FPGA implementation of the DCD algorithm

Jie Liu, Ben Weaver and George White

Department of Electronics, University of York, UK

Abstract: This paper describes an experimental implementation of the Dichotomous Coordinate Descent (DCD) algorithm on a Xilinx Virtex-II Pro FPGA. The DCD algorithm is a computationally-efficient method for solving linear systems of equations – a common task in the field of digital signal processing. Our implementation is described, and our implementation results and resource requirement figures are stated. It is shown that the DCD algorithm forms a highly efficient method for solving linear systems of equations on FPGA, requiring 650 logic slices for a core to solve systems of equations of size N=50.

1. Introduction.

Obtaining the solution of a system of linear equations is a common task in many signal processing areas, for example adaptive filters employing the Wiener-Hopf equations [1]. This paper describes an experimental implementation of the Dichotomous Coordinate Descent (DCD) algorithm [2, 3] – a computationally-efficient iterative linear equation solver.

2. The DCD Algorithm.

The DCD algorithm is a hardware-efficient iterative method of solving linear systems of equations of the form $\mathbf{Rh}=\boldsymbol{\beta}$ where $\mathbf{R}$ is an $N\times N$ matrix and $\mathbf{h}$ and $\boldsymbol{\beta}$ are $N\times 1$ vectors. We denote $R(i,j)$, $h(j)$ and $\theta(j)$ elements of matrix $\mathbf{R}$ and vectors $\mathbf{h}$ and $\boldsymbol{\theta}$ respectively, and $R(:,j)$ the jth column of matrix $\mathbf{R}$. Like many other iterative algorithms, the initial guess at the result vector, $\mathbf{h}$, is zero. The first set of iterations determine the most significant bits (MSBs) for all elements of $\mathbf{h}$ using a step size parameter d initialised to some value H, and subsequent sets of iterations determine lower order bits until a suitable number of bits M are obtained.

If an iteration updates an element of the solution vector (such an iteration is labelled "successful"), an auxiliary vector $\boldsymbol{\theta}$ (which is initialised to $\boldsymbol{\beta}$) is also updated. The computational load of the algorithm is mainly due to these successful iterations, and to limit this (with an uncertain error of the solution), a limit for the number of successful iterations It_{MAX} can be predefined. If there is no such limit, or the limit is high enough, the accuracy of the solution is $2^{-M}H$. The operation of the algorithm is described as pseudocode in Figure 1. It is seen that if the step size d is a power of two, then only multiplications by factors of power of two are needed. These can be replaced by bit shifts in hardware implementation, and so the DCD algorithm can be implemented without explicit multiplication or division (often the source of numerical instability) operations, which are well-known to require significantly higher chip area and power consumption in hardware implementation than addition and bit-shift operations [4]. The complexity of the DCD algorithm for a particular system of equations depends on many factors, but for given It_{MAX} and M, the peak complexity can be shown to be N ($2\,It_{\text{MAX}} + M$) shift-accumulate (SAC) operations [5].

3. Architecture Description.

We have implemented an experimental FPGA core for solving systems of linear equations by DCD, based on a Xilinx Virtex-II Pro Development System [6]. The board is powered by an XC2VP30 FPGA (FF896 package, speed grade 7), which features 13696 logic slices each containing two D-type flip-flops and two four-input look-up tables (LUTs). The FPGA also has 136 block-RAM components, each offering 18kbits of storage [7]. We use VHDL to describe our core, and it is synthesised and downloaded to the target using the Xilinx ISE 7.1i software package. A block diagram overview of our FPGA DCD implementation is shown in Figure 2. The whole design operates from a single 100MHz clock and we make use of the FPGA DCM (Digital Clock Manager) and Global Clock Distribution Network to ensure there is uniform delay between the system clock source and each logic slice. Our design uses an RS232 serial link running at 115.2kbps between the FPGA and a host computer. Our

initialise: $\boldsymbol{\theta} = \boldsymbol{\beta}$, $\mathbf{h} = \mathbf{0}$, $d = H$, $counter = 0$
for $m = 1 : M$
  $d = d/2$
  do
    $flag = 0$
    for $j = 0 : N-1$
      if $|\theta(j)| > (d/2)R(j, j)$
        $flag = 1$; $counter = counter + 1$
        $h(j) = h(j) + \text{sgn}(\theta(j))\ d$
        $\boldsymbol{\theta} = \boldsymbol{\theta} - \text{sgn}(\theta(j))\ d\ R(:, j)$
        if $counter > It_{\text{MAX}}$ then stop
  repeat while $flag = 1$

Figure 1. Operation of the DCD algorithm

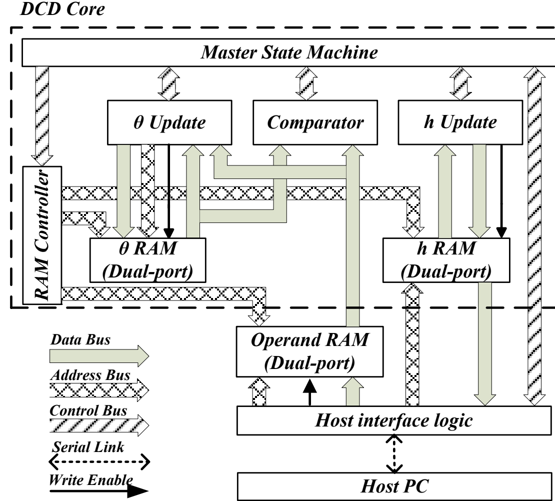

Figure 2. Block diagram of the DCD implementation

test set-up involves feeding randomly generated positive definite systems of equations of size N=50 into the FPGA from the host PC. For each system of equations, the accuracy of the FPGA-computed solution is checked against the same solution computed by a Matlab-based implementation of the DCD algorithm designed to give the same bit precision – in our case 32 bits.

The main DCD core consists of seven sub-modules: A master state machine, internal $\boldsymbol{\theta}$ RAM, internal **h** RAM, a RAM controller, a comparator, $\boldsymbol{\theta}$-update logic, and **h**-update logic. In addition to the main core, our implementation includes a host interface module. This is responsible for receiving test data from the host PC, placing it in FPGA block memory, and triggering execution of the main DCD process. After the DCD completes, the module is also responsible for reading the DCD result from FPGA block memory and transmitting it back to the host PC. The operand data (the **R** matrix and $\boldsymbol{\beta}$ vector), internal iteration data ($\boldsymbol{\theta}$ vector) and the result data (**h** vector) are stored in three separate memory components. The elements of both the matrix and vectors are mapped in a sequential and contiguous manner. Throughout the design, block-RAM components are chosen as they are faster and more power efficient than their distributed logic counterparts. The three memory components are all dual-port. In the case of operand RAM and result RAM, this is to separate the DCD logic from the interface logic, but in the case of the internal $\boldsymbol{\theta}$ RAM, dual-port memory is used so that data may be read, modified, and written back to the same memory in a pipelined fashion.

3.1 Master State Machine

The master state machine drives the other submodules and provides DCD control parameters such as step size, iteration count, and element indices. The key part of the DCD algorithm is the two-part “compare-update” iteration. There are, in practice, extra states for controlling index looping, but the operation of the DCD core can adequately be described using just these two “compare” and “update” states. The “compare” phase takes three clock cycles (one to perform the comparison, one to interpret the result of the comparison, and one cycle to test whether the final element of the $\boldsymbol{\theta}$ vector has been reached).

3.2. RAM Controller

The RAM controller drives the DCD Core side address ports of the operand RAM and **h** RAM, and both address ports of the $\boldsymbol{\theta}$ RAM. During the “compare” part of the DCD iteration, the RAM controller provides the address of $R(j, j)$ in operand RAM, the address of $\theta(j)$ in $\boldsymbol{\theta}$ RAM, and the address of $h(j)$ in **h** RAM. The “update” part of the iteration is conditional. If the previous comparison indicates a successful iteration, then the RAM controller will sequentially increment the $\boldsymbol{\theta}$ and **R** addresses from 0 to (N-1) until all N elements of $\boldsymbol{\theta}$ have been updated. It will then address $\theta(j+1)$ and $R(j+1, j+1)$ for the

next comparison. However, if the iteration is unsuccessful, the RAM Controller will address $\theta(j+1)$ and $R(j+1, j+1)$ immediately for the next comparison.

3.3. COMPARATOR

Figure 3 shows the layout of the comparator sub-module. The main function of the sub-module is to find the index of $\max(\theta(j), -\theta(j), (d/2)R(j, j))$. The values of $\theta(j)$ and $R(j, j)$ are provided from **θ** RAM and operand RAM respectively. The comparator state machine negates $\theta(j)$ and provides a decremented step size parameter d_{EXP}, which is used by a simple bit-shift device to compute $(d/2)R(j, j)$. The value d is defined as $2\hat{}d_{\mathrm{EXP}}$ and so in practice the d_{EXP} value is used directly without the need for realising the value of d. A three-input comparator component finds the maximum of the three computed values, and the index of this value is fed back to the state machine. The comparison takes just a single clock cycle.

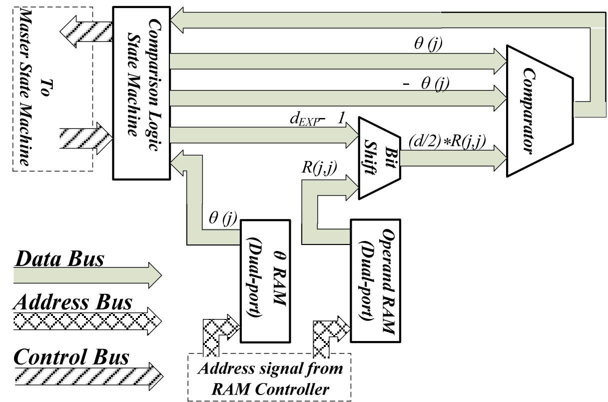

Figure 3. Comparator component block diagram

3.4. UPDATE LOGIC SUBMODULES

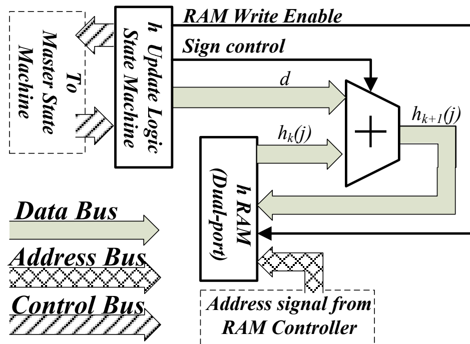

Figure 4. h-update logic

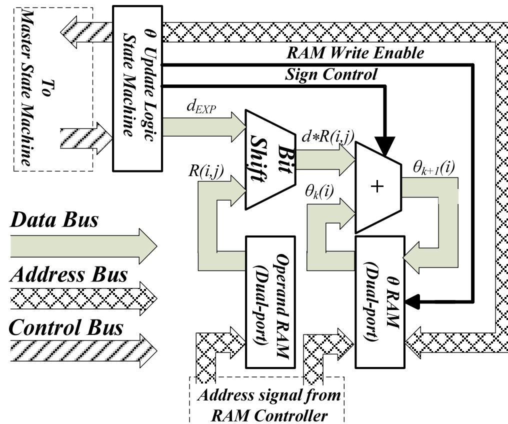

Figure 5. θ-update logic

Block diagrams of the **h**-update and **θ**-update submodules are shown in Figure 4 and Figure 5 respectively. The **θ**-update and **h**-update submodules are similar in that they are both responsible for initialising **θ** and **h** RAM respectively at the start of the DCD process, and also for updating their contents during the DCD process itself. Initialising the memories is performed concurrently. The **θ** RAM is filled with scaled β data from the operand RAM, whilst the **h** RAM is filled with zeros clearing the result of any previous computation. The update process is rather different for the two sub-modules. The h update process involves reading $h(j)$ from the read section of one port of **h** RAM, adding or subtracting the step size d, and writing the new $h(j)$ value back via the write section of the same port. Each of these operations takes one cycle, giving a three-cycle update. This update executes concurrently with the update of the **θ** vector, which takes many more cycles to complete.

The update of **θ** RAM is more involved, and requires cycling through all N elements of the **θ** vector. Each element is read from one port of **θ** RAM, updated according to the equation $\theta(k)=\theta(k) \mp dR(j,k)$ (where k runs from 0 to N–1), and written back through the second memory port. Similarly to updating **h** RAM, each of these operations takes a single cycle but they are pipelined to give an effective

element update time of one cycle (10ns in our case) with a latency of three cycles. The whole vector update time is N+3 cycles.

4. Results

Our design has been implemented on a XC2VP30 FPGA. To date, upwards of 10,000 equations have been solved and no deviation from our reference solutions have been detected. The FPGA resources used by our design are shown in Table 1. The overall time taken for the DCD algorithm to complete will depend on the demanded accuracy and the particular set of equations being solved, but it will be upper bounded by the It_{MAX} value which limits the number of successful iterations. Each successful iteration takes N+3 clock cycles. The logic requirement of the DCD core remains relatively constant over a wide range of N whilst the RAM requirement scales directly with the size of operand data.

Element	Availability	Total Usage	Interface Usage	DCD Usage
Slices	13696	855 (6.24%)	205 (1.50%)	650 (4.75%)
Slice Flip-Flops	27392	484 (1.77%)	197 (0.72%)	287 (1.05%)
Slice LUTs	27392	1404 (5.13%)	232 (0.85%)	1172 (4.28%)
IO Blocks	556	10 (1.80%)	10 (1.80%)	0 (0.00%)
Block RAMs	136	18 (13.24%)	8 (5.88%)	10 (7.35%)
Global Clocks	16	2 (12.50%)	2 (12.50%)	0 (0.00%)
DCMs	8	1 (12.50%)	1 (12.50%)	0 (0.00%)

Table 1. FPGA Usage Figures

5. Conclusions and Future Work

The architecture presented gives a starting point for using DCD to solve linear systems of equations using FPGAs. Our DCD core requires 650 logic slices on a Xilinx Virtex-II Pro FPGA. This is a particularly low value as no multiplier or division components are used, and exemplifies the suitability of the DCD algorithm for hardware implementation. The number of clock cycles required to perform a successful DCD update is approximately the same as the size of equations being solved. The number of updates required to solve a system of equations is variable, but can be capped by the value It_{MAX}. Our implementation has shown no computational differences against a Matlab-based reference DCD implementation offering the same bit precision.

There are many ways to optimize this design, such as performing the vector $\boldsymbol{\theta}$ updates in parallel. At present, the size of equations that can be handled is limited by on-chip memory. In future, we may wish to solve very large systems of equations which will call for interfacing to off-chip memory, and upgrading the host-to-FPGA link to use a high speed IO port.

References.

[1] S. S. Haykin, “Adaptive Filter Theory”, Fourth Edition, Prentice Hall, New Jersey, 2002.

[2] Y. V. Zakharov and T. C. Tozer, “Multiplication-free iterative algorithm for LS problem”, IEE Electronics Letters, Volume 40, Issue 9, 29th April 2004, pp. 567-569.

[3] Y. V. Zakharov, B. Weaver, T. C. Tozer, “Novel Signal Processing Technique for Real-time Solution of the Least Squares Problem”, 2nd International Workshop on Signal Processing for Wireless Communications, June 2004, pp. 155-159.

[4] A. R. Omondi, “Computer Arithmetic Systems”, First Edition, Prentice Hall, New Jersey, 1994.

[5] Y. Zakharov, F. Albu, “Coordinate Descent Iterations in Fast Affine Projection Algorithm”, IEEE Signal Processing Letters, Volume 12, Number 5, May 2005, pp. 353-356.

[6] “Xilinx XUP Virtex II Pro Development System”. See http://www.xilinx.com/univ/xupv2p.html.

[7] Xilinx Datasheet DS083 (v4.5), “Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet”, October 2005. See http://direct.xilinx.com/bvdocs/publications/ds083.pdf.