformations. Section 4 presents our method and illustrated with an example. Section 5 shows our experimental results presenting the latency and area penalties found on nine known and one homemade benchmark. Section 6 presents our conclusions and future work.

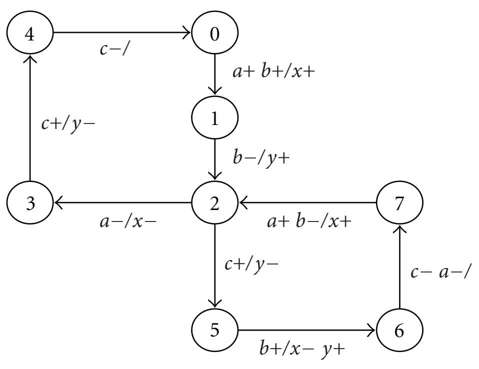
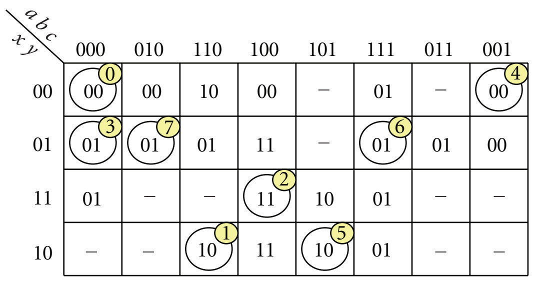


Figure 1: BM specification.



xy \ abc	000	010	110	100	101	111	011	001
00	00 (0)	00	10	00	–	01	–	00 (4)
01	01 (3)	01 (7)	01	11	–	01 (6)	01	00
11	01	–	–	11 (2)	10	01	–	–
10	–	–	10 (1)	11	10 (5)	01	–	–

Figure 2: BM flow map.

3. Burst-Mode Specification

The BM specification is represented as a state transition diagram. Each transition is triggered by an input burst (single- or multiple-input changes) causing the occurrence of an output burst (that may be empty or nonempty). It is necessary to define an initial state. State transitions are represented by arcs, which are labeled with their corresponding input/output bursts. The signals are always transition sensitive ($0 \rightarrow 1$, or $1 \rightarrow 0$). Input bursts may not be empty. The input signals are monotonic, changing only once during each state transition. The BM specification has to obey the polarity property, the unique entry point and the maximal set property [23].

Figure 1 shows a BM specification. The input signals are a, b, and c while the output signals are x and y. For example, state transition $7_{[a+\,b-/x+]} \rightarrow 2$ means that if a changes from 0 to 1 and b changes from 1 to 0, the output x will change from 0 to 1. State 0 is the initial state. Figure 2 shows the corresponding burst-mode flow map (2D map) [27]. Several tools, like Minimalist [28], 3D [27], and ATACS [29] have been proposed to synthesize controllers from a textual description of the burst-mode specification. These

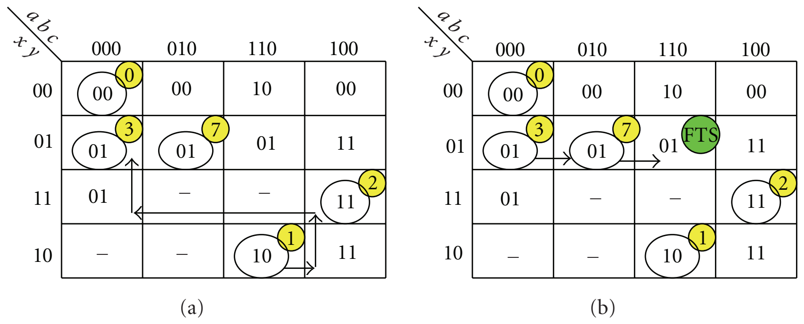


Figure 3: Part of the BM flow map of Figure 2: (a) path T2; (b) path T3 (final total state—FTS).

tools generate an independent *netlist* of the technology (next-state equations of the type sum of products).

BM asynchronous controllers may be subject to sequential hazards. Essential hazards, like transient essential hazard or steady-state essential hazard, are inherent to the sequential function and are not necessarily associated to a particular implementation of the circuit. The concept of essential hazard has been originally defined by Unger [30] in connection with fundamental-mode controllers.

This concept has been generalized for BM controllers and may be explained using the total state concept. A total state $A(I, O)$ is a vector composed of all the input (I) and output (O) signal values in the specification. A total state corresponds to one single cube (cell) on a burst-mode flow map (see Figure 2). For example, the total state 2 of the Figure 2 is $(a, b, c, x, y) = 10011$. There may be $n!$ paths on the BM flow map corresponding to the transition from total state $A(I_1, O_1)$ to total state $B(I_2, O_2)$ (labeled with an input/output burst I_b/O_b), n being the number input signals in I_b. These paths cover the set $\{A, B, C, \ldots, N\}$, where A is the initial total state, B is the final total state, and $C, \ldots, N$ are intermediary total states.

3.1. Essential Hazard

Generalized Unger Rule [30] (GUR): the Triple Sequential Input Burst

Let $A(I_1, O_1)$ and $B(I_2, O_2)$ be two total states in the BM flow map and I_b/O_b the input/output burst that activates the transition $A \to B$. Let N be the number of the input burst signals. Consider the following transitions sequence:

T1: $A_{[Ib]} \to B$; (transition 1 is $A \to B$ activated by I_b)

T2: $B_{[Ib\_\text{inverted\_polarity}]} \to C_i$ ($i = 1, \ldots, k$ are possible final states);

T3: $C_{[Ib]} \to D_j$ ($i = 1, \ldots, n$ are possible initial states), ($j = 1, \ldots, m$ are possible final states).

Definition 1. There is a potential *steady-state essential hazard* in the $A \to B$ transition if, applying the GUR rule, any final total state D_j $(j = 1, \ldots, m) \neq B$.

Definition 2. There is a potential *transient essential hazard* in the $A \to B$ transition if, applying the GUR rule, there is a total state I ($\neq A$ and $\neq B$) on any path of transitions $B \to C_i$ or $C_i \to D_j$ that produces an output signal different from any value occurring on any path of transition $A \to B$.

A potential essential hazard can be detected applying the GUR rule from any initial state. For example, Figures 3(a) and 3(b) show two paths for the $0 \to 1$ state transition on the BM flow map of Figure 2. Consider the $0_{[b+a+/x+]} \to 1$ path on Figure 3(a). According to the GUR rule we must apply the following activation sequence: T1($b+a+$), T2($b-a-$), T3($b+a+$). The corresponding paths on the BM flow table are

T1: $abcxy = \{00000 \to 01000 \to 11000 \to 11010\}$,

T2: $abcxy = \{11010 \to 10010 \to 10011 \to 00011 \to 00001\}$,

T3: $abcxy = \{00001 \to 01001 \to 11001\}$.

As the final total state (11001) after the last activation (T3) is different from the final total state (11010) after the first activation (T1), then a steady-state essential hazard has occurred. Figure 3(a) shows the path T2 and Figure 3(b) shows the path T3.

3.2. BM-EHF Condition

An input signal in a BM specification is a *context signal* in an $A \to B$ transition if it does not change during this transition (it is not on the label) while it is a *trigger signal* if it is labeled during this transition. The input burst of each state transition can be represented by an input transition cube (ITC). For example, the ITC for state transition $7 \to 2$ on Figure 1 is $abc = 220$ (2 means do not care). In this example a and b are trigger signals while c is a context signal (whose value is 0).

Definition 3. Let A and B be a pair of total states in a BM specification and I_b/O_b be the input/output burst for the $A \to B$ transition. Let E_s be one input signal ($E_s \in I_b$). E_s is an *essential signal* if it is a context signal on all transitions incident on state A and is a trigger signal on the transition $A \to B$.

For instance (see Figure 1), a, b, c are not essential on transitions $4 \to 0$, $1 \to 2$, and $2 \to 3$ because they are trigger

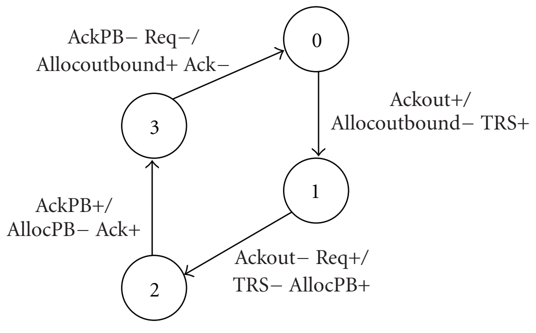


Figure 4: BM-EHF specification.

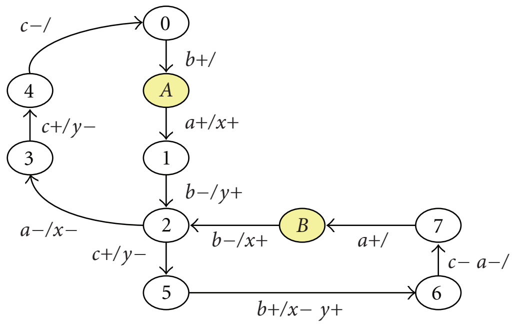


Figure 5: Concurrency reduction.

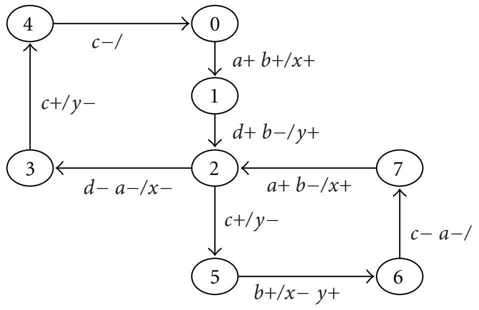


Figure 6: Inserting essential signal.

signals on transitions 3 → 4, 0 → 1, and 7 → 2. Signal b is essential on transition 7 → 2 because it is a context signal on transition 6 → 7. On transition 6 → 7 both a and c are essential signals.

Lemma 1. *A BM specification is essential hazard free (BM-EHF) if and only if for each state transition labeled by I_b/O_b, if $O_b \neq \varnothing$, there must be at least one essential input signal.*

Proof. Let T1$(A \rightarrow B)$ and T2$(B \rightarrow C)$ be two sequential state transitions of a BM-EHF specification. ITC_{T1} and ITC_{T2} are their respective input transition cubes. Suppose that the transition T2 input burst does not contain as essential signal. Then $\text{ITC}_{\text{T1}} \subseteq \text{ITC}_{\text{T2}}$, which means that the C final total state belongs to a path on ITC_{T1}. This fact violates Definitions 1 or 2. □

Figure 4 shows the *HP-mp-for-pkt* benchmark [12, 13]. On all transition labels there is at least one essential signal. Therefore, it is a BM-EHF specification.

There are two ways to transform nonessential hazard-free BM specifications into a BM-EHF specification.

3.3. Reduction of Input Burst Concurrency

The transformation consists of decomposing the input burst labeled on a state transition generating two-state transitions. For example, Figure 5 shows a reduced concurrency BM-EHF specification equivalent to the BM specification in Figure 1 in which the concurrency has been reduced. Analyzing the BM specification in Figure 1, we found state transitions 1 → 2 and 2 → 3 without essential signals. Decomposing state transitions $0_{[b+\,a+/x+]} \rightarrow 1$ into $0_{[b+/]} \rightarrow A_{[a+/x+]} \rightarrow 1$ and decomposing state transition $7_{[a+\,b-/x+]} \rightarrow 2$ into $7_{[a+/]} \rightarrow B_{[b-/x+]} \rightarrow 2$, we obtained the BM-EHF specification shown in Figure 5. It is EHF because transitions 4 → 0 and 7 → B contain empty output bursts while all other transitions contain essential signals.

3.4. Insertion of Essential Signals

This transformation consists of inserting the smallest number of dummy essential signals in all state transitions without essential signal. For example, Figure 6 shows an BM-EHF specification equivalent to the specification in Figure 1 in which adummy essential signal d has been added to state transitions 1 → 2 and 2 → 3. This transformation has a higher cost than the previous one because it increases the number of input signals (I_b), modifying the interaction with the external environment.

If one observes the 2 → 3 state transition in Figure 6, the conclusion is that a is essential on transitions 1 → 2 → 3, while d is essential on transitions 7 → 2 → 3.

3.5. Super-State Condition

Lemma 1 is a necessary and sufficient condition for an essential hazard free *specification* but not for hazard-free *implementation*. The super-state concept will guarantee the latter condition.

Definition 4 (super-state). Consider an input burst $I_b(a, b, \ldots, n)$ and an output burst $O_b(x, y, \ldots, m)$. We call a *super-state* the set of single total states defined by all 0/1 combinations of a subset S_{Ib} of the input burst signals, keeping fixed the remaining input signals and all the output signals.

Definition 5 (essential super-state). Consider a BM-EHF specification in which a total state F is reached by a set of N incident transitions $\{I_{t\_i}\}$ $i = 1, \ldots, N$. Each incident transition I_{t_i} is activated by an input burst I_{b_i}. Each input burst is labeled with a subset of the input signals set $\{I_s\}$. An *essential super-state* is the super-state defined by the union

	xy \ abc	000	010	110	100	101	111	011	001
$d = 0$	00	00 ⓪	00	10	00	——	——	——	00 ④
	01	01 ③	——	——	01 ③	——	——	——	00
	11	01	——	11 ②	11 ②	——	——	——	——
	10	10 ①	10 ①	10 ①	10 ①	——	——	——	——
$d = 1$	00	——	——	——	——	——	01	——	——
	01	01 ③	01 ⑦	01	11	01 ⑥	01 ⑥	01	——
	11	11 ②	11 ②	11 ②	11 ②	10	01	——	——
	10	——	——	10	11	10 ⑤	01	——	——

Figure 7: BM flow map with essential super-states (BM-ESS).

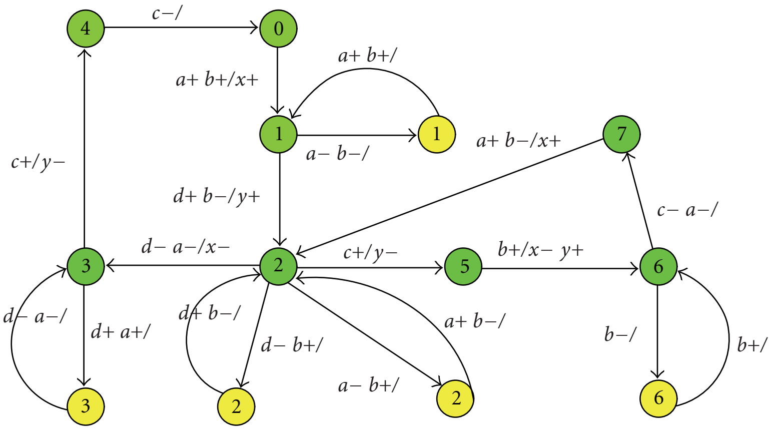


Figure 8: BM-EHF specification with essential super-states (BM-ESS).

$S_{\cup Ib} = \cup\{I_{b\_i}\}$ of all input signals active on the incident transitions set $\{I_{t\_i}\}$, labeled *with nonempty output bursts.*

An essential super-state BM flow map is derived from a BM-EHF specification by applying Definition 5 to all total states. Figure 7 is such a map for the specification in Figure 6. Cells in red are used to compose essential super-states. For example, the $0_{[a+b+/x+]} \rightarrow 1$ transition creates essential super-state 1 composed of four total states: $abcdxy = [000010, 010010, 100010, 110010]$. State 110010 is the final total state. Total state 2 may be reached from either state 1 or state 7. Applying Definition 5, we find that it must be composed of six total states (essential super-state 2): $abcdxy = [000111, 010111, 100111, 110111, 100011, 110011]$. This set of total states can be described by a cube (super-state transition cube—SSTC). Figure 8 shows the description in states transition diagram of the BM-ESS flow map.

Proposition 1. *If a total state F in a BM-EHF specification is reached by one or more incident transitions labeled with empty output bursts, then F is an essential super-state.*

Proof. Let $T1(A \rightarrow F)$ be a state transition with an empty output burst. $SSTC_A$ and $SSTC_F$ are super-state transition cubes for final total states A and F. As A must be essential, and as $SSTC_A[\text{output}] = SSTC_F[\text{output}]$ because both output bursts are empty, then F is also an essential super-state. □

Lemma 2. *The BM-EHF specification has an EHF implementation if and only if for ∀ total state A ∈ BM-EHF it is an essential super-state.*

Proof. Let $T1(B \rightarrow A)$ and $T2(A \rightarrow C)$ be state transitions with output bursts. $SSTC_A$ and $SSTC_C$ are the super-state transition cubes of final total states A and C. Suppose that the $T2(A \rightarrow C)$ input burst does not contain an essential signal. Then $SSTC_A[\text{input}] \subseteq SSTC_C[\text{input}]$ hence $SSTC_A[\text{output}] \neq SSTC_C[\text{output}]$. This means that A cannot be an essential super-state because this would violate Definition 5. □

4. Metodolology

Our method begins from the BM specification and implements the asynchronous controllers in the architecture of Huffman with feedback output. The synthesis procedure has five steps.

(1) If the BM specification satisfies Lemma 1 to go for Step (3), otherwise, Step (2).

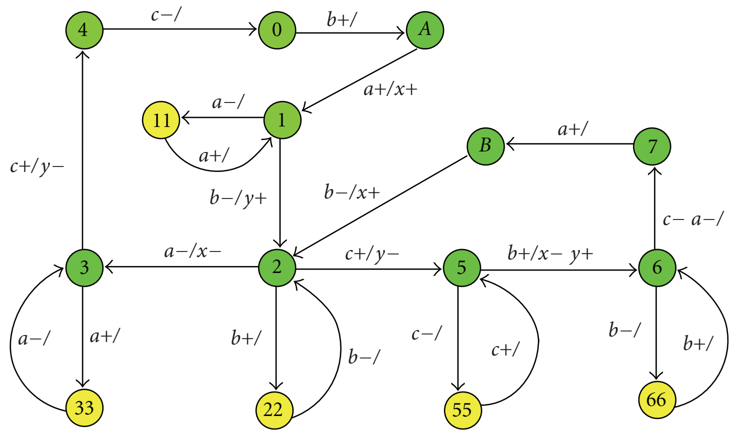
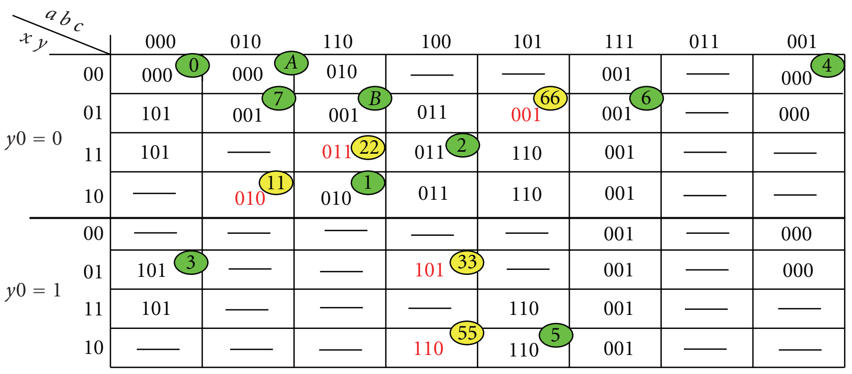


Figure 9: BM-ESS specification of Figure 5.



	xy \ abc	000	010	110	100	101	111	011	001
$y0 = 0$	00	000 0	000 A	010	—	—	001	—	000 4
	01	101	001 7	001 B	011	001 66	001 6	—	000
	11	101	—	011 22	011 2	110	001	—	—
	10	—	010 11	010 1	011	110	001	—	—
$y0 = 1$	00	—	—	—	—	—	001	—	000
	01	101 3	—	—	101 33	—	001	—	000
	11	101	—	—	—	110	001	—	—
	10	—	—	—	110 55	110 5	001	—	—

Figure 10: BM flow map with essential super-states (BM-ESS).

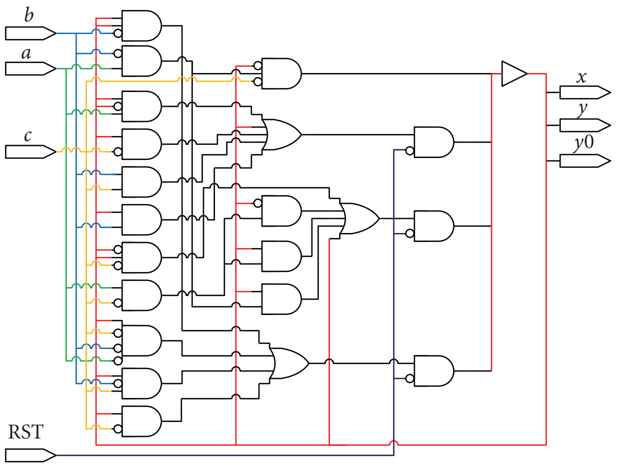


Figure 11: Logic circuit: RTL view—Altera.

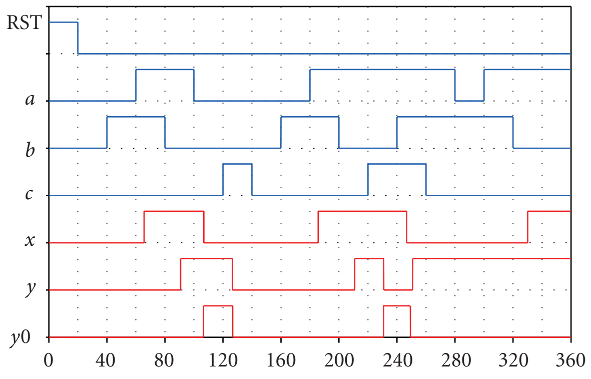


Figure 12: Simulation of the logic circuit of Figure 5.

(2) Apply in the BM specification the functional transformations that satisfy Lemma 1 (Sections 3.3 and 3.4).

(3) Generate the BM-EHF specification with essential super-states (BM-ESS) according to Section 3.5 (applying Definition 5).

(4) Use the Minimalist tool that starts from the BM-ESS specification and produces the equations of next-state hazard-free (sum of products—*netlist*).

(5) Use the Quartus tool [31] that starts from the *netlist* in structural VHDL.

The BM specification shown in Figure 1 has been used to illustrate our method. Figure 5 shows BM-EHF specification (Steps (1) and (2)). Figure 9 shows the BM-ESS specification (Step (3)). Steps (4) and (5) accomplish the automatic

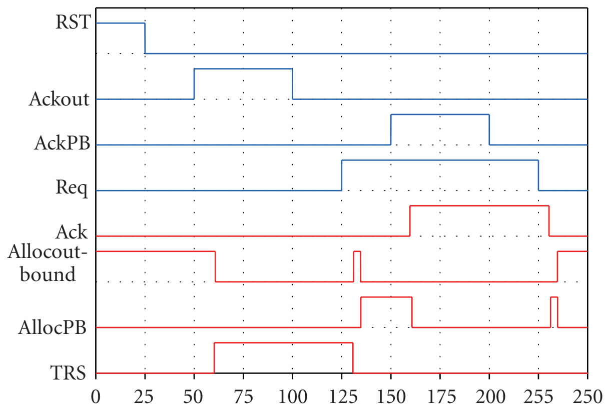


FIGURE 13: Simulation of the mp-for-pkt: with transient essential hazard.

TABLE 1: BM specifications show data.

	BM specification	
Design	States/trans.	In/out
Call-proc	12/15	3/3
Chu133	4/4	3/3
Chu150	5/5	3/3
Diff-Alu1	7/9	3/5
Dram-ctrl	12/14	7/6
Figure 1	8/9	3/2
Hp-ir-if	7/7	5/5
Mp-for-pkt	4/4	3/4
QR42	4/4	2/2
Rcv-setup	6/7	3/2

TABLE 2: Experimental Results to Huffman Machine The column State vars shows the variables of state.

		Huffman machine	
Design	State vars	Total of LUTs	Latency (ns)
Call-proc	0	8	10,888
Chu133	2	5	10,650
Chu150	0	3	10,017
Diff-Alu1	3	17	11,081
Dram-ctrl	0	10	11,633
Figure 1	0	4	10,675
Hp-ir-if	0	7	10,900
Mp-for-pkt	0	5	10,719
QR42	1	6	9,699
Rcv-setup	0	4	11,104

synthesis. One-state variable $y0$ was required to solve the existing conflicts (see Figure 10) [28]. Figure 11 shows logic circuit (RTL view—Altera). Figure 12 shows result of simulation of the circuit that was obtained by our method (hazard-free waveforms).

TABLE 3: BM-ESS specifications lead to the following data.

	BM-ESS specification		
Design	States/trans.	In/out	Dummy signals
Call-proc	22/40	3/3	0
Chu133	6/8	3/3	0
Chu150	10/15	3/3	0
Diff-Alu1	14/23	3/5	1
Dram-ctrl	22/36	7/6	0
Figure 5	14/19	3/2	0
Hp-ir-if	7/7	5/5	0
Mp-for-pkt	6/8	3/4	0
QR42	8/12	2/2	1
Rcv-setup	9/13	3/2	1

TABLE 4: Experimental results show Huffman machine—EHF (*Minimalist Tool did not complete the synthesis).

		Huffman machine—EHF	
Design	State vars	Total of LUTs	Latency (ns)
Call-proc	0	6	10,898
Chu133*	—	—	—
Chu150	0	7	11,606
Diff-Alu1	3	24	11,879
Dram-ctrl	0	16	12,271
Figure 5	1	11	10,855
Hp-ir-if	1	14	11,454
Mp-for-pkt	0	8	10,948
QR42	1	7	10,769
Rcv-setup	1	6	10,263

5. Discussion & Results

5.1. Discussion

Figures 13 and 14 show, respectively, the simulation results and the logic circuit of the mp-for-pkt benchmark whose specification is shown in Figure 4. The synthesis was performed using the Minimalist tool followed by the Quartus tool. Figure 13 shows two glitches, one on the *Allocoutbound* output and one on the *AllocPB* output. For example, the glitch on signal *Allocoutbound* occurs on state transition $1_{[\text{Ackout}-\text{Req}+/\text{TRS}-\text{AllocPB}+]} \rightarrow 2$. Figure 14 shows the behavior in the logic circuit of the state transition $1 \rightarrow 2$. The reason of the glitch: input signal *Req+* acts in the paths 1 and 2, where the change in the path 1 arrives first in LUT-5 (see Figure 14). This glitche can also be identified in the BM flow map. Thespecification is EHF (Lemma 1 is satisfied) but the implementation is not (Lemma 2 is not satisfied), causing a transient essential hazard shown in Figure 15 (to apply GUR rule—T2: $2_{[\text{Req}-\text{Ackout}+]} \rightarrow 1 - \textit{Allocoutbound}\ 0 \rightarrow 1 \rightarrow 0$).

The result of simulation of the circuit that was obtained by our method shows that the glitches have been eliminated (see Figure 16). The area penalty was 8 LUTs against 5 LUTs in the first solution. The latency penalty was 2,2%.

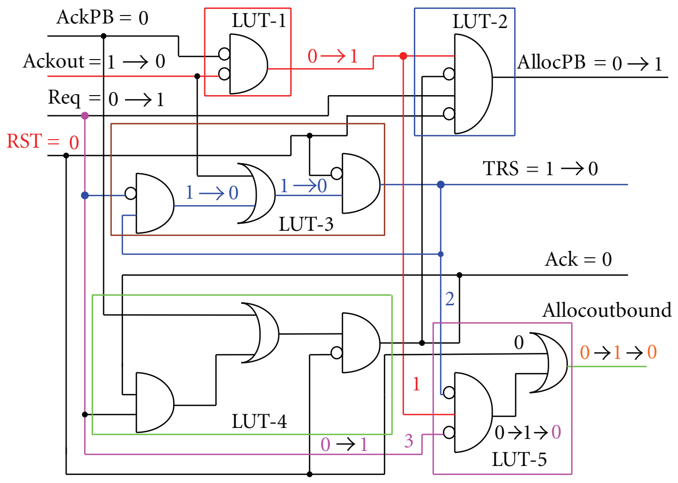
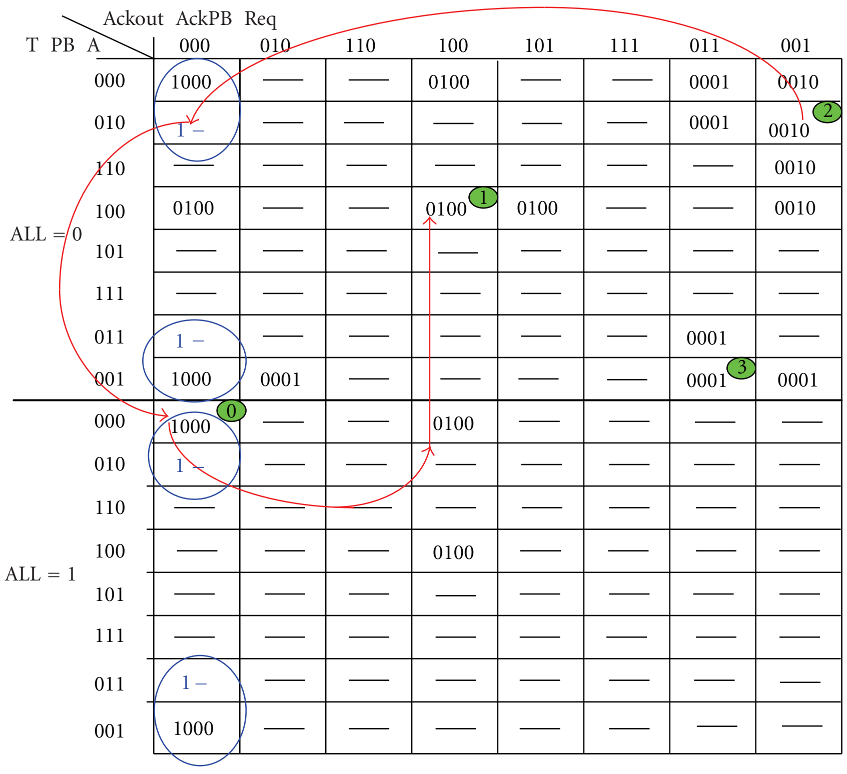


Figure 14: Logic circuit of the mp-for-pkt: map view—Altera (behavior 1 → 2).


Ackout AckPB Req
T PB A
ALL = 0
ALL = 1
0
1
2
3

ALL	T PB A	000	010	110	100	101	111	011	001
ALL = 0	000	1000	—	—	0100	—	—	0001	0010
	010	1 –	—	—	—	—	—	0001	0010
	110	—	—	—	—	—	—	—	0010
	100	0100	—	—	0100	0100	—	—	0010
	101	—	—	—	—	—	—	—	—
	111	—	—	—	—	—	—	—	—
	011	1 –	—	—	—	—	—	0001	—
	001	1000	0001	—	—	—	—	0001	0001
ALL = 1	000	1000	—	—	0100	—	—	—	—
	010	1 –	—	—	—	—	—	—	—
	110	—	—	—	—	—	—	—	—
	100	—	—	—	0100	—	—	—	—
	101	—	—	—	—	—	—	—	—
	111	—	—	—	—	—	—	—	—
	011	1 –	—	—	—	—	—	—	—
	001	1000	—	—	—	—	—	—	—

ALL = Allocoutbound T = TRS
PB = AllocPB A = Ack

Figure 15: BM flow map showing a transient essential hazard.

5.2. Results

We applied our theory to 9 known [8, 9, 12, 13] and one homemade benchmark. Table 1 presents the number of input and output signals, states, and transitions for each benchmark. Table 2 presents the area and timing results for these benchmarks synthesized as Huffman machines (with feedback output) before applying our theory. Syntheses performed using Minimalist followed by Quartus. The area was measured in terms of the number of LUTs while the latency was derived from simulations of the circuits already fitted on an EP2C35F672C7 device from Altera (Cyclone II family).

Table 3 presents the number of inputs and output signals, states, transitions, and dummy signals for the same benchmarks after applying the functional transformations

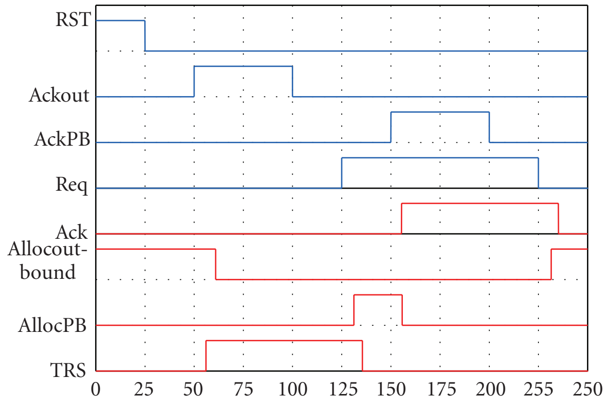


Figure 16: Simulation of the EHF version of the mp-for-pkt.

required to satisfy Lemmas 1 and 2. Table 4 shows the same results for the benchmarks after adhering to Lemmas 1 and 2.

As expected we found an area penalty (average of 54%), a latency penalty (average of 4,8%), and a state variables penalty (average of 75%). The *call-proc* benchmark showed a smaller area (less LUTs) and the *rev-setup* benchmark showed a reduced latency time. However, the area penalties did not impact significantly the FPGA usage (≅1%) still leaving enough free space for a datapath and other components that could be placed on the same device.

6. Conclusions

This work presented two conditions that, if satisfied, guarantee that burst-mode asynchronous controllers can be mapped on any commercial LUT-based FPGA without incurring in essential hazards.

When these conditions are not satisfied, we presented functional transformations that may be used to solve the problem. In this case, there is an area (mainly are added state variables—75%) and a latency penalty. However, our experimental results on a set of known benchmark showed low latency penalty (4,8%) and low FPGA occupation overhead (≅1%). This type of burst-mode controllers may be combined with a self-timed datapath that have already been successfully synthesized on commercial FPGAs, in order to create fully asynchronous processor on FPGAs.

References

[1] C. J. Myers, *Asynchronous Circuit Design*, John Wiley & Sons, New York, NY, USA, 2001.

[2] S. Hauck, “Asynchronous design methodologies: an overview,” *Proceedings of the IEEE*, vol. 83, no. 1, pp. 69–93, 1995.

[3] S. H. Unger, “Hazards, critical races, and metastability,” *IEEE Transactions on Computers*, vol. 44, no. 6, pp. 754–768, 1995.

[4] S. Hauck, S. Burns, G. Borriello, and C. Ebeling, “An FPGA for implementing asynchronous circuits,” *IEEE Design & Test of Computers*, vol. 11, no. 3, pp. 60–69, 1994.

[5] D. M. Chapiro, *Globally-asynchronous locally-synchronous systems*, Ph.D. thesis, Stanford University, Stanford, Calif, USA, 1984.

[6] S. Burns and A. Martin, “Syntax-directed translation of concurrentprograms into self-timed circuits,” in *Proceedings of the 5th MIT Conference on Advanced Research in VLSI*, J. Allen and T. Leighton, Eds., pp. 35–50, MIT Press, Cambridge, Mass, USA, March 1988.

[7] I. E. Sutherland, “Micropipelines,” *Communications of ACM*, vol. 32, no. 6, pp. 720–738, 1989.

[8] T.-A. Chu, *Synthesis of self-timed VLSI circuits from graph-theoretic specifications*, Ph.D. thesis, Department of EECS, MIT, Cambridge, Mass, USA, June 1987.

[9] S. M. Nowick, K. Y. Yun, and D. L. Dill, “Practical asynchronous controller design,” in *Proceedings of the IEEE International Conference on Computer Design (ICCD ’92)*, pp. 341–345, Cambridge, Mass, USA, October 1992.

[10] S. M. Nowick and B. Coates, “UCLOCK: automated design of high-performance unclocked state machines,” in *Proceedings of the IEEE International Conference on Computer Design (ICCD ’94)*, pp. 434–441, Cambridge, Mass, USA, October 1994.

[11] S. M. Nowick, M. E. Dean, D. L. Dill, and M. Horowitz, “The design of a high-performance cache controller: a case study in asynchronous synthesis,” *Integration, the VLSI Journal*, vol. 15, no. 3, pp. 241–262, 1993.

[12] A. Marshall, B. Coates, and F. Siegel, “Designing an asynchronous communications chip,” *IEEE Design & Test of Computers*, vol. 11, no. 2, pp. 8–21, 1994.

[13] A. Davis, B. Coates, and K. Stevens, “Automatic synthesis of fast compact self-timed control circuits,” in *Proceedings of the IFIP Working Conference on Asynchronous Design Methodologies*, Manchester, UK, April 1993.

[14] E. Brunvand, “Using FPGAs to implement self-timed systems,” *The Journal of VLSI Signal Processing*, vol. 6, no. 2, pp. 173–190, 1993.

[15] K. Maheswaran and V. Akella, “Hazard-free implementation of the self-timed cell set in a Xilinx FPGA,” Tech. Rep., University of California, Davis, Calif, USA, 1994.

[16] Y. Zafar and M. M. Ahmed, “A novel FPGA compliant micropipeline,” *IEEE Transactions on Circuits and Systems II*, vol. 52, no. 9, pp. 611–615, 2005.

[17] A. Ejnioui, “FPGA prototyping of a two-phase self-oscillating micropipeline,” in *Proceedings of the IEEE Computer Society Annual Symposium on VLSI (ISVLSI ’07)*, pp. 437–438, Porto Alegre, Brazil, March 2007.

[18] N. Huot, H. Dubreuil, L. Fesquet, and M. Renaudin, “FPGA architecture for multi-style asynchronous logic,” in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE ’05)*, vol. 1, pp. 32–33, Munich, Germany, March 2005.

[19] X. Jia and R. Vemuri, “The GAPLA: a globally asynchronous locally synchronous FPGA architecture,” in *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM ’05)*, pp. 291–292, Napa, Calif, USA, April 2005.

[20] S. C. Smith, “Design of an FPGA logic element for implementing asynchronous NULL convention logic circuits,” *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 6, pp. 672–683, 2007.

[21] D. Sokolov and A. Yakovlev, “Clockless circuits and system synthesis,” *IEE Proceedings: Computers and Digital Techniques*, vol. 152, no. 3, pp. 298–316, 2005.

[22] H. Jacobson, *Asynchronous circuit design: a case study of a framework called ACK*, M.S. thesis, Luleå University of Technology, Luleå, Sweden, 1996.

[23] S. M. Nowick, "Automatic synthesis of burst-mode asynchronous controllers," Tech. Rep. CSL-TR-95-686, Stanford University, Stanford, Calif, USA, 1995.

[24] S. M. Nowick and D. L. Dill, "Exact two-level minimization of hazard-free logic with multiple-input changes," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, pp. 986–997, 1995.

[25] P. Siegel, G. De Micheli, and D. Dill, "Automatic technology mapping for generalized fundamental-mode asynchronous designs," in *Proceedings of the 30th ACM/IEEE Design Automation Conference (DAC '93)*, pp. 61–67, Dallas, Tex, USA, June 1993.

[26] B. Lin and S. Devadas, "Synthesis of hazard-free multilevel logic under multiple-input changes from binary decision diagrams," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 14, no. 8, pp. 974–985, 1995.

[27] K. Y. Yun and D. L. Dill, "Automatic synthesis of extended burst-mode circuits. I. (Specification and hazard-free implementations)," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 18, no. 2, pp. 101–117, 1999.

[28] R. M. Fuhrer, S. M. Nowick, M. Theobald, N. K. Jha, and L. A. Plana, "Minimalist: an environment for the synthesis and verification of burst-mode asynchronous machines," in *Proceedings of the International Workshop on Logic Synthesis (IWLS '98)*, Lake Tahoe, Calif, USA, June 1998.

[29] H. M. Jacobson and C. J. Myers, "Efficient algorithms for exact two-level hazard-free logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 21, no. 11, pp. 1269–1283, 2002.

[30] S. H. Unger, "Hazards and delays in asynchronous sequential switching circuits," *IRE Transactions on Circuit Theory*, vol. 6, no. 1, pp. 12–25, 1959.

[31] Altera Corporation, http://www.altera.com/.

Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2008, Article ID 736203, 10 pages
doi:10.1155/2008/736203

Research Article

The Coarse-Grained/Fine-Grained Logic Interface in FPGAs with Embedded Floating-Point Arithmetic Units

Chi Wai Yu,[1] Julien Lamoureux,[2] Steven J. E. Wilton,[2] Philip H. W. Leong,[3] and Wayne Luk[1]

[1] *Department of Computing, Imperial College London, London SW7 2AZ, UK*
[2] *Department of Electrical and Computer Engineering, University of British Columbia, Vancouver, British Columbia, Canada V6T1Z4*
[3] *Department of Computer Science and Engineering, Chinese University of Hong Kong, Hong Kong*

Correspondence should be addressed to Chi Wai Yu, cyu@doc.ic.ac.uk

Received 7 July 2008; Accepted 30 October 2008

Recommended by Gustavo Sutter

This paper examines the interface between fine-grained and coarse-grained programmable logic in FPGAs. Specifically, it presents an empirical study that covers the location, pin arrangement, and interconnect between embedded floating point units (FPUs) and the fine-grained logic fabric in FPGAs. It also studies this interface in FPGAs which contain both FPUs and embedded memories. The results show that (1) FPUs should have a square aspect ratio; (2) they should be positioned near the center of the FPGA; (3) their I/O pins should be arranged around all four sides of the FPU; (4) embedded memory should be located between the FPUs; and (5) connecting higher I/O density coarse-grained blocks increases the demand for routing resources. The hybrid FPGAs with embedded memory required 12% wider channels than the case where embedded memory is not used.

1. Introduction

Significant improvements in the performance, logic density, and power efficiency of field-programmable gate arrays (FPGAs) have made them useful for implementing nearly any type of digital application. In early FPGAs, significant improvements were made by optimizing the fine-grained programmable logic and routing architecture of the FPGA. Today, further improvements are being made by embedding coarse-grained elements such as memories, multipliers, and processors within the fine-grained programmable fabric of the FPGA. Coarse-grained elements can implement a specific function more efficiently than fine-grained programmable logic. However, since they are not as flexible, they only benefit applications which utilize them. This limits the types of embedded blocks which are commercially viable in general-purpose FPGAs to very common circuit elements such as memories, adders, and multipliers. For domain-specific FPGAs, however, additional embedded blocks may make sense. For example, an FPGA that is built specifically to implement applications containing a significant amount of floating point computation would benefit from embedded floating point units. This was explored in [1], in which a domain-specific FPGA that incorporates coarse-grained floating point units (FPUs) was described. The results in [1] show that the embedded floating point units lead to an 18 times density improvement for a set of floating point datapath circuits.

An important consideration when adding coarse-grained embedded elements to an FPGA is the interface between the coarse-grained and fine-grained resources. If this interface is not flexible enough, the usefulness of the embedded block will be reduced, since connections to and from the block will be expensive. On the other hand, if the interface is too flexible, it will require too much area and delay, possibly negating the density and performance advantages of including the embedded block, and resulting in unnecessary overhead for applications that do not use the embedded component.

In this paper, we examine this interface. We focus on architectural issues, such as the location of the embedded elements, and the interconnect between the embedded elements and the fine-grained fabric. Our approach is presented in the context of the embedded floating point blocks described in [1]. Specifically, the key contributions of this paper are the following:

(i) a set of parameters that describes the interface between coarse-grained and fine-grained programmable logic in FPGAs;

(ii) an empirical framework to model the impact of coarse-grained architectural parameters in terms of performance, density, and power consumption;

(iii) an empirical study that examines:

  (1) where the coarse-grained FPUs should be embedded within FPGAs;

  (2) where the pins of the FPUs should be on the periphery;

  (3) how flexible the interconnect between the FPUs and the fine-grained logic should be;

  (4) what shape the FPU should have;

(iv) a study of a hybrid FPGA interface containing embedded memories and FPUs including:

  (1) where embedded memories used by the FPUs should be located;

  (2) how flexible the interconnect between the FPUs, embedded memories and the fine-grained logic should be.

Although this study focuses on FPGAs with embedded FPUs, its findings may be applicable to other types of embedded computational blocks.

A preliminary version of this work was presented in [2]. This paper further expands the study by considering hybrid FPGAs with more than one type of coarse-grained block. This is important because the different coarse-grained blocks have different I/O density, area, and speed. The connection of those blocks should affect the performance and routing resources required in the hybrid FPGA.

This paper is organized as follows. Section 2 describes related work. Section 3 illustrates the interface between coarse- and fine-grained logic and presents corresponding parameters to describe this interface. Section 4 then presents the empirical framework used to evaluate different interface schemes. Finally, Section 5 presents our results and analysis, and Section 6 summarizes our conclusions.

2. Background

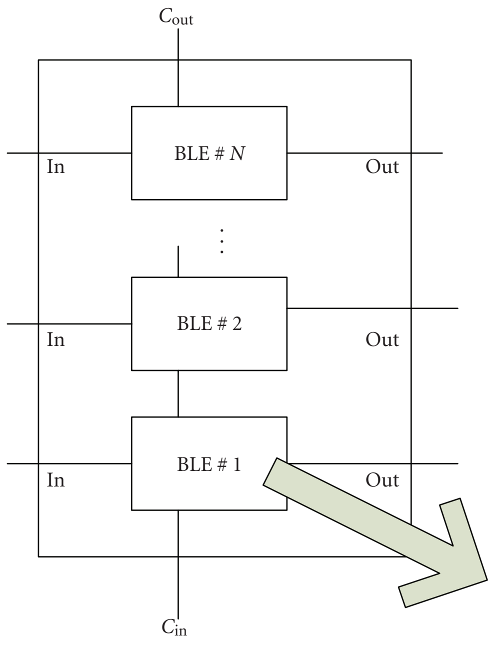
Conventional island-style FPGAs consist mainly of a fine-grained programmable fabric that is made up of configurable logic blocks (CLBs), programmable routing resources, and programmable I/Os. The CLBs consist of one or more k-input lookup tables (k-LUT) and fast local interconnect. Each k-LUT can implement any single output function with k inputs or less. The routing resources implement the interconnect between the CLBs and the I/Os.

A significant number of studies have focused on optimizing this type of FPGA architecture to minimize area, critical-path delay, and power consumption. As an example, the study described in [3] compares different aspects of segmented routing architectures, such as wirelength distribution, switch block implementation, and connection block flexibility, with the goal of creating a fast and area-efficient general-purpose FPGA architecture.

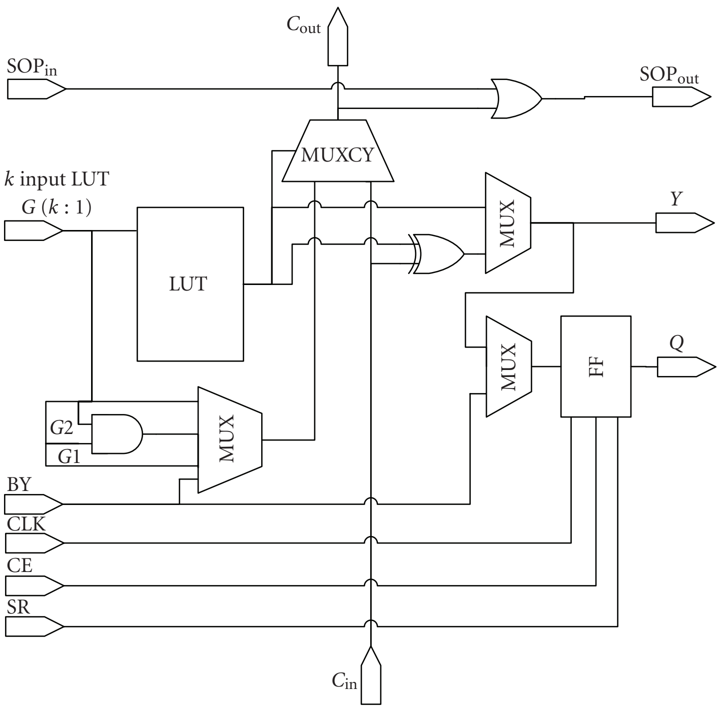
More recent work has focused on adding coarse-grained blocks within the fine-grained fabric. Examples of this include embedded arithmetic multipliers [4, 5] and embedded processors [5]. Coarse-grained blocks improve area and delay since they can implement specific functions more efficiently than the fine-grained logic [6]. On the other hand, coarse-grained blocks waste area when they are not used by an application. FPGAs vendors must consider this tradeoff to determine the type and number of coarse-grained blocks that should be embedded within their devices.

In order to take further advantage of coarse-grained blocks, domain-specific hybrid FPGAs target a specific application domain. In doing so, greater area and delay savings can be achieved for certain types of applications since the amount of coarse-grained logic can be tailored for those applications. A number of recent approaches have been proposed in the literature. In [7], a coarse-grained architecture with bus-based interconnect has been shown to reduce area for datapath circuits. In [8], a tool that generates a domain-specific reconfigurable fabric that is tailored to a specified set of application has been proposed. In [9], the QUKU architecture which merges coarse-grained reconfigurable processing element array and FPGA architectures has been described. This two-level reconfigurable architecture provides active support for fast and efficient dynamic reconfiguration. Enzler et al. [10] has proposed a framework for the cycle-accurate performance evaluation of hybrid reconfigurable processors on the system level, which is based on data-streaming applications. In [1], a domain-specific hybrid FPGA architecture that targets floating point arithmetic applications by incorporating floating point units within a fine-grained programmable fabric has been presented; this architecture is shown to be 18 times more area-efficient than a purely fine-grained architecture for floating point arithmetic applications.

One of the key parts of an FPGA with embedded coarse-grained blocks is the routing structure between the embedded blocks and the fine-grained logic resources. If the coarse-grained/fine-grained interface is not flexible enough, many applications will be unroutable. On the other hand, if the interface is overly flexible, the routing resources will be slower and consume more area than is necessary. Although a number of studies have proposed new coarse-grained blocks and hybrid FPGA architectures, few have examined the interface between the coarse-grained blocks and fine-grained fabric in significant detail. In [11], the local routing resources that connect CLBs to the FPGA routing resource are shared with the embedded blocks to minimize the overall area penalty when adding the embedded blocks. This technique, called *shadow clustering*, is useful for embedded blocks with similar I/O pin densities as the existing CLBs; however, for embedded blocks which has higher I/O pin densities than the existing routing resources are not sufficient. In [12], the interface between embedded memory blocks and



(a) A CLB



(b) A BLE

Figure 1: A configurable logic block and the basic logic element inside.

fine-grained programmable logic is examined. Memories are quite different from computation blocks, and so we expect that the interface presented in [12] would not be suitable for our problem.

3. Coarse/Fine-Grained Interface

In this section, we describe the architecture of the blocks used in this work. We first present our assumptions regarding the fine and coarse-grained logic and then give a description of a generic interface architecture with parameters that cover the space of architectures considered.

3.1. Fine-Grained FPGA Assumptions

We assume that the fine-grained resources in the FPGA consist of a grid of identical configurable logic blocks (CLBs), each containing N basic logic elements (BLEs). Each BLE contains a k-LUT and flip flop. The CLB also contains support for carry chains, shift registers, internal multiplexers, and XOR gates. Figure 1 shows the CLB and BLE modelled.

The CLBs are connected using horizontal and vertical channels, as described in [3]. Each channel contains W parallel routing tracks of length 1 and is connected to neighbouring CLBs using a connection block, and intersecting channels using a switch block. We use the subset switch block (also known as disjoint) with $Fc_{\text{switch}} = W$, $Fs = 3$, $Fc_{\text{output}} = 1$, $Fc_{\text{input}} = 1$, and $Fc_{\text{pad}} = 1$ [3].

3.2. Coarse-Grained Block Assumptions

In general, FPGA-based floating point application circuits can be divided into control and datapath circuits. The datapath occupies most of the area in the form of FPUs. The required processing mainly consists of addition, subtraction, and multiplication. We adopt the coarse-grained floating point blocks described in [1]. The datapath circuit is implemented in this floating point block. The floating point multiplier block is a fixed-function block. The floating point adder block can be configured for either floating point addition or subtraction. Each block has a reconfigurable registered output and associated control input and status output signals. A wordblock contains N identical bitblocks. Bitblock contains two 4-input LUTs and a reconfigurable output register. Bitblocks within a wordblock are all controlled by the same set of configuration bits, so all bitblocks within a wordblock perform the same function. A wordblock, which includes a register, can efficiently implement operations such as addition and multiplexing.

In our assumption, each coarse-grained block contains two double precision floating point adders, two double precision floating point multipliers, and five wordblocks which can efficiently implement operations such as addition and multiplexing as shown in Figure 2.

In addition to FPUs, we also consider embedded memories. Specially, we consider block-selected RAMs (BRAMs) as described in [13]. The details of floating point blocks and BRAMs are shown in Table 1.

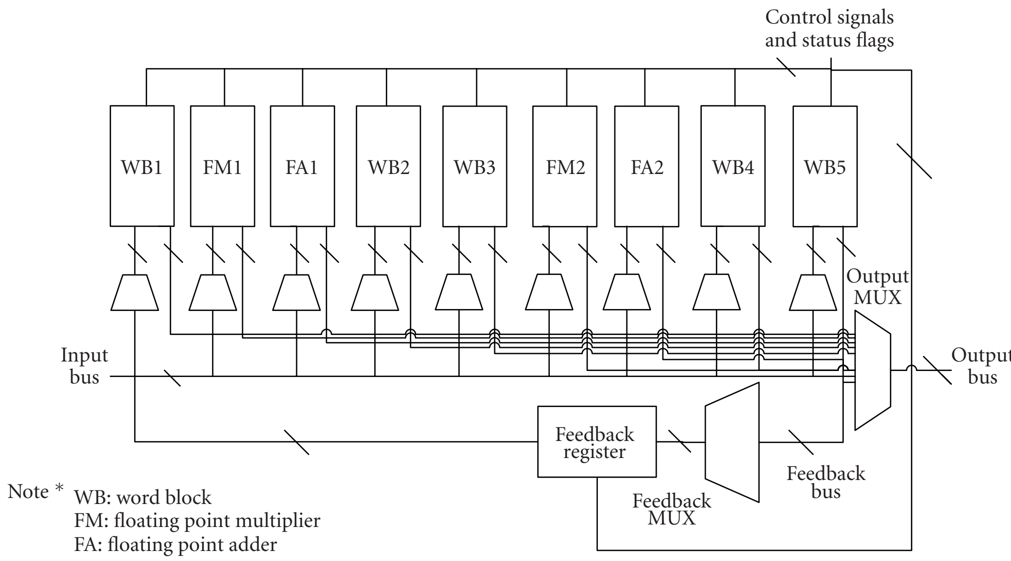


Figure 2: Coarse-grained unit modelled in this paper.

Table 1: Statistic of the coarse-grained blocks used.

	BRAM	Floating point unit
Number of input	90	286
Number of output	64	258
Area (no. of CLB)	8	182
Delay (nanoseconds)	2.1	9.2

3.3. Coarse-Grained Interface

Based on our detailed area model, we estimate that our embedded FPU consumes roughly the same amount of area as 182 tiles. Each tile represents a CLB and its associated interconnect, buffer, and configuration bit. To embed an FPU, we remove a 13 × 14 grid of CLBs, and replace them with a single EB. Figure 3 shows an example of replacing 3×3 grid of CLBs by a single embedded block (EB). We assume that the EB pins connect to the routing architecture through connection blocks, similar to those used for CLBs. Although other connection patterns are possible (see, e.g., [12]), this pattern allows us to minimize the number of changes to the existing FPGA routing architecture, so that we can leverage the significant amount of previous work on FPGA routing structures. We also assume that the gridded routing fabric extends over the embedded block, as shown in Figure 3. Given the large number of metal layers available in modern CMOS processes, it is reasonable that tracks can easily be placed on top of the embedded blocks. In Figure 3, the four switch blocks required at the interface of the horizontal and vertical channels must coexist with the embedded block; the embedded block, which takes the same area as nine CLBs, includes the area of these four switch blocks. Although it would be possible to consider architectures in which the grid is "broken" [14], it would require changes to the detailed routing architecture. In addition to FPUs, the memories are embedded in the hybrid FPGA under the same assumption. However, the area and the delay of the memories are different to FPUs, which is shown in Table 1. The size of BRAM is 2×4 tiles.

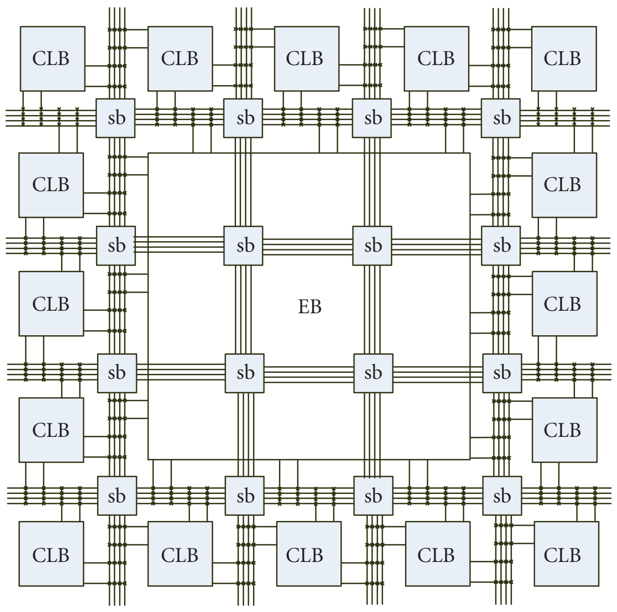
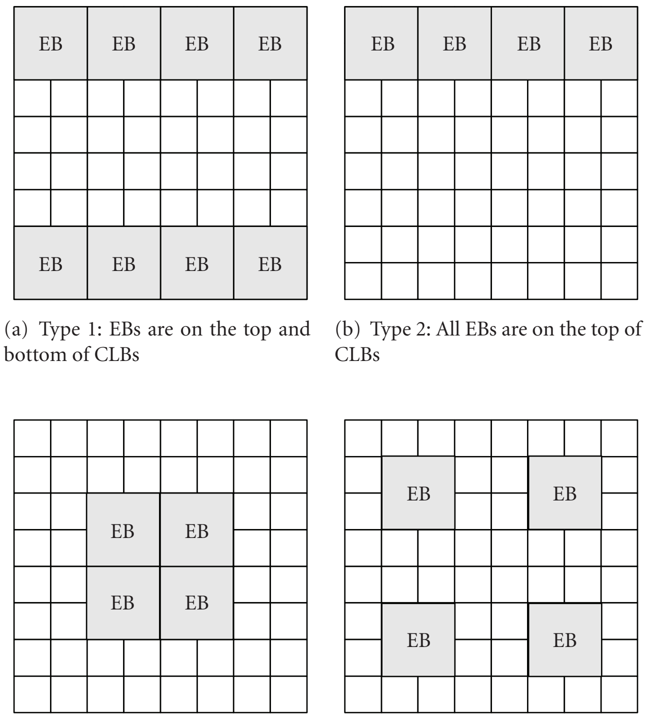


Figure 3: Connection between coarse- and fine-grained units through switch box (sb).

3.4. Interface Parameters—Single EB Type

In this paper, we consider a range of interface architectures. First, we explore the single EB-type hybrid FPGA. To describe the space of single EB-type architectures that we consider, we define the following parameters.



(a) Type 1: EBs are on the top and bottom of CLBs

(b) Type 2: All EBs are on the top of CLBs

(c) Type 3: EBs are in the middle of CLBs

(d) Type 4: EBs are surrounded by sea of CLBs

Figure 4: Various positions of the EBs relative to the fine-grained CLBs.

(1) Single EB Position

The embedded blocks can be placed in various places within the FPGA. In this paper, we consider the positions as shown in Figure 4.

(2) Single EB Pin Location

Figure 5 shows several strategies for positioning the pins of each EB. Strategy (a) has the highest I/O density, but may be suitable if signals from the I/O block are to be combined using a small set of CLBs. Strategies (b), (c), and (d) have lower I/O density, but may result in longer connections if signals from more than one side of the EB are to be connected to the same CLB(s).

(3) Single EB Channel Width

The width of the channels surrounding the EB has a significant impact on the routability of the device. Since our EB has a large number of pins, congestion around the EB may happen so it is desirable to relieve this congestion by using wider channels.

(4) Single EB Shape

Several layouts of each embedded block are possible. We consider various aspect ratios.

3.5. Interface Parameters: Multiple EB Types

We also study the interface between multiple EB types and the fine-grained fabric. In this case, connections exist between the two types of EBs and also between EBs and CLBs. The best interface architecture may be different from the single type EB FPGA. Therefore, we investigate the following parameters for the hybrid FPGAs with two types of embedded blocks.

(1) Multiple EB Position

We arrange the different EB types in various ways as shown in Figure 6. We consider three different arrangements. The first type places the smaller EBs in columns next to the larger EBs. The second type places the smaller EBs around a group of larger EBs. The third type places the smaller EBs around individual larger EBs.

(2) Multiple EB Channel Width

Embedding additional EBs may change the amount of routing resources that are needed. The connections between EBs are usually bus-based which require more routing resources. It is because if the I/O density of the additional EB is larger, more wires are needed to connect to another EB within a certain area. And the congestion in this area increases and may reduce the performance of the FPGA.

4. Methodology

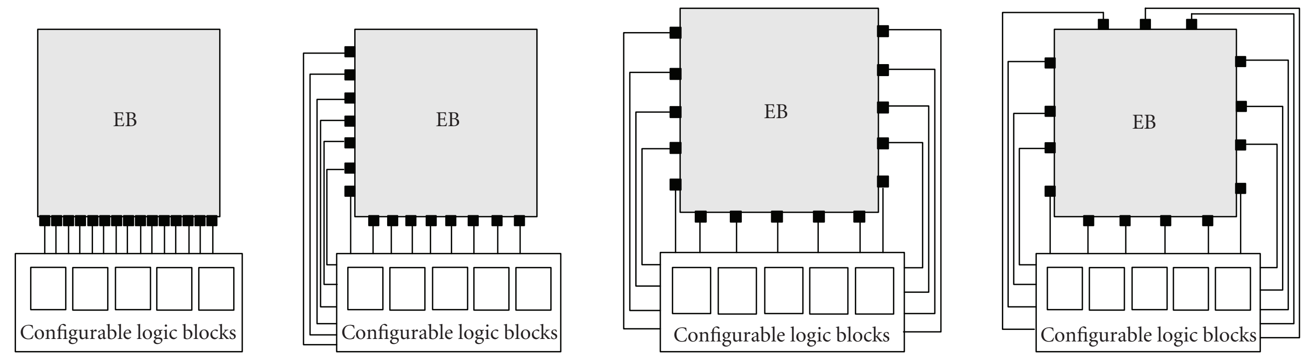
We employ an empirical methodology to examine the impact of the interface parameters described in the previous section. This section describes the benchmark circuits, the CAD tools, and the model that are used.

4.1. Domain-Specific Benchmark Circuits

First we use six double precision floating point benchmark circuits [15] with only one kind of coarse-grained embedded block. They are (1) *bfly*: the basic component of fast Fourier transform: $z = y + x*w$ using complex numbers; (2) *dscg*: a digital sine-cosine generator; (3) *fir4*: a 4-tap finite impulse response filter; (4) *mm3*: a 3×3 matrix multiplication circuit, (5) *ode*: an ordinary differential equation solver; (6) *bgm*: a datapath to compute Monte Carlo simulations of interest rate model derivatives priced under the Brace-Gątarek-Musiela (BGM) framework.

These benchmarks are chosen since they each involve a significant amount of floating point computation. Since *bfly*, *dscg*, *fir4*, *ode*, and *mm3* contain a small number of fine-grained units, each core is replicated four times and are connected together. For example, a *dscg* benchmark contains four *dscg* cores connected together. All circuits use a single global clock. The amount of FPUs and CLBs used for each benchmark circuit is shown in Table 2.

For the experiment involving embedded memories, we add BRAMs to the benchmark circuits. It is more realistic



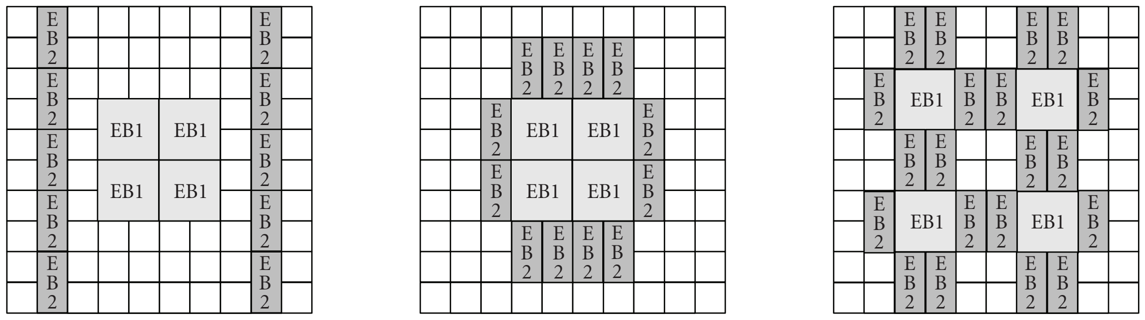
(a) CLBs connect to one side of EB with highest I/O denisty

(b) CLBs connect to two sides of EB

(c) CLBs connect to three sides of EB

(d) CLBs connect to four sides of EB with lowest I/O denisty

Figure 5: Different pin positions in EB.



(a) Type 1: Column based small EBs near large EBs

(b) Type 2: Group of large EBs is surrounded by small EBs and CLBs

(c) Type 3: Each large EB is surrounded by small EBs and CLBs

Figure 6: Various positions of the multiple EBs.

to store the input and output data of the applications in internal BRAMs rather than store the data in of-chip memories. The BRAM data lines are connected to primary input/output of the benchmark circuits. The BRAM address lines are connected to counters which are also added to the benchmark circuits. The adders do not affect the performance because they are implemented using fast carry chains, the delay of which is small compared to the delay of the BRAM and the FPU. The benchmark circuits now contain two different types of EB: (1) FPUs and (2) BRAMs. The number of BRAMs used in each benchmark circuit is shown in Table 2.

4.2. VPH: Versatile Place and Route for Hybrid FPGAs

We use the evaluation tool VPH to explore our architectures. VPH is a modified version of the VPR tool, with support for embedded blocks, complex logic blocks, carry chains, and constraint files [16]. The tool is available at [17]. In the VPH design flow, shown in Figure 7, applications and coarse-grained elements are written in a high-level hardware description language (VHDL) and synthesized to a mapped library netlist in VHDL format using Synplicity's Synplify Premier 8.5 tool. The library netlist contains the usage and connection of simple units such as registers, LUTs, internal multiplexors, and internal inverters. The basic logic block packing tool, VPHpack, packs these units into basic logic elements (BLEs). VPHpack clusters BLEs into CLBs.

Table 2: Amount of FPU, CLB, and also BRAM used in each benchmark circuit.

Benchmarks	*bgm*	*dscg*	*bfly*	*ode*	*mm3*	*fir4*
No. of CLB	6433	649	884	430	876	282
No. of FPU	7	8	8	8	8	8
No. of BRAM	18	22	40	25	12	12

A user constraint file (.ucf) is used to specify the FPGA area and the absolute position of each embedded block. A separate constraint file for each embedded block is used to specify the area, the pin position, and the timing information for the EB; the area and delay information for each block is obtained using Synopsys Design Compiler V-2004.06. As in VPR, an architecture file specifies the fine-grained FPGA's architectural parameters, such as timing delay of the LUT. Using these files, the VPH tool performs placement, routing,

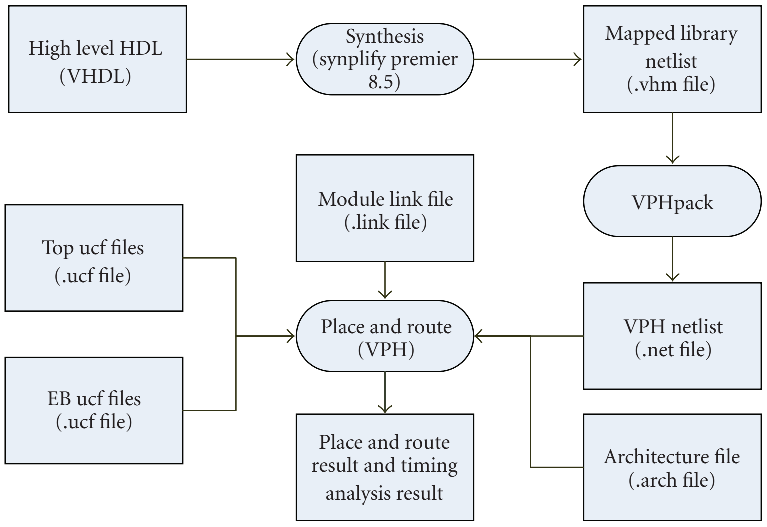

High level HDL (VHDL)
Synthesis (synplify premier 8.5)
Mapped library netlist (.vhm file)
VPHpack
Module link file (.link file)
Top ucf files (.ucf file)
Place and route (VPH)
VPH netlist (.net file)
EB ucf files (.ucf file)
Place and route result and timing analysis result
Architecture file (.arch file)

Figure 7: Design flow of exploration using VPH.

and timing analysis to produce area and delay estimates for each benchmark circuit.

5. Results and Discussion—Single EB Type

In this section, the impact of the interface parameters in Section 3 on hybrid FPGAs is studied. In the experiments conducted, the default architecture parameters are (1) CLB with 2× 4-LUTs, (2) type 3 EB position (Figure 4) as it gives the best performance for the first experiment, (3) channel width 80; since the maximum I/O density of the EBs is 42 pins per slice width, we choose 80 to be the channel width to facilitate routing, (4) size of the floating point unit is 13 × 14 CLBs. We use higher routing effort than our preliminary version of work in [2]; therefore, the experiments result can achieve higher speed than our previous work.

5.1. Single EB Position Results

We first examine how the position of the EBs affects the overall performance of the device. As shown in Figure 4, we consider positioning the EBs both around the periphery of the device, as well as in the centre. Intuitively, positioning the EBs in the centre will lead to shorter wirelengths for wires that connect multiple EBs. However, positioning the EBs around the periphery may cause less congestion since the EBs will be more spread out.

Figure 8 shows the results for each of the positioning strategies described in Figure 4. The best strategy is type 3, in which the EBs are in the centre of the device, surrounded by a sea of CLBs. It achieves at most 14.4% in speed improvement compared to other positioning strategies. The critical path of our circuits tends to include nets that connect multiple EBs; thus placing the EBs close to each other is beneficial.

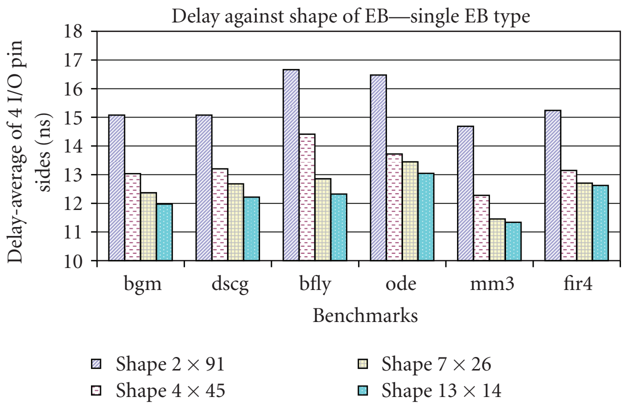


Figure 8: Delay against various EBs positions for single EB type FPGA, as defined in Figure 4.

5.2. Single EB Pin Location Results

We next consider the effect of I/O pin position on the periphery of each EB. As shown earlier, pins can be distributed evenly around the EB, or can be concentrated on one or more sides of the block. Intuitively, distributing the pins evenly will lead to a lower I/O density, possibly reducing congestion but may lead to longer wirelengths if pins from more than one side of the EB are connected.

The results are shown in Tables 3 and 4. The critical path of the circuit is slightly smaller if all pins are placed on a single side of the embedded block. In several of our benchmarks, the critical path includes a path from one EB, through a register in a CLB, into another EB. These connections are shorter if the pins are close together. On the other hand, Table 4 shows that the routing demand in each channel can be reduced by distributing the pins evenly around each EB. Compared to the configuration in which all pins are on one side of the block, evenly distributing the pins

Table 3: Critical path delay in ns for different EB's I/O positions as shown in Figure 5 for single EB-type FPGA. The percentage shows the deviation from the 1 side result.

Circuits	1 side	2 sides	3 sides	4 sides
	(42 wires/clb)	(21 wires/clb)	(14 wires/clb)	(11 wires/clb)
bgm	12.01 (0%)	11.92 (−0.7%)	11.93 (−0.7%)	12.03 (0.2%)
dscg	12.12 (0%)	12.34 (1.8%)	12.13 (0.1%)	12.28 (1.3%)
bfly	12.42 (0%)	12.38 (−0.3%)	12.30 (−1.0%)	12.19 (−1.9%)
ode	13.02 (0%)	13.30 (2.2%)	12.79 (−1.8%)	13.06 (0.3%)
mm3	11.22 (0%)	11.53 (2.8%)	11.31 (0.8%)	11.29 (0.6%)
fir4	12.63 (0%)	12.79 (1.3%)	12.41 (−1.7%)	12.67 (0.3%)
Average	12.23 (0%)	12.37 (1.1%)	12.14 (−1.9%)	12.25 (0.9%)

Table 4: Minimum channel width (number of wires) for different I/O positions for single EB-type FPGA as shown in Figure 5. The percentage shows the deviation from the 1 side result.

Circuits	1 side	2 sides	3 sides	4 sides
bgm	44 (0%)	44 (0%)	30 (−32%)	27 (−39%)
dscg	43 (0%)	44 (2%)	30 (−30%)	33 (−23%)
bfly	44 (0%)	44 (0%)	38 (−14%)	37 (−16%)
ode	43 (0%)	44 (2%)	35 (−19%)	33 (−23%)
mm3	45 (0%)	45 (0%)	29 (−36%)	30 (−33%)
fir4	42 (0%)	44 (5%)	32 (−24%)	29 (−31%)
Average	43.5 (0%)	44.2 (1.6%)	32.3 (−26%)	31.5 (−28%)

reduces the channel width by 39%. We conclude that this is the best choice.

5.3. Single EB Interconnect Flexibility

We next consider the width of the channels surrounding the EBs. Intuitively, there is a high pin density on each side of each EB; this may place additional demands on the routing fabric near the EBs. If the fabric cannot provide the required flexibility, circuitous routes may be required, leading to an increased delay.

The results in Figure 9 show the effect of EB to CLB channel width on delay. For routable circuits, rather surprisingly, the average variation is less than 3%. We believe that this is due to critical paths being routed efficiently, so once the circuit is routable, channel width does not affect delay.

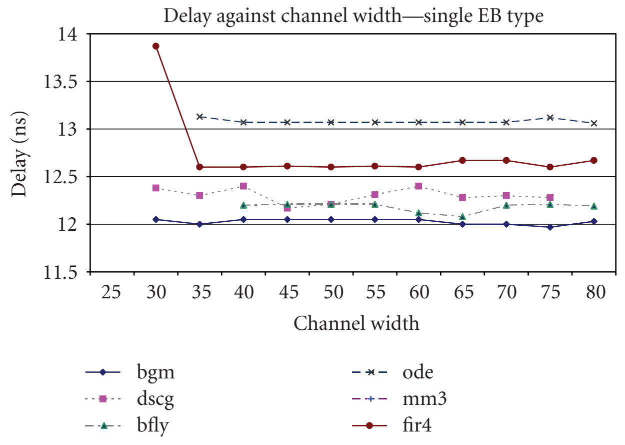


Figure 9: Delay against channel width for single EB-type FPGA.

5.4. Single EB Aspect Ratio

Finally, we consider how the aspect ratio of each EB affects the overall performance of the FPGA. In this experiment, the area of EB is fixed, but the aspect ratio is changed. Intuitively, changing aspect ratio will change the distance between pins on different EBs; this leads to a change in the delay of the nets connecting these pins.

We modify the shape of the EBs from rectangular (2×91) to square (13×14); the width and height are counted in the number of CLBs. The results in Figure 10 show that square EBs are the most efficient for all applications and result in a 14.4% speed improvement compared to the 2×91 shape. Square EBs lead to a better worst-case delay between the EBs, shortening the critical path in our benchmark circuits.

6. Results and Discussion: Multiple EB Types

After finding the best parameters for the single EB-type case, we examine how embedding more than one type of EB affects performance and routing demand. In our experiments, we explore the EB position and the interconnect flexibility of this system. The size of BRAM is 2×4 CLBs in these experiments.

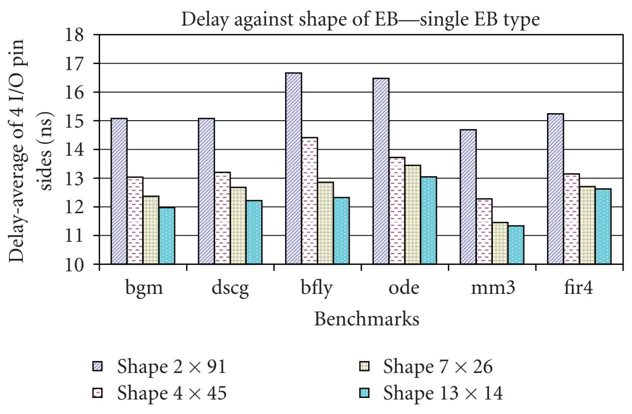


Figure 10: Delay against various EBs' shape for single EB-type FPGA.

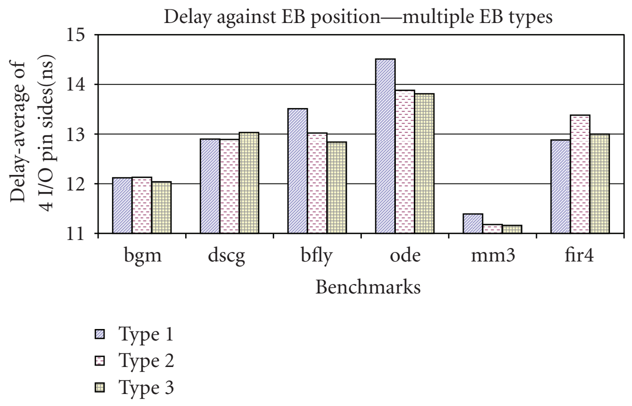


Figure 11: Delay against various EBs positions for multiple EB types FPGA, as defined in Figure 6.

6.1. Multiple EB Position Results

We first explore the effect of the BRAMs position on the floating point hybrid FPGA. Figure 11 shows the best location between floating point units which corresponds to type 3 in Figure 6. This configuration performs better than the traditional column-based BRAM (type 1 in Figure 6) used in Xilinx devices because the connections between floating point units and BRAMs are reduced in this case. In a similar way shown in Section 5.1, placing the embedded blocks closer together reduces the length of the bus-based connections between the embedded blocks which improves performance and reduces congestion.

6.2. Multiple EB Interconnect Flexibility

Finally, we investigate routing resources for the multiple EB system. Figure 12 shows delay for different channel widths. The channel width is observed to be nearly constant which is similar to the case discussed in Section 5.3. Table 5 shows the increase in channel width required when embedded BRAM is introduced. Since both FPUs and BRAMs have large I/O requirements, an increase in channel width of 12% over the case without embedded BRAM is required.

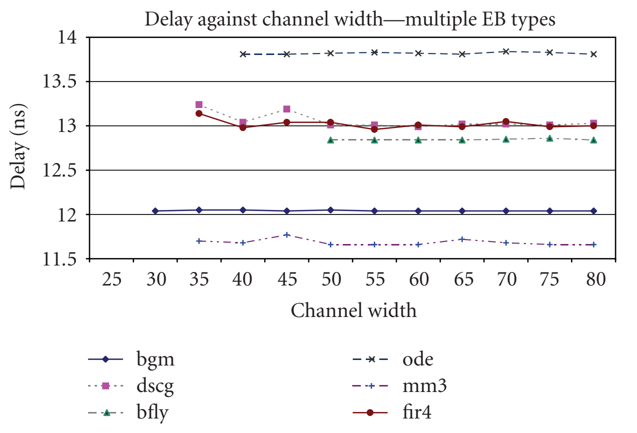


Figure 12: Delay against channel width for multiple EB types FPGA.

Table 5: Minimum channel width of multiple EB types FPGA.

Circuits	Minimum channel width	Channel width increase (%) (I/O pos.: 4 sides, type 3 in Figure 4)
bgm	27	0
dscg	35	6.06
bfly	50	35.14
ode	36	9.09
mm3	32	6.66
fir4	33	13.79
Average	35.5	11.79

7. Conclusion

This paper investigates the architecture of the programmable interconnect between coarse-grained blocks and the fine-grained fabric in domain-specific FPGA with embedded floating point blocks. Specifically, we first examine the position of the embedded blocks (EBs) within the FPGA, the placement of the pins on the periphery of the EB, the width of the routing channels surrounding the EB, and the aspect ratio of the EB for single EB type FPGA. After that we explore the EBs position and the channel of multiple EB types FPGA. We find that (a) the EBs should be positioned close to each other in the middle of the chip, (b) the EB's pins should be distributed evenly around the EB, (c) the width of the channels surrounding the EB have little impact on circuit speed, (d) a square EB leads to the most efficient implementations (e) smaller EBs should be located between large EB to achieve higher speed, and (f) embedding higher I/O density EB types leads to more routing resources being consumed. Although our results are specific to the architecture studied, we believe that they can be applied to FPGAs containing other types of embedded blocks. Current and future work includes extending our methodology to cover other embedded blocks for different domain-specific applications.

Acknowledgment

The support of UK Engineering and Physical Sciences Research Council (EP/D060567/1 and EP/C549481/1), Agility Celoxica and Xilinx are gratefully acknowledged.

References

[1] C. H. Ho, C. W. Yu, P. H. W. Leong, W. Luk, and S. J. E. Wilton, "Domain-specific hybrid FPGA: architecture and floating point applications," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 196–201, Amsterdam, The Netherlands, August 2007.

[2] C. W. Yu, J. Lamoureux, S. J. E. Wilton, P. H. W. Leong, and W. Luk, "The coarse-grained/fine-grained logic interface in FPGAs with embedded floating-point arithmetic units," in *Proceedings of the 4th Southern Conference on Programmable Logic (SPL '08)*, pp. 63–68, San Carlos de Bariloche, Argentina, March 2008.

[3] V. Betz, J. Rose, and A. Marquardt, *Architecture and CAD for Deep-Submicron FPGAs*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1999.

[4] Altera Corp., "Stratix III Device Handbook, Vol. 1," 2006.

[5] Xilinx Inc., "Virtex-5 Family Overview - LX, LXT, and SXT Platforms," 2007.

[6] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 26, no. 2, pp. 203–215, 2007.

[7] A. Ye, J. Rose, and D. Lewis, "Architecture of datapath-oriented coarse-grain logic and routing for FPGAs," in *Proceedings of IEEE Custom Integrated Circuits Conference (CICC '03)*, pp. 61–64, San Jose, Calif, USA, September 2003.

[8] K. Compton and S. Hauck, "Totem: custom reconfigurable array generation," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 111–119, Rohnert Park, Calif, USA, April-May 2001.

[9] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: a coarse grained paradigm for FPGA," in *Proceedings of the Dagstuhl Seminar 06141*, Dagstuhl, Germany, April 2006.

[10] R. Enzler, C. Plessl, and M. Platzner, "System-level performance evaluation of reconfigurable processors," *Microprocessors and Microsystems*, vol. 29, no. 2-3, pp. 63–73, 2005.

[11] P. Jamieson and J. Rose, "Enhancing the area-efficiency of FPGAs with hard circuits using shadow clusters," in *Proceedings of IEEE International Conference on Field Programmable Technology (FPT '06)*, pp. 1–8, Bangkok, Thailand, December 2006.

[12] S. J. E. Wilton, J. Rose, and Z. G. Vranesic, "The memory/logic interface in FPGAs with large embedded memory arrays," *IEEE Transactions on Very Large Scale Integration Systems*, vol. 7, no. 1, pp. 80–91, 1999.

[13] "Virtex-II Plaform FPGAs: Complete Data Sheet," (v3.4), March 2005, http://direct.xilinx.com/bvdocs/publications/ds031.pdf.

[14] T. Wong and S. J. E. Wilton, "Placement and routing for non-rectangular embedded programmable logic cores in SoC design," in *Proceedings of IEEE International Conference on Field-Programmable Technology (FPT '04)*, pp. 65–72, Brisbane, Australia, December 2004.

[15] C. H. Ho, P. H. W. Leong, W. Luk, S. J. E. Wilton, and S. Lopez-Buedo, "Virtual embedded blocks: a methodology for evaluating embedded elements in FPGAs," in *Proceedings of the 14th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, pp. 35–44, Napa, Calif, USA, April 2006.

[16] C. W. Yu, "A tool for exploring hybrid FPGAs," in *Proceedings of the International Conference on Field Programmable Logic and Applications (FPL '07)*, pp. 509–510, Amsterdam, The Netherlands, August 2007.

[17] http://www.doc.ic.ac.uk/~cyu/vph.zip.