

**LOCAL DECODING OF WALSH CODES  
TO REDUCE CDMA DESPREADING COMPUTATION**

Matt Doherty

TA: Kyeong-Jae Lee

6.111 Introductory Digital Systems Laboratory

May 18, 2006

**Abstract**

As field-programmable gate arrays (FPGAs) continue to become more powerful and more flexible, computer scientists are looking to them to combine the flexibility of software with the performance afforded by the specialization of hardware. An algorithm for local decoding of CDMA Walsh codes has been translated from software to hardware and the problems and benefits associated with the translation have been analyzed. In software, the algorithm saves valuable computation, but in hardware, this computation savings results directly in power savings as well. The system uses a VGA display to show, in real time, the decoder's bit-error rate, the received signal's signal-to-noise-ratio, and an estimate of the power being consumed by the device. The project is a successful proof-of-concept not only that complex software algorithms can be implemented in hardware, but also that significant power savings can be realized in CDMA base stations today through the use of a different decoding algorithm.

## **Table of Contents**

|                                       |    |
|---------------------------------------|----|
| 1.0 Introduction                      | 3  |
| 2.0 Module Description/Implementation | 4  |
| 3.0 Testing/Debugging                 | 9  |
| 4.0 Conclusion                        | 10 |
| 5.0 Bibliography                      | 10 |

## **List of Figures**

|                               |   |
|-------------------------------|---|
| 1. Labkit block diagram       | 4 |
| 2. Walsh decoder block digram | 4 |
| 3. Bahl's 256-chip FHT design | 5 |
| 4. Bahl's single FHT stage    | 6 |
| 5. Screenshot of VGA display  | 7 |

## 1.0 Introduction

As field-programmable gate arrays (FPGAs) continue to become more powerful and more flexible, computer scientists are looking to them to combine the flexibility of software with the performance afforded by the specialization of hardware. The project described herein is one such computer scientist's implementation of a software algorithm in hardware and analysis of the problems and benefits associated with the translation.

Code-division multiple access (CDMA) is a wireless standard that relies on orthogonal Walsh codes to multiplex signals transmitted simultaneously over the same frequency to the same base station. To recover the bits transmitted over the reverse link, base stations must correlate the received symbols with all possible Walsh codes.

In 2005, Chan et al. invented several novel classes of algorithms to reduce computation in software implementations of the IS-95 reverse link by exploiting software's inherent flexibility over hardware. The algorithms function by processing only a fraction of the despread signal to decode Walsh codewords, relying on a feedback loop to choose the fraction of despread signal to process in order to maintain a target bit error rate (BER).

Because FPGAs are more flexible than application-specific integrated circuits (ASICs), the same algorithms can be implemented in hardware in an FPGA to reduce computation. This reduction in computation results directly in power savings because FPGA power consumption is tightly correlated with its gates' switching activity. This report details the design and implementation of the "novel and elegant" Generalized Local Decoding of Walsh codewords in an FPGA and estimation of the power savings of the algorithm in hardware. (Chan et al.)

## 1.1 System Behavior

Upon *reset*, the FPGA steps through a test vector of Walsh chips. The decoder determines the six bits that correspond to every Walsh codeword of length 64 chips. The user is presented with a graph and three gauges. The graph shows the decoder's bit error rate (BER) over time. The three gauges show the instantaneous decoder BER, signal-to-noise ratio (SNR) of the test vector, and power usage of the device.

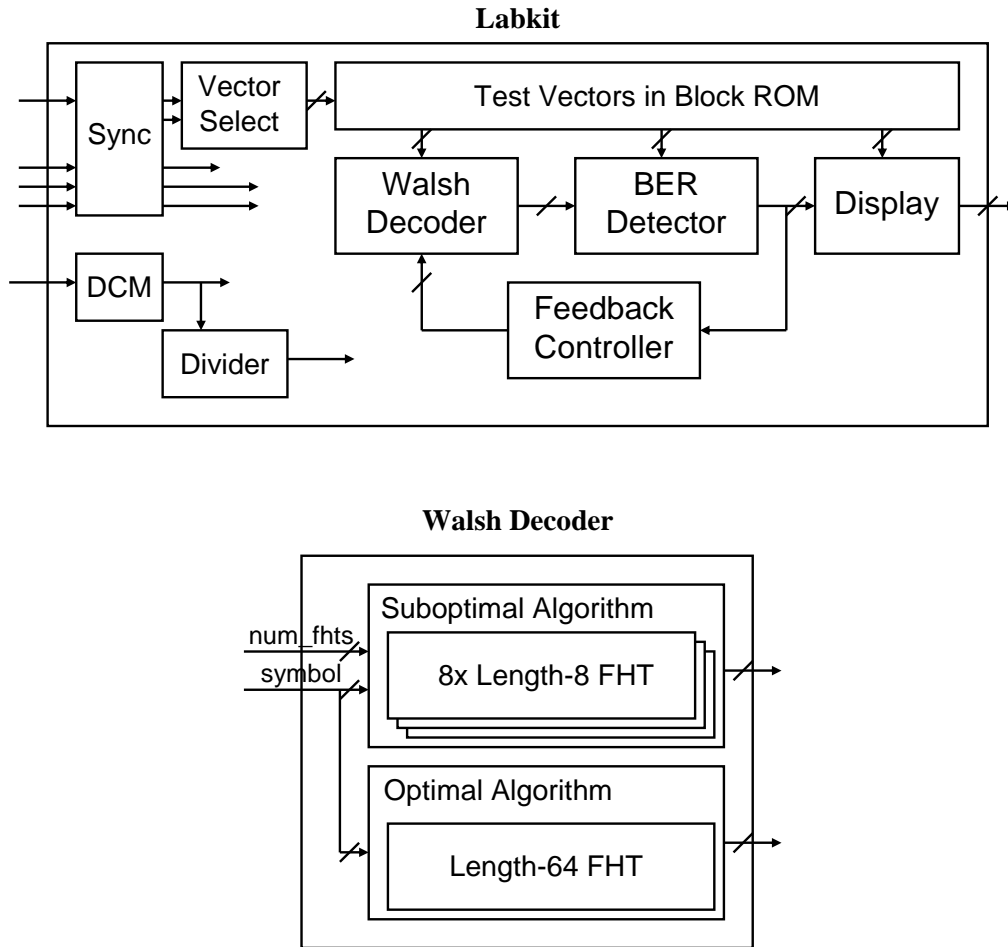
Using button 3, the user can cycle among three decoding algorithms. The first is the optimal Walsh decoder, which feeds all 64 symbols to a length-64 Fast Hadamard Transform (FHT) that decodes the Walsh codeword. It uses the most power because it correlates every combination of the input symbols with every Walsh code. But it also results in the lowest average BER and thus serves as a benchmark against which the performance of the other algorithms can be validated.

The second is the generalized local Walsh decoder. Using the left and right buttons, the user can choose the number of length-8 FHTs with which the algorithm decodes the Walsh codeword. As few as two and as many as eight length-8 FHTs can simultaneously estimate the six transmitted bits. Since not all of the combinations of symbols are used in this estimation, the algorithm is "suboptimal." However, as long as the signal is relatively free of noise, as few as 16 symbols can be used to decode the Walsh codeword while maintaining a low BER. Especially at the single-FHT setting, this decoder's gates switch much less frequently than those of the optimal decoder, and so it uses much less power.

Third, the adaptive generalized local Walsh decoder is simply a generalized local decoder with a feedback loop wrapped around it. With this feedback loop, the algorithm maintains a target BER while using as few length-8 FHTs as possible to decode the Walsh codeword.

Using the up and down buttons, the user can adjust the signal-to-noise ratio (SNR) of the test vector. In general, a lower SNR results in a higher BER because of the larger number of symbols that are received incorrectly on average.

## 1.2 Block Diagrams



Figures 1 and 2. The lab kit block diagram, and detailed view of the Walsh decoder block.

## 2.0 Module Description/Implementation

### 2.1 Walsh Decoder

The three decoding algorithms are instantiated within the Walsh decoder block. This module sets *reset* high for all but the selected algorithm, and outputs the six decoded bits when the selected algorithm sets its *ready* flag high.

### 2.2 Suboptimal Algorithm

The suboptimal decoder is the crux of the project. It was originally designed for and implemented in software. Albert Chan from Vanu, Inc. was gracious enough to provide an Octave script that simulates the IS-95 reverse link and performs local decoding on the received symbols. Using Chan et al.'s design and this script, the multiple-FHT local decoder was designed.

During cycles 0-63 after a *reset* signal, the decoder buffers the symbols from the ROM into a memory. Then, from cycle 64 to cycle 67, different combinations of symbols are fed into the eight length-8 FHTs. During cycles 67-70, the outputted correlation coefficients are summed component-wise and the largest two values (one for the first three bits, and one for the second three bits) and corresponding indices are held. On cycle 71, the decoder outputs the two three-bit indices of the largest correlation coefficients and sets its *ready* flag high.

Depending on the number of FHTs used, the suboptimal decoder sets the *reset* signal high on some number of its eight length-8 FHTs. The *reset* signal prevents the FHTs from switching, thereby saving power under high SNR conditions.

### 2.3 Optimal Algorithm

The optimal decoder is not significantly different from the suboptimal decoder. It too buffers incoming symbols during cycles 0-63 after a *reset* signal. Then, during cycles 64-95, it feeds the 64 symbols to the length-64 FHT. The decoder then holds the largest correlation coefficient that the FHT outputs during cycles 95-126. On cycle 127, it outputs the six-bit index of the largest correlation coefficient and sets its *ready* flag high.

### 2.4 Fast Hadamard Transform

To reconstruct the bits transmitted over the reverse link, the base station performs an FHT on the received real-valued symbols. The FHT correlates the received codeword with all possible Walsh codewords of its length and returns the bits corresponding to the most-likely candidate.

The design for the hardware FHT is adapted from Bahl's "Design and Prototyping a Fast Hadamard Transformer for WCDMA." It uses the same butterfly structure as a Fast Fourier Transform (FFT) to generate intermediate correlations, and uses shift registers, enabled at various times, to correlate every symbol together with every code. At each stage from the input, the shift registers are half the length of those in the previous stage. This design is illustrated in figure 3.

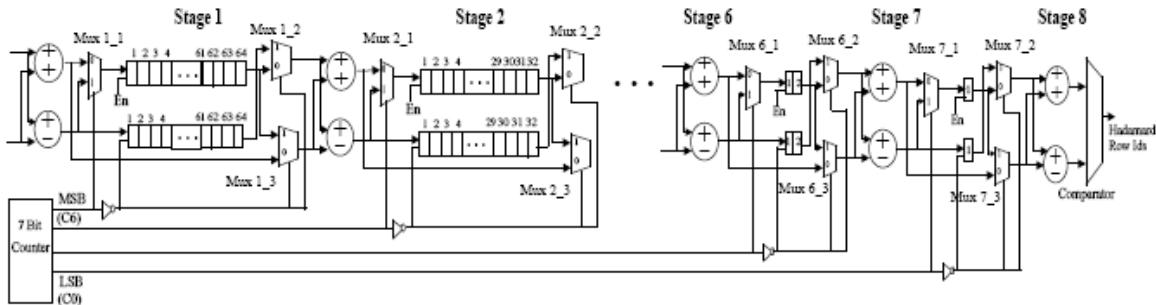
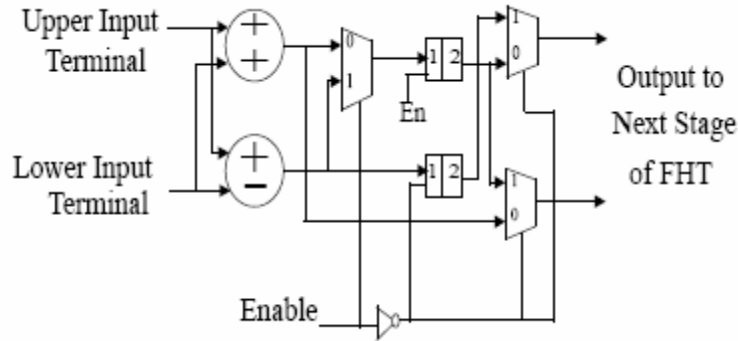


Figure 3. The 256-chip FHT design described by Bahl.

#### 2.4.1 FHT Stage

A single stage of the FHT is abstracted into its own module. This abstraction allows variable-length FHTs to be constructed without retesting individual stages or ever seeing intermediate correlations. Figure 4 shows the FHT stage design given by Bahl.



**Figure 4.** The FHT stage described by Bahl.

### 2.4.2 Length-8 FHT

The length-8 FHT is used by the suboptimal Walsh decoder and is simply a subset of the length-256 design given by Bahl. During cycles 0-3, symbols 0-3 should be fed into the upper input terminal, and symbols 4-7 into the lower input terminal. During cycles 3-6, correlations arrive on the output terminals. On cycle 3, correlation 0 is on the upper output terminal and correlation 1 is on the lower output terminal. On cycle 4, correlation 2 is on the upper terminal and 3 on the lower, and so on. The correlation index with the highest coefficient is the most likely decoding of the three bits, and this value is also the FHT's confidence in its answer.

### 2.4.3 Length-64 FHT

The length-64 FHT is used by the optimal Walsh decoder and, again, is simply a subset of Bahl's length-256 design. During cycles 0-31, symbols 0-31 are fed into the upper input terminal, and symbols 32-63 into the lower input terminal. During cycles 31-62, the output terminals hold the correlation coefficients enumerated as described above. In other words, the first two correlation coefficients are outputted 31 cycles after the first two symbols are inputted.

## 2.5 Bit Error Rate Detector

The BER detector finds the number of mismatched bits by summing the exclusive-ors of the bits from the decoder  $a$  and the actual transmitted bits  $b$ :

$$\sum_{i=0}^5 a_i \oplus b_i.$$

In a more realistic configuration, this module would not exist, as it is impossible to detect the number of bit errors with perfect accuracy.

## 2.6 Feedback Controller

The feedback controller uses the BER found in the BER detector to select the number of FHTs with which to decode the input codeword. It is a simple proportional controller: if the BER is different from the target BER, the number of FHTs is increased or decreased proportional to the difference  $BER_{actual} - BER_{target}$ .

In a more realistic configuration, this controller would use the FHT's confidence rather than the BER to choose the decoding scheme. If the confidence is too low, then the number and size of the FHTs is increased until either the confidence is high enough or the codeword is decoded optimally. Moreover, a better controller would be proportional-derivative (taking into account the derivative of the error), since real-world SNR can vary very quickly.

## 2.7 Algorithm Selector

This module takes a signal from a button and sets the control signals necessary to select another algorithm. For each button press, the module selects the next algorithm in series: local, optimal, then adaptive.

### 2.8 Vector Select

The vector select module takes up and down button signals and outputs a memory address for the test vector ROM. The SNR defines the vector base, then for each enable signal received, the codeword base is incremented. And at each clock cycle, the symbol base is incremented. Together, the vector, codeword, and symbol bases constitute 10 bits that can address 1024 10-bit symbols.

### 2.9 Test Vector ROM

The test vector ROM holds two 8-codeword vectors, for a total of 1024 10-bit symbols. Of these 1024, 512 are received at 0 dB SNR (simulated), and 512 are received at -5 dB SNR (simulated). Eight 64-symbol codewords constitute each vector.

The test vector ROM was constructed using the Octave script given by Chan to generate the noisy Walsh chips, and a Python script by the author to scale and format the chips into a .coe file for Xilinx IP/CoreGen. Both scripts are available in Appendix A.

### 2.10 Display

The display module assembles the graph, three gauges, and text captions. Its graphical output is shown in figure 5.

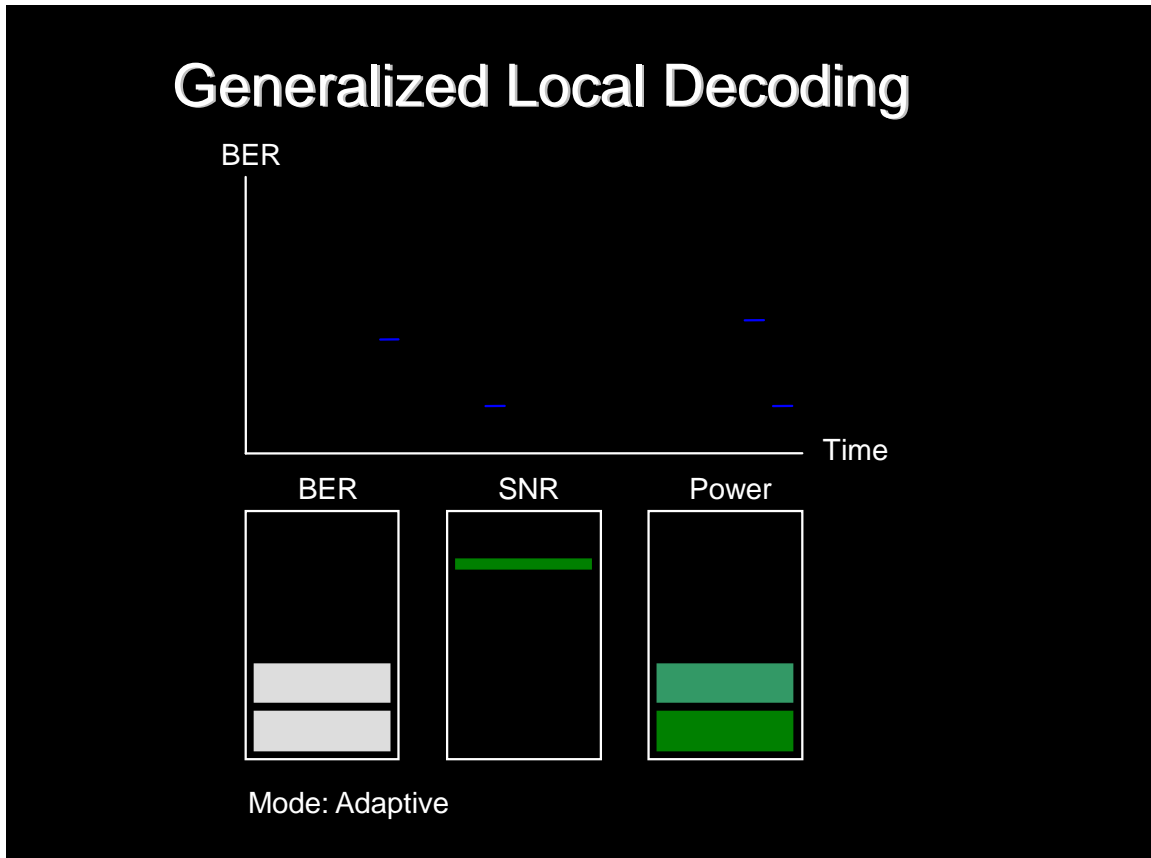


Figure 5. Screenshot of project's VGA display.

### **2.10.1 Horizontal Line**

This module outputs a nonzero *color* if and only if the *pixel* value is within its *x* parameter range and *line* value is equal to its *y* parameter. It is used to display the time axis of the graph and the horizontal boundaries of the gauges.

### **2.10.2 Vertical Line**

This module outputs a nonzero *color* if and only if the *line* value is within its *y* parameter range and *pixel* value is equal to its *x* parameter. It is used to display the BER axis of the graph and the vertical boundaries of the gauges.

### **2.10.3 Dot Graph**

The dot graph stores incoming BERs in a 3-bit-wide shift register. On every *enable* signal, it shifts in a new BER and shifts all its stored values back. When the *line* value corresponds to a stored value at an index that corresponds to the *pixel* value, it outputs a nonzero *color*. Thus, it shows the decoder's BER over time.

### **2.10.4 BER Gauge**

This module takes a 3-bit value as input and represents it with rectangles. A value of zero corresponds to zero displayed rectangles. It shows the instantaneous BER.

### **2.10.5 SNR Ratio Gauge**

This module takes a 3-bit value as input and represents it with a narrow rectangle. A value of zero corresponds to the topmost location. It shows the instantaneous SNR.

### **2.10.6 Power Gauge**

This module takes a 3-bit value as input and represents it with rectangles that vary in color from green on the bottom to red on the top. A value of zero corresponds to zero displayed rectangles. It shows an approximation of the instantaneous power usage.

### **2.10.7 Character String Display**

The character string display was taken in large part from last year's 6.111 web site. It determines whether the current VGA line and pixel counts align with those that should be on to display a character as specified by the font ROM. If so, it sets the output pixel bits high; else, it sets them low. A scale factor was added to the size of the bitmapped font to vary.

### **2.10.8 Font ROM**

The font ROM was taken from last year's 6.111 web site. It consists of ones that correspond to set pixels and zeros that correspond to unset pixels for each ASCII character. It is indexed by ASCII values and *line* count.

## **2.11 VGA**

The VGA module coordinates the timing of signals to the VGA display operating at 640x480 resolution at 60 Hz. It is important because the display modules rely on it to time their output signals correctly for the physical display.

## **2.12 DCM**

The DCM module takes as input the 6.111 lab kit's 27.5 MHz clock and outputs a 26.6 MHz pixel clock. The module is important because the display's resolution of 640x480 pixels and refresh rate of 60 Hz requires signals to switch at this frequency.



### 2.13 Divider

The divider uses the 26.6 MHz pixel clock to output a 5 Hz enable signal. This module is important because it times the decoding of individual codewords and the display of statistics.

### 2.14 Debounce/Synchronizer

Input from physical devices must be debounced and synchronized to the clock before use. Debouncing eliminates the high frequency noise in the signal that occurs when physical contacts come together or separate; synchronization significantly lowers the probability of reading a metastable invalid value from the input.

## 3.0 Testing/Debugging

Several issues arose that were related directly to the hardware description language (HDL), Verilog. First, implementation constraints actually precluded a more modular design. The buffering of the 64 input symbols in a codeword should occur outside the decoders, not within them. Because the decoders have to perform this function individually, the same code had to be tested and debugged twice.

In this case, one especially nonsensical restriction was at fault: Verilog cannot register-index another register. Specifically, the following code does not compile:

```
reg [7:0] index;
reg [255:0] wide_register;
assign wide_register[index] = 1;
```

But the following is a trivial workaround:

```
reg [7:0] index;
reg [255:0] wide_register;
assign wide_register[0] = (index == 0) ? 1 : 0;
assign wide_register[1] = (index == 1) ? 1 : 0;
...
assign wide_register[255] = (index == 255) ? 1 : 0;
```

Unfortunately, it requires typing quite a bit, or writing a script to generate the code. Oddly, the compiler is situated perfectly in the toolchain to perform the static elaboration automatically, but it does not. (Static elaboration is simply the ability of a compiler to expand, or elaborate condensed code at compile time.)

Because this workaround was not used, a wide, flat register could not be passed from the buffer to the decoder module. Thus, the decoder modules had to buffer the symbols themselves in memories.

The second major issue was also related to the compiler's inability to statically elaborate: Verilog cannot index a register with an integer, even if the integer is always well-defined. Specifically, the following code does not compile:

```
reg [7:0] buffer;
integer i;
always @(posedge clock) begin
    for (i = 0; i < 8; i = i + 1) begin
        buffer[i] <= ~buffer[i];
    end
end
end
```

Again, there is no reason that the compiler cannot unroll this loop itself. For the suboptimal decoder module, a separate Python program had to be written to unroll loops. Since the indices were fairly complex, it would have been painstaking to write and debug them by hand.

Finally, timing remains a major problem for digital circuits. Originally, this project's design called for up to 16 length-8 FHTs in the suboptimal decoder. But the component-wise summation of all 16 FHTs' outputs proved too time-intensive to fit in a clock cycle. Unfortunately, this summation was difficult to pipeline because the FHTs' outputs are only available for one cycle. The solution, then, was to reduce the maximum number of FHTs to eight, thereby halving the critical path. Pipelining the decoder would have required a different design from the ground-up (even the shift registers would have to change), or a more advanced HDL like Blusepec that can better describe datapaths.

#### **4.0 Conclusion**

It is apparent at lower SNRs that the adaptive generalized local decoder achieves a significantly lower BER than the single-FHT generalized local decoder while using significantly less power than the optimal decoder. Thus, it is a compromise between the low-power and high-BER single-FHT generalized local decoder, and the high-power and low-BER optimal decoder. The project was a success.

Unfortunately, accurate power usage data was not available. XPower, Xilinx's power estimation tool, uses a simulation file to aid in estimation. However, ModelSim does not allow block ROMs to be simulated along with other modules, and the decoder requires an input vector to do any decoding. A workaround is simply to store such a vector as registers in the test module, but that is left as an exercise for the future.

Overall, decent modularity and explicit prevention of error propagation made system integration relatively smooth. But the implementation of a software-optimized algorithm has yielded much insight into the reasons software engineers avoid FPGAs like the plague. The reasons can be summed up quite simply: software engineers have grown accustomed to much better tools.

"If I have seen further, it is by standing on the shoulders of giants," Isaac Newton said. Xilinx would be wise to stand on the shoulders of those who have incrementally improved software compilers over the years. C++ and Java compilers, for instance, halt compilation if an undeclared variable is detected (likely a typo) or an argument list is mismatched (another likely programmer error). They also unroll loops and go far out of their way to elaborate code statically.

But software does some other interesting things as well. The Java Runtime Environment, for example, dynamically optimizes programs as they are running. The result is that Java's byte-compiled code can actually run faster than the most highly-optimized compiled machine code. In short, I look forward to the day when FPGAs can dynamically reconfigure themselves to optimize for changing conditions in a layer beneath that in which programmers write application-specific code, so algorithms like adaptive local decoding are faster and easier to implement.

#### **5.0 Bibliography**

Chan, A., Feldman, J., et al. "Local Decoding of Walsh Codes to Reduce CDMA Despreading Computation." Available online at <http://web.mit.edu/matttd/Public/walsh.pdf>.

Bahl, S.K. "Design and Prototyping a Fast Hadamard Transformer for WCDMA." Available online at [http://web.mit.edu/matttd/Public/FHT\\_FPGA.pdf](http://web.mit.edu/matttd/Public/FHT_FPGA.pdf).