High Level Abstractions for Implementation of Software Radios

J. B. Evans, Ed Komp, S. G. Mathen, and G. Minden
Information and Telecommunication Technology Center
University of Kansas, Lawrence, KS 66044-7541

ABSTRACT

A significant portion of most DSP designs consist of structured components like adders and multipliers. These components are most efficiently mapped to programmable hardware (FPGA) by intelligent module generators. We use this mapping in the implementation of a software radios, reconfigurable over multiple modulation schemes and a wide range of parameters.

1. Introduction

The real time throughput requirements of high performance DSP systems often dictate hardware intensive solutions. A system implementation based on a DSP microprocessor often fails to exploit the inherent parallelism present in the algorithm. But pure hardware solutions have high overhead in development and implementation time. Field Programmable Gate Arrays (FPGA) provide a rapid prototyping platform, with no non-recurring engineering costs. In practice, efficient FPGA solutions also have relatively high development costs and are relatively inflexible. In this paper, we demonstrate techniques for implementing rapidly reconfigurable radio processing functions on FPGAs, that also exhibit good performance characteristics.

Our system combines features of module generators, which allow more abstract and concise definition of systems, and rule-based selection of mappings to generate optimal designs based on specific attributes and constraints for individual designs.

2. Beyond Module Generators

Module generators take advantage of the regular patterns which exist in many of the core functions common in DSP applications. They can often generalize over the width of inputs and/or number of stages. Most existing module generator based systems, however, only allow one to capture a specific algorithm within a single module/operation definition. If multiple algorithmic alternatives exist for a particular operation, a distinct module must be defined for each algorithm. Then, when using an operation at a higher level, the user must choose a specific implementation at definition time.

In our system, definitions are expressed as rule definitions. One can define an arbitrary number of alternatives for any rule, with different selection criteria. The optimal implementation for each operator (definition) is selected (based on rule transformations) when generating the circuit for a complete design. Waiting as late as possible to bind an operation to a specific implementation provides the maximum benefit, because more design specific information is available on which to base the decision.

Most user rule definitions are expressed as the composition of existing rules. This supports a natural, hierarchical definition of increasingly complex systems. These capabilities make our system appear very similar to module generator based systems

which are increasingly common in today's CAD environment. It shares with them, the ability to more quickly define correct and concise definition of solutions through abstractions, reuse of code through parameterized common sub-components. In addition, however, these user definitions transparently acquire the optimizations associated with rules being composed.

3. Rule Based Evaluation

Our compiler accepts design descriptions defined by rule transformations written in a functional language called Flash. The compiler reduces the input to successively simpler descriptions based on repeated application of rule transformations. It then generates a low level VHDL description of the design. This is then passed on to commercial VHDL synthesis tools to produce a net list for the target FPGA, which in turn, is placed and routed by vendor specific tools.

Rule selection is based on a variety of compile-time information, including:

- Input type(s)
- Input width
- Binary representation of known values
- Parameter values
- Compile time constants
- Global settings, such as "optimize for space"
- Target Hardware

There may be an arbitrary number of implementations for any particular operation which are distinguished by their selection criteria. This flexible rule-based approach allows us to select the optimal implementation for a generic operation for the specific application.

As a specific example, numerous rules are defined for multiplication. A few of these are listed here:

- Both arguments are known values. No circuitry is required for this operation. The multiplication can be performed in the compiler, and the multiplier output replaced with this known value.
- One argument is a known value. A KCM (Constant Coefficient Multiply) algorithm can be applied.
  - Number of 1's in the known value binary representation is less than cutoff, implement the KCM with a shift and add algorithm.
  - Otherwise, use a table-lookup implementation for the KCM.
- Arguments are equal, but unknown; use a lookup table.
- Otherwise, perform a general shift and add implementation.

All definitions in Flash are defined in terms of rules. As a result, optimizations can be applied at arbitrary levels of abstraction. This allows one to take advantage of information available only at more global or abstract view of an algorithm.

As a specific example, consider implementation of a Finite Impulse Response (FIR) filter. The typical implementation is a regular sequence of Multiply Accumulate (MAC) blocks separated by latches. Each MAC multiplies the input by a constant (tap) associated with that stage of the filter. Since these taps are represented with a finite number of bits, multiple taps may have equivalent representations. Recognizing this fact, allows one to define an implementation of the FIR which first examines the binary representation for the taps, and generates a multiplication circuit only for each distinct valued tap (rather than a multiplication circuit for every tap).

Flash is supplied with a core set of optimization rules for the basic boolean operations on bits. In addition we have developed a core library of elementary DSP functions including elementary arithmetic functions and some higher level functions such as FIR filters. Typically, users begin the definition of a specific system based on these core rule libraries.

All rules are entirely external to the Flash compiler itself. Users can augment this set with new function definitions or additional optimization rules for existing functions.

4. Implementation of Digital Radios

Radio processing functions for various modulation and demodulation schemes map well to digital operations. With a relatively small collection of basic modules, including:

- Signal mixers, using multiplication;.
- Filtering. with FIR filters;
- Multi-stage delays;
- Carrier recovery;
- Local carrier generation, via ROM tables of the sine or cosine functions

a wide variety of digital radios can be implemented.

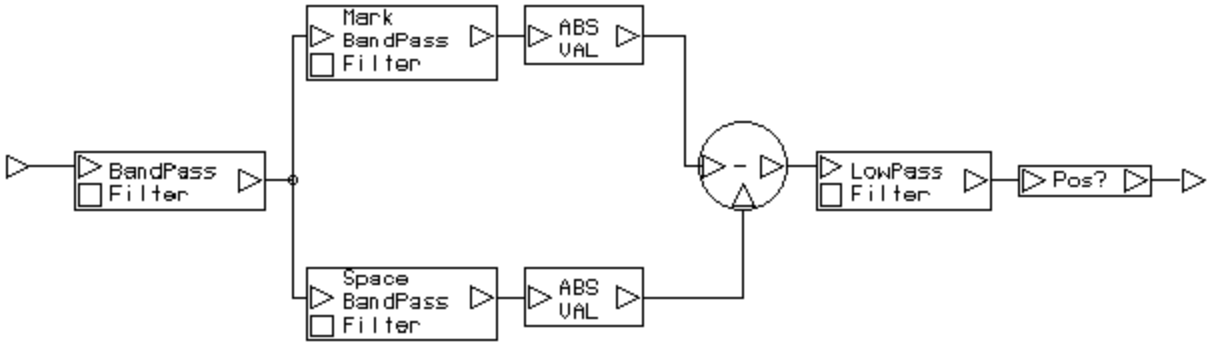
Our design environment has been successfully tested with the implementation of the following digital radios: AM; LSB; USB; FM; BFSK; BPSK; and QPSK.

Our primary objective has been to implement a set of highly configurable radios to address some of the issues in the adaptive computing systems environment. At the same time we hope to be able to at least approximate the performance of hand-coded designs written for a specific parameter set.

The following section briefly describes the design of one of these radios in greater detail.

4.1 BFSK Demodulator

In BFSK (Binary Frequency Shift Keying) modulation, zeros and ones are sent



at narrowly separated mark and space frequencies. Figure 1 shows a block diagram of a simple BFSK demodulator.

The Flash implementation, shown below, parameterizes the design over the system sampling rate and the mark and space transmit frequencies. In this example, the order of the filters and the amount of precision used for the tap coefficients have been set internally. Each of these values, appears as a constant in a lower level rule application which is easily changed in a text editor.

```
(define (bfsk sample-freq mark-freq
              space-freq data-freq)
  (term-rules
    [(input : fixedpt) => fixedpt
     ;; Transition BW for mark/space filters
     (let ((sb-adj (- (/ (+ mark-freq
                            space-freq) 2)
                      mark-freq)))
       (bind ((filtered-input
                ;; BP filter to remove noise
```

```
        ;; outside space/mark freq
        (app (bandpass-filter-remez
              sample-freq
              (* .5 mark-freq)
              mark-freq space-freq
              (* 1.5 space-freq)
              16 input.f)
             input)))
  ;; LP filter before output
  (app (lowpass-filter-remez
        sample-freq data-freq
        (1.5 data-freq) 12 input.f)
       (@ subtract
          (app absolute
     ;; BP filter at mark-freq.
     (app (bandpass-filter-remez
           sample-freq
           (- mark-freq sb-adj)
           (- mark-freq data-freq)
           (+ mark-freq data-freq)
           (+ mark-freq sb-adj)
           16 input.f)
          filtered-input))
(app absolute
      ;; BP filter at space-freq.
     (app (bandpass-filter-remez
           sample-freq
           (- space-freq sb-adj)
           (- space-freq data-freq)
           (+ space-freq data-freq)
           (+ space-freq sb-adj)
           16 input.f)
          filtered-input)
     filtered-input)))))]))
```

5. **Results**

Our design environment has been successfully tested with the implementation of a variety of digital radios. These radio descriptions were abstracted over design constants like carrier frequencies and data rates. In this way we obtain rapid reconfiguration within a wide range of design specifications. The reconfiguration time required is less than half an hour - which is the time required for the compilation and for the FPGA vendor tools to perform place and route.

Our system facilitates design re-use and permits concise definitions of complex systems. The table below compares the number of lines of Flash code required to define each radio to the number of lines of VHDL code generated for each. Though hand crafted VHDL could be more concise, the size of generated VHDL output is a rough indication of the compactness of our source representation.

Design	*Functional Description*	*Generated VHDL*
AM	16	306
FM	45	520
BFSK	50	571
BPSK	82	972
QPSK	72	1334

In the following implementation results, the hardware target for all implementations was a Xilinx XC4085XLA FPGA. Size is reported in allocated CLBs (Configurable Logic Blocks) and speed as the maximum clock rate as reported by the place and route tools.

To evaluate the performance gain of selected higher level design rule optimizations, we generated circuits for the same system definition, but with different sets of rule transformations in place. This allows us to evaluate exactly the improvement attributable to the additional rule(s).

The results in the following table summarize the results for implementation of a 22 tap FIR low pass filter, using 10 bits of binary precision for the coefficients. The Basic design results were generated using a FIR transformation based on a standard series of 22 multiply accumulate blocks, The MinMult design results were generated using a FIR transformation which minimizes the number of multiply circuits.

Design	*Size* CLBs	*Speed Mhz*
Basic	401	36.30
MinMult	346	37.00

For comparison to hand-coded VHDL, other members of the team implemented some of the radios directly in VHDL. These implementations were written for a specific

set of parameters, allowing the authors to make optimizations based on bit-widths, filter tap coefficients, etc.

The following table shows these results for the BFSK demodulator.

Design	*Size* CLBs	*Speed* *Mhz*
HandCoded	855	29.13
Flash	737	30.87

The Flash implementation is both smaller and can run at a higher clock rate.

From the perspective to source language definitions (Flash compared to VHDL), the Flash implementation is very clearly superior. Only 50 lines of Flash were required to define the BFSK demodulator, while the hand-coded VHDL includes over 90 FILES (and 6000+ lines of code).

In addition the hand-coded VHDL implementation was customized for a particular bit width, specific filter coefficients. Modification of any system parameter, or a variation in accuracy requirements would require significant re-design for large portions of the 6000 source lines. In contrast, any combination of these changes could be altered in the Flash design by modification of the associated parameter.

We note that the amount of VHDL code is significantly larger than the number of lines of VHDL generated through the Flash design. This is largely due to the coding style employed for the hand-coded implementation. The author wisely chose to separate the various components into separate entities in order to perform unit testing, etc. Combining separate entities in a hierarchical fashion in VHDL involves significant overhead when simply counting lines of code. In addition, a utility program to generate table-lookups for KCMs was employed which generated approximately 150 lines of VHDL for each KCM.

6. Conclusion

This paper summarizes the results pertaining to the use of intelligent module descriptions and a design compiler for DSP based designs. The design reuse and exploitation of application domain specific optimizations are strong factors in favor of this approach. Specific instances of the radio designs were generated from our generic descriptions, allowing rapid reconfiguration of the programmable hardware to perform different radio functions and still provide final circuit implementations with efficiencies comparable to hand-coded, parameter specific implementations.

.

ACKNOWLEDGEMENT: *This work is funded by DARPA contract number DABT63-97-C-0032 as part of the Adaptive Computing Systems Initiative.*