FINITE DIFFERENCE TIME DOMAIN: A CASE STUDY USING FPGAs

Wang Chen, Miriam Leeser
Department of Electrical and Computer Engineering
Northeastern University

This chapter presents a reconfigurable hardware accelerator that implements the FDTD method. We first present background, including applications of the FDTD method. We then provide analysis and design details of the FPGA accelerator for FDTD.

32.1 THE FDTD METHOD

Modeling electromagnetic behavior has become a requirement for key technologies such as cellular phones, mobile computing, lasers, and photonic circuits. The finite-difference time-domain (FDTD) method, which provides a direct, time domain solution to Maxwell's equations in differential form with relatively good accuracy and flexibility, has become a powerful method for solving a wide variety of electromagnetic problems [1–3]. The main drawback to FDTD is its high computational complexity.

32.1.1 Background

The discovery of Maxwell's equations was one of the outstanding achievements of nineteenth-century science. The equations give a unified and complete theory for understanding electromagnetic (EM) wave phenomena. Solving Maxwell's equations is an important method for investigating the propagation, radiation, and scattering of EM waves.

The FDTD method, first introduced by Yee in 1966 [4], is a way to solve Maxwell's equations. The differential form of these equations and constitutive relations can be written as follows:

$$\nabla \times \vec{E} = -\frac{\partial \vec{B}}{\partial t} - \sigma_m \vec{H} - \vec{M} \tag{32.1}$$

$$\nabla \times \vec{H} = \frac{\partial \vec{D}}{\partial t} + \sigma_e \vec{E} + \vec{J} \tag{32.2}$$

$$\nabla \cdot \vec{D} = \rho_e; \quad \nabla \cdot \vec{B} = \rho_m \tag{32.3}$$

$$\vec{D} = \epsilon \vec{E}; \qquad \vec{B} = \mu \vec{H} \tag{32.4}$$

In equations 32.1 through 32.4, the following symbols are used:

$\vec{E}$: electric field	$\vec{H}$: magnetic field
$\vec{D}$: electric flux density	$\vec{B}$: magnetic flux density
$\vec{J}$: electric current density	$\vec{M}$: equivalent magnetic current density
ϵ: electrical permittivity	μ: magnetic permeability
σ_e: electric conductivity	σ_m: equivalent magnetic conductivity

First, the FDTD method replaces $\vec{D}$ and $\vec{B}$ in equations 32.1 and 32.2 with $\vec{E}$ and $\vec{H}$ according to the constitutive relations in equation 32.4, which yields Maxwell's curl equation.

$$\mu \frac{\partial \vec{H}}{\partial t} = -\nabla \times \vec{E} - \sigma_m \vec{H} - \vec{M}; \quad \epsilon \frac{\partial \vec{E}}{\partial t} = \nabla \times \vec{H} - \sigma_e \vec{E} - \vec{J} \tag{32.5}$$

All of the curl operators are then written in differential form and replaced by partial derivative operators, as shown in equation 32.6, with the $\vec{E}$ and $\vec{H}$ vectors separated into three vectors in three dimensions (i.e., $\vec{E}$ is separated into E_x, E_y, E_z, and $\vec{H}$ is separated into H_x, H_y, H_z):

$$curl\vec{F} = \nabla \times \vec{F} = \hat{x}(\frac{\partial F_z}{\partial y} - \frac{\partial F_y}{\partial z}) + \hat{y}(\frac{\partial F_x}{\partial z} - \frac{\partial F_z}{\partial x}) + \hat{z}(\frac{\partial F_y}{\partial x} - \frac{\partial F_x}{\partial y}) \tag{32.6}$$

We then can rewrite Maxwell's curl equations into six equations in differential form in rectangular coordinates.

$$\mu \frac{\partial H_x}{\partial t} = \frac{\partial E_y}{\partial z} - \frac{\partial E_z}{\partial y} - \sigma_m H_x - M_x; \qquad \epsilon \frac{\partial E_x}{\partial t} = \frac{\partial H_z}{\partial y} - \frac{\partial H_y}{\partial z} - \sigma_e E_x - J_x \tag{32.7}$$

$$\mu \frac{\partial H_y}{\partial t} = \frac{\partial E_z}{\partial x} - \frac{\partial E_x}{\partial z} - \sigma_m H_y - M_y; \qquad \epsilon \frac{\partial E_y}{\partial t} = \frac{\partial H_x}{\partial z} - \frac{\partial H_z}{\partial x} - \sigma_e E_y - J_y \tag{32.8}$$

$$\mu \frac{\partial H_z}{\partial t} = \frac{\partial E_x}{\partial y} - \frac{\partial E_y}{\partial x} - \sigma_m H_z - M_z; \qquad \epsilon \frac{\partial E_z}{\partial t} = \frac{\partial H_y}{\partial x} - \frac{\partial H_x}{\partial y} - \sigma_e E_z - J_z \tag{32.9}$$

Second, in preparation for "discretizing" the model in the next step, the model size, unit size, and unit timestep must be determined. The FDTD method establishes a model space, which is the physical region where Maxwell's equations are solved or the simulation is performed. The model space is then discretized to a number of cells, and the time duration, t, is discretized to a number of timesteps. The unit cell size should be small enough to ensure the accuracy of the result, but large enough to minimize the number of cells in order to save computation resources.

Although half of the EM wavelength is an upper bound of the cell size by the Nyquist sampling theorem, the cell size is often set to less than one-tenth of the EM wavelength for better results [1]. The model size depends on the number of cells in the model space, which is usually inversely proportional to the size of the unit cell. The unit timestep is calculated by following the Courant condition, which states that it must be less than the time the EM wave spends traveling to the adjacent unit cell. For a ground-penetrating radar example, assuming a central frequency of 1.25 GHz, the central wave length is 0.24 m. We set the unit cell size to 0.012 m, which is one-twentieth of the central wave length, for good simulation quality. The timestep can be set to 0.02 ns, which meets the Courant condition.

Every cell in the model space has its associated electric and magnetic fields. The material type of each cell is specified by its permittivity ϵ, permeability μ, and conductivity σ. The three-dimensional grid shown in Figure 32.1, the "Yee cell" [4], is helpful for understanding the discretized EM model space. The Yee cell is a small cube that can be treated as a single cell picked from the discretized model space; Δx, Δy, and Δz are the three dimensions of this cube. We use (i, j, k) to denote the point whose real coordinate is $(i\Delta x, j\Delta y, k\Delta z)$ in the model space. Instead of placing the E and H components in the center of each cell, the E and H field components are interlaced so that every E component is surrounded by four circulating H components and every H component is surrounded by four circulating E components.

Maxwell's equations in rectangular coordinates—equations 32.7 through 32.9—can be clearly illustrated by Yee's cell. For example, the H_x component located at point $(i, j+\frac{1}{2}, k+\frac{1}{2})$ is surrounded by four circulating E components, two E_y components, and two E_z components, matching equation 32.7, which states that the H_x component increases directly in response to a curl of E components in the x direction. Similarly, the E_x component increases directly in response to the curl of the H components, as shown in Figure 32.2, also matching equation 32.7. We represent an electric component E_z at the discretized three-dimensional coordinate $(i\Delta x, j\Delta y, (k+\frac{1}{2})\Delta z)$ as $E_z|_{i,j,k+\frac{1}{2}}$, and when the

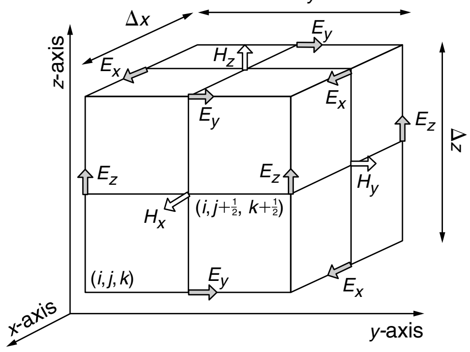

FIGURE 32.1 ■ The geometrical representation of the three-dimensional Yee cell.

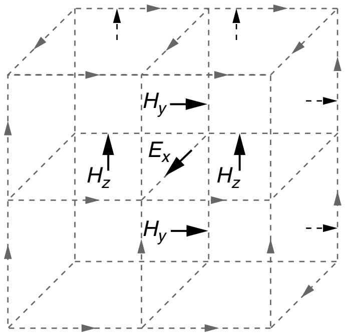

FIGURE 32.2 ■ Example of electric and magnetic components on a 4-cell grid.

current time is in the discretized N^{th} timestep, we denote the same component as $E_z|^N_{i,j,k+\frac{1}{2}}$.

Third, all of the partial derivative operators in equations 32.7 through 32.9 are replaced by their central difference approximations, as illustrated in equation 32.10. The second-order part of the Taylor series expansion is discarded to keep the algorithm simple and reduce the computational cost. Also, the variable without partial derivative can be approximated by time averaging, as shown in equation 32.11, which has a similar structure to the central difference approximation.

$$\frac{\partial f(u_o)}{\partial u} = \frac{f(u_o+\Delta u)-f(u_o-\Delta u)}{2\Delta u} + O[(\Delta u)^2] \tag{32.10}$$

$$f(u_o) = \frac{f(u_o+\Delta u)+f(u_o-\Delta u)}{2} \tag{32.11}$$

For example, equation 32.7 is changed to

$$\mu\frac{H_x(t_0+\frac{\Delta t}{2})-H_x(t_0-\frac{\Delta t}{2})}{\Delta t} = \frac{E_y(z_0+\frac{\Delta z}{2})-E_y(z_0-\frac{\Delta z}{2})}{\Delta z} - \frac{E_z(y_0+\frac{\Delta y}{2})-E_z(y_0-\frac{\Delta y}{2})}{\Delta y} - \sigma_m\frac{H_x(t_0+\frac{\Delta t}{2})+H_x(t_0-\frac{\Delta t}{2})}{2} - M_x \tag{32.12}$$

After these modifications, the FDTD method turns Maxwell's equations into a set of linear equations from which we can calculate the electric and magnetic fields in every cell in the model space. We call these equations *the electric and magnetic field updating algorithms*. Six field-updating algorithms form the basis of the FDTD method. For example, the field-updating algorithm for the H_x

component, derived from equation 32.12 or equation 32.7, is given by

$$\left(\frac{\mu}{\Delta t}+\frac{\sigma_m}{2}\right)H_x\Big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{N+\frac{1}{2}}=\left(\frac{\mu}{\Delta t}-\frac{\sigma_m}{2}\right)H_x\Big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{N-\frac{1}{2}} + \frac{1}{\Delta z}\left[E_y\Big|_{i,j+\frac{1}{2},k+1}^{N}-E_y\Big|_{i,j+\frac{1}{2},k}^{N}\right]-\frac{1}{\Delta y}\left[E_z\Big|_{i,j+1,k+\frac{1}{2}}^{N} - E_z\Big|_{i,j,k+\frac{1}{2}}^{N}\right]-M_x\Big|_{i,j+\frac{1}{2},k+\frac{1}{2}}^{N} \tag{32.13}$$

32.1.2 The FDTD Algorithm

The FDTD algorithm, whose flow diagram is shown in Figure 32.3, is based on these equations. It first establishes the model space and specifies the material properties and the excitation source. The source can be a point source, a plane wave, an electric field, or another option depending on the application. The algorithm then runs through the electric and magnetic updating algorithms on every cell in the model space and loops through every timestep. The output of the FDTD algorithm can be any electric or magnetic field data from any cell in any timestep.

The electric and magnetic fields depend on each other. As we can see from equation 32.13, the current timestep's magnetic field depends on the electric fields in the surrounding cells. Similarly, the current timestep's electric field depends on the magnetic fields in the surrounding cells. Because of this dependence between the electric and magnetic fields, we cannot update them

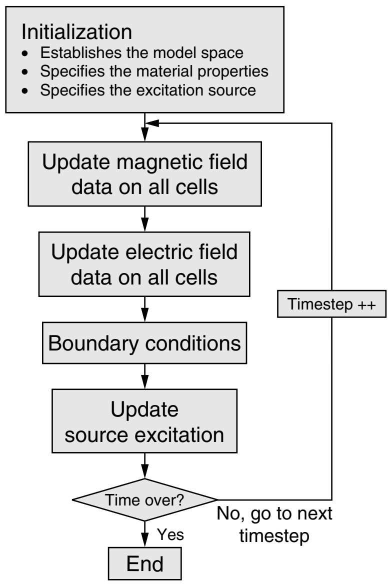
Initialization
• Establishes the model space
• Specifies the material properties
• Specifies the excitation source
Update magnetic field data on all cells
Update electric field data on all cells
Boundary conditions
Update source excitation
Time over?
Yes
No, go to next timestep
Timestep ++
End

FIGURE 32.3 ■ The flow diagram of the FDTD algorithm.

in parallel. So the FDTD algorithm updates the electric and magnetic fields in an interlaced manner, timestep by timestep, until the job finishes. First, all the magnetic fields in all cells in the model space are updated; next, all the electric fields in all cells, then the source excitation and boundary conditions, are given to the model space; finally the algorithm goes to the next timestep and starts from the magnetic fields again.

The boundary condition computation consists of special algorithms to deal with the unit cells located on the boundary of the model space. The preceding electric and magnetic updating algorithms work accurately in the interior of the model space; however, because the cells on the boundary do not have the adjacent cells needed, the algorithm does not work properly and, as a result, there are algorithm-introduced reflections on the boundary. Special techniques, called absorbing boundary conditions (ABC), are necessary to deal with boundary cells, to prevent nonphysical reflections from outgoing waves, and to simulate the extension of the model space to infinity. The development of efficient ABCs is very important for the FDTD method.

The perfect matched layers (PMLs) ABC [5] sets the outer boundary of the model space to an absorbing material medium layer, which absorbs most of the impinging wave and has low reflection for most incidence angles. The UPML (uniaxial PML) ABC [3]—a modification of PML—uses a generalized formulation on the entire FDTD model space that integrates the boundary condition and electric field updating algorithms, simplifies the FDTD algorithm, and makes a good model for hardware datapath design. Although UPML introduces extra computation and memory consumption, the quality of the uniaxial PML is especially good for dispersive media, which is useful in solving many realistic problems (e.g., the dispersive soil found in modeling ground-penetrating radar and medical studies of EM waves' effects on dispersive human tissue).

The FDTD algorithm is an accurate and successful modeling technique for computational electromagnetics. It is flexible, allowing the user to model a wide variety of EM materials and environments on most scales. It is also easy to understand, with its clear structure and direct time domain calculation. However, FDTD is data and computationally intense. It needs to visit all the cells in every step of the calculation, forcing a large working set. The amount of data in the FDTD model space can be very large for large model sizes, creating a heavy burden on both memory storage and access. The computation is also intense for each cell in the FDTD model space, including updating six electric and magnetic fields and the boundary conditions. This complexity makes the FDTD algorithm run slowly on a single processor—modeling an electromagnetic problem using the FDTD method can easily require several hours. Without powerful computational resources, FDTD models are too time consuming to be implemented on a single computer node. Accelerating FDTD with inexpensive and compact hardware will greatly expand its application and popularity, which is the purpose of an FPGA implementation.

The FDTD algorithm can be viewed as a cellular automata (CA) (see Section 5.2.5). A cellular automaton is a discrete model that consists of an infinite or finite grid of cells, where the state of every cell at discrete time *t* is a function of the states of a finite number of neighborhood cells at discrete time $t-1$. Every cell has the same rule for updating. The updating algorithm loops through the whole discrete model and then goes to the next discrete time $t+1$. The FDTD algorithm exactly fits the definition of a CA. First, it creates a discrete model space, discretizing both physical space and time with a uniform grid. Second, every cell in the model space follows the same rule (six uniform updating algorithms) for updating the electric and magnetic fields. Finally, the calculation loops though cells to simulate the phenomenon of the whole model space through time. A hardware implementation of the FDTD method is thus a template hardware design for most CA problems.

32.1.3 FDTD Applications

The FDTD method is an important tool for investigating the propagation, radiation, and scattering of EM waves. Before the 1990s the cost of solving Maxwell's equations directly was high and most of the related research was for military–defense purposes. For example, engineers used huge parallel supercomputing arrays to model the radar wave reflection of airplanes by solving Maxwell's equations, trying to develop an airplane with a low radar cross-section [6]. The difficult task of solving Maxwell's equations has had more economical solutions since 1990 with the development of fast computing resources applied to the FDTD method. Now FDTD has spread to many areas, including discrete scattering analysis, antenna and radar design [3], EM wave phenomena analysis on multilayer circuit boards [6], subsurface sensing and ground-penetrating radar (GPR) detection [7,8], studies of EM wave phenomena in the human body, and the study of breast cancer detection using EM waves [9,10]. We apply our FDTD solution to landmine detection using GPR, breast cancer detection, and spiral antenna modeling.

Ground-penetrating radar

The FDTD method has been used to simulate GPR applications for buried landmine detection [7,8]. A three-dimensional FDTD model, as shown in Figure 32.4, simulates the wave propagation and scatter response of three-dimensional GPR geometries with realistic dispersive soil along with air, metal, and dielectric media. The UPML ABC produces good results for this application. The three-dimensional model has been validated by experiments performed with a commercially available GPR system and realistic soil.

Breast cancer detection

Because of the large difference in electromagnetic properties between malignant tumor tissue and normal fatty breast tissue, microwave breast cancer detection has attracted much interest because it may overcome some of the shortcomings of X-ray detection. Accurate computational modeling of microwaves in human tissue with the FDTD method is promising for breast cancer detection research. Researchers built a three-dimensional model of the human breast [9,10], shown in Figure 32.5, that includes a semi-ellipsoid geometric representation of the

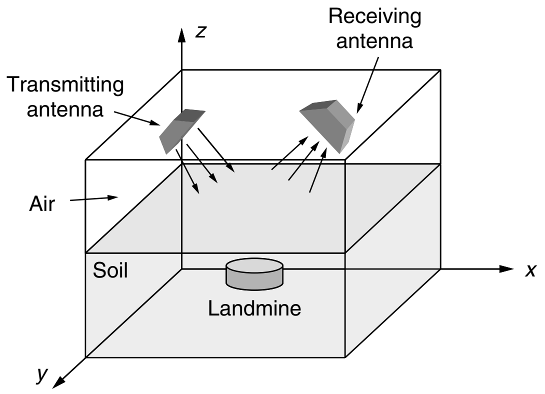

FIGURE 32.4 ▪ A three-dimensional FDTD application of landmine detection using GPR.

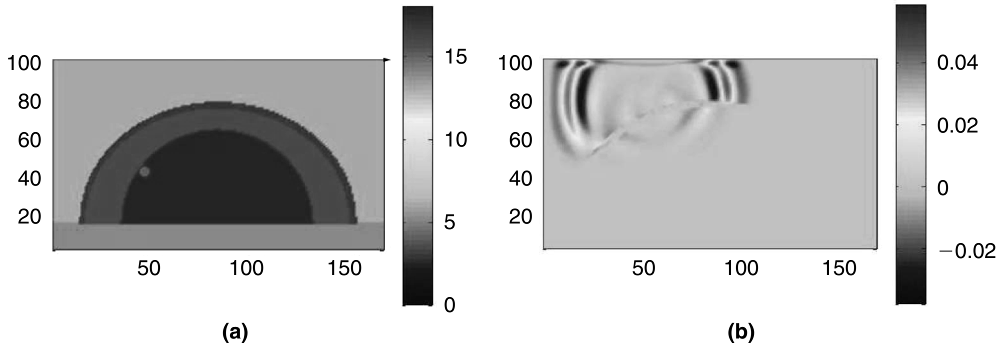

FIGURE 32.5 ▪ Three-dimensional FDTD application of microwave breast cancer detection: (a) geometry map; (b) simulated model space.

breast and a planar chest wall. The modeling is in the range of 30 MHz to 20 GHz, and the UPML ABC is implemented.

Spiral antenna model

The spiral antenna is a popular frequency-independent antenna. As shown in Figure 32.6, we use the FDTD method to simulate the radiation of the Archimedean spiral antenna as an example of its application to antenna design.

Clearly, FDTD is a powerful tool that can be used in many different applications. However, its data-intense and computationally intense properties make it run slowly on a single processor.

The reconfigurable hardware implementation of the FDTD method can greatly accelerate the running speed of the algorithm and maintain its accuracy and flexibility. For example, the breast cancer detection FDTD algorithm running on a single processor may require hours, while the hardware implementation delivers results in minutes, enabling a medical device that

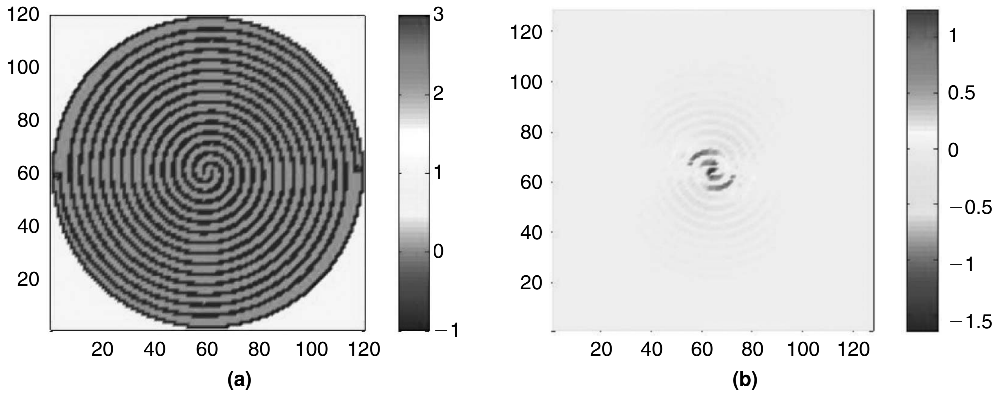

FIGURE 32.6 ■ (a) The floorplan of the spiral antenna model; (b) an FDTD-simulated two-dimensional space.

delivers an answer during the examination. With the help of faster computing technology, the FDTD method will be applied to more research areas and applications.

32.1.4 The Advantages of FDTD on an FPGA

Compared to software running on a general-purpose processor, the advantages of an FPGA implementation are evident—faster speed, smaller size, lower power consumption; the last two advantages are significant, especially compared to a large computer cluster.

Compared to an ASIC finite-difference time-domain design, the FDTD field-programmable gate array (FPGA) implementation has the advantage of flexibility while accelerating the algorithm. The FDTD method models a wide variety of electromagnetic problems that are difficult to cover with a single hardware design. With an FPGA, a designer can modify the model size, the materials, and the parameters and even introduce new updating algorithms and boundary conditions easily. While the ASIC may outperform the FPGA as to speed, size, and power, the reconfigurable property of an FPGA makes it more suitable for the FDTD algorithm.

We can achieve fast computation in an FPGA for finite-difference time-domain because FDTD has properties that make it very suitable for hardware implementation. These properties are its favorable structure for pipelining and parallelism and its constrained data ranges, which are good for fixed-point representation. They make the FDTD method especially suitable for FPGA implementation.

Parallelism and deep pipelining

The FDTD algorithm repeats the same electric and magnetic updating algorithms on every cell of the model space. These calculations are independent between each cell. As long as there are adequate hardware resources, the fields for several cells can be calculated in parallel. Also, although the electric and magnetic updating algorithms depend on each other, the hardware design can still run these calculations in parallel with a carefully designed memory interface. The parallelism between electrical and magnetic fields and the parallelism between space cells make the FDTD algorithm very suitable for parallel hardware implementation, which is a key method for hardware acceleration.

The six electric and magnetic updating algorithms can also be constructed with deep pipelining because they repeat the same calculation on each cell. *Deep pipelining*, another key method for hardware acceleration, maximizes data throughput and greatly increases overall design performance.

Most cellular automata have properties similar to the FDTD algorithm with repeated, independent computation on every cell of the model space. The CA computation can be constructed with deep pipelining, and the parallelism between discrete cells is the same as that available in any CA problem.

Fixed-point arithmetic

Floating-point representation provides high resolution and large dynamic range, but it can be costly. In hardware design, floating point uses slower arithmetic components and consumes more area. In contrast, fixed-point components have much faster speed and occupy less area. In applications where data resolution and dynamic range can be constrained, such as the FDTD algorithm, fixed-point arithmetic can provide similar precision and much faster speed than floating-point arithmetic.

The majority of the data in the FDTD algorithm is the six EM field variables and nine intermediate field variables for each cell in the model space. Since all the calculations in the FDTD method are linear, we can maintain the EM field data at a certain level of magnitude by normalizing the incoming source field magnitude. For example, if the source fields are between -1 and 1, all the EM field variables are between -1 and 1. In rare cases, we simulate the model space with a focus lens to magnify the EM field data. In this case we can estimate the EM data range and still keep the variables between -1 and 1 by normalizing the source field. Since all the EM field variables can be controlled in a fixed range, a fixed-point representation can be used for better performance with a relatively low error rate.

The uniaxial PML FDTD algorithm must be optimized for fixed-point representation. Several parameters in the algorithm have a much different order of magnitude than the EM fields. They may not be representable in fixed point directly or may result in a large error when quantized. Additional error can arise from arithmetic calculations with these parameters in fixed point. These errors can be canceled by making a few changes to the original FDTD algorithm. For example, very large and small coefficients can be multiplied together to create a medium-value coefficient to be used in the new equation. The modification has no effect on the result of the algorithm.

Careful analysis is important for fixed-point quantization to avoid errors. For normalized EM field values that range between -1 and 1, the data tends to be accurate to a relative error of 0.5 percent. The resolution of the fixed-point representation is determined by its data bit width. The longer the bit width, the higher the resolution, so the smaller the error. However, longer bit-width data uses more hardware resources. After careful study of the FDTD algorithm and representative data, we can pick a suitable bit-width with relatively small error (see also Chapter 23).

In conclusion, the FDTD algorithm is very suitable for hardware implementation. The FPGA implementation of the finite-difference time-domain method will empower many FDTD applications in medical, military, and other areas by providing fast, small, low-power, and inexpensive implementations. Many cellular automata, which share similar properties, are also suitable for FPGA hardware implementation. The FDTD hardware design we present in the next section is a good example of hardware implementations for CA.

32.2 FDTD HARDWARE DESIGN CASE STUDY

The FDTD algorithm has a clear structure for hardware design. For each cell in the model space, it reads the electric and magnetic data out of the memories, passes them through the updating algorithms, and writes the results back to the memories. The algorithm repeats this processing until it completes the model space; then it goes to the next timestep and does the same calculations again.

It is easy to separate any hardware design into datapath, memory interface, and control logic. For FDTD, the datapath implements all the electric and magnetic updating algorithms; the memory interface controls data reading, writing, and caching; and the control logic uses a finite-state machine (FSM) to control the progress of the whole design. However, because of its complexity, an efficient hardware implementation of FDTD is not straightforward. The FDTD algorithm is data intense. The electric and magnetic updating algorithms interface a lot with the input and output memories, which creates a heavy burden on the memory interface and data bandwidth. Also, the EM field dataset for the whole model space can be very large for a large model size (a $100\times100\times100$ model may require 60 MB of memory space), meaning that local FPGA memory is insufficient to contain the entire problem.

The FDTD algorithm is also computationally intense. Every EM field has its own updating algorithms and boundary conditions. A special interlaced mechanism is used between the electric and magnetic updating algorithms, making them depend on each other. Many problems arise when considering the pipelining and parallelism of the datapaths. The FDTD algorithm is complex enough to reach the resource limits of most advanced FPGAs available on the market. Consideration of fixed-point quantization and resource performance trade-offs is very important for efficient hardware design.

One of the main purposes of a hardware implementation is to achieve better performance. To implement the FDTD algorithm on an FPGA efficiently, we need to consider the following:

- Determining the right precision for fixed-point representation (Section 32.2.2)
- Determining the memory hierarchy and designing the memory interface and cache module (see Memory hierarchy and memory interface subsection of Section 32.2.3)
- Determining the pipelining and parallelism by considering the trade-off between resources and performance (see Pipelining and parallelism subsection of Section 32.2.3)

It is important to analyze the data structures, algorithm structure, hardware architecture, and resource limits before design of hardware implementation. This section introduces a target reconfigurable platform, the WildStar-II Pro FPGA board, and lists its detailed specifications. Then we choose the suitable fixed-point representation by analyzing the quantization error of a fixed-point FDTD algorithm and the hardware resource limits. Then we go through the problems in the FDTD hardware implementation and provide detailed solutions and analyses. By carefully considering the trade-offs between hardware resources and performance, we can design the FDTD accelerator with the memory interface, pipelining, and parallelism optimal to the current FPGA computing board.

32.2.1 The WildStar-II Pro FPGA Computing Board

The FPGA board used here is a WildStar-II Pro/PCI reconfigurable FPGA computing board from Annapolis Micro Systems [12]. Its main features are summarized in Table 32.1; a block diagram of this board is shown in Figure 32.7. There are two Xilinx Virtex-II Pro FPGAs, each with 328 embedded 18×18 signed multipliers and 328×18-Kb BlockRAMs.

The embedded multipliers are much faster than a multiplier component implemented with reconfigurable logic, so it is best to use them if possible. The BlockRAMs are the fastest memory the designer can use in an FPGA design, operating as fast as 200+ MHz on the Virtex-II Pro chip. Critical data interchange and interfacing can be programmed using the BlockRAMs. A pair consisting of an embedded multiplier and a BlockRAM shares the same data and address buses in the Xilinx Virtex-II architecture, so once the embedded multiplier is used, we cannot use its

TABLE 32.1 ▪ The main features of the WildStar-II Pro FPGA board

FPGA chips	Two Xilinx Virtex-II Pro XC2V70 FPGAs (33,088 slices, 328 embedded multipliers, and 5904 Kb BlockRAM)
Memory ports	Twelve DDRII SRAM ports totaling 54 MBytes (6×4.5 MBytes for each FPGA chip)
Memory bandwidth	Eleven GB/s memory bandwidth (6×72 bits for each FPGA chip)
PCI interface	133 MHz/64-bit PCI-X up to 1.03 GB/s

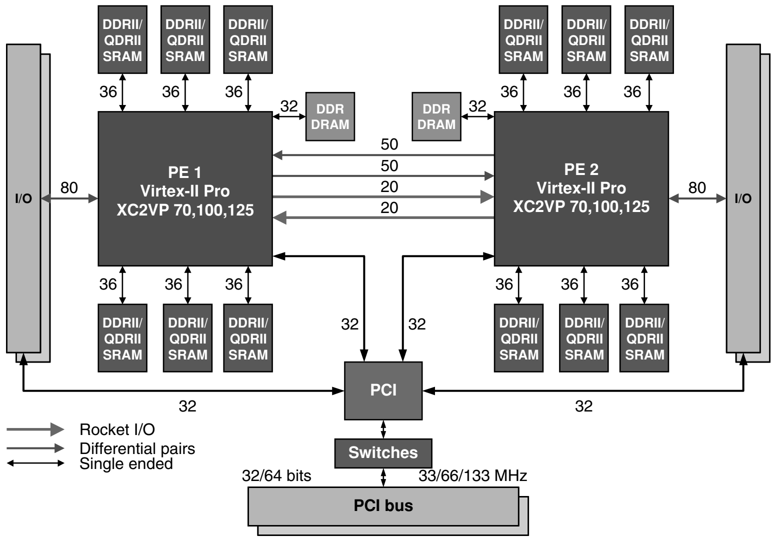

FIGURE 32.7 ■ A block diagram of the WildStar-II Pro FPGA board.

corresponding BlockRAM, and vice versa. Thus, the sum of the total number of embedded multipliers and BlockRAMs used must be less than 328.

Each FPGA is connected to six independent onboard memories, which are 1-M$\times$36-bit DDRII SRAM that have 72-bit data bandwidth and speeds up to 200 MHz. The size of each SRAM is 36 Mbits, or 4.5 MBytes, so the total SRAM attached to each FPGA is 27 MBytes. The WildStar-II Pro board is connected to the desktop computer via a PCI-X interface, with a DMA data transfer rate up to 1 GB/s between the host PC and the FPGA.

The WildStar-II Pro is a typical commercial off-the-shelf (COTS) FPGA computing board, which is widely available and easy to set up. These boards normally contain one or two FPGA chips. Each FPGA chip may be connected to several onboard memories consisting of SRAM or DRAM. The computing boards are often PCI boards for a desktop computer or PCMCIA cards for a laptop. Data and control signals can be transferred between the FPGA computing board and the host PC via either standard PCI transfer or fast DMA transfer. The FDTD hardware design is based on the WildStar-II Pro board but can be easily modified for other COTS FPGA boards.

32.2.2 Data Analysis and Fixed-point Quantization

Because of its limited data range and favorable algorithm properties, the FDTD method is suitable for fixed-point arithmetic (see Section 32.1.4). To use fixed-point representation with the algorithm, we need to first decide its representation and the right data precision.

For simplicity, we use a 2's complement fixed-point representation that has a fixed number of digits before and after the binary point. Because the EM

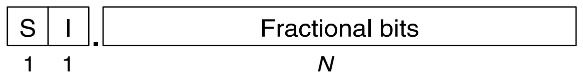

FIGURE 32.8 ■ The data structure of the fixed-point representation.

field data in the FDTD algorithm fits in the range −1 to 1, and the results of the intermediate calculations (i.e., add, subtract, and multiply) fit in the range −2 to 2, we set the fixed-point data structure as one sign bit S, one integer bit I before the binary point, and N fractional bits F_i after the binary point, as shown in Figure 32.8. The fixed-point data value is $V = -S \cdot 2 + I + \frac{1}{2^N}\sum_{i=0}^{N-1} 2^i F_i$. The data range given by this representation is between −2.0 and 1.999.

The data precision depends on the smallest absolute value that can be represented. Because the binary point position is fixed, the smallest absolute value is 2^{-N}, which depends solely on the bit width N of the fractional part. To determine the right value for N, we need to consider the trade-off between quantization error and resource costs. To avoid quantization error, which is the difference between the fixed-point and corresponding floating-point data, a longer data bit width is preferable. However, longer data bit widths require larger and slower arithmetic components and put more burden on memory bandwidth and data storage. The problem is how to pick the optimal data bit width such that the fixed-point FDTD algorithm generates acceptable quantization error and consumes a reasonable amount of hardware resources.

To determine this, we wrote the FDTD algorithm in C code both in double-precision floating-point and fixed-point arithmetic and compared the results. Fixed-point representation is simulated by long integers in C, which have a 32-bit maximum bit width. We used two long integer variables to represent one fixed-point datum up to 64 bits. Based on this representation, we created add, subtract, and multiply components for each fixed-point bit width. The C code simulates the fixed-point arithmetic and produces results that are exactly the same as the hardware output. Thus, this C code also can be used for hardware results verification.

By comparing floating-point and the corresponding fixed-point data results for the same model space, we can calculate the relative error, defined in equation 32.14, over the time period that the algorithm runs.

$$\textit{Relative error} = \frac{|\textit{floating-point data} - \textit{fixed-point data}|}{|\textit{floating-point data}|} \tag{32.14}$$

We studied the following six experimental FDTD models to investigate quantization errors:

- The two-dimensional and three-dimensional soil media–based GPR landmine detection models
- The two-dimensional and three-dimensional human tissue media–based tumor detection models
- The two-dimensional and three-dimensional spiral antenna models

TABLE 32.2 ■ Detailed specifications of the experimental FDTD models

	2D landmine detection	3D landmine detection	2D breast detection	3D breast detection	2D spiral antenna	3D spiral antenna
Size	150×100	50×50×50	240×140	80×60×40	120×120	120×120×25
Time duration	2000	2000	2000	2000	2000	2000
Source	Plane wave		Point source		Point source	
Media	Soil, air, dielectric		Human tissue, dielectric		Metal, air, dielectric	

TABLE 32.3 ■ Relative error between fixed-point and floating-point representation

		Timestep (%)					Average
Bit width	Field	400	600	1000	1400	1600	across timestep (%)
29	E_x	9.187	3.503	0.280	0.182	0.558	2.742
	H_y	12.440	0.124	1.431	0.244	0.264	2.901
	H_z	2.706	1.925	0.472	0.200	0.235	1.108
31	E_x	3.861	0.941	0.058	0.032	0.110	1.001
	H_y	3.681	0.025	0.295	0.042	0.001	0.809
	H_z	1.905	0.461	0.105	0.039	0.046	0.511
33	E_x	2.155	0.209	0.016	0.010	0.031	0.484
	H_y	2.101	0.007	0.077	0.012	0.014	0.442
	H_z	1.479	0.120	0.029	0.010	0.013	0.330
35	E_x	1.729	0.063	0.004	0.002	0.008	0.361
	H_y	1.420	0.002	0.021	0.003	0.004	0.290
	H_z	1.314	0.030	0.007	0.003	0.003	0.271

The specifications of these models are listed in Table 32.2. For all of them, we studied the average relative errors between the floating-point and the fixed-point results. This section analyzes the GPR model results. The other model spaces are similar.

Table 32.3 shows average relative errors for the fractional data bit-width range from 29 to 35 bits in the two-dimensional GPR landmine detection model. E_x, H_y, and H_z are electric and magnetic field data. The relative errors are plotted in Figure 32.9. Those of both electric and magnetic field data decrease as bit widths increase. However, the rate of decrease slows as the bit widths increase. Considering both the relative error and the bit-width cost, a 33-bit fractional part is a good choice for the trade-off between data precision and hardware resources. The average absolute error for this representation is on the order of 10^{-8} for magnetic field data and on the order of 10^{-6} for electric field data; the average relative error is about 0.3 to 0.5 percent. Thus, this representation satisfies the accuracy requirement that the relative error is less than 0.5 percent.

In addition to quantization error analysis, we need to consider the resource limits of the real hardware device in determining the fixed-point data bit width. The FDTD model space will be stored in the onboard SRAMs on the WildStar-II

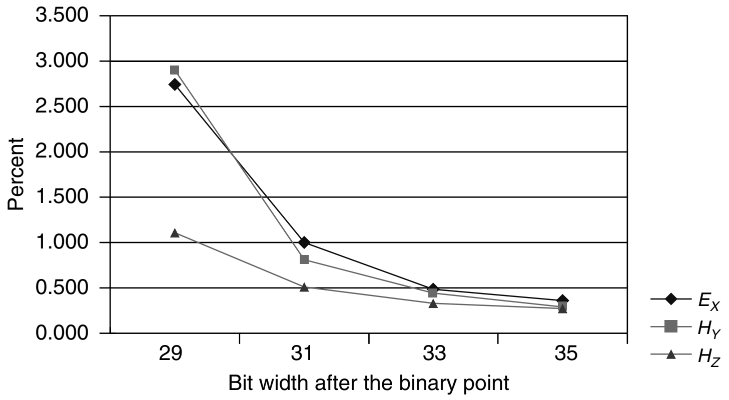

FIGURE 32.9 ■ The relative error between fixed-point and floating-point arithmetic for different bit widths.

Pro FPGA board. The SRAM memory chip we used has size 512K × 36 bit. The data is stored in the memory in units of 36 bits. Any data more than 36 bits wide will take two memory units. To keep the memory interface working efficiently, we want to set the data bit width less than or equal to 36 bits.

The embedded multiplier provided on the Xilinx Virtex II-Pro FPGA chip, an 18×18-bit 2's complement signed multiplier, is much faster than the multiplier component implemented by normal reconfigurable logic. Four embedded multipliers can form a 35×35-bit signed multiplier. However, to construct a 36×36-bit signed multiplier, nine embedded multipliers are needed. Because the number of multipliers is limited and very useful in the FDTD algorithm, it is uneconomical to use a 36×36 multiplier or 36-bit data. A data bit width of 35 bits is more efficient for the embedded multiplier. Because the fixed-point quantization error analysis performed in the last section also recommends a data bit width of 35, we choose 35 bits of data as the fixed-point data structure based on both quantization error and resource limits.

32.2.3 Hardware Implementation

After choosing the fixed-point data representation, we then study two very important problems in the FDTD hardware implementation: memory interfacing and pipelining and parallelism.

Memory hierarchy and memory interface

Because the EM field data is proportional to the number of cells in the FDTD model space, the dataset can be very large. Every cell in the FDTD model space has 6 EM field data and 9 intermediate field data for the UPML computation, adding up to 15 field data. An FDTD model space may have millions of cells, require hundreds of megabytes of memory space, and easily exceed the limits of the memory available inside the FPGA chip. Therefore, the data must be stored in larger memories, which are normally slower than the fast on-chip memories, outside the FPGA chip.

The data stored in the slower memories needs to be transferred to the processing core in the FPGA. The processing core is composed of six electric and magnetic updating algorithms, which require very large amounts of input data. In the worst case, three electric updating algorithms require 36 input data and three magnetic ones require 18, adding up to 54 input data for each dispersive UPML FDTD cell. In other words, to make sure that the processing core works at full speed, we need to transfer 54 input data from off-chip memory to the FPGA for each cell. The data transfer puts a heavy burden on the interface between the off-chip memories and the FPGA design. To provide the necessary data at the right time and to optimize the efficiency of the memory interface, we need to determine how to organize the memory resources efficiently by considering the size, speed, and interface bandwidth of each memory resource.

There are three levels of memory hierarchy, based on the WildStar-II Pro/PCI FPGA computing board:

- The fast and wide data-width on-chip memory (BlockRAM) integrated on the FPGA chip
- The fast but limited data-width onboard memory located on the FPGA computing board
- The slow memory for the FPGA to access located in the host PC

BlockRAMs are programmable memories that are integrated inside modern FPGA chips. A Xilinx Virtex-II Pro XC2V70 FPGA contains 328 BlockRAMs, 18 Kb each, with a maximum data width of 36 bits. They can be implemented as small memory blocks or cascaded to form large memory blocks. They also can be programmed to be different depths and widths to fit the hardware design and data structures. They are fast memory units in terms of latency, with only one clock cycle delay for clock cycles up to 200 MHz.

Although BlockRAMs are fast and flexible memory resources, there is much less BlockRAM available compared to off-chip memory. So normally we do not fit the entire model space's data into BlockRAMs. Instead, they are used to build cache modules that read from and write to off-chip memories continuously and feed data to the processing core. What's more, the BlockRAMs are true dual-ported RAM units, and a group of BlockRAMs can provide a very wide data width to the processing core when aggregated together. For example, 54 BlockRAMs on the input side can provide a 54×36-bit data width every clock cycle, which allows the FDTD processing core to run at full speed. The *data width* is the number of bits that can be transferred in one clock cycle. Along with clock frequency, data width determines the data transfer speed (bandwidth) of the memory interface.

Onboard memories, which directly communicate with the FPGA chip, are relatively slower than BlockRAMs in terms of latency, but they are usually much larger in size, varying from megabytes to hundreds of megabytes. The interface between the memory chips and the FPGA chips follows the read/write cycles of the specific memory chips, which are normally single-ported data access with limited data transfer width. Because of the heavy data access required by the FDTD algorithm, the onboard memory bandwidth is very important to the performance of the FDTD design.

As discussed before, the six electric and magnetic updating algorithms need 54 input data for each FDTD cell, which is around 54×36-bit×100 MHz = 194 Gb/s—far beyond the onboard memory bandwidth of typical FPGA boards. The input data of a single cell have to be transferred to the updating algorithms in several clock cycles, while the updating algorithms can calculate results with a throughput of one cell per clock cycle. So, the onboard memory data transfer bandwidth is the bottleneck of the FDTD design. Memory bandwidth is an important specification in choosing the FPGA computing board for a finite-difference time-domain implementation. To solve this bottleneck, we introduce the managed-cache module that is explained in the next subsection.

The memories in the host PC can be accessed by the FPGA via the PCI or other interfaces. These interfaces are normally slower than the two memory interfaces we have discussed, so we treat the memories in the host PC as the slowest memory, no matter what the actual speed. This memory can be used for data initialization at design startup and data retrieval at the end. At the start of processing, the model space data are loaded from the host PC to the onboard memory and loaded back to memory in the PC at the end of the design. If the onboard memory is not big enough to hold the whole model space, the memory in the host PC will be the primary memory and the data need to be transferred to and from the onboard memory throughout the entire calculation, slowing down the whole design. The size of onboard memories is thus another critical specification in choosing an FPGA computing board.

The memory hierarchy and memory interface structure used in this design is shown in Figure 32.10. We use one FPGA and six onboard memories on the WildStar-II FPGA board. The FDTD field data stored in the onboard memories are sent to the electric and magnetic field–processing cores for calculation via

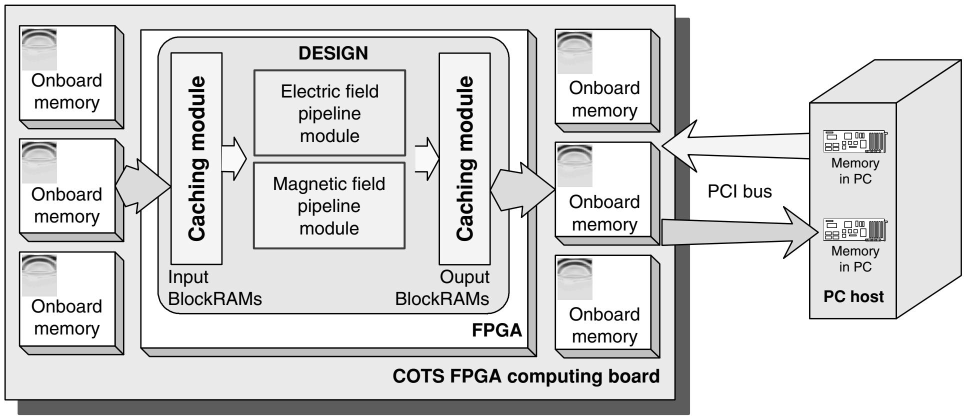

FIGURE 32.10 ■ A structural diagram of the memory interface.

the caching modules built using the BlockRAMs on the FPGA chip. The 3-level memory hierarchy formed from the host PC, the onboard memories, and the BlockRAM caching modules ensure that the electric and magnetic field updating algorithms work at optimal speed.

As shown in Figure 32.10, the BlockRAM caching modules are split into two parts: input and output. The six onboard memories, which are used to store EM field data, are split into two parts also. The entire FDTD model space of the previous timestep is stored in the input onboard memories, and the calculation results, which comprises the data in the current timestep, will be stored in the output onboard memories. In the next timestep, the role of the onboard memories is swapped. The original output onboard memories, which store the current timestep's data, will be connected to the input caching module and the original input onboard memories will be connected to the output module to store the next timestep's result.

The separation of the input and output onboard memories eliminates the need for simultaneous read/write access to the same memory. Because the onboard memories are single ported, shifting between reading and writing to the same memory will create overhead and greatly reduce the speed of the design. By separating input and output memory, we can read from and write to the onboard memories at the same clock cycle, and continue reading and writing a group of data on every clock cycle. So, although the separation of the memory interface does not change the memory bandwidth, the data-transfer rate of the memory interface is increased. Also, the separation makes the structure of the memory interface clearer and the swapping mechanism avoids the extra effort of transferring data from output memories to input memories at the end of every timestep. This swapping of input and output memories is a common hardware design technique to increase throughput.

Managed-cache module

As introduced in the previous section, onboard memory data bandwidth is limited on the FPGA computing board, so the EM field data cannot be transferred to the FPGA fast enough to allow the processing core to run at full speed. To solve this memory transfer bottleneck, we need to introduce the managed-cache module, which is an important part of the memory interface design.

Memory transfer bottleneck Although the FDTD processing core requires a large amount of input data, the input data for each cell are the EM field data in their nearest-neighbor cells. For two cells located near each other in the FDTD model space, some of the nearest-neighbor cells are the same. The cache module between the onboard memories and the hardware processing cores is designed to avoid reading the same data multiple times from onboard memories.

All of the input data for each cell are from their near neighbors, which means the data are located in a small cubic window around the current cell. If the managed-cache module is designed to be larger than this cubic window, when we calculate the fields of the next cell, the processing core can get all the necessary input data from the cache module. Among the input data, only a little is new, so we only need to fetch the new data from the onboard memories every clock cycle, which greatly reduces the data-transfer burden. At the same time, some of the old data becomes obsolete. In the managed-cache module, we can replace the obsolete data with the new data fetched from onboard memory.

Ideally, we keep the processing core running at full speed so that it calculates one cell's EM data per clock cycle. The managed-cache module needs to be designed to provide all the necessary input data for the processing core, while fetching only one new cell's data from onboard memory every clock cycle. Since every UPML FDTD cell has 15 field data and the processing core needs up to 54 field data inputs, an ideal managed-cache module will fetch 15 field data from onboard memory every clock cycle and provide a data width of 54 field data to the processing core, solving the memory bandwidth bottleneck problem by reducing the number of fetches to 15 every clock cycle, which is 15×36-bit$\times100$ MHz = 54 Gb/s. This rate can be supported by the WildStar-II Pro FPGA computing board. We explain how to realize this ideal cache module in the next two subsections.

Dataflow and processing core optimization To simplify the explanation of how to optimize the dataflow and how to optimize the processing core, we start from a two-dimensional FDTD algorithm, which can be directly reduced from the three-dimensional FDTD algorithm by considering only one plane in the three-dimensional model. The two-dimensional algorithm updates three EM field data instead of six, handling much less data transfer and calculation, but it keeps the same algorithm structure and datapath. For a two-dimensional model plane of size $N\times N$, we assume that each N cell row is a basic processing unit. Calculating one row of data means updating all EM field data for this row.

The cache modules separate the whole dataflow of the FDTD design into three processes: (1) `READ` from the input onboard memory and store to the input cache module; (2) read from the input cache module, `CALCULATE`, and write the result to the output cache module; (3) read from the output cache module and `WRITE` to the output onboard memory. These three processes can be run in parallel since the cache module can be read from and written to at the same time (i.e., because the cache modules are built from dual-ported BlockRAMs). The parallelism of `READ`, `CALCULATE`, and `WRITE` means that the FDTD design can, at the same time, `READ` one row of data, `CALCULATE` the previous loaded row, and `WRITE` out the results of the row before that. We can understand this as systemwide pipelining in the dataflow. Each process is a pipeline stage. Rows of data are pushed into this 3-stage pipeline, one at a time. Compared to running the three processes serially, this optimized dataflow structure increases the throughput by a factor of 3.

For a two-dimensional plane of size $N\times N$, a simple 2-row cache module (size $2\times N$) realizes the `READ`/`CALCULATE`/`WRITE` pipelining. As shown in Figure 32.11, the data can be `READ` from input onboard memory and stored in the second input cache row while the `CALCULATE` process works on the previously loaded data in the first input cache row. The result is stored in the first output cache

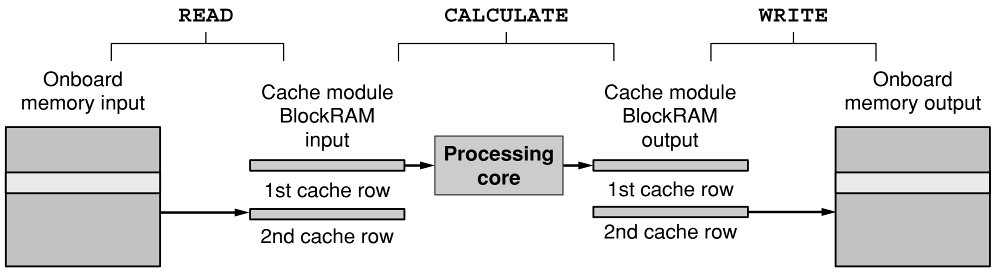

FIGURE 32.11 ■ A structural diagram of the simple 2-row cache module.

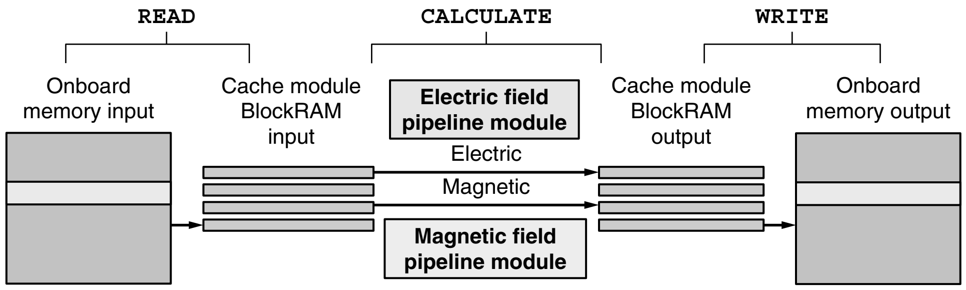
READ
CALCULATE
WRITE
Onboard memory input
Cache module BlockRAM input
Electric field pipeline module
Electric
Magnetic
Magnetic field pipeline module
Cache module BlockRAM output
Onboard memory output

FIGURE 32.12 ■ A structural diagram of the two-dimensional managed-cache module.

row while the previous row's result is read from the second output cache row and WRITTEN to output onboard memory. This cache module structure can be applied to other CA designs.

Furthermore, for FDTD implementation the managed-cache module enables parallel implementation of the electric and magnetic updating algorithms in the implementation of the processing core. Because of the data dependency of the electric updating algorithm on the magnetic updating algorithm—the former needs the current result of the latter—we cannot directly update the M-field and E-field in parallel until we introduce two extra rows in the managed-cache module (see Section 32.1.2). Why two extra rows?

The electric updating algorithm needs to have newly updated magnetic data in the current cell and newly updated magnetic data in the cell below as inputs. So, the electric updating algorithm needs to wait until the magnetic updating algorithm finishes two rows of computation. As long as the cache has two extra rows to save the newly calculated magnetic data, we can run the magnetic updating algorithms two rows ahead of the electric updating algorithms and partially overlap their computation. This is illustrated in Figure 32.12.

For a two-dimensional model space of size $N\times N$, the managed-cache module stores four rows ($4\times N$) of field data. While the READ process is working on the fourth cache row, the magnetic updating algorithm can work on the data in the third row, which was just read from the memories by the last READ. At the same time, the electric updating algorithm can work on the first cache row, which is two rows after the magnetic algorithm. Finally, `WRITE` also works on the fourth row, sending out both calculation results from the electric and magnetic updating algorithms. The four rows of field data roll over in the cache modules until the entire model space is calculated. This 4-row cache module improves the total computation time by a factor of almost 2, or $(N+2)/(2N+2)$, by partially parallelizing the electric and magnetic updating implementations.

Thus, the managed-cache module optimizes the design here in two ways: (1) systemwide pipelining of the design dataflow, and (2) processing-level parallelism of the electric and magnetic updating algorithms.

Expansion to three dimensions Here we expand the two-dimensional cache module design to three dimensions. The memory interface and the cache modules are more complex in the three-dimensional FDTD hardware implementation, which handles many more data transfers and calculations. There are two possible approaches for upgrading the cache module to three dimensional. The first is a direct upgrade of the two-dimensional memory interface, as shown in Figure 32.13. Instead of a 4-row cache module, we need to build a 4-slice cache module. Here we `READ` one slice, `CALCULATE` one slice, and `WRITE` out one slice of data at each time interval. However, a $4\times100\times100$ cache module consumes more than 1200 18-Kb BlockRAMs, which is over three times all the BlockRAMs on the targeted Virtex-II Pro XC2V70. This approach is not feasible for large three-dimensional model spaces.

The second approach reduces the size of the cache module to 4×3 rows of field data by cutting the model space into slices and then into rows. As shown in Figure 32.14, the cache module reads three rows of field data at each time interval, goes through the current vertical slice until it finishes, and then goes to the next vertical slice in the model space. Instead of a 4-slice cache module, we only need to build a 4×3 row cache module. This method minimizes BlockRAM consumption; however, it sacrifices overall design speed to achieve larger model space compatibility. We `READ` three rows of data at each time interval to `CALCULATE` only one row of results. This is because we need the current row

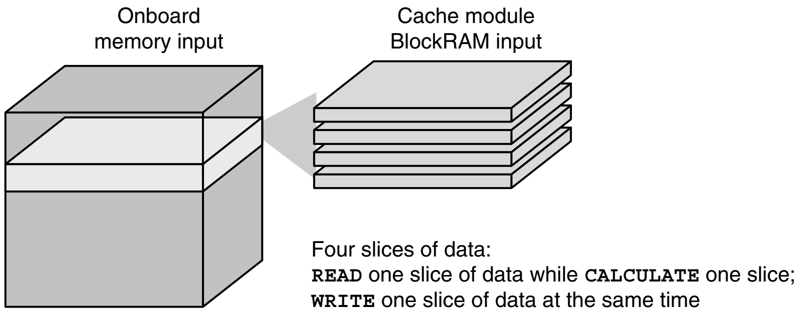
Onboard memory input
Cache module BlockRAM input
Four slices of data:
READ one slice of data while CALCULATE one slice;
WRITE one slice of data at the same time

FIGURE 32.13 ■ A structural diagram of the 4-slice caching design.

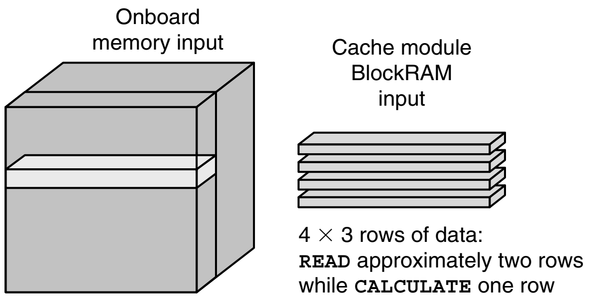
Onboard memory input
Cache module BlockRAM input
4 × 3 rows of data:
READ approximately two rows while CALCULATE one row

FIGURE 32.14 ■ A structural diagram of the 4×3 row caching module.

and adjacent two rows of data to calculate the current row's results. Because only one row of results is calculated from the field-updating pipelines, the READ process is longer than the CALCULATE and WRITE processes. At this point the other two processes need to wait for the READ process.

This waiting process slows down hardware design. Fortunately, we do not need to READ all three rows (45 data per cell) to start processing since the field-updating algorithm only needs part of the data in adjacent rows. We only need to READ approximately two rows of data (36 data per cell), CALCULATE one row, and WRITE one row at each time interval. Due to the limited number of BlockRAMs, the second approach is more practical. From the preceding analysis of the managed-cache modules, we conclude that the efficiency of the memory interface plays a key role in the performance of the complete FPGA design. The speed and manner in which the memory interface handles the input data often limits the speed of the entire design.

Pipelining and parallelism

Given an efficient memory interface and proper fixed-point data representations, the designer next needs to adjust the architecture and optimize design performance by considering pipelining and parallelism.

As discussed before, we can implement the electric and magnetic updating algorithms in parallel with the correct cache structure. We can also implement the three key processes—READ, CALCULATE, and WRITE—in parallel by separating the input and output memory interfaces and building dual-ported cache modules. In hardware design, parallelism translates to faster speed; however, it also "costs" more in hardware resources. The FDTD algorithm is large enough to reach the resource limits of the most advanced FPGAs on the market. One of the important problems in FDTD hardware design is determining the design architecture by considering the trade-offs between resources and performance. The hardware resource limit of each FPGA chip and computing board is different. The resource–performance trade-off analysis here is based on the targeted WildStar-II Pro FPGA computing board.

Pipelining The FDTD algorithm repeats the same electric and magnetic updating algorithms, which are independent of each other, on every cell of the model space. The algorithms can be implemented with complex combinational logic with long delay. Building them with deep pipelining helps reduce the clock cycle and increase the throughput of the hardware design. Because of the advantages, we pipeline all the updating algorithms. The embedded multipliers, which are the slowest components in the datapath, can also be pipelined to several stages to reduce delay. Because the lengths of the electric and magnetic updating pipelines are different, state machines are used to control the start and end of the pipelines and to synchronize them.

Parallelism Because the updating calculations on every cell in the FDTD model space are independent of each other, as long as there are adequate hardware resources, the computation of two or more FDTD cells can be implemented in parallel. However (see Section 32.2.3), the bandwidth of the memory interface is the bottleneck of the FDTD hardware design. The memory data width here is 3×72 bits, which can transfer six 35-bit field data inputs at each clock cycle. This memory bandwidth needs 6 clock cycles to prepare one cell's 36 input data when using the 4×3 row cache module. Can this memory interface handle the increased parallelism?

Running two cells in parallel actually saves memory bandwidth per cell. As shown in Figure 32.15, two adjacent FDTD cells share a portion of their nearest-neighbor cells. For each single cell, we need to read three rows of data (36 field data per cell) from the onboard memories, which is when running two cells in parallel, we only need to read four rows of data, or 24 data per cell. Because the bottleneck of the design is the memory bandwidth, the 2-cell parallelism mechanism improves the performance of the whole design. We can use the ratio between input data and result data as a metric to measure the efficiency of the memory interface. After implementing 2-cell parallelism, the input–result ratio decreases from 6:1 to 4:1.

Running two cells in parallel creates an extra burden on the cache size and the calculation pipelines, however. The cache module needs to hold 4×4 rows of data at the same time instead of 3×4 rows. Fortunately, the Virtex-II Pro XC2V70 FPGA has adequate BlockRAMs for the 4×4 row cache, but there is no space for increasing the cache beyond this, which is why we choose not to run three cells in parallel, even though this would further save memory bandwidth per cell and improve the input–result ratio.

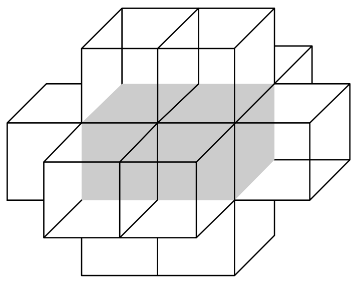

FIGURE 32.15 ■ Running two cells in parallel.

Also, the Virtex-II Pro FPGA XC2V70 does not have enough reconfigurable logic to implement all the updating pipelines in parallel. Instead, because the memory interface takes four clock cycles to transfer enough input data for one cell's calculation, the number of parallel updating pipelines can be reduced. The calculation core can run several updating algorithms serially in one updating pipeline, taking more than one clock cycle to finish the calculation for one cell. The serial calculation reduces the level of parallelism, saves reconfigurable logic, and still maintains the performance of the hardware design.

Two hardware implementations The preceding input–result ratio is calculated based on the input data needed for the uniaxial PML FDTD algorithm. This algorithm treats the whole model space as UPML cells and provides a uniform structure for both the UPML cells and the non-UPML center cells, as shown in Figure 32.16. However, the UPML FDTD algorithm requires nine extra field data for each cell in the model space, which adds overhead to the memory interface. The cells in the center of the model space that are not located in the UPML layer can be calculated by the normal FDTD algorithm, which has only six field data for each cell. Small modifications to the UPML updating pipelines can make the new updating pipelines work on both the UPML cells and non-UPML center cells.

Therefore, we can save memory bandwidth and memory space on the center cells by combining the UPML and center cell algorithms in the hardware design. The input–result ratio of a center cell is 3:1 and will be 2:1 after applying 2-cell parallelism. For the normal model space, where half the cells are center cells and the other half are UPML cells, the overall input–result ratio will decrease to approximately (4:1 + 2:1)/2 = 3:1, raising the performance of the hardware design.

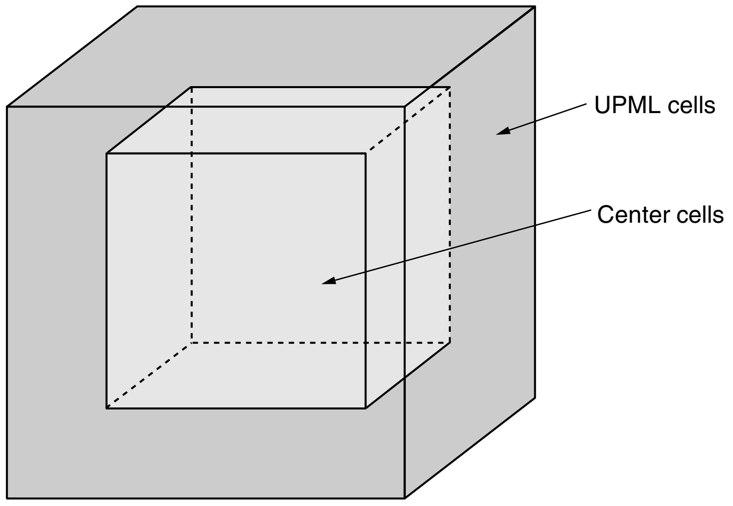

FIGURE 32.16 ■ Uniaxial PML boundary condition cells and non-uniaxial PML center cells in the model space.

We have two hardware implementations for the uniaxial PML FDTD algorithm. The first implementation treats the whole model space as UPML cells, with a simpler design structure and an input–result ratio of 4:1. The second implementation, which includes center cell and UPML cell calculations, has a more complex memory interface and better performance (the input–result ratio depends on the number of center cells and UPML cells).

The analysis of resources and performance trade-offs here is based on the WildStar-II Pro FPGA computing board. For other FPGA devices, the analysis is similar. A wider onboard memory data width, which can ease the memory bottleneck, will raise the design performance proportionally. A bigger FPGA chip, which can hold larger cache modules and more updating pipelines, will speed up the hardware design by calculating more cells in parallel.

32.2.4 Performance Results

A comparison of performance results for three-dimensional FDTD software and hardware implementations is shown in Table 32.4. The sample model is a $50\times50\times50$ three-dimensional uniaxial PML FDTD algorithm model with 500 timesteps of FDTD iteration. The fixed-point FDTD hardware design, which treats all cells as UPML boundary cells, runs at 90 MHz on the WildStar-II Pro FPGA board. The UPML FDTD FPGA implementation is 16 times faster than the floating-point Fortran software implementation running on a 3.0-GHz PC. Hardware times are measured on the board and include the time to transfer data between the FPGA board and the host PC at the start and end of computation. The hardware design speedup can increase to 25 times with the implementation that combines the center and UPML region. The Virtex-II Pro XC2V70 FPGA chip is almost fully utilized because the FDTD hardware design occupies 99 percent of the reconfigurable slices, 51 percent of the BlockRAMs, and 46 percent of the embedded multipliers. There are two Xilinx Virtex-II Pro FPGAs on a WildStar-II Pro FPGA board. Dual-FPGA parallel implementations of the FDTD algorithm are expected to double the speedup.

TABLE 32.4 ■ Three-dimensional FDTD hardware implementation performance results

	Software floating-point Fortran code on 3.0 GHz PC	**Hardware fixed-point design running at 90 MHz**	**Hardware fixed-point design running at 90 MHz**	**Hardware fixed-point design running at 90 MHz**
		All cells as center cells	All cells as UPML boundary cells	Combined center and UPML region
Runtime (sec)	49	1.59	2.985	1.89
Million nodes/sec	1.27	39.31	20.93	33.07
Speedup	1	30.9	16.5	25.9

32.3 SUMMARY

Implementing the FDTD algorithm in hardware greatly increases its computational speed. The speedup is due to three major factors: fixed-point representation, custom memory interface design, and pipelining and parallelism. FDTD is a data-intense algorithm; the bottleneck of the hardware design is its memory interface. With the limited bandwidth between the FPGA and data memories, a carefully designed custom memory interface allows for full utilization of the memory bandwidth and greatly improves performance. The FDTD algorithm is also a computationally intense algorithm; by considering the tradeoffs between resources and performance, we implement as much pipelining and parallelism as possible to speed up the design.

The FDTD algorithm is also a cellular automata, sharing a similar algorithmic structure with many other CA problems. The hardware design techniques and memory interface architecture presented in this chapter can be applied to a wide range of other CA problems to achieve speedup on an FPGA and to provide fast, small, low-power, and inexpensive implementations.

References

[1] K. S. Kunz, R. J. Luebbers. *The Finite Difference Time Domain Method for Electromagnetics*, CRC Press, 1993.

[2] A. Taflove, S. C. Hagness. *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 2nd ed., Artech House, 2000.

[3] A. Taflove. *Advances in Computational Electrodynamics: The Finite-Difference Time-Domain Method*, Artech House, 1998.

[4] K. Yee. Numerical solution of initial boundary value problems involving Maxwell's equations in isotropic media. *IEEE Transactions on Antennas and Propagation* 16, 1966.

[5] J. P. Berenger. Three-dimensional perfectly matched layer for the absorption of electromagnetic waves. *Journal of Computational Physics* 127, 1996.

[6] A. Taflove. Reinventing electromagnetics: Emerging applications for FD–TD computation. *IEEE Computational Science and Engineering* 2(4), 1995.

[7] B. Yang, C. Rappaport. Response of realistic soil for GPR applications with two-dimensional FDTD. *IEEE Transactions on Geoscience and Remote Sensing*, June 2001.

[8] P. Kosmas, Y. Wang, C. Rappaport. Three-dimensional FDTD model for GPR detection of objects buried in realistic dispersive soil. *SPIE Proceedings* 4742, April 2002.

[9] P. Kosmas, C. Rappaport. Modeling with the FDTD method for microwave breast cancer detection. *IEEE Transactions on Microwave Theory and Technology* 52(8), 2004.

[10] P. Kosmas, C. Rappaport. Use of the FDTD method for time reversal: Application to microwave breast cancer detection. *SPIE Proceedings Computational Imaginary* 5299, 2004.

[11] Xilinx, Inc. *Virtex-II Pro and Virtex-II Pro X Platform FPGAs: Complete Data Sheet*, 2004.

[12] Annapolis Micro Systems. *WildStar-II Hardware Reference Manual*, 2004.