LEAST-TIME PATH FAST MARCHING METHOD FOR SEISMIC TRAVEL TIME COMPUTING: THEORY AND IMPLEMENTATION

XIAONING ZHANG

Least-Time Path Fast Marching Method for Seismic Travel Time Computing: Theory and Implementation

by

© Xiaoning Zhang
Master of Engineering

A thesis submitted to the
School of Graduate Studies
in partial fulfillment of the
requirements for the degree of
Master of Engineering.

Department of Electrical and Computer Engineering
Memorial University of Newfoundland

May 8, 2009

St. John's Newfoundland

Contents

Abstract

The main contribution of this thesis is the development of Least-Time Path Fast Marching Method and the design of the algorithm implementation frame on a digital hardware platform. This work imports application specific digital circuit design technology into the domain of computational geophysics problem solving.

In the thesis, firstly, geophysics knowledge is reviewed and theoretical fundamental is introduced. From the theory, the Least-Time Path Fast Marching Method that computes seismic travel time is developed. In the algorithm design section, the issues on parallel algorithm design and algorithm 3D extension are discussed. Software simulations are run for verifying the algorithm, while parallel programming solution on a multiprocessor platform is introduced as well. At the end, a digital circuit implementation frame for the algorithm is proposed and a prototyping system is built on Xilinx FPGA.

This thesis is not only an implementation report of a digital design project, but also includes consideration and discussion on the future direction of reconfigurable computing applications and methodology.

Acknowledgements

Give all my greatest praises to God, without his love and encouragement, I can hardly move even one step forwards in my life and research.

Grateful to Dr. Bording who is not only the supervisor in my study, he instructed me skills, gave me vision of the research and taught me the passion of career as well. It is always my biggest pleasure to be a disciple of his. Thanks to Dr. Venkatesan, it was his recommendation leading me to Dr. Bording's group.

My appreciation also goes to the professors Dr. Cheng Li, Dr. Fang Wang and post-doc fellow Dr. Andreas Atle who gave their precious suggestions and supports to my work.

I would like to thank my group colleagues, four office mates and all other friends who have given the time, assistance and patience so generously.

Last but not least, I give thanks to my funding agencies: Husky Chair Endowment, IBM Faculty, SUR grants, ACENET, and ACOA.

List of Abbreviation

The following abbreviations are used in this thesis.

- **BRAM: Block RAM.**
- **BUFG: Global buffer.**
- **CMT: Clock management tiles.**
- **DSP: Digital System Processor.**
- **FDM: Finite Difference Method.**
- **FSM: Finite State Machine.**
- **FPGA: Field Programmable Gate Array.**
- **GPU: Graphics processing unit.**
- **GUI: Graphical User Interface.**
- **H/S: Hardware and Software co-design.**
- **HDL: Hardware Description Language.**
- **IDE: Integrated development environment.**

- IP: Intellectual Property.
- LTPFM: Least-Time Path Fast Marching method.
- LUT: Lookup tables.
- MPI: Message Passing Interface.
- NoC: Network on Chip.
- PCIe: PCI express.
- PAR: Placing and Routing.
- RICE: Radial Incidence Computing Element.
- RTL: Register transfer level.
- SRL: Shift Register LUT.
- TWT: Two way time.
- TP: Ray Tracing is known as two-point ray tracing.
- WE PSDM: Wave Equation Pre-stack Depth Migration.
- WFC: Wavefront construction method.

List of Figures

Chapter 1

Introduction (Geophysics)

1.1 Subsurface Exploration

Today, the world has intense need for oil and gas resources, and these needs are driving the fast growing research in geo-sciences and related fields.

Geo-scientists in the petroleum industry - including geologists, geophysicists, geochemists and paleontologists - study what has happened to rocks that may be buried thousands of meters below the surface, how those rocks were formed and affected by events stretching back millions of years, and how to identify traps where oil and gas have accumulated within rock formations.

On the other hand, computer engineers assist on the topics of geophysics computing and seismic data visualization by creating devices and developing new methods and algorithms. The basis of their work is the understanding of subsurface exploration theory, upon which they can bring computer technology into. This first chapter introduces the application of our algorithm and the related geophysics theory to help readers understand the meaning of our work. Moreover, from this introduction, you may get some ideas where to find the niche in petroleum industry for computer engi-

neering research.

1.1.1 Role of Geophysics in Oil and Gas Exploration

Borehole Tool Method

Finding commercially valuable accumulations of hydrocarbons is rarely simple: usually, they are to be found at depths of at least several thousand feet below the ground surface. In some area of the earth, it is possible to infer the geology at these depths from that of the rocks exposed at the surface, but it is more common for the near-surface geology to bear little or no relation to the deeper structures. Ultimately, knowledge of this deeper geology can come only from drilling, but a deep borehole is expensive and in many areas (especially offshore) the delineation of structure by closely-spaced drilling is unthinkable.

However, geophysics can assist exploration in some ways: seismic reflection method, which is a major measure extensively used in search for oil and gas, is capable of giving a resolution of at least a few tens of meters vertically and a few hundred meters horizontally. This increases tremendously the knowledge that can be gained from a few exploration boreholes. To introduce Seismic reflection method, let's review the concept of velocity model first.

1.1.2 Velocity Model of the Earth

In geophysics, the earth is usually thought as a layered structure. The geophysicist regards it as a series of layers with rather abrupt changes in velocity of seismic wave transmitting between them vertically, but only gradual changes laterally, along a layer.

In a velocity model[9], it is assumed that the seismic wave travels at a specific

velocity value inside each layer. These layers are not flat layers one above another. Actually, the long term movement of earth continents causes thrust movements and this can tilt these flat layers. The velocity values can range from less than 1,000 meters per second to as much as 8,000 meters per second or more.

The features of velocity model used in geophysics research can be illustrated as in Fig. 1.1:

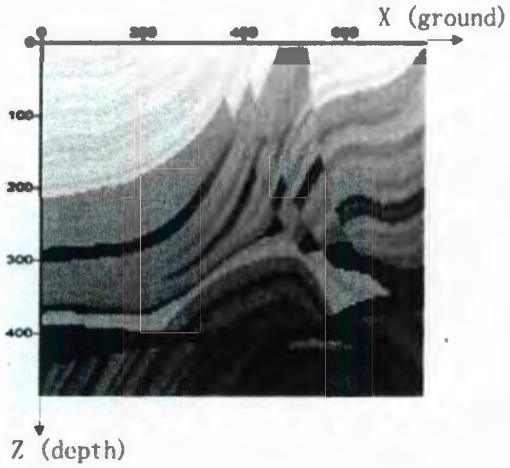


Figure 1.1: Marmousi model

1.1.3 Seismic Reflection Method

Seismic reflection method explores this layered structure by bouncing sound waves off the interfaces between these various velocity layers. This is analogous to the way in which a ship's echo sounder measures the depth of the sea from the time taken for a pulse of sound to return to the ship after reflection at the seabed: A record can be produced automatically showing the time for the wave to travel from the ship down to the seabed and back again at different points along the ship's path. By multiplying the time by half the velocity of sound in sea water, the time scale on the record can

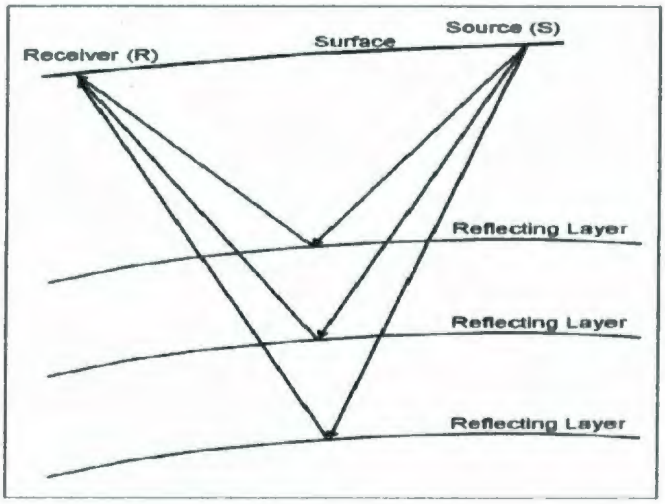


Figure 1.2: Schematic of the seismic reflection method. Figure from Mc Quillin, 1984, [1].

be marked off directly in depth. Similarly, seismic waves are reflected at interfaces where rock properties change; and the round-trip travel time, together with velocity information, gives the distance to the interface. The profile on the interface can be determined by mapping the reflection at many locations.

In Fig.1.2. physical process of seismic reflection is illustrated. The ray paths through successive layers are shown. There are commonly several layers beneath the earth's surface that contribute reflections to a single seismogram (a gathering of seismic traces. explanation will be given later). The unique advantage of seismic reflection data is that it permits mapping of many horizon or layers with each shot.

Historically, seismic reflection method is the principal method by which the petroleum industry explores for hydrocarbon-trapping structures in sedimentary basins. Its extension to deep crustal studies began in the 1960s. and since the late 1970s reflection technology has become the principal procedure for detailed studies of the deep crust.

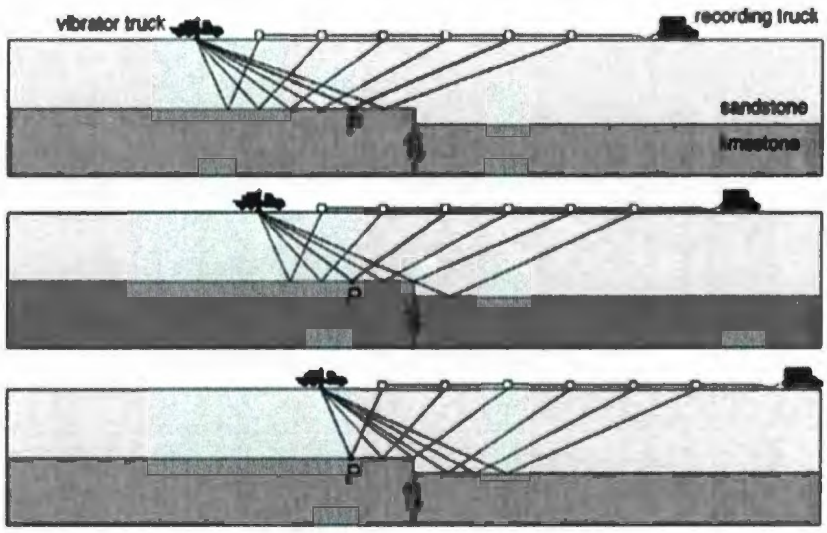


Figure 1.3: Seismic data acquisition. Figure from Mc Quillin, 1984, [1].

Seismic Data Acquisition

The very first step to apply seismic reflection method is to collect an amount of raw seismic data from field survey. In some cases seismic waves from natural earthquake source are received. By processing these data, underneath structure can be imaged. On the other hand, in the man-made experiment, as opposed to earthquake seismology, where the location and time of the source is an unknown that needs to be solved for, seismic reflection profiling uses a controlled source to generate seismic waves. On land, large truck-mounted vibrators are used as a source (in Fig.1.3), and occasionally dynamite is used. At sea, large arrays of airguns, which rapidly eject compressed air, are deployed. The reflected signals are recorded by geophones or hydrophones at sea, which are similar to ordinary microphones.

As in Fig.1.3, During a seismic survey, a cable with receivers attached to it at regular intervals is laid out along a road or towed behind a ship. The source moves along the seismic line and generates seismic waves at regular intervals such that

points in the subsurface are sampled more than once by rays impinging on that point at different angles. As a shot goes off, signals are recorded from each geophone along the cable for a certain amount of time, producing a series of seismic traces. The seismic traces for each shot (called a shot gather) are saved on computers in the recording truck.

Seismic Data Processing

With an array of seismic traces in computer, digital data processing of raw seismic data is a main task of geophysicists. Before seismic data can be sent to oil companies to be interpreted for prospecting, geophysicists apply a series of techniques to convert the field recordings into usable seismic sections. The techniques are various and complicated upon different applications: for example, before stacking the seismic traces, the static corrections need to be done to correct the data for the effect of near-surface time delays; and both before and after stack, various filtering processes are applied to improve the 'signal to noise ratio' and increase vertical resolution. Especially, before all kinds of processing, we shall finally convert seismic section into a form showing the true spatial disposition of the reflecting surface, through the process of migration. An example of typical processing sequence from raw trace data to interpretable seismic graph is shown in Fig.1.4.

Role of Computer Engineering

There is no unique processing sequence which can be applied to all seismic data; it is always necessary to balance improvement in quality against processing cost. Researchers keep devising new algorithms and methods to satisfy all kinds of processing tasks. This active research field is also where our computer engineers can play on skills to develop new computing devices and high performance solutions to facilitate

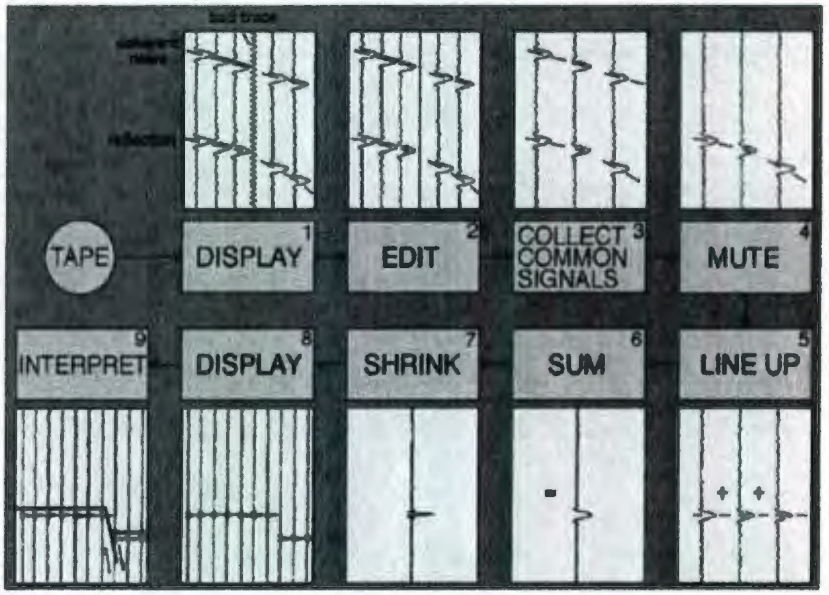


Figure 1.4: Seismic data processing

the computing intensive processing jobs.

Interpretation and Prospecting

Finally, oil companies buy the processed data; and their geophysicists and geologists do the interpretation on these seismic data. Seismic prospecting for oil depends not only on high technology but also on the interpretive powers and experience of the practicing geophysicists and geologists. To a large extent, the techniques of interpretation can only be acquired as a result of long experience translating the results of the physical experiments into geologically valid models of earth structure and then assessing the probability that any such structure might trap an accumulation of hydrocarbons.

At this stage, the oil and gas exploration procedures have been gone through roughly. The next section focuses on the topic of seismic migration.

1.2 Seismic Migration

Because readers of this thesis may lack of geophysics background, so introduction in this section obviates from excessively using jargons and abstract math.

In short, seismic migration is a wave equation based process that removes distortions from reflection records by moving events to their correct spatial locations, and by collapsing energy from diffractions back to their scattering points[10]. Migration is a tool used in seismic processing to get a picture of underground layers. It involves geometric repositioning of returned signals to show an event (layer boundary or other structure) where it is being hit by the seismic wave. Seismic Migration is a central step in the seismic data processing flow. It represents the culmination of "standard" processing, and it provides input for several relatively exotic non-standard processes.

Migration can be taken as a seismic imaging process which provides a geometrical image of the subsurface reflectors, as well as quantitative estimates of the reflection coefficients[11]. According to given seismic data and different accuracy and computing expense requirements, various migration techniques are developed from different ways of solving wave equations.

Among all these methods, Kirchhoff migration has been widely used to perform Pre-Stack Depth Migration (PSDM), while Wave Equation Pre-Stack Depth Migration (WE PSDM) has become competitive in recent years. WE migration has theoretical advantages over Kirchhoff migration with implicitly dealing with the approximations often made by the Kirchhoff methods. But WE method is more costly, and until recent years it has transformed into a economically feasible method because of the advancement in computing technology. Kirchhoff was the overwhelming historical pre-stack depth migration method of choice. Although making approximations on the imaged data, and greater imaging accuracy offered by other migration methods,

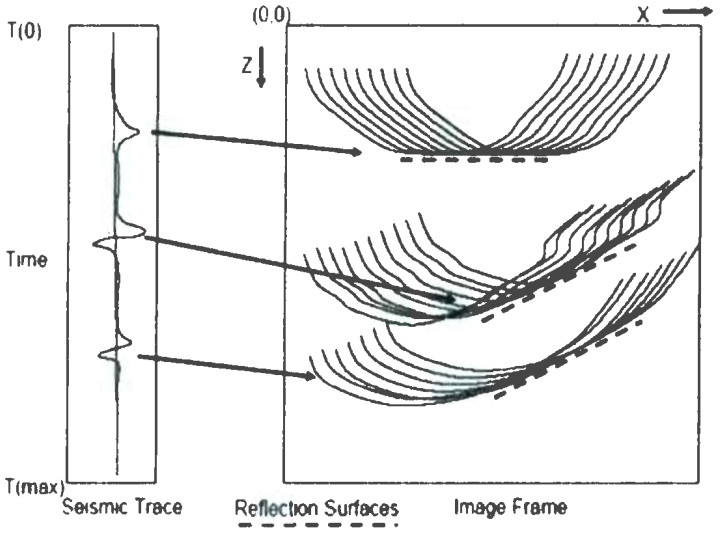


Figure 1.5: Smear the recorded wave samples back to the reflection point

Kirchhoff migration promises to remain a method of choice for pre-stack migration, especially in 3D migration. Generally speaking, Kirchhoff Migration is economical, reliable, versatile, and easy to implement and understand[12].

1.2.1 Kirchhoff Migration

To perform Kirchhoff migration one needs to first compute travel time through a velocity depth model. Then for every travel time of the computed travel time table, a sample of the input data is matched. The amplitude of this sample is scattered along the migration impulse response corresponding to a wave front or location of equal travel time as in Fig.1.5. Mathematically, this can be expressed as a process of spatial convolution. Crude amplitude decay is then considered along the impulse response curve. Impulse responses are stacked to recreate a seismic reflector where energy "stacks in" and theoretically cancel out each other in the same stack process outside this zone. This constructive summation creates the Kirchhoff image.[12].

Take an example[2] of migration of a single seismic trace[1] :

Given a source and a geophone on the free surface, and a single dipping reflector in a homogeneous acoustic medium, there will be only one primary reflection recorded in the seismic trace Fig.1.6. For convenience, multiples and direct waves will be ignored, namely, there is only one pulse in the seismic trace sample representing the main wave front. The arrival time of this event is equal to the travel time for energy to propagate from the source to the reflection point P and from P to the geophone, the two-way-time TWT between source point and receiver. The dashed line in Fig.1.6 depicts the associated specular ray. To backproject the observed energy to its subsurface reflector, the first step is to compute the travel times based on both the source point and receiver poitn. Simply speaking, travel time is the time that seismic wave spends on travel within the subsurface. If the travel time of the wave from source point to the reflection point and receiver point has been known, in the seismic trace record, it is not difficult to find out the corresponding amplitude sample which matches the reflection event happened underneath. From this point, you may already have an intuition on the significant role of travel time in the migration processing. Conventionally, there are two methods to compute travel time: ray tracing and solving the Eikonal equation. The travel time computation will be introduced in later sections.

The next step in Kirchhoff migration is to smear the observed energy to its primary reflection point. This is a blind operation because we know nothing about where the true reflection point is. As a result, the event has to be migrated to all possible reflection points. As in Fig.1.7, such image points r are those whose reflection travel

[1] A seismic trace is the seismic data recorded by one of the geophones: it records the amplitude and timing of the seismic wave reflection from subsurface received on the ground. A seismic trace represents the response of the elastic wave-field to velocity and density contrasts across interfaces of layers of rock or sediments as energy travels from a source through the subsurface to a receiver or receiver array.

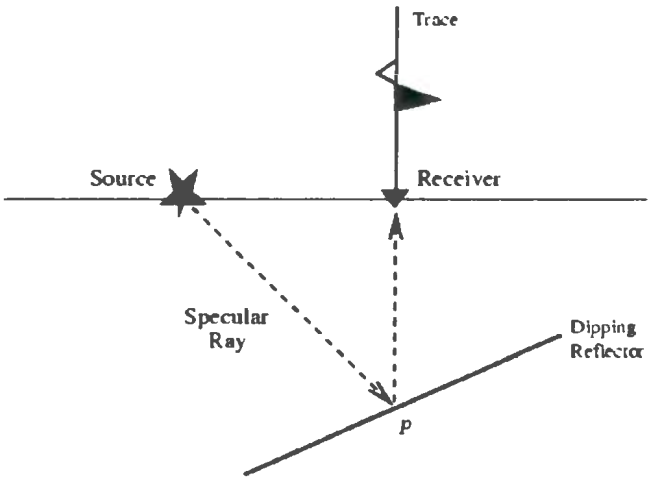


Figure 1.6: Seismic forward modeling for a homogeneous medium, where a single dipping reflector is embedded. The only event being observed is a primary reflection. From H. Sun, 2001[2].

times $\tau_{s1}+\tau_{1g}$ are equal to the observed travel time of the event. Thus, the event energy is smeared to points r_1, r_2, and r_3. In addition, all of the candidate points fill up an entire ellipsoidal zone as shown in Fig.1.7. When the medium is heterogeneous, the migration aperture becomes a quasi-ellipsoid. Fig.1.7 depicts a full-aperture Kirchhoff Migration, where the observed event is migrated to an entire ellipsoid.

Fig.1.8 shows that the dipping reflector in Fig.1.6 can be clearly resolved after many traces have been migrated. The common tangent of the quasi-ellipses depicts the curve of the layer interface. Stacking[2] of migrated traces helps strengthen the true reflectors and attenuate migration artifacts. Namely, the common tangent part of the quasi-ellipses will be highlighted. Accompanying with many other technical measures such as truncating the migration aperture and applying anti-aliasing filters,

[2]Roughly speaking, stacking can be understood as gathering together the received wave samples for the same reflection point recorded in different seismic traces.

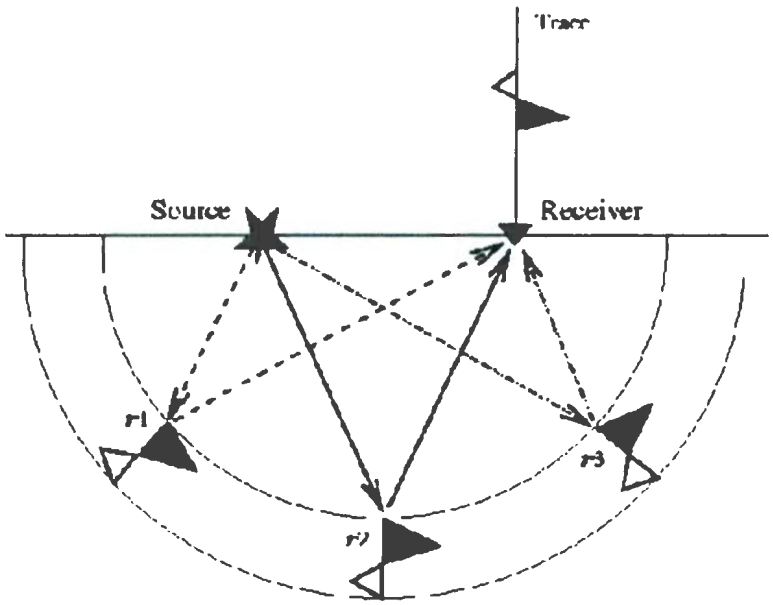


Figure 1.7: Full-aperture Kirchhoff Migration of a single trace. Each time sample on the trace is smeared along an ellipse, and each event is smeared along an ellipsoidal zone. Taken from H. Sun, 2001[2].

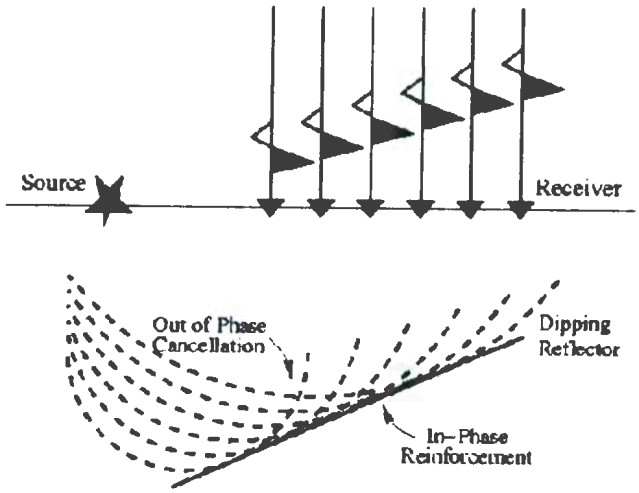


Figure 1.8: Migration of several seismic traces. The energy from the wings of the half-ellipses incoherently cancel, while the energy from the common tangent to the half-ellipses coherently superimpose. From H. Sun, 2001[2].

the migration can give an image of the subsurface structures.

From this example, it is easy to see the important role which travel time plays in Kirchhoff migration. The travel time table or say travel time map work as the reference scales to interpolate the wave samples in the spatial convolution. To compute the travel time fields is one of the core tasks in Kirchhoff migration process. In next section, let's see what travel time is and the difficulties to compute travel times.

1.2.2 Travel Time

The first step to apply Kirchhoff migration is to compute travel time fields. For 2D subsurface area, we can take it as a plane with the coordinate axis X pointing to the direction along the ground surface and coordinate axis Z pointing downward in depth. By meshing this interception plane of the earth into evenly distributed grids, Finite Difference Method (FDM) can be applied to do computation and numerical analysis

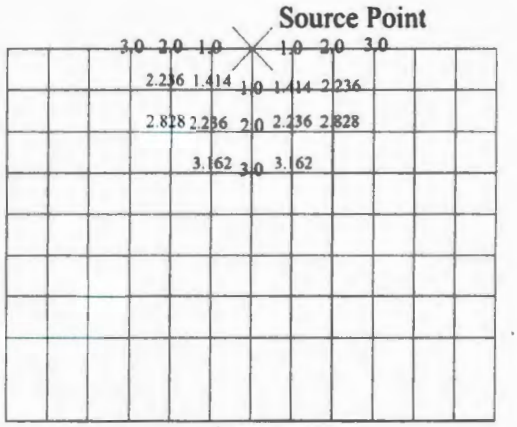


Figure 1.9: Travel time table.

to decide the physical properties of the field. Travel time field can be considered as a travel time map or say travel time table (as shown in Fig.1.9): each point on the grid has a value representing the time that the seismic wave spends on traveling from source point to itself. There is a numerical approximation that the small area around a grid point holds the same travel time as the value on this grid point. The travel time table, in data structure, can be an array of numbers.

The travel time contour which indicates the points hold the same travel time values can be plotted according to the travel time values on grid points(Fig.1.10).

If the velocity is constant, travel time value is just a scaled value of distance. If the coordinates of a grid point are known. the distance can be achieved by applying Pythagorean Theorem in the simple geometry. Nevertheless, it is impractical to expect the earth holding a constant velocity everywhere. Similar to the model shown in Fig.1.11, a geology structure can have complicated features: Scientists have developed efficient ways to compute the travel times in complex velocity models. Major methods are reviewed in next section.

Moreover, travel time data is also required by many other seismic modeling and

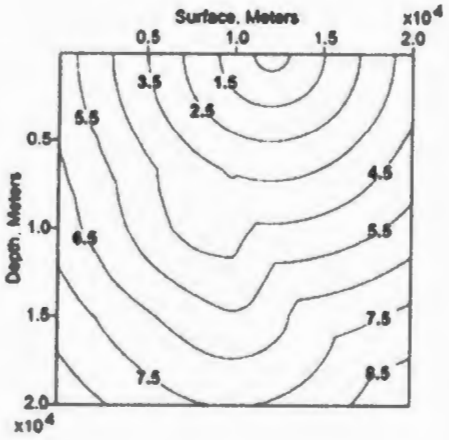
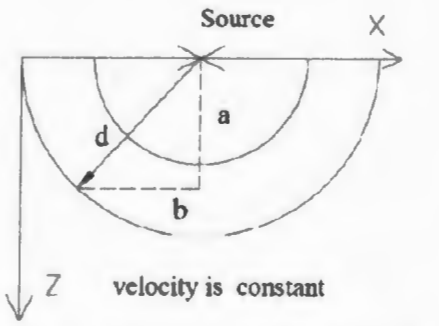


Figure 1.10: Travel time contours.



According to Pythagorean theorem, $d = \sqrt{a^2 + b^2}$, and if velocity v is a constant in this model, thus, travel time $t = d / v$.

Figure 1.11: Travel time in constant velocity field.

inversion applications. For instance, the inverse migration and velocity estimation[3].

1.2.3 Travel Time Computation

Travel time calculation is computational intensive. Research is advancing toward accurate, robust, efficient and economical affordable methods. Generally speaking, all these method can fall into two major categories: Ray tracing method and methods that are based on a direct numerical solution of the Eikonal equation using finite differences (we refer it as Eikonal equation solving method in this thesis).

1.2.4 Ray Tracing Method

Ray tracing is a well practiced graphic art. The concept that seismic energy of infinitely high frequency follows a trajectory (or say ray path) determines the ray tracing equations. Physically, these equations describe how energy continues in the same direction until it is refracted and reflected by velocity and density variations.

There are two kinds of ray tracing problems, namely initial value and boundary value ray tracing. Initial value ray tracing is numerically rather stable and fast; however, in geophysics and seismological applications such as earthquake location and velocity structure inversion, Two-Point (TP) ray tracing is a more commonly adopted method. Two-Point ray tracing is a boundary value problem[13]. TP ray tracing in a heterogeneous isotropic medium has been investigated by many seismologists and mathematicians (Julian and Gubbins, 1977[14]; Cerveny et al., 1977[15]; Pereyra et al., 1980[16]).

Under TP ray tracing, there are Ray Shooting methods and Ray Bending methods[5].

[3]Velocity estimation is the inverse problem of conversing time migration velocities to true seismic velocities (the velocity model built in Cartesian depth coordinates) by geometrical spreading process.

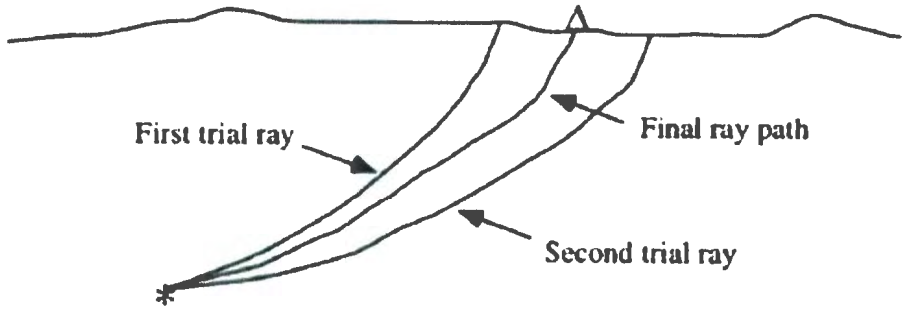


Figure 1.12: Ray Shooting Method, from D. A. Waltharn, 1988 [3].

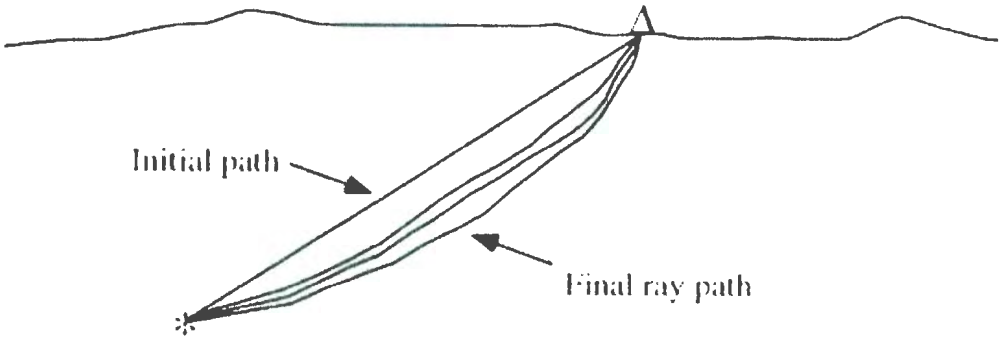


Figure 1.13: Ray Bending Method, from D. A. Waltharn, 1988 [3].

The shooting method fixes one end of the ray path (source point), and takes initial incidence angle and initial azimuth, and then use ray path equation to find the coordinates of another end point (Fig.1.12). It is like a fan of rays is shot from one point in the general direction of the other. The correct path and travel time to connect the two points may then be approached with successively more accurate guesses. Whereas, bending method fixes the two ends of the ray, and takes some initial estimate of the ray path and perturbs it until it satisfies a minimum travel time criterion (Fermat's principle or the principle of least time[17][3]) (Fig.1.13). Bending method is an efficient way to trace a ray between two given end points.

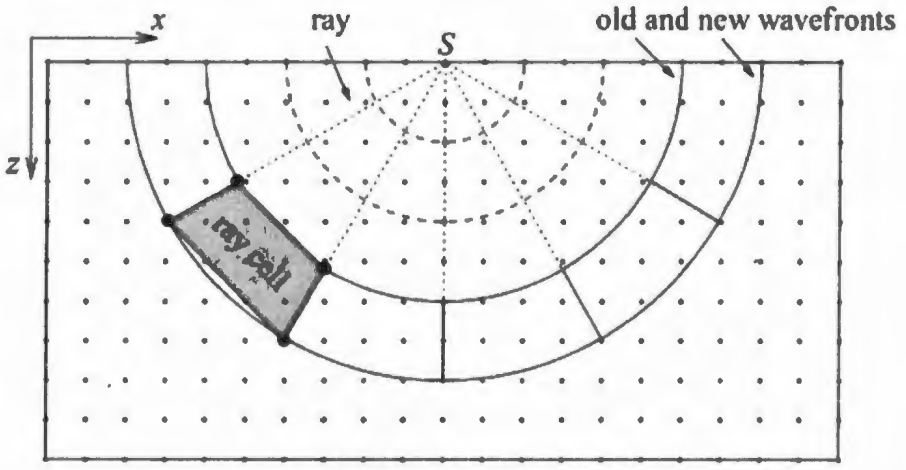


Figure 1.14: Graphical description of the Wavefront Construction methods. The travel times at nodes are computed by ray tracing. The travel times at grid points are estimated within a ray cell. From Vinje. V.. 1993, [4].

Furthermore, instead of tracing only one single ray. the computing can propagate a ray field. To raise the efficiency of these methods, adjacent rays are grouped into ray tubes, the ray density of the ray field is checked at certain positions. and if necessary, new rays are inserted. The ray field may be examined at constant depth levels, at interfaces, at the final surface, or at the wavefronts.

Methods which check the ray field at wavefronts are called wavefront construction (WFC) methods, which is introduced by Vinje. Iversen and Gjoystddal, 1993[4]. and became the other milestone in the progress of ray tracing. The main characteristics of WFC (see Fig.1.14) are: (1) wave propagation. the propagation of the ray field with a constant travel time step: (2) the insertion of a new ray between two adjacent rays if illumination is insufficient: (3) the estimation of travel times with a ray cell, which is the part of the ray tube between the last two constructed wavefronts[18].

In general. the use of ray tracing method followed by interpolation of travel times

onto a regular grid is a popular and robust method for computing diffraction curves for Kirchhoff migration. There are two reasons to review ray tracing method in this section:

1. Wavefront Construction methods and Finite Difference methods solving Eikonal Equation are two widely used methods to calculate travel times. Our method raised in this thesis is one of the Eikonal solving methods: before introducing our algorithm, it is better to understand existing algorithms.

2. Some comments given to ray tracing methods is that[19] ray tracing methods can accurate in describing both travel times and ray paths, but require expensive global wavefront construction and travel time interpolation from these wavefronts to grid points. Indeed, from the ray tracing method, high accuracy travel time tables can be achieved (theoretically, by ray tracing, 'exact' solutions can be achievable by analytical solving ray tracing equation). These high accuracy results are usually taken as the reference to verify the travel time results generated from new methods.

On the other hand, an alternative to ray tracing is directly solving the eikonal equation with finite differences at each grid point, without computing ray paths. Solving the Eikonal equation on such a grid simplifies the problem of interpolating times onto the migration grid[19]. This procedure also eliminates several issues associated with ray tracing, such as: shooting versus TP ray tracing, how many rays to trace, and the type of velocity model (gridded or interface-based) to use. A further reason to study gridded solutions of the Eikonal equation is for travel time calculations in three dimensions, where even the simplest ray-based methods can be horribly complex.

1.2.5 Eikonal Equation Solving

In 1988, John Vidale published his famous paper *Finite-Difference Calculation of Travel Times*[5], which opened the way to calculate travel times on a regular velocity grid by solving the Eikonal equation using finite differences. Following his work, many researchers (as well as our group) designed their own Eikonal solvers to compute travel times.

Vidale first extended the finite difference schemes described by Reshef and Kosloff (1986) to compute travel times of first arrivals in a 2D isotropic media model (this is the same assumption used in our algorithm, namely, compute the first arrival travel times in a media model in which the vertical velocity and the lateral velocity are the same on every grid point); then he expands the problem into 3D model two years later (1990).

Eikonal equations are obtained from the elastic-wave equations by searching for plane harmonic solutions and applying the high-frequency approximation of ray theory[6]. The propagation of 2D geometric rays and the propagation of 2D wavefronts can be guided by the Eikonal equation of ray tracing.

$$(\frac{\partial t}{\partial x})^2 + (\frac{\partial t}{\partial z})^2 = s(x,z)^2 \tag{1.1}$$

In the same velocity layer of a model, or in vicinity of a grid point where the velocity in that small region can be thought as the same, the slowness s (the reciprocal of velocity) is a constant. Eq.1.1 therefore can be written as Eq.1.2.

$$(\frac{\partial t}{\partial x})^2 + (\frac{\partial t}{\partial z})^2 = s^2 \tag{1.2}$$

Local Scheme of Extrapolation

Considering the geometry in Fig.1.15, assume the travel times between the points A (t_0), B_1 (t_1), and B_2 (t_2) and the source point are known, and the travel time t_3 between point C_1 and the source point A is sought. Using two finite differences Eq.1.3 and Eq.1.4 to approximate differential terms, Eq.1.5 can be derived. It evaluates the travel time of point C_1 using the travel times from the source to points A, B_1, and B_2, in a plane wave approximation[1] as shown in Fig.1.16. This Eq.1.5 is called **Extrapolation Formula**, which determines the local computing scheme that evaluate the travel time of a uncomputed point based on the other three known travel times on the points in the same grid.

$$\frac{\partial t}{\partial x} = \frac{1}{2h}(t_0 + t_2 - t_1 - t_3) \tag{1.3}$$

$$\frac{\partial t}{\partial z} = \frac{1}{2h}(t_0 + t_1 - t_2 - t_3) \tag{1.4}$$

$$t_3 = t_0 + \sqrt{2(hs)^2 - (t_2 - t_1)^2} \tag{1.5}$$

On the other hand, Vidale also derived Eq.1.6 to deal with the extrapolation of the wavefronts which have high curvature. x_s and z_s are the coordinates of the virtual source point of the circular wavefront with high curvature, t_s is the origin time for the virtual source[5]. It is straightforward that, generated from a point wave source, the wavefronts at close-origin region have higher curvature than the ones at far-origin region as shown in Fig.1.17. The combined use of Eq.1.6 in close-origin region and Eq.1.5 in far-origin region is called a "mixed" scheme by Vidale, which can provide

[1]When transmits far from the source point and curvature of the wavefront becomes low, the wave can be modeled as plan wave.

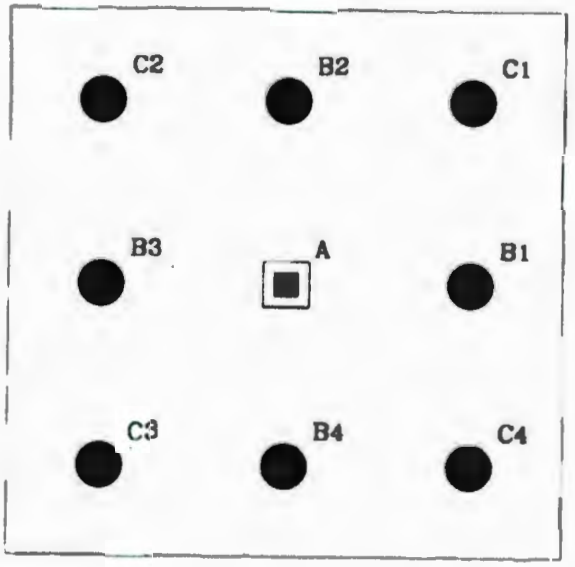


Figure 1.15: The source grid point A and the eight points in the ring surrounding point A, from Vidale, 1988, [5].

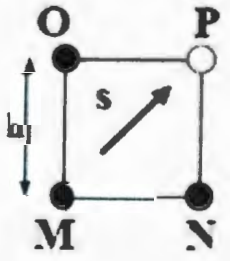


Figure 1.16: Plane wave approximation Scheme. The travel times at three corners M,N,O of a grid cell are used to estimate the travel time at the fourth corner P

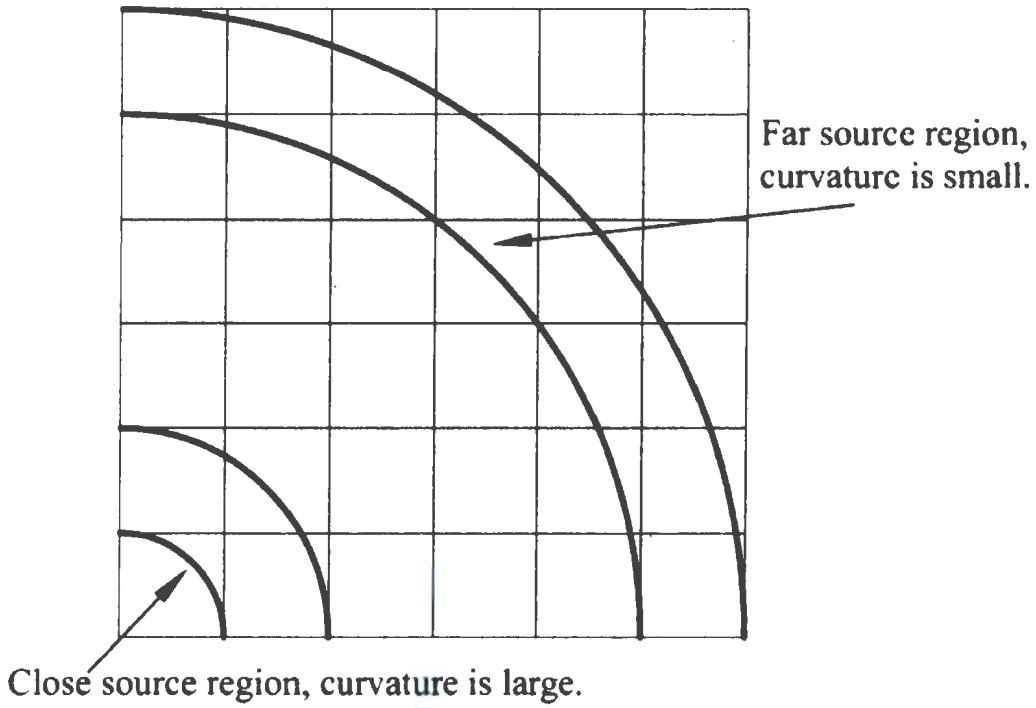


Figure 1.17: Illustration of the curvature change.

better accuracy. For purposes of achieving high speed rather than great accuracy, Eq.1.5 may be used exclusively in a "simple" scheme.

$$t_3 = t_s + s\sqrt{(x_s + h)^2 + (z_s + h)^2} \tag{1.6}$$

Inductive Scheme

The inductive scheme of adding a ring of travel times to those already calculated is the second key point of the algorithm. It determines how to spread and extend the computation to all the grids in the domain.

Vidale's inductive scheme is as shown in Fig.1.18. Consider the ring of radius 5, where all travel times inside the ring are known, but travel times on and outside the

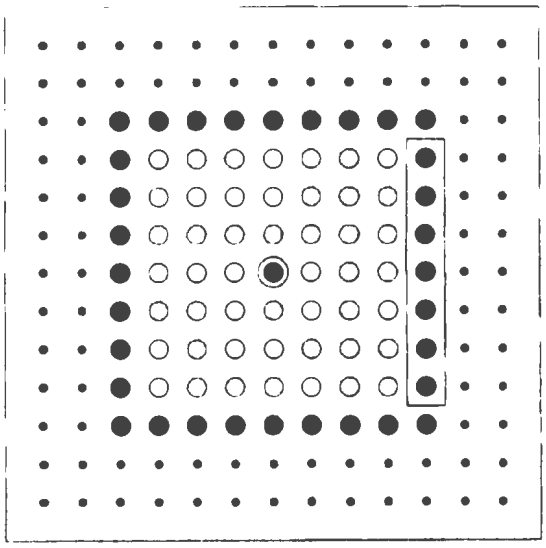


Figure 1.18: Picture of the 2-D grid as the numerical calculation of travel time is progressing. The ring of points shown as filled circles are about to be timed. The hollow circles indicate points that have had their travel time calculated. The double circle in the middle shows the source point. The dots are not yet timed, nor will they be timed until the ring of filled circles is done. Figure is from Vidale, 1988, [5].

ring are unknown. Solution will proceed on the four sides sequentially, followed by the four corner points. Computing can start arbitrarily with the right side, and find travel times for the points within the four sides of the rectangular.

The procedures can be shown in Fig.1.19. First of all, the points in the row are examined in order from left to right, and the points that are at a relative minimum are identified. Note that there are usually several relative minima on one line (as in Fig.1.20).

Then, in step (1), a relative minimum in the currently computed row is used to evaluate the point adjacent to it. A relative minimum is assumed if there is a relative minimum in the time for the adjacent point in the adjacent row that has

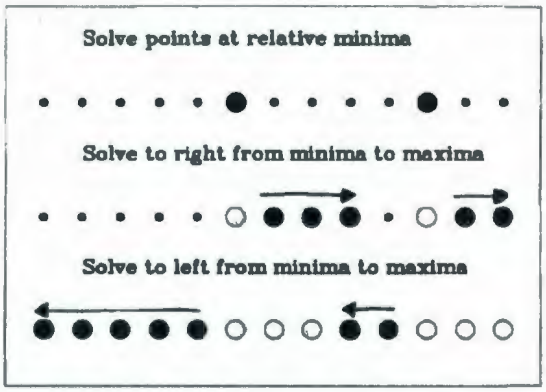


Figure 1.19: Sequence of solution of one edge of the ring: first, the points that are just outboard of those at a relative minimum. Next, we sweep to the right and solve the points from each relative minimum until either a relative maximum or the edge is encountered. Finally, we sweep to the left from each relative minimum until reaching a relative maximum or edge. These three steps will find the times for the entire edge.

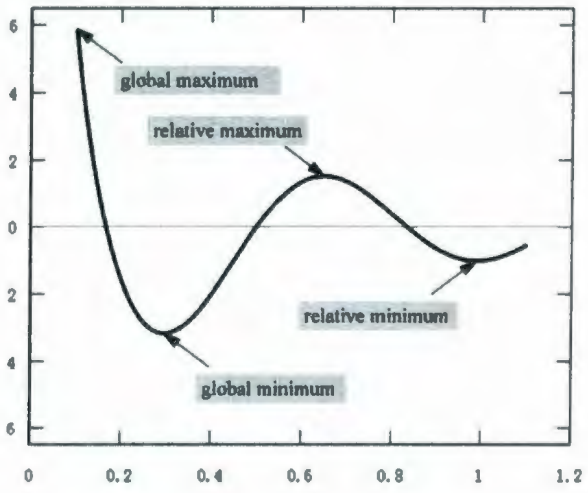


Figure 1.20: Relative minima and relative maxima.

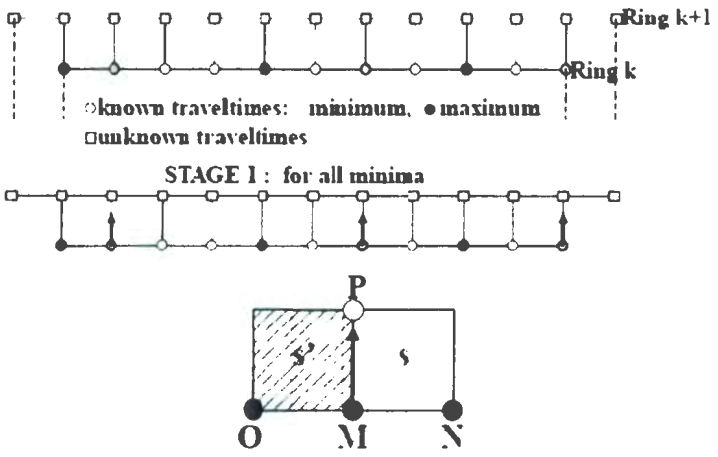


Figure 1.21: The expanding square ring process: the initial stage is the emission time at a source point on a regular grid. Thereafter, travel times are determined along successive square rings centered at the source, using travel times on the previous ring.

already been solved in the previous ring. To time the first point on an edge, a non-centered finite-difference of Eq.1.2 must be used. Vidale used the plane-wave formula as Eq.1.7: Where t_3 is the time to be found, t_0 is the relative minimum time in the inside row, and t_1 and t_2 are the times on either side of the point whose time is t_0. The Fig.1.21 may give you a better illustration of this idea. Remember that as the local Extrapolation scheme of Eq.1.5 is used, three points in a grid have to be known to evaluate the fourth one, that is why there is this step to compute point P as Fig1.21 shown.

$$t_3 = t_0 + \sqrt{(hs)^2 - 0.25(t_2 - t_1)^2} \tag{1.7}$$

In step (2)(as shown in Fig.1.22), starting at each relative minimum point, solution progresses along the row finding the time for each point until the relative maximum is encountered. The computing is iteratively following Eq.1.5 that in com-

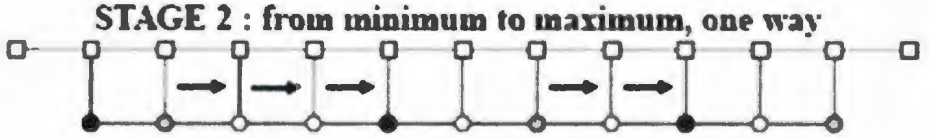


Figure 1.22: The minimum-to-maximum travel time progression: local travel time minima and maxima along a side of a determined ring: first from right to left, from each minimum to the next maximum.

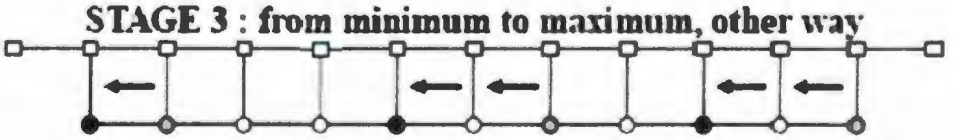


Figure 1.23: Do the computing in a reverse way: from left to right, from each minimum to the next maximum.

putation of each grid, the point on the fourth corner is evaluated based on the other three known ones (as Fig.1.16 shown, the local Extrapolation scheme).

In step (3) (as shown in Fig.1.23), upon completion of the left-to-right sweep through the row, the row is swept through in the reverse direction, and the remaining untimed points are solved in order from the relative minima to the relative maxima. One thing you may notice is that grid points in front of local maxima[5] will be assigned two travel times[6], in step (2) and step (3). But only the smallest is kept: this is equivalent to considering two geometric rays coming from either side and only counting the one that arrives first[5].

Once all the four sides are solved in this way, the times for the corners may be found, and it proceeds to the next ring outward. By applying this method iteratively, the entire two-dimensional grid is filled with travel times like a square wavefront

[5]Vidale's paper say "minima" here. But I agree to the idea in [6], in which said it is "maxima".

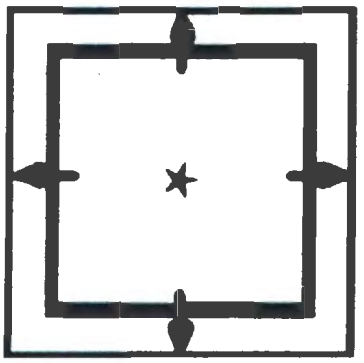


Figure 1.24: Square wavefront spreading pattern.

spreading out (see Fig.1.24).

Limitations of Vidale's Method

Prior to Vidale's work, travel times were mainly computed using ray tracing. While these ray-tracing methods offer a high degree of accuracy, they also pose interpolation problems in shadow areas and areas where multiple caustics develop. The use of finite difference travel times solved the problem of interpolating in shadow zones, but new issues ensued[20]. In Vidale's Finite Difference method, the way that the computation evolving is not an exact mimic of the seismic wave transmitting in the nature, so there are limitations would make the method fail in the models that have strong velocity variations. It is obvious that seismic waves are not spreading as the shape of a square; rather it is a better way to mimic the wave transmitting by following the Fermat's Principle. Two points need to be noticed in Vidale's method: (1) by theory, the solution must follow causality, that is, the time for the part of the ray path leading to a point must be known before the time of the point can be found; (2) in practice, solving for progressively earlier times along a row results in an instability.

1.2.6 Fast Marching Method

Following Vidale's work, a lot of travel time computation methods are raised at early 90s. After in depth study, researchers concentrated on the core of this problem: computing travel times is equivalent to tracking an interface advancement with a speed normal to itself given by the supplied velocities. The goal in such interface advancement is to deal accurately and robustly with the formation of cusps and corners, topological changes in the propagating interface, and stability issues in computing space.

In the 1980s, the level set method[21] was developed by the American mathematicians Stanley Osher and James Sethian. Generally speaking, level set method is a numerical technique which can follow the evolution of interfaces. It became popular in many disciplines, such as image processing, computer graphics, computational geometry, optimization, and computational fluid dynamics. In 1996, based on previous work, Sethian raised fast marching methods[21], which specifically aimed at the solution of the Eikonal equation. His technique hinge on the construction of entropy-satisfying weak solutions by using numerical schemes borrowed from the technology of hyperbolic conservation laws and aimed at constructing the correct viscosity solution of the appropriate partial differential equations[20]. He discussed fast marching method on the travel time computation issues in the geophysics application by publishing several papers from early 90s until recently[20] [22] [23] [24]. The method we have developed in this thesis is actually originated from consideration of the fast marching method. Before presenting our algorithm, fast marching is introduced first as theory foundation.

In Sethian's method, he used the one called finite difference upwind stencil[21] [20] to locally solve and advance the eikonal equation. He used this stencil in combination

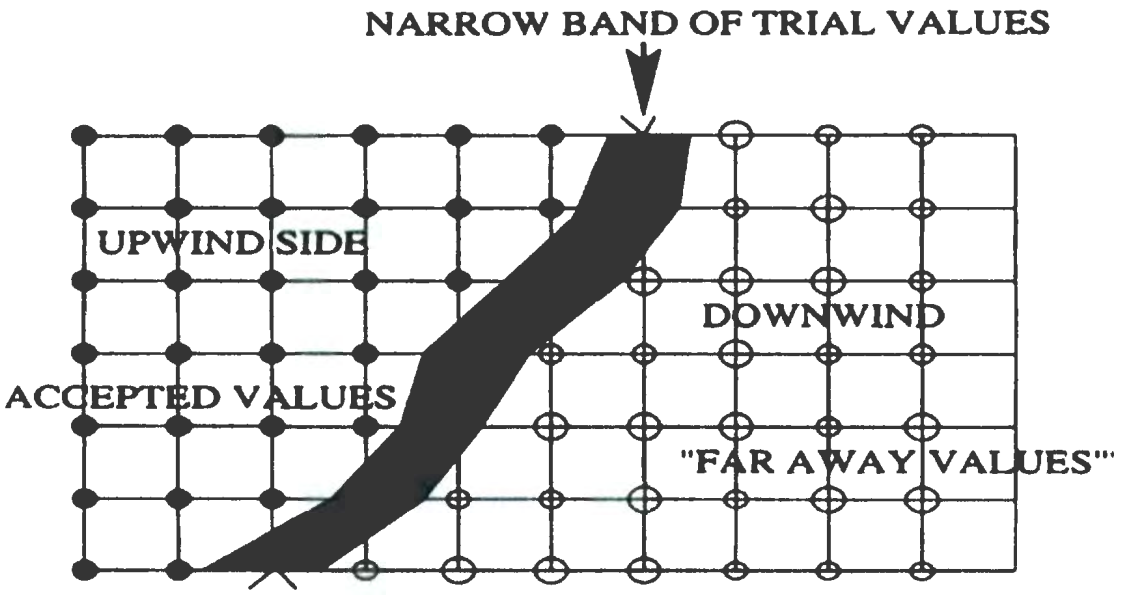


Figure 1.25: Wavefront evolving of the fast marching method. From I. Lecomte, 2000, [6].

with a wavefront construction technique based on building a narrow band around the travel time wavefront. The travel time values are stored on a heap, with the minimum value at the top of the heap. The wavefront is always advanced by using the minimum travel time value in the heap. The cost of a heap operation is $\log(N_{NB})$: where N_{NB} is the total number of travel time values in the narrow band.

The fast marching method solves eikonal equation by directly mimicking the advancing wavefront (see Fig.1.25). Every point on the computational grid is classified into three groups: points behind the wavefront on the "upwind" side, whose travel times are known and accepted; points on the wavefront in a narrow band area, whose travel times have been calculated, but are not yet accepted; and points ahead of the wavefront on the "downwind" side. The algorithm then proceeds as follows:

1. Choose the point on the wavefront in the narrow band set with the smallest

travel time.

2. Accept this travel time into the set on the upwind side.

3. Advance the wavefront, so that this point is behind it, and adjacent points are either on the wavefront or behind it.

4. Update travel times for adjacent points on the wavefront by solving Eikonal equation numerically.

5. Repeat until every point is behind the wavefront.

The update procedure in step 4 requires the solution of a quadratic equation which is the numerical approximation of eikonal equation. The numerical approximation scheme used here is finite difference upwind stencil which constructs an entropy-satisfying approximation to the travel time gradient (eikonal equation says that travel time gradient is the inverse of velocity at that point). For the concepts such as entropy condition and entropy-satisfying approximation, please refer to [21] and [20] for details. They are concepts under the level set and fast marching theory. This section is trying to make reader have an intuition of the inductive scheme of the Sethian's method (rather than the mathematic derivation for the numerical scheme), because in our algorithm, the inductive scheme is similar to Sethian's, but we adopt a distinct numerical scheme for local extrapolation.

Chapter 2

Algorithm Design

In this chapter, we first present Vidale's and Sethian's work, and then we developed **Least-Time Path Fast Marching Method**. It originates from the result of ray tracing, finite difference and fast marching. In local extrapolation scheme of the algorithm, according to ray tracing theory, the formulae are developed based on simple geometry, following Fermat's principle of least travel time; for introductive scheme of the algorithm, following the ideas of Sethian's fast marching method, our algorithm avoids the causality problem of Vidale's finite difference method.

Moreover, two versions of our algorithm are proposed. The sequential version is based on classic implementation techniques, while a fully parallel version can be operated on Networking Computing on FPGA Array or other parallel computing platform.

2.1 Local Scheme of Extrapolation

Reviewing Vidale's local scheme in Fig.2.1, his numerical approximation of Eikonal equation is straightforward: Eq.2.1 expresses the assumption that the travel time from

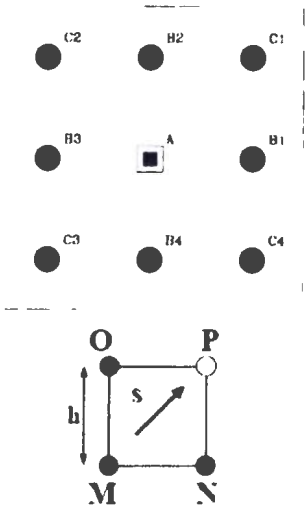


Figure 2.1: Vidale's local scheme. From Vidale, 1988, [5].

one grid point to its neighbor is the contribution of the time seismic wave transmits through a half length of the grid spacing at the velocity of point, and the time in which it travels through the other half length at the velocity of its neighbor point. Eq.1.3 and Eq.1.4 are all derived from this basic assumption. This relation does not consider the geometry of ray tracing, and it is a coarse approximation.

$$t_i = \frac{h}{2}(s_{Bi} + s_A) \tag{2.1}$$

Whereas, in our method, the problem is first studied inside a grid. By partitioning a grid into two triangles, the relationship among the travel time values of three grid points t_a, t_b , and t_c in a single triangle is considered (see Fig.2.2).

As shown in Fig.2.3, in finite difference method, the area of interests is partitioned into evenly distributed grids. The travel time values are discretized onto the grid points, even though the real ray path may not pass the grid points in practice. In this way, inside each grid, inside each triangle the incidence position on the edge is

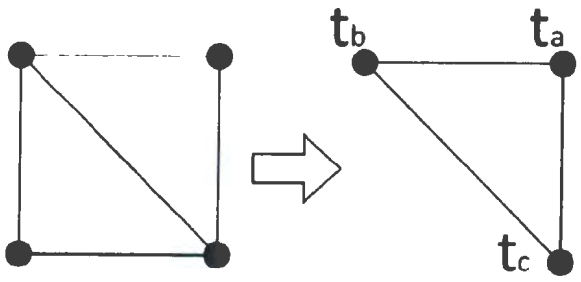


Figure 2.2: Geometry inside one triangle.

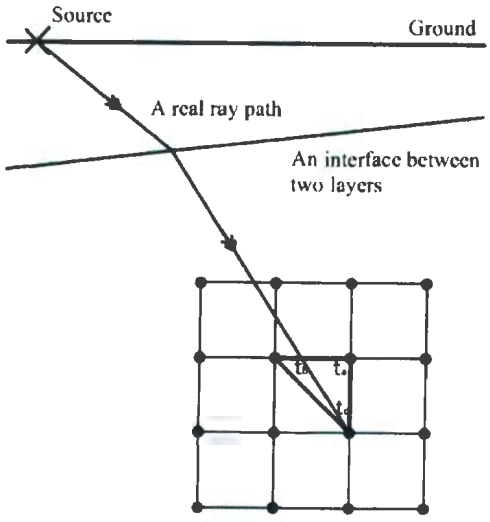


Figure 2.3: Finite difference meshing.

decided by the travel time values on the two grid points (see Fig.2.4). The "ray" mentioned here can be understood in two ways: it can be taken as the seismic wave after high frequency approximation; or as the normal of the travel time wavefront[1].

[1]The travel time computed in this thesis is, in fact, the first arrival travel time of the seismic wave. There are comments that the first arrival may not be the portion that carries most energy of the seismic wave. However, I will not raise a discussion of this pure geophysics problem in this thesis.

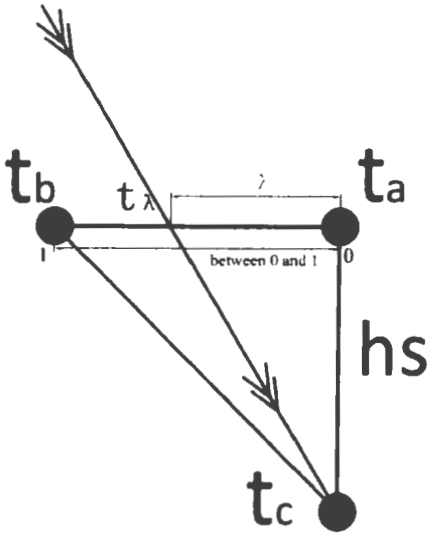


Figure 2.4: Linear interpolation.

2.1.1 Formula Derivation following Linear Interpolation and Fermat's principle

As in Fig.2.3, the coefficient λ is defined as a fraction ranging from 0 to 1. If we assume the wave source locates at the upper left of the triangle, thus, because the wave will first reach point b before arriving at point a, the travel time value t_a on point a will be always larger than the t_b on b. Herein, $t_a - t_b$ is positive, and $t_b - t_a$ is negative. The value of λ should satisfy the following Eq.2.2.

$$\lambda = \frac{t_\lambda - t_b}{t_a - t_b} \tag{2.2}$$

t_λ is the travel time at incidence point, and can be expressed as in Eq.2.3 by applying linear interpolation scheme. λ works as a linear interpolation coefficient.

$$t_\lambda = t_a + \lambda(t_b - t_a) \tag{2.3}$$

Comparing to Vidale's scheme in Eq.2.1 which simply takes arithmetic average, the linear interpolation in Eq.2.3 is more flexible and accurate.

In the method, square finite difference grid is used to mesh the field, in which four edges of a grid are the same in length. This length is denoted as h. Meanwhile, if we make the assumption that the seismic wave is transmitting in an isotropic media, thus, the velocity in each direction is the same, and the slowness s (the inverse of the velocity) is the same in every direction. In this way, $h \cdot s$ is the length of an edge of a square grid. Based on this context, the formula Eq.2.1 for t_c can be derived on the geometry in Fig.2.4.

$$t_c = t_a + \lambda(t_b - t_a) + \sqrt{1 + \lambda^2} \cdot h \cdot s \tag{2.4}$$

This formula gives out an expression of t_c concerning t_a, t_b, h, s and λ. If these variables are known, t_c can be evaluated from this formula.

The coefficient λ is unknown in Eq.2.1. However, it can be evaluated from an implicating condition by following Fermat's Principle. According to Fermat's Principle, the path taken between two points by a ray of light is the path that can be traversed in the least time, in nature, the ray will choose the incidence point along the edge ab (see Fig.2.5) to go through a path that can make the travel time t_c the minimum. In other words, the real ray path should be the path whose λ can make Eq.2.4 be evaluated as a minimum of t_c.

Based on this analysis, the partial derivative of Eq.2.1 with λ results in Eq.2.5. The partial derivative should be equal to zero when value of T_c reach a minimum[2].

[2]According to calculus, an extremum can be reached when the derivative of a function is equal to zero; however, monotropy analysis tells us that the function here will be a minimum when derivative is equal to zero.

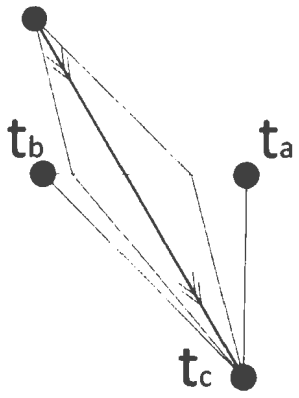


Figure 2.5: Least travel time path for ray tracing.

$$\frac{\partial t_c}{\partial \lambda} = (t_b - t_a) + \frac{\lambda}{\sqrt{1+\lambda^2}} \cdot h \cdot s = 0 \tag{2.5}$$

Herein, from Eq.2.5 we can derive the following expression of λ concerning T_a, T_b, h and s.

$$\lambda^2 = \frac{(t_a - t_b)^2}{h^2 s^2 - (t_a - t_b)^2} \tag{2.6}$$

Furthermore, Eq.2.4 can be transformed into the form of Eq.2.7.

$$t_c = t_a + hs \cdot \sqrt{1+\lambda^2} \cdot (1 - \frac{(t_a - t_b)^2}{h^2 s^2}) \tag{2.7}$$

Substituting λ^2 in Eq.2.7 with the expression 2.6. Finally, we can get a formula of T_c as Eq.2.8.

$$t_c = t_a + \sqrt{h^2 s^2 - (t_a - t_b)^2} \tag{2.8}$$

This formula is only concerning t_a, t_b, h and s. h is predefined, and s is read from an known velocity table as input. Eq.2.8 builds up a scheme that, among 3 grid points in a triangle, the third one can be evaluated with the other two known travel times.

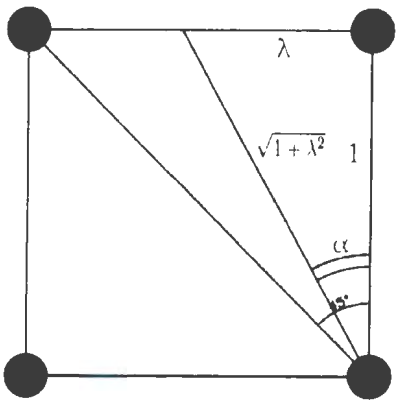


Figure 2.6: Range of angle α.

This scheme is based on geometric interpolation and Fermat's principle. Comparing to Vidale's scheme which needs 3 known travel times to evaluate the fourth one, our Eq.2.8 needs the same number of arithmetic operations as Vidale's Eq.1.5. So our local scheme has similar computational complexity (same operations such as addition, multiplication and square root), but needs less operands in the formula.

2.1.2 Determine the Function Interval

Eq.2.8 is the main formula for our scheme. However, before using it, its effective interval has to be determined. As in Fig.2.6, the angle α should be ranged from 0°to 45°, because the grid is square and the triangle is isoceles triangle. The fraction λ thus, needs to satisfy the following relation in Eq.2.9.

$$0 \leq \lambda \leq 1 \tag{2.9}$$

Squaring Eq.2.9 on either sides, Eq.2.10 can be achieved.

$$0 \leq \lambda^2 \leq 1 \tag{2.10}$$

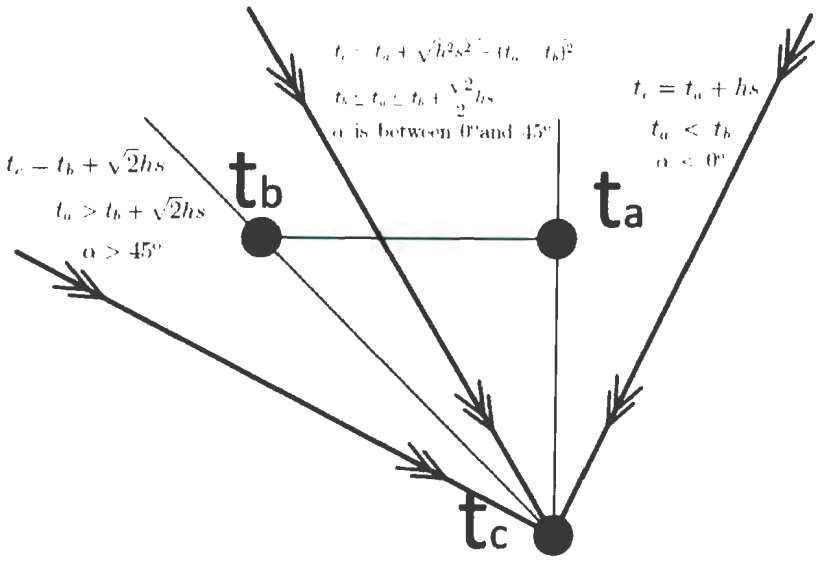


Figure 2.7: Function interval.

Substituting λ^2 with right side of Eq.2.6, get

$$0 \le \frac{(t_a - t_b)^2}{h^2s^2 - (t_a - t_b)^2} \le 1 \tag{2.11}$$

After reduction, Eq.2.12 can be achieved from Eq.2.11.

$$t_b \le t_a \le t_b + \frac{\sqrt{2}}{2}hs \tag{2.12}$$

This equation gives out the effective interval of our main formula Eq.2.8 (see Fig.2.7).

Eq.2.8 is only applied when α is between 0°and 45°. When $t_a < t_b$, $\alpha < 0°$, the ray is actually come from a source on upper right of the triangle, thus, Eq.2.13 can be used to compute the travel times under this condition. In the same way, Eq.2.14 is used to compute the travel times when $\alpha > 45°$and $t_a < t_b + \frac{\sqrt{2}}{2}hs$.

$$t_c = t_a + hs \tag{2.13}$$

$$t_c = t_b + \sqrt{2}hs \tag{2.14}$$

2.1.3 Rice Computing Unit

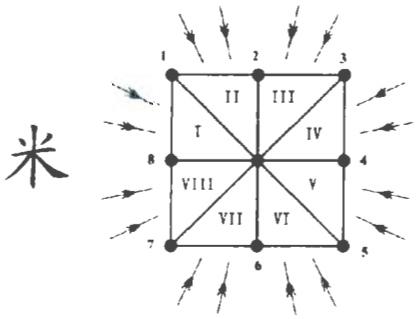


Figure 2.8: Rice computing unit.

When determining the function interval, there is a problem that the formula Eq.2.8 can only deal with the computation in a 45°arc; thus how to compute the rays which may come from every direction in subsurface area? The solution is shown in Fig.2.8: for a single grid point, combine 8 computing triangles together, and apply the computing jointly to this grid point at same time. After getting 8 results, the minimum among them is taken as finial answer. Because 8 triangle computing units can cover the whole 360°arc (see Fig.2.8), the true answer should be caught by one of the 8 triangles. In practice, according to Fermat's Principle, rays always travel through the least time path, so the final answer should be the minimum among the 8 results. This combined computing unit of 8 triangles in Fig.2.8 looks like a Chinese character "rice". So it is called "rice computing" and "rice computing unit", or the Radial

Incidence Computing Element (RICE).

Rice computing evaluates one result from 8 inputs plus the information of h and s. However, it is not necessary for all of the 8 surrounding points to be known. If any of the 8 surrounding points are unknown, it can be set as "infinity"; and the triangle which hold the "infinity" will never "win" through the process of electing minimum.

All the travel times known and unknown are stored in a data array. If the coordinates of a target point are known, by applying $+1$ or -1, the coordinates of the 8 surrounding points of the target point can be achieved. The 8 triangle computing can be done sequentially on a single CPU computer in a traditional language like C or Fortran; while the 8 computing units can also be implemented in parallel on the platform such as FPGA and ASIC, or using parallel programming language constructs.

2.2 Inductive Scheme

The inductive scheme of our algorithm is similar to Sethian's fast marching method.

The grid spacing h should be assigned at first. This parameter can largely impact on the meshing, and further on the accuracy of the Finite Difference Method (FDM). Vidale proposed an accuracy estimation as Eq.2.15, which represents the timing error divided by the transit time across the cell [5].

$$E = \frac{ta - tb}{hs} \tag{2.15}$$

The other input to the algorithm is the velocity model. It is a data array holding the travel time values of every grid point.

Before the algorithm begins, the parameter h and the velocity array are precomputed. The algorithm begins with setting a set of seed points which are sur-

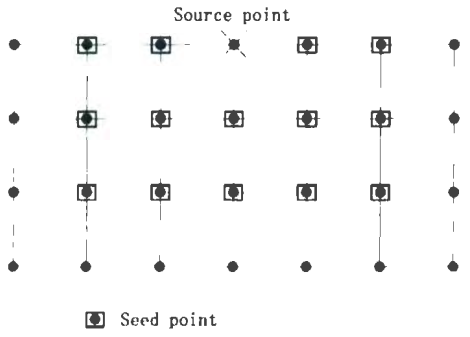


Figure 2.9: Initial set of seed points.

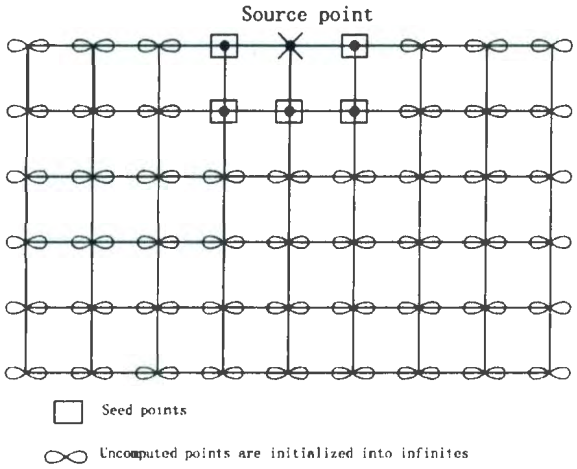


Figure 2.10: Initialization.

rounding the source point. Because this initial set of points are very close to the source, their travel times can be pre-calculated. The amount of seed points to be used is determined by application practice (see Fig.2.9).

Before starting, the travel time array is first initialized into infinities on each point; and the initial set of seed points are assigned into the form as shown in Fig.2.10.

In course of computing, the computation evolves from seed points outwards. There are three classes of points during the process (see Fig.2.11), the first set contains fixed points whose value are computed and have been popped off from the active set. The

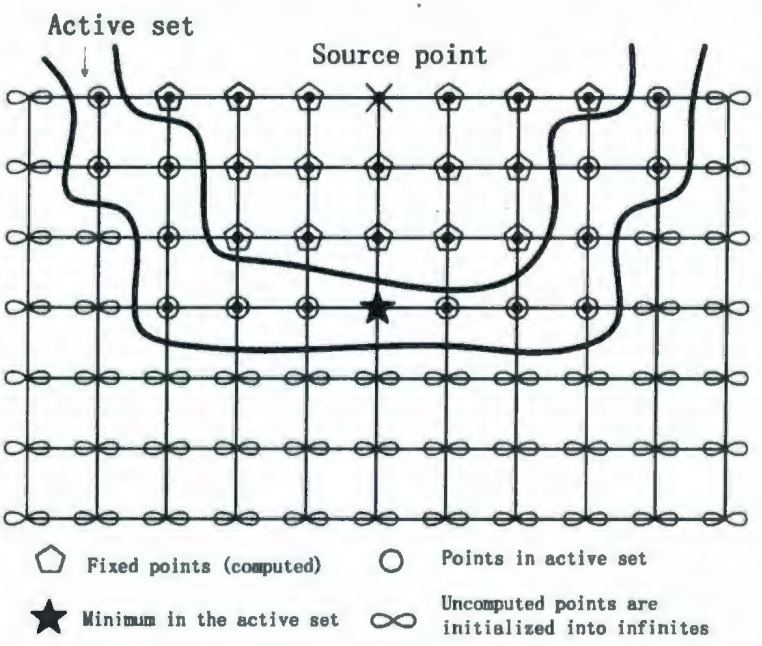


Figure 2.11: Wavefront evolving.

second set is active set. The points in active set are computed; however, before writing them into fixed set, all the values in the active set have to be sorted. In data structure, all the active set values are put into a heap; and the minimum should be always returned to the top of the heap by applying certain sorting algorithm in the heap. The active set actually contains points around the evolving wavefront. With wavefront evolving, active set moves forward to sweep across the whole field, and this is why it is called "active". The points denoted with "∞" are uncomputed points; before computation evolving to these points, they keep their initialization value "infinity".

In Fig.2.11, the minimum travel time point in the active set is marked off with a star sign. This point will be the starting point of the next computing round. Because seismic wave always attempts to travel through the point holding the minimum travel time, so the computation always begin with the current minimum travel time point

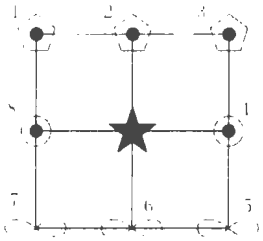


Figure 2.12: Eight neighbor points.

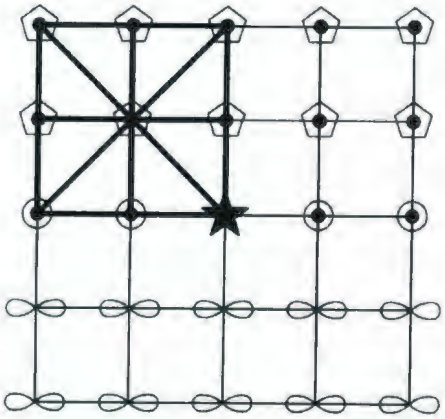
in the active set. This way follows the least travel time principle, and it is a mimic of the phenomenon in the nature. For Vidale's method in which the computing is from local minimum to local maximum along an edge of a square. It is obviously not the way the wave would advance in the nature, and would cause some causality problems.

After determining the minimum travel time point in active set at beginning of a computing iteration, the 8 neighbor points can be found out based on the coordinates of the starting point as Fig.2.12 shown. These 8 neighbor points can be the points in either fixed set, active set or uncomputed set. On each neighbor point, a rice computing is applied, which is introduced in last section, and the steps can be shown as Fig.2.13.

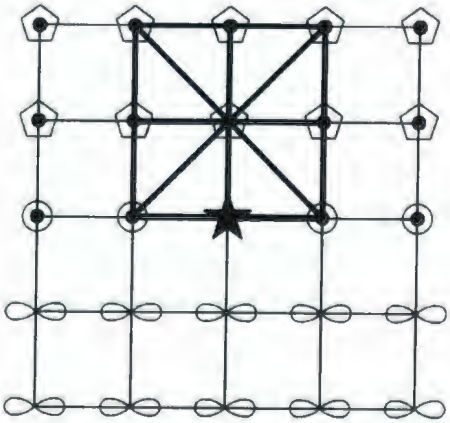
2.2.1 Soundness of the result

In a single triangle of a rice computing unit, if the two input points t_a and t_b are in fixed set or active set and none of the paramters t_a t_b and hs is infinity, the result for t_c is said to be **"good"**. In contrast, if either of the two points t_a t_b and hs is infinity, the result from the triangle will be infinity. For some rice computing, all 8 triangles output infinities for the results, and therefore the result from rice computing unit is infinity.

If a rice computing unit outputs a "good" result and the result point is currently

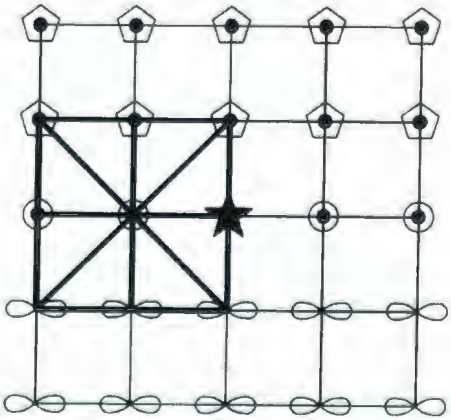


"Rice computing" for neighbor point 1.



"Rice computing" for neighbor point 2.

... ...



"Rice computing" for neighbor point 8.

Figure 2.13: Apply rice computing 8 times.

not in the fixed set and active set, this point should be written into active set, and pushed into the heap to be sorted for future use; otherwise, if the result of rice computing is infinity which is not "good", the result point should stay in the uncomputed set.

After all 8 computations in Fig.2.13 finish, the start point which is the current minimum in the sorting heap should be put into fixed set. The algorithm repeats this process iteratively until there are no points left uncomputed in the field.

The 8 rice computing on 8 neighbor points can be sequentially done one by one, as shown in Fig.2.13. In this case, one rice computing unit is reused all the time. For the other option, all 8 rice computing units are built in parallel, and all 8 computations take place at the same time. This second solution is difficult to implement on a single processor platform, however, on the platform such as FPGA, the computing unit can be built on the same chip to exert the parallel potential of the hardware platform.

2.3 Summary of Algorithm Flow

Based on the previous derivation, the algorithm can be summarized into following steps:

1. Initialize the elements in the travel time array to be infinity, and assign a set of seed points to the travel time array and sorting heap.

2. Pop the minimum off the heap as starting point.

3. Calculate the coordinates for the 8 neighbor points of the starting point.

4. Apply rice computing on each neighbor point based on local extrapolation scheme.

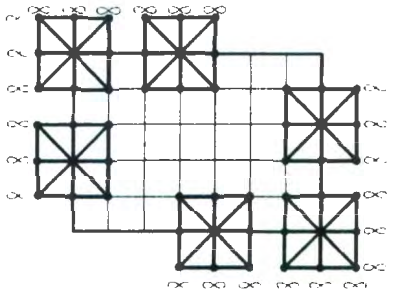


Figure 2.14: To compute the point on boundary, assign infinities to the points beyond boundary as inputs of the "rice compute unit".

5. Write the least time computing results into active set (push into the heap to sort), and put the starting point into fixed set.

6. Repeat from step 2 until all the points in the field are covered.

There are several points to be noted when going through these steps.

For the points on the edges of the plane, rice computing cannot be directly deployed on them, because they do not have exact 8 neighbor points (as Fig.2.14 shown). The solution is to set an edge detection mechanism: each time when getting a minimum from sorting heap as starting point, the coordinates of its neighbor points will be calculated and examined. If any index appears to be negative, the value of the point is set to be infinity. Thus, this assigned infinity will generate infinity as result through "rice computing". Obviously, infinity cannot win through the minimum electing during rice computing, so it is indeed no impact to the computing. However, by adding these extra infinities the same pattern of the rice computing unit with 8 input points can be used for all grid points including the ones located on the edge of the field. In this way the digital logic can be simplified for implementation on FPGA's.

The computed points have to be distinguished from uncomputed points. The computed points include the points in fixed set and active set: they are all behind the wavefront. An "flag bit" array which has the same size of the travel time array is set, in which each element only has one bit in it. Uncomputed point has its flag set to be '0', while '1' for the computed one. Each time when the algorithm reaches step 5, the results can be written back to corresponding set after checking the flag bit.

In this way, by jointly using ∞'s and flag bit array, there is no need to design specific logic to detect boundary points on the edge of the domain.

2.4 Fully Parallel Algorithm

2.4.1 Problems with Sequential Dependency

There are several key parts that should be included in the system to implement the algorithm: the travel time array, velocity array, flag bit array, rice computing unit, sorting heap and corresponding memory access control modules. One of the characteristics of this system is that the execution of the algorithm steps have to strictly follow certain order: Though, as mentioned in previous section, there can be some parallelism at small scale in this version of the algorithm. For example, implement 8 rice computing units at the same time, instead of using only one every time. However, the sequential dependency at top level of the algorithm makes it difficult to parallel the computation of several rounds together into one round. Each iteration of the computation has to go through all the 5 steps sequentially. Furthermore, the sorting process which is time and resource consuming is inevitable in every computing iteration. Moreover, there are too many memory accesses happening in one computing round to fetch and write back the parameters and inputs from memory array.

This strong sequential dependency and frequent memory access may not have too much impact on the sequential program designed with C or Fortran; however, it can be really a disaster for digital designs facing ASIC's and FPGA's. Because as we known, platform such as FPGA's have preference on the design problems that can be solved by using the design patterns in style of cell networks or pipelines. On the contrary, problems with strong sequential dependency are usually put into programs using microprocessors. Strong sequential dependency can prevent pipelining the algorithm on the FPGA's.

However, in this thesis, this sequential algorithm is still implemented in FPGA's. This is because our research is exploratory, we do not have enough chip resources to develop a mature commercial product; moreover, it is necessary for us to first apply the classical and mature digital design methodology to make an example implementation to evaluate the complexity and feasibility of the solution for future commercial use.

2.4.2 Hardware/Software Co-design Solution

There are several technology trends appearing in recent years' of FPGA development. Hardware/Software co-design (H/S co-design) is one of these issues, and it can be an ideal solution to our sequential algorithm.

Nowadays, all of the several mainstream FPGA production companies provide soft core processors [25] which can be implemented into their FPGA's as common digital design modules. There are no further external interface problems need to be handled by the users. Integrated development environment (IDE) [26] [27] program on the PC allows user to customize the soft processors with specified features. The soft core of processor can be implemented into VHDL models, so the difficulties of initialization

the processor is restricted into the VHDL language level. This technique allow users to flexibly specify the microprocessor which is best suited for their application, and easily design peripheral modules working jointly with the microprocessors. This solution can fully take advantage of the on chip FPGA resource (user can decide capability of the processor and configure it), and provide flexibility to solve the problems with high sequential dependency, such as our algorithm. In the system design, the portion which is highly sequential can be written in high level language that can be compiled into the soft processor to avoid complex FSM (Finite State Machine); and for other logical parts the traditional VHDL design methodology is kept, and used to achieve function parallelism and high efficiency.

2.4.3 Network on Chip

To further raise the computing efficiency and improve the feasibility of the FPGA implementation, a modified version from the sequential algorithm is proposed. This new method originates from the idea of **network on chip** (known as NoC, Fig2.15)[28] [29], though itself is not strictly following the NoC paradigm.

In this new algorithm, the local extrapolation scheme from sequential algorithm is kept. In other words, rice computing is still used for extrapolation. However, instead of using one or eight rice computing units, the rice computing unit is put on each grid point in the array. In this way, there is actually an array of rice computing units working jointly together at same time for each computing iteration(see Fig.2.16).

Every time the rice computing unit on each grid point computes in its own 8 triangles according to formula, and elects the minimum among the 8 results. Because all the units behave in the same manner that they only take care of their 8 neighbor points in vicinity, the computing will be passed layer by layer outwards, just as

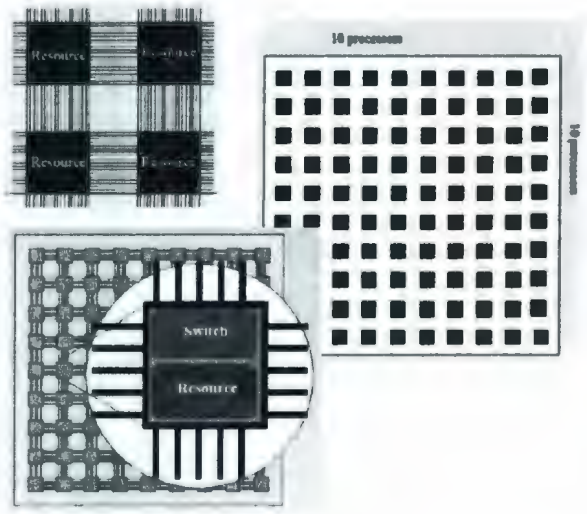


Figure 2.15: Network on chip.

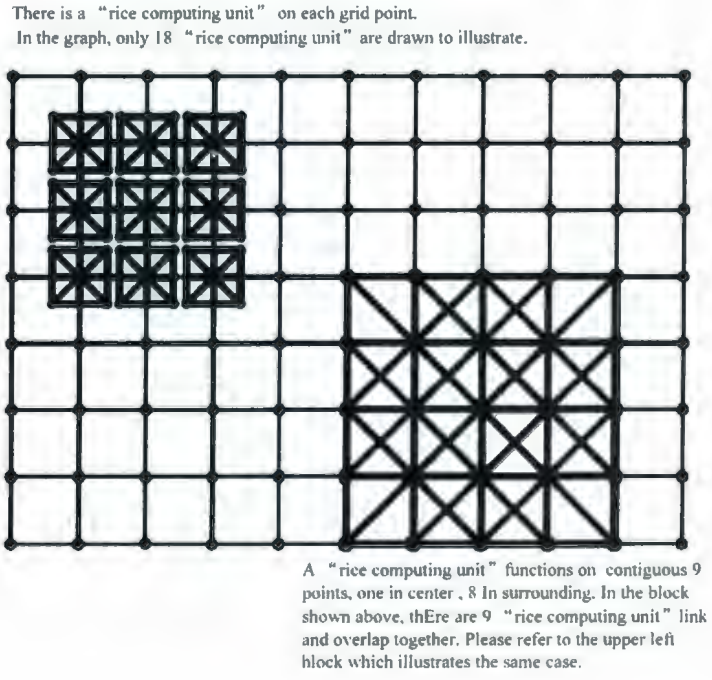


Figure 2.16: Rice computing unit array.

wavefront extending. Because on the hardware platform, such as FPGA, there can be many computing cells running independently and simultaneously; this computing array pattern can exert the potential of FPGA's. The only problem now is how many computing cells can the FPGA hold.

With the system structure shown in Fig.2.16, the algorithm flow as well as the control logic can be largely simplified, because it does not need to explicitly trace the evolving wavefront by keeping an active set in this algorithm. There is no need to build the challenging sorting heap, this saves a lot of design work, and reduces the size of the complicated system. All one needs to do in this parallel algorithm is to allow updating the rice computing unit array round by round, and the wavefront will evolve. The necessary system modules only include rice computing unit array (may include thousands of rice computing units), and velocity input array. The work flow of this fully parallel algorithm can be summarized into following steps:

1. Initialize the elements in the rice computing unit array to be infinities, and assign a set of seed points.

2. Update the array by running all rice computing units round by round.

3. Repeat step 2 until all the points in the entire field is computed.

2.4.4 Feasibility Discussion

The major difficulty of this fully parallel solution is its consuming need for FPGA resources. If the computing is operated in precision of single precision floating point number, even the high end FPGA platform such as Xilinx Vertex-5 can only hold dozens of computing cells which evaluate the formula Eq.2.8, Eq.2.13 and Eq.2.14. It means one FPGA chip can only process the computing on dozens of grid points.

nevertheless, in practical application the grid size can be hundreds by hundreds or even larger.

In the past, this kind of solution is unthinkable, because it is too expensive to implement. However, in recent years, the density of IC chips has risen, and the price of FPGA product becomes more and more affordable. In this context, some researchers began experimenting FPGA arrays. This is a new idea raised in the field of reconfigurable computing[30] [31] [32] [33]. It adopts multiple FPGA's to form an array: each FPGA chip takes responsible for computing in a block, and there are communications between FPGA chips. This idea is pretty close to the parallel programming technique such as MPI (Message Passing Interface); the difference is that control logic is directly built on the chips with digital circuit rather than software programming with high level language in an operating system.

This technique has its many advantages.

1. Obviously, this method is fast. It is "genuine" parallelism that you can have many processing cells working at the same time. The computation which needed a bunch of loop statements in software programming; now can be finished in one time updating.

2. The control logic is relatively simple and robust. Designer may not need to design complicated FSM to reuse limited resource. Instead, more effort is put on the network switching design.

3. The FPGA array can be a scalable design. According to different problem size (for example, different grid number in various applications,) user can add in or remove chips to fit the size of the problem. This solution is best suited for linear extendable problem in the research field such as graphics and scientific

computing. Our finite difference method can be an ideal application of this solution.

1. This is method is also economical. FPGA's are actually affordable device for many research groups compared to ASIC's. Especially, the reconfigurable feature of FPGA can make the same platform be used for different problems after simply adjusting.

As collections of FPGA processing systems grow in size, the feasibility of large rice unit array for travel time computation will be realized.

2.5 3D Extension of the Algorithm

Geophysics imaging technology is advancing toward the direction of 3D. Nearly all the research on algorithms in this field would consider the expansibility to 3D. Vidale, Sethian as well as other researchers published their 3D version algorithm one or two years later after the initial proposal. A critical quality of a good algorithm is that it should be easily upgraded to 3D without bringing in too much complexity.

For our algorithm, it is extendable. The 3D version of it can be discussed from following three aspects.

1. For the local extrapolation scheme, this 3D version can inherit rice computing derived in section 2.1. The same rice computing unit can be put on the xy, yz, xz coordinate planes intercepting at the grid point. There will be 3 rice computing units with 24 triangles (see Fig.2.17).

   However, because in 3D, there are 26 points surrounding 1 point in the center. So there can be more triangle relations developed among these spatial points (see

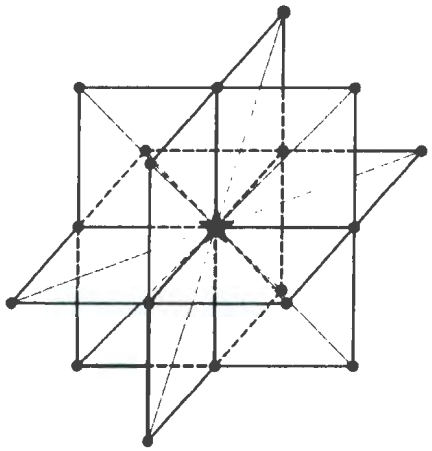


Figure 2.17: Rice computing unit in 3D.

Fig.2.18). These triangles have different shapes and angles with the triangles inside rice computing unit. Mathematics has to be developed for these triangles again based on the same geometric principles and derivation procedures.

Because there are more computing triangles in a 3D cube than in a 2D grid, the single "computing cube" is more complicated to build. However, it may be not necessary to use all of the triangle relations among the contiguous grid points. The trading off between complexity and accuracy by choosing right triangles to compute is a key point in this 3D algorithm design.

2. For inductive scheme, the 3D algorithm can fully inherit the scheme from 2D version. Computing can evolve following the steps described in section2.3. However, the active set becomes a surface rather than a contour in 2D case. There can be many more points to be sorted in the sorting heap. The intensive storage demand and large scale sorting operation can be challenges in building 3D algorithm.

3. For fully parallel algorithm, the thinking of parallelizing many "computing

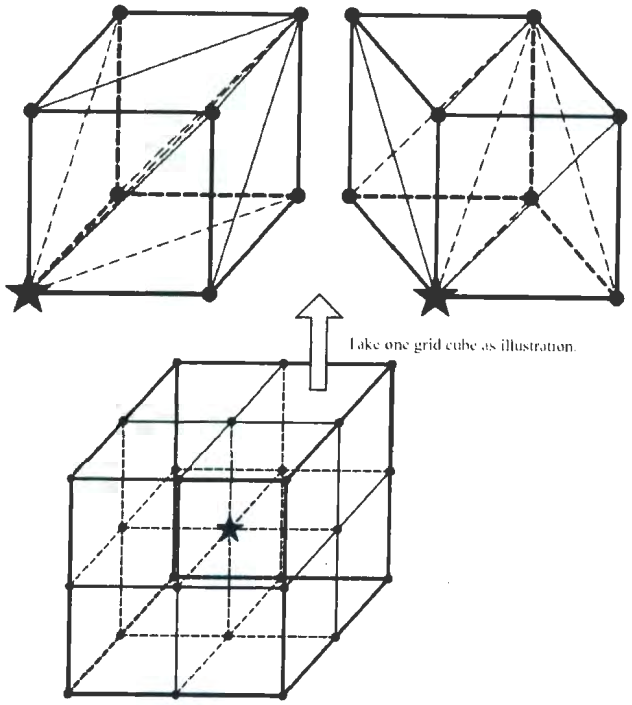


Figure 2.18: Triangles among spatial points.

cube" on grid points can be transplanted into 3D case. However, it has to face the same problem caused by the intensive expense of building 3D "computing cube". Even one of the "computing cube" units includes too many arithmetic operations. The capacity of the FPGA is too small to hold many of this "computing cube" units. In this case, further optimization should be done on the design of digital arithmetic operations.

Here, the theoretical fundamentals of the algorithm is developed for 2D. In following chapters, the software and hardware implementation issues will be discussed.

Chapter 3

Software Simulation and Parallel Program Design

In the previous chapter, the travel time cellular algorithm is developed from mathematic concepts to execution procedure. The algorithm can be implemented on all kinds of platforms. Respectively, each implementation solution has its own features, so algorithm details may need to be modified to fit certain data structure and data manipulation flow. In this chapter, the algorithm is developed to run in high level programming languages. The languages can be Fortran, C or Java, and the implementation platforms can be PC, PC network or multi-core cluster.

There are two purposes for running the software simulations: First, high performance programming is still the major measure to solve the problems in the field of scientific computing. It is necessary to discuss the software solution for a newly developed algorithm. Second, software simulation is usually applied to verify the algorithm. The languages such as Java and SystemC are object oriented; they are good to model the physical software or hardware modules in the working system.

In the project, MATLAB is used as the simulation language. MATLAB is not

an object oriented language, but program is organized into function blocks which corresponds to the modules constituting the system. In the first section, I will give an overview of the modules in the system, to investigate the roles they play and how they are combined together.

3.1 System Overview

There are 2D data structures holding three arrays in the system: travel time data matrix, also called "main matrix", stores the travel time values on each grid point, and it is an abstraction of the meshwork; velocity matrix is a parameter matrix provided outside the system as velocity model input, and it has the same size as the main matrix with each element representing the velocity value on a grid point. Actually, it can save the velocity or directly save the parameter hs which is the product of grid spacing and slowness (the reciprocal of the velocity value); flag bit matrix has one bit for its each element to label whether the grid point is computed (in fixed or active set) or not.

Sorting Engine

Sorting engine is a heap, holds and sorts the travel time values in active set. At the beginning of each round of computing, the sorting engine pops off the coordinates of the minimum travel time point from active set to be the starting point. Because it requires "starting point" to be chosen among a number of points in every computing round, the sorting efficiency is critical to the total performance of the system.

Rice Computing Unit

The rice computing unit is built by using the arithmetic functions provided by MATLAB. The default operation precision in MATLAB is double precision floating point number. Rice computing unit receives the starting point coordinates from sorting engine and calculates to find out the neighbor points of the starting points. As mentioned in previous sections, the rice computing on 8 neighbor points can be done together or separately in sequence. "To be done together" means that the computing unit loads the data block of 25 grid points required by 8 rice computing at once (see Fig.3.1), this behavior is an analog of the parallelism in hardware circuit; while "to be done separately in sequence" means loading in 8 grid points from main matrix to feed one rice computing unit at a time, and repeat this loading and computing for 8 times to compute the 8 neighbor points. In our program, the former one is chosen to be implemented. The 8 "computing triangles" in a rice computing unit is running the formulae Eq.2.8, Eq.2.12 and Eq.2.13.

System Structure

Assembling the modules together, the system can be shown as in Fig.3.2. There is a data flow going through every part of the system for each iteration. The computing begins with the release of a starting point from sorting engine. According to its indexes, the indexes of the points surrounding this starting point can be calculated and the whole block of 25 travel time data on grid points in vicinity of the starting point are fetched. This data block with 25 travel time values are fed into the computing unit. As shown in Fig.3.3, these 25 points may belong to different sets: some of them are computed points in the fixed set or active set, which have already had computed travel time values attached to them; while some of the points are still uncomputed,

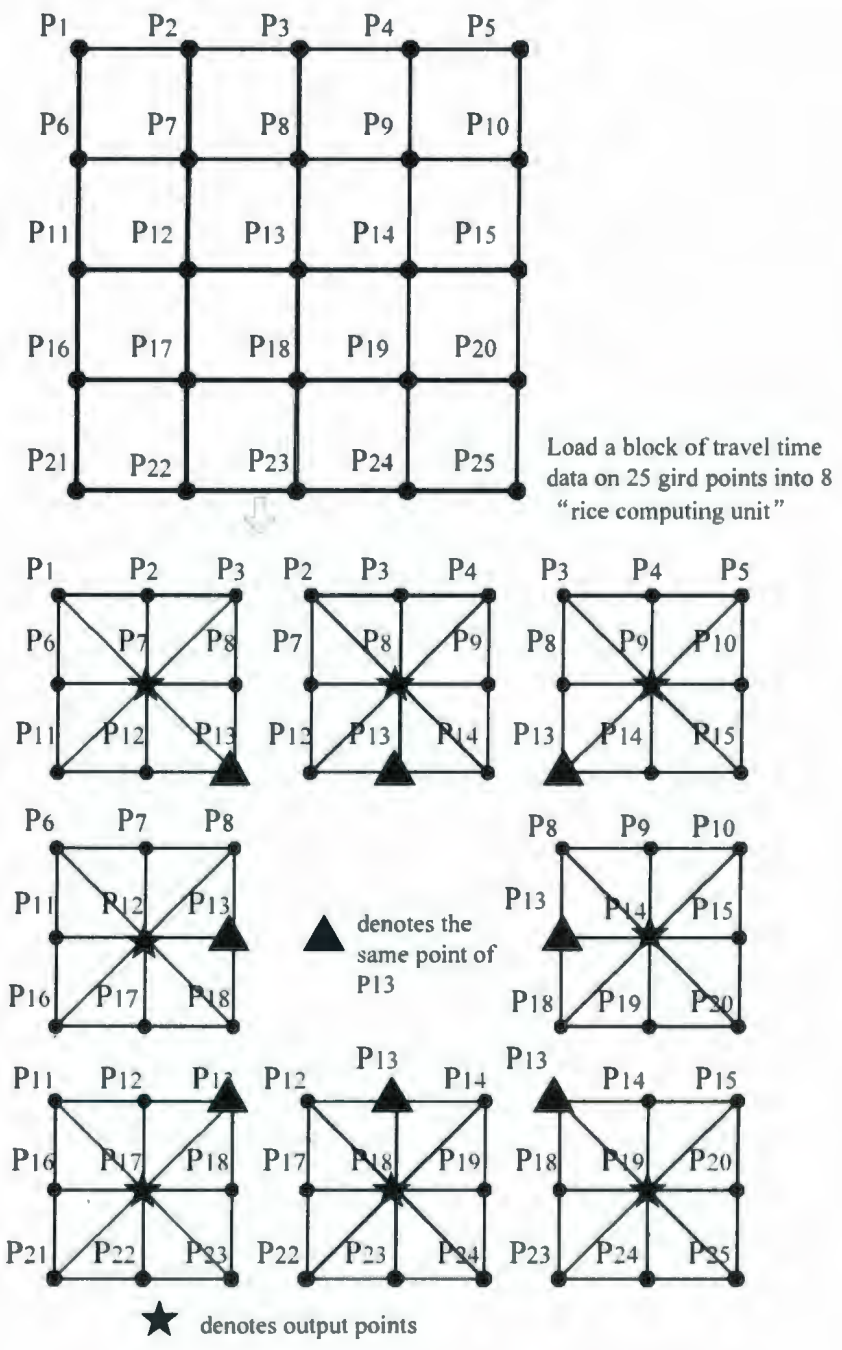


Figure 3.1: Load 25 points into 8 rice computing units.

and hold the values which are initialized as infinity. However, no matter whether the points are computed or not, the data block will be loaded into the computing unit from the travel time array. Meanwhile, the velocity values on the 8 grid points that locate on the center of the 8 rice computing units are also loaded from velocity array. In the computing unit, there are 8 rice computing units (in a single CPU program, we actually reuse one rice computing algorithm 8 times). The mapping of the points to input ports is shown in Fig.3.1. In a rice computing unit, when computing Eq.2.8, Eq.2.12 and Eq.2.13, the velocity of the center point is used by assuming the velocity is a constant in the small vicinity of the 9 grid points. The results coming out of the "rice computing unit" are the new travel time values on the 8 neighbor points of the starting points. Flag bit array can provide the information on these points to tell whether they are computed or not. Finally, the good[1] results are written back to travel time array and pushed into sorting heap; the elements in the flag bit array will be updated as well. As far as this point, one round of the computation is completed, and the next round begins with the release of the next starting point from sorting heap.

The relation between these function modules in the program can be described by class diagram. Class diagram is the design assistant tool for object oriented programming. It can describe the structure of a system by showing the system's classes, their attributes, and the relationships between the classes (see Fig.3.4).

[1]The meaning of "good" is explained in the section 2.2.

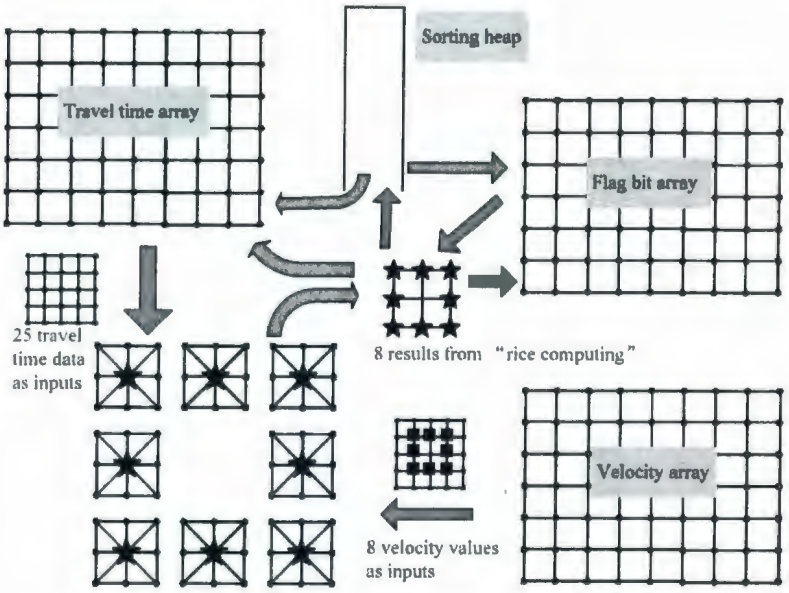


Figure 3.2: System assembly: there several major modules in the system as illustrated, the data flow between modules are illustrated with arrows.

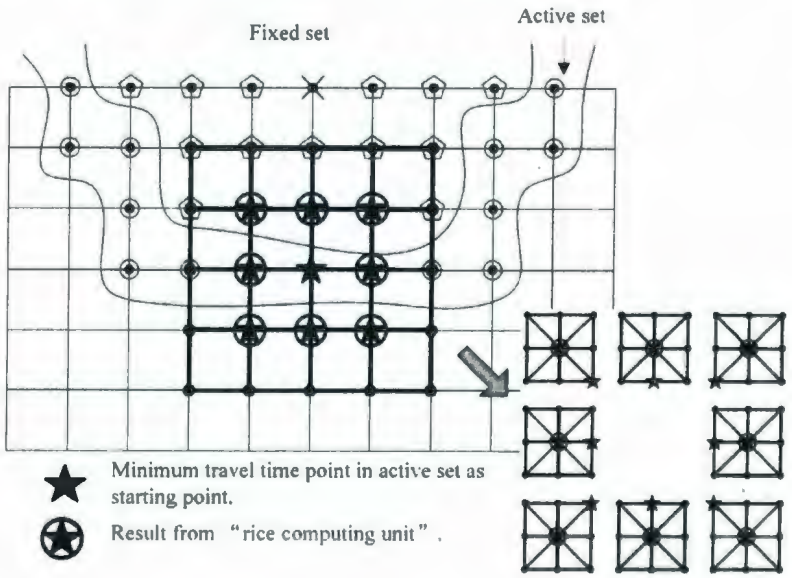


Figure 3.3: Process of extrapolation using 8 rice computing unit from the minimum travel time point in active set as starting point.

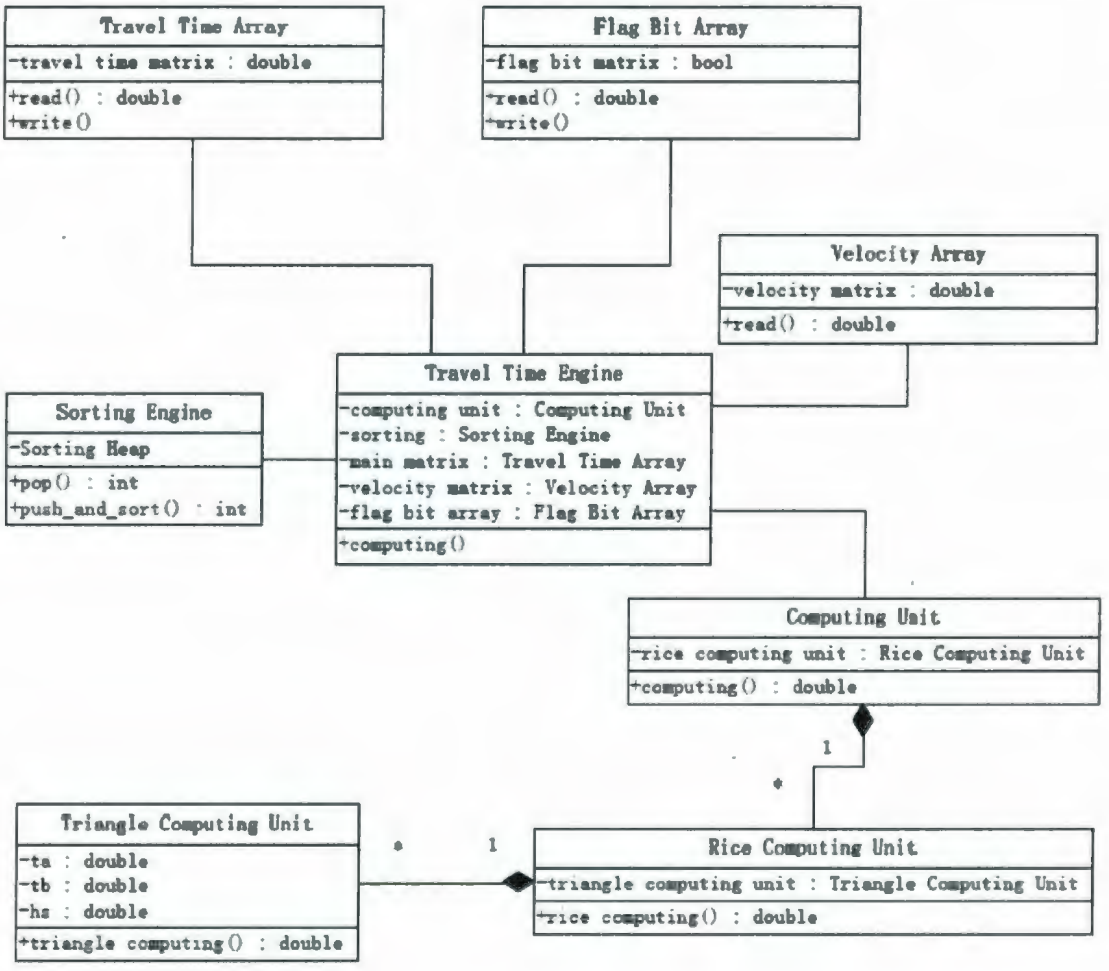


Figure 3.4: Class diagram.

3.2 Module Implementation

3.2.1 Computing Unit Implementation

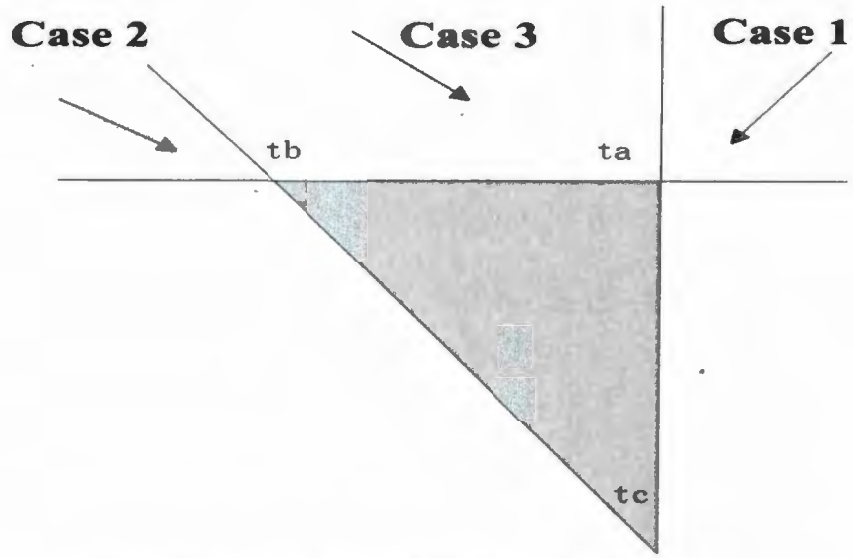
A "rice computing unit" is composed of 8 "triangle computing unit" as shown in Fig.2.7 and Fig.2.8 in the section of 2.1.2 and 2.1.3. Each computing triangle can only take care of incoming rays in 15°range. There are three formulae in a row to perform the computation inside a triangle. These three formulae deal with three branches according to computing effective intervals as the MATLAB code shown in Fig.3.5. The last step in rice computing is to pick up the minimum among the 8 results as the final for the rice computing unit.

3.2.2 Sorting Engine

The sorting operation has to be performed in every working iteration. Therefore, the efficiency of the sorting is essential to the whole system performance.

Sorting Algorithm Selection

To choose a proper sorting algorithm in our implementation, the application scenario of the sorting has to be considered. In our algorithm, the data to be sorted are pushed into the heap at the end of each computing iteration, and the number of input keys can be 8 at most (there are 8 results generated from 8 rice computing units). At the beginning of the iteration, the numbers in the sorting heap should be sorted, or at least the minimum in the heap should be decided. The computing requires this minimum to be the starting point. The input of the sorting is not a whole array of numbers which are given at once together. Instead the numbers to be sorted are given step by step. This is a typical **online sorting**.




```
function [ tc ] = triangle(tb, ta, hs)
% computation in each triangle
% h is the length of the grid side;
% s is slowness the inverse of velocity;
If(ta== Infinity ||tb== infinity)
% dealing with uncomputed inputs
    tc=infinity; % a large enough number

else
 if  ta < tb                          ①
    tc = ta + hs

 elseif ta > tb + √2hs                ②
    tc = tb + √2hs
else
    tc = ta + √(h²s² − (ta − tb)²)    ③
end
end
end
```

Figure 3.5: Computing triangle formulae.

Several algorithms can be used in online sorting as introduced in the last section. However, the insertion sort is finally used for our design. There are reasons for doing this:

1. The algorithm such as merge sort can also be used in online sorting. However, it is difficult to exactly estimate the number of elements to be sorted for our algorithm. The merge sort is too complex for a large amount of inputs. Online merge sort need the other sorting algorithm to pre-sort the input pieces, and then merge it into a final list. This approach can be expensive in time and space if the received pieces are small compared to the sorted list.

   At this point, our insertion sort which is based on a high speed bus in hardware implementation can be easily scalable in length. The extension of the heap can be done by appending cells after rear of the heap, rather than adding merge stages in the merge sort.

2. Insertion sort is relatively efficient for mostly sorted lists. In our travel time computing, with the wave front evolving, the travel time values are actually getting larger. The incoming numbers to the sorting list is roughly in an order from small to large.

3.3 Stop Criteria

In the system, once started, the computing is going on iteratively round by round, and wavefront is evolving outwards from the initial set. The question on when the computing finishes must be answered: there should be a stop criteria.

As introduced previously, before computing started, the travel time matrix has to be initialized into infinity. So one way to decide whether computing finishes is

checking whether there is any infinity left in the travel time matrix. We can exam the travel time values on the grid points which are on the edge of the computing field, because upon the causality, these points should be computed at last.

The other way to decide when to stop the computing is that a number of iterations can be simply set. This number does not need to be exact. For example, if running the program on a computing field contains 100×100 grid points, the number of iterations can be 10000. In this way, there would not be complex logic involved in judging whether the computing finishes in each round of computing. This way works best for testing the system.

Another way of program termination is based on least time points in the sorting heap. If the sorting heap is empty, the program is finished.

3.4 Simulation Result

The algorithm runs in different test velocity models for verification. For the first test case, the simulation runs in the simplest constant velocity model, in which the velocity is constant on each grid point. The result output from the program is a travel time value matrix. Contours can be plotted according to the values of the matrix. Fig.3.6 shows a set of travel time contours, in which the source point is assigned on the ground in the middle of the range.

One reason to run the simulation in constant velocity model is that it is possible to calculate the exact travel time values on each grid point in the constant velocity field by using the Pythagorean proposition. In this way, the results from our algorithm can be compared to the exact travel times, to further evaluate the error terms. The error distribution for the travel time matrix in Fig.3.6 is shown in Fig.3.7. The error is a percentage calculated with the formula Eq.3.1.

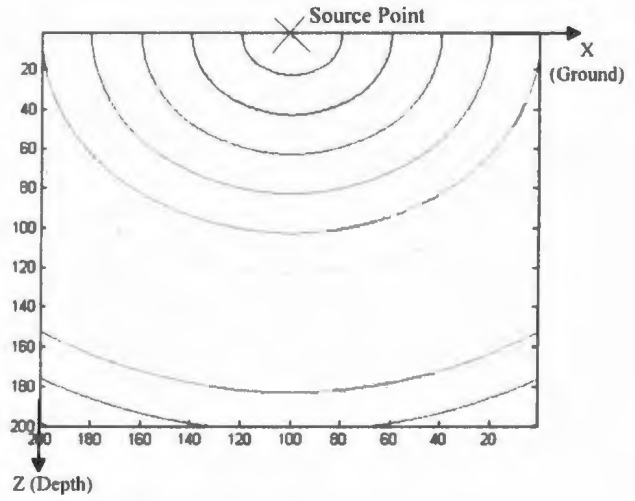


Figure 3.6: Travel time contours.

$$err = \frac{T - T_{exact}}{T_{exact}} \times 100\% \tag{3.1}$$

It may be noticed that the percentages for the relative errors are high on the grid points which are close to the source point, while they are decreased upon the increasing of the distance to the source point. The points at the far end from the source point have lower errors. This seems against common sense that the numerical errors should be accumulated to increase at far-end of the computing field. In fact, the reason is that the formulae Eq.2.8, Eq.2.13 and Eq.2.14 are directly derived from the geometry. It is based on the high frequency approximation of the seismic wave. In the algorithm, the wave path is taken as the ray path without considering too much about the curvature of the wavefront. Indeed our formulae are derived for the plane wave. In the region close to the source point, the curvature is high (see Fig.3.6, the contours at close-source region have higher curvature). So our formulae work relatively bad on these points. However, after the wave spreads out into the computing field, the curvature becomes lower, and our formulae give a better accuracy.

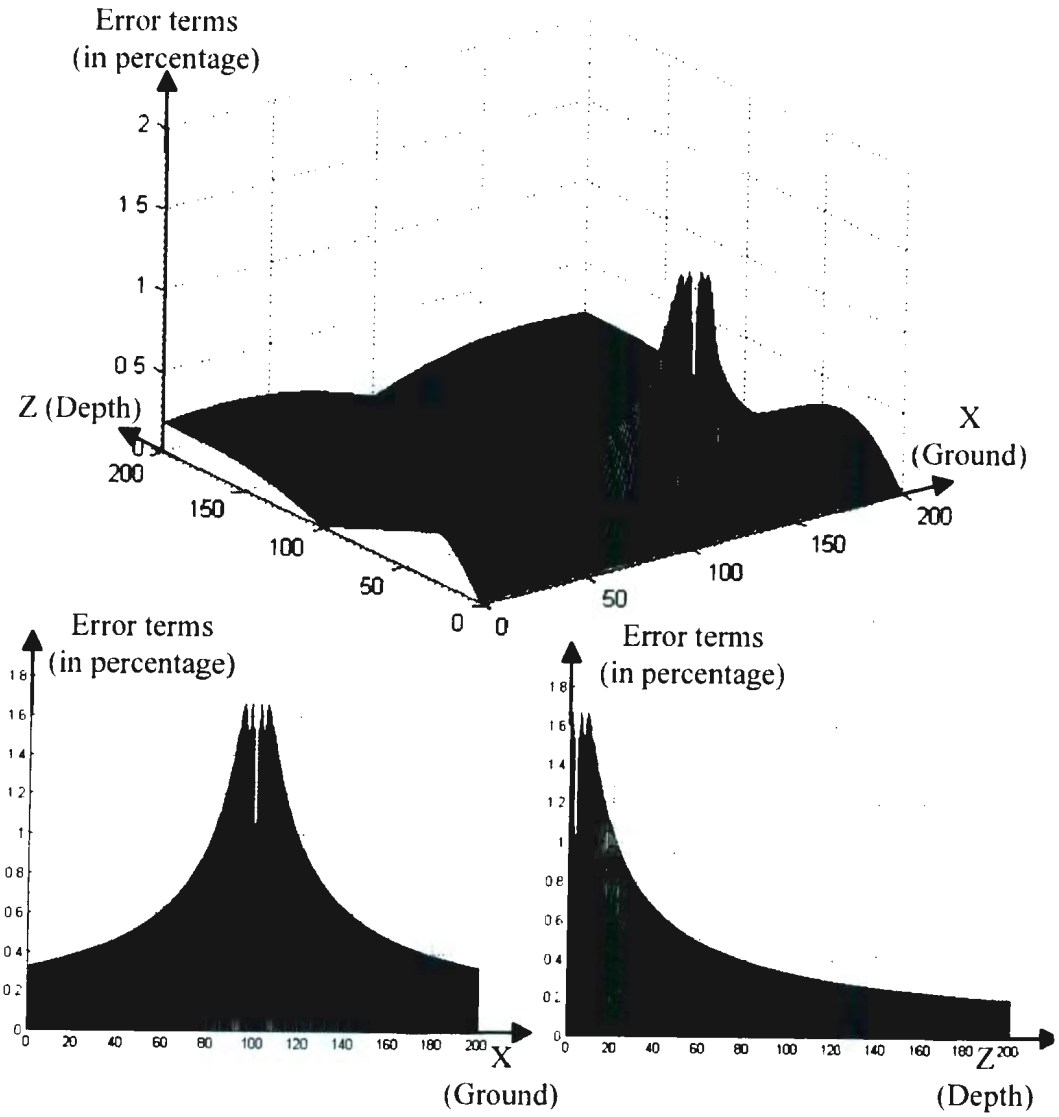


Figure 3.7: Error plot generated by comparing the result from our LTPFM method with the exact travel times using Pythagorean Proposition (Errors in percentage).

For these computing errors, a "mix scheme" can be used to improve the accuracy. As introduced in previous section 1.2.5, in Vidale's local extrapolation scheme, Eq.1.6 is developed to deal with the high curvature situation. In our algorithm, this Eq.1.6 can be adopted by assigning certain region which is close to source point. In this region, the Eq.1.6 is applied; while in the far-source end, our formulae Eq.2.8, Eq.2.13 and Eq.2.14 are used. In this way, the "mix scheme" can work fine in everywhere in the computing field.

In section 2.2, we have introduced the timing error divided by the transit time across the cell as Eq.2.14, which is proposed by Vidale in [5]. This error is a measure of the computing accuracy and the effect of finite difference meshing (whether the grid size h is properly assigned). The plot of our results at h 10 $meter$ and $velocity = 1000\ meter/second$ is shown in Fig.3.8.

To further verify the algorithm, two source points are assigned at underground positions in a constant velocity model. The correct resulting contours should be two set of symmetric circles, as shown in Fig.3.9.

Finally, the program runs in a more complex velocity model Dablain model as in Fig.3.10. After running the program on this model, the result contours are shown as in Fig.3.11.

3.5 Program design for fully parallel method

For the simulation of fully parallel algorithm, because the sequential programming platform cannot execute all the rice computing in real parallelism, "for loops" are applied to reuse a rice computing unit iteratively. Fig.3.12 illustrates the structure of the program.

There are two travel time arrays in the system. The data are read from one input

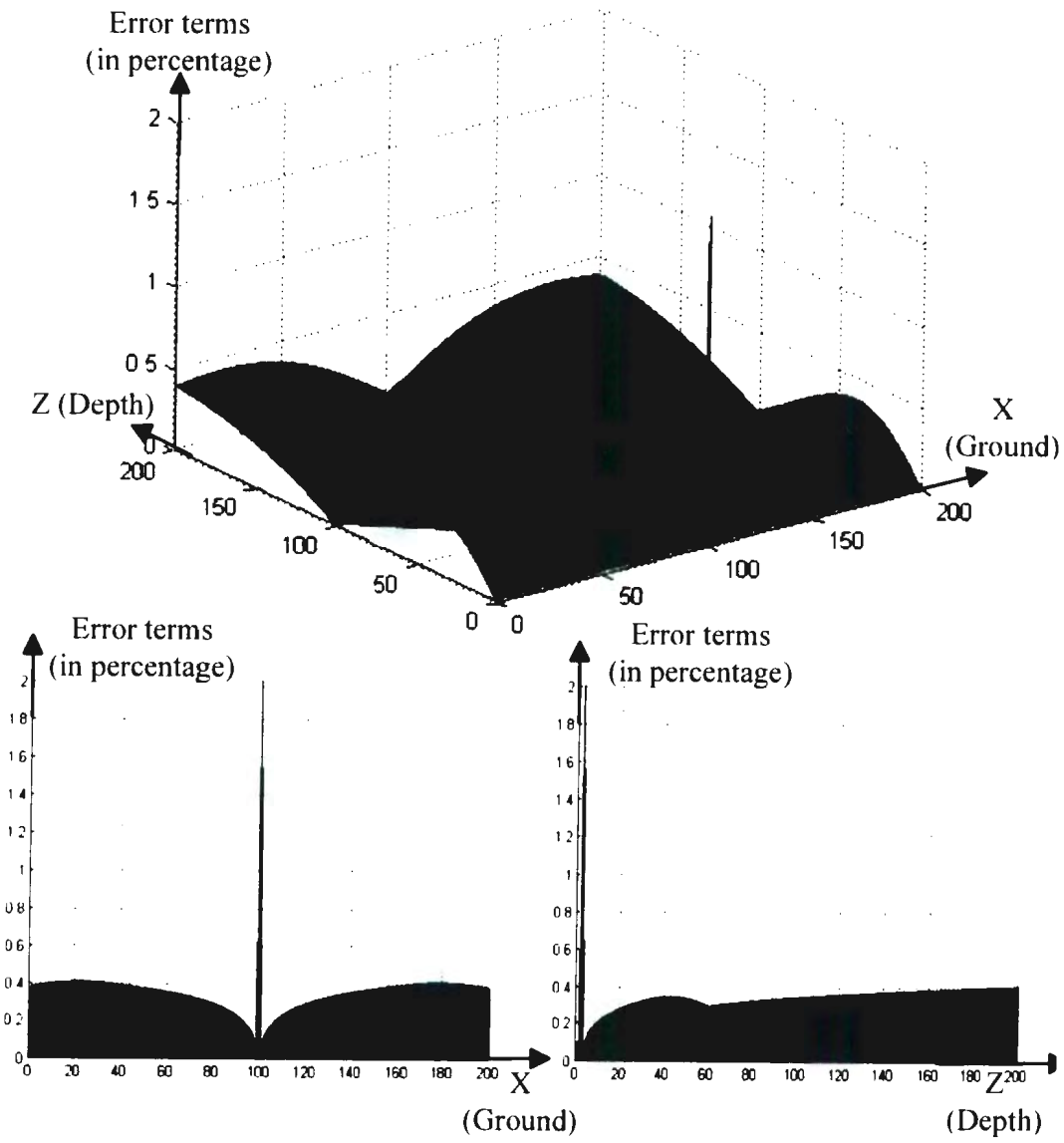


Figure 3.8: Timing error divided by the transit time across the cell. $h = 10$ *meter* and $velocity = 1000$ *meter/second*.

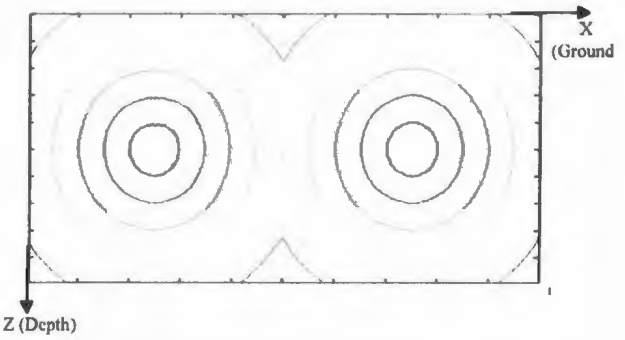


Figure 3.9: Two source points experiment.

matrix as the input to feed into rice computing unit, and the result are written into other matrix. At the end of the computing iteration, the result matrix is copied back into the input matrix. In this way, it can simulate the computing process on the parallel hardware platform with thousands of rice computing units, that the input matrix won't change during one round updating (computing iteration); though there is only one computing unit in this simulation program.

There is no doubts that MATLAB implementation of this method will take much longer running time than the sequential version of our algorithm. Because in each updating iteration, rice computing has to be repeated on every grid point no matter whether the computation is useful. However, imagining in hardware implementation, each updating round only spends the time of running rice computing for once without any further time expense, thus, the algorithm will be really fast.

3.6 Simulation Result for parallel algorithm

In a constant velocity model, the parallel program outputs the contours as shown in Fig.3.13 (at this time, the source point is located at the center of the research field).

The error term distribution (in percentage) comparing to exact solution can be

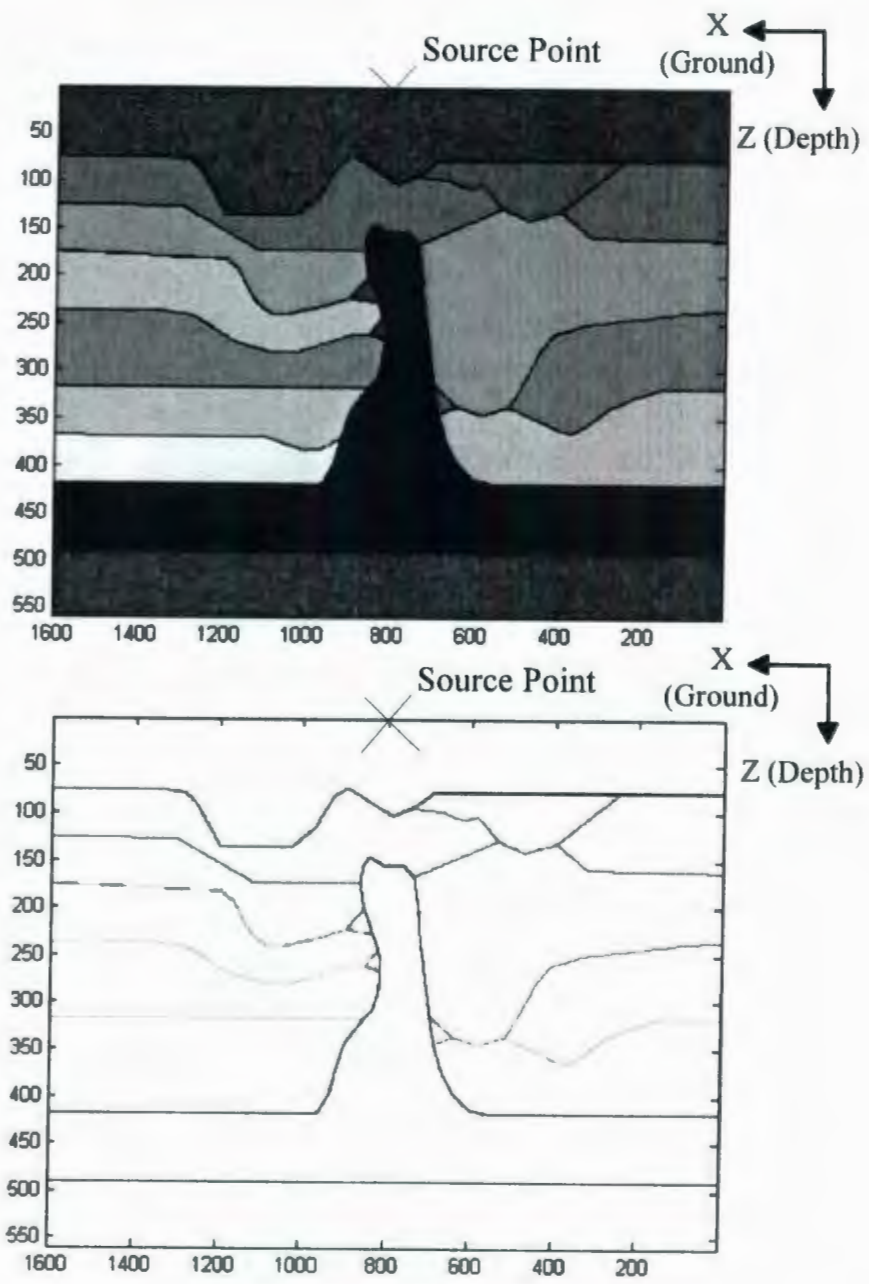


Figure 3.10: Dablain model.

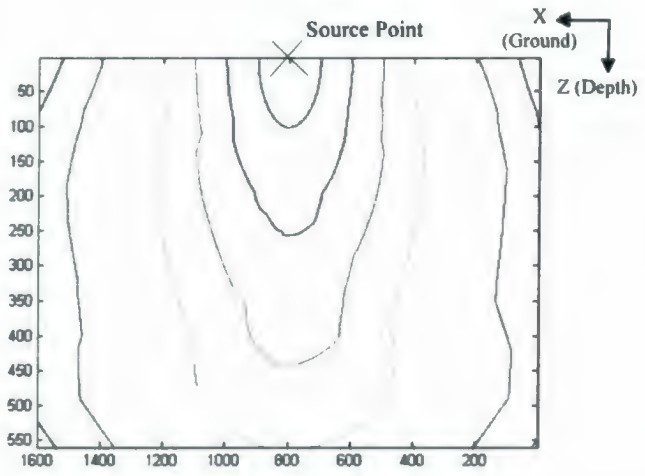


Figure 3.11: Travel time contours on Dablain model.

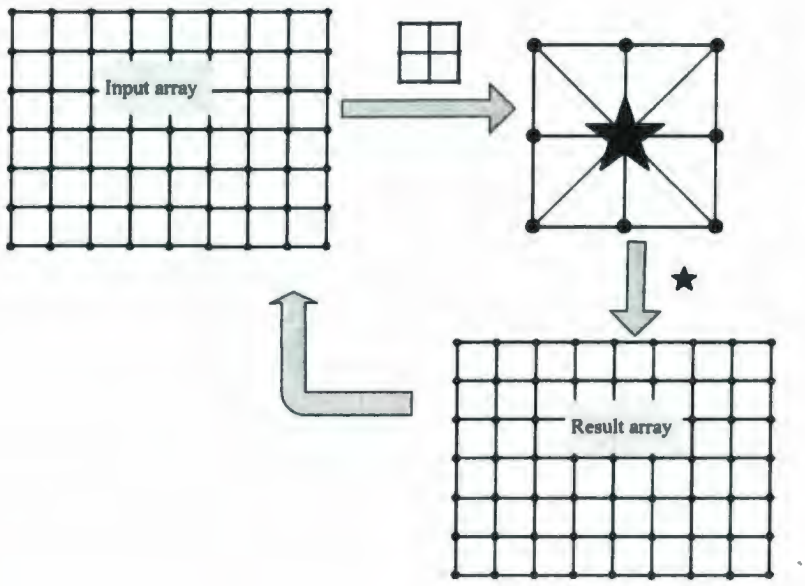


Figure 3.12: Structure of the fully parallel program.

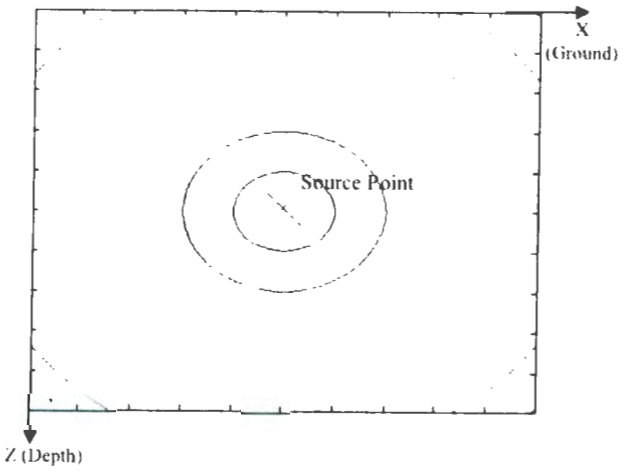


Figure 3.13: Travel time contours in constant velocity model run by parallel program.

shown as in Fig.3.14.

The travel time contours resulting from double source points experiment is look like this in Fig.3.15:

Finally, the parallel version of the program runs in Dablain model, to compare with the result from sequential program. The contours are shown in Fig.3.16.

The difference plot between this solution and the one from sequential program can be shown as in Fig.3.17.

Processing it into relative ratio in percentage, the difference plot is shown as in Fig.3.18.

3.7 Parallel Programming

This parallel version of the algorithm can be implemented with parallel programming techniques. Parallel programming is programming that allows you to explicitly indicate how to partition and distribute the computation into different processors to be

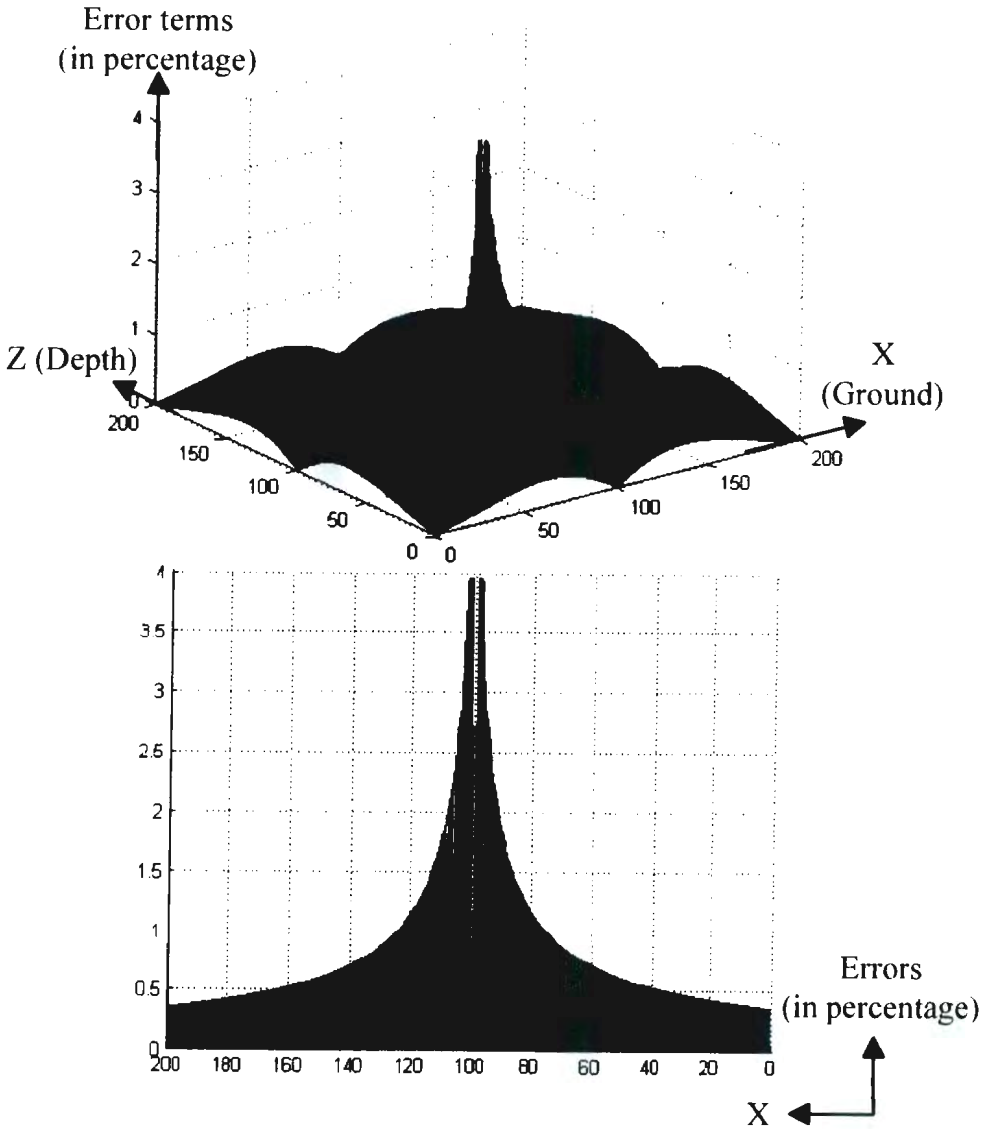


Figure 3.14: Error distribution comparing to exact solutions (in percentage).

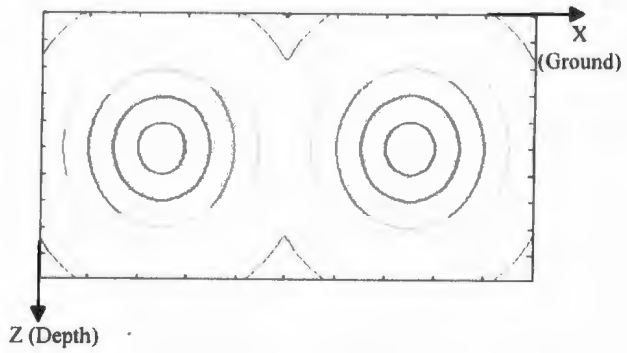


Figure 3.15: Travel time contours generated from double source points experiment.

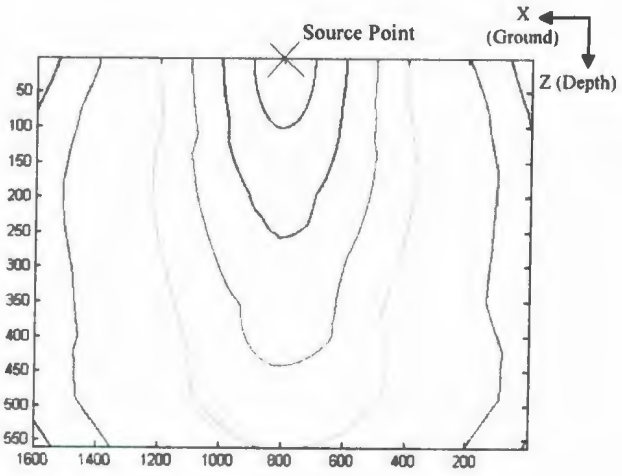


Figure 3.16: Travel time contours generated from Dablain model.

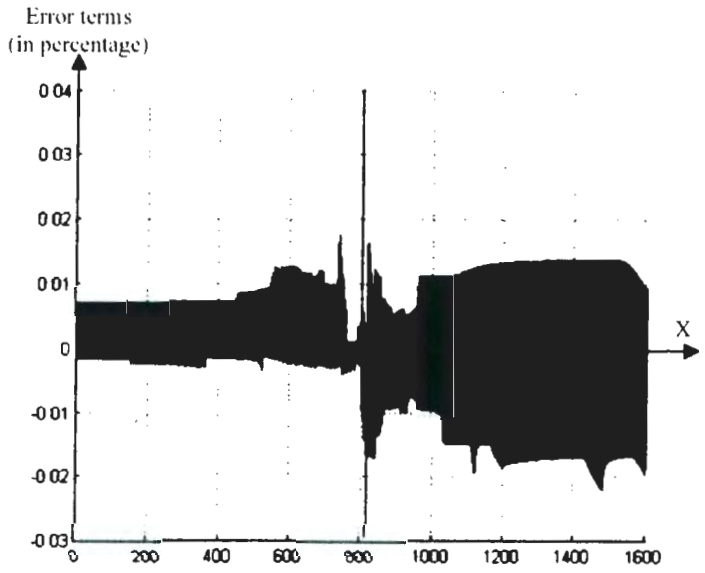


Figure 3.17: Difference between sequential algorithm and parallel algorithm.

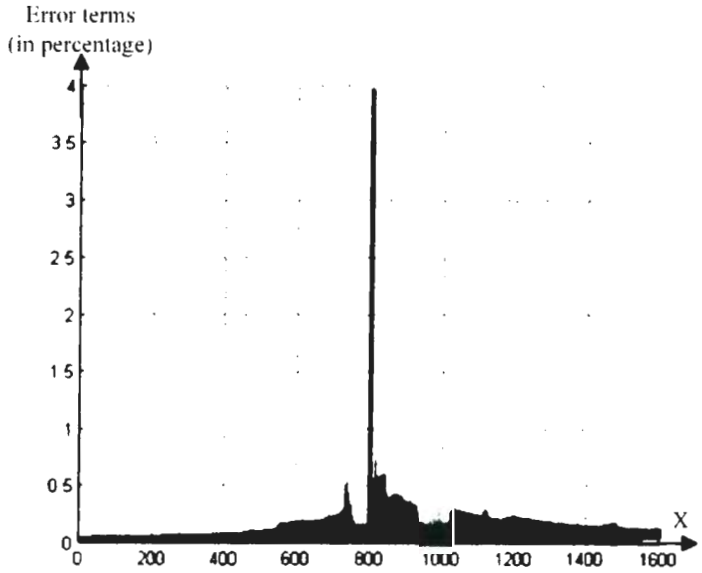


Figure 3.18: Difference between sequential algorithm and parallel algorithm in percentage.

executed concurrently[31]. MPI (Message Passing Interface) is a standard specification for message passing libraries. Libraries consistent with this standard allow the user using Fortran, C or C++ to realize multi-core programming on the platforms such as network of workstations and multi-core cluster.

3.7.1 Methodology

Partitioning

One of the most important issues for parallel programming design is **partitioning**. Partitioning is the process of dividing the computation and the data into pieces (also known as primitive task), so that the computation can be assigned to different processors. One approach is known as Domain Decomposition in which designer first divides the data into pieces and then determine how to associate computations with the data. Function Decomposition is the other approach in which designer first divides the computation into pieces and then determine how to associate data items with the individual computations. These two methods are complementary to each other.

The goal of the partitioning is to identify as many primitive tasks as possible, because the number of primitive task is an upper bound on the parallelism we can exploit (obviously, the limit case is to assign one of the primitive tasks on one processor).

Communication

After identifying the primitive tasks, the next step is to determine the **communication** between them. The primitive tasks usually share data with others. For example, the primitive task in our algorithm is one rice computing, and the adjacent

rice computing units share travel time inputs. Furthermore, the computing units need the computing results from its neighbors as input. The data have to be transferred among these primitive tasks. Communication among tasks is the overhead of the computing. A good way to design the program is to minimize this overhead.

Agglomeration

In partitioning, we try to identify as much parallelism as possible. However, the number of primitive tasks can exceed the number of processors by several orders of magnitude. In our example, there are thousands of grid points for rice computing, but the cluster only has 36 processors available), simply creating these tasks would be a source of significant overhead. **Agglomeration** is the process of grouping tasks into larger tasks in order to improve performance or simplify programming. The goal of agglomeration should be lower communication overhead; maintain the scalability of the parallel design; reduce software engineering costs by largely taking advantage of the existing sequential program.

Mapping

Finally, **mapping** is the process of assigning tasks to processors. The goal is to maximize processor utilization and minimize inter-processor communication. Processor utilization is the average percentage of time the system's processor are actively executing tasks necessary for the solution of the problem. Processor utilization is maximized when the computation is balanced evenly, allowing all processors to begin and end executing at the same time. Conversely, processor utilization drops when one or more processors are idle while the remainder of the processors are still busy.

3.7.2 Program design

According to the four aspects of the design methodology, an MPI program can be designed as follows to implement our parallel algorithm.

As discussed, the primitive task in our design is rice computing. In ideal case, the design should implement as many rice computing units as the number of grid points. However, in parallel programming platform, the number of processors is much less than the number of primitive tasks. On the other hand, in our algorithm, each "rice computing" is equal to another, and there is no locality preference. This makes the agglomeration easier.

2D partition is applied on the computing field to make the field a 2D array of sub-blocks (a 1D partition is also possible, see Fig.3.19). The number of sub-blocks can be as many as the number of available processors, in that the computation of each sub-block is taken by one of the processors.

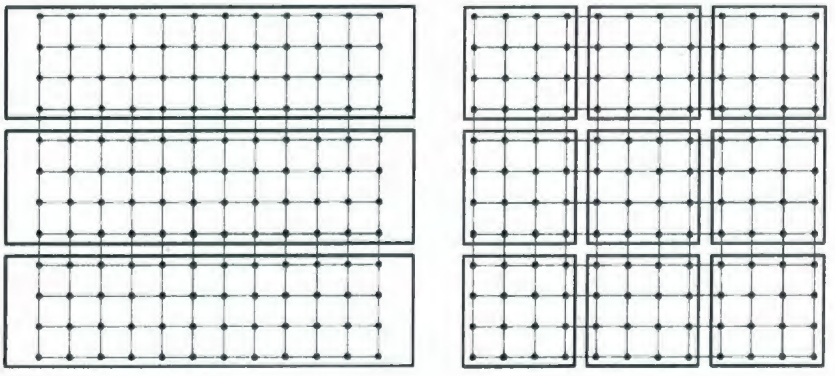


Figure 3.19: Partition the computing in a finite difference filed into sub-block, each processor take responsible for one sub-block.

The computation of the grid points on the edge of each sub-block needs the travel

time values on the adjacent points which is stored in the other sub-block of memory managed by a different processor. Thus, a communication scheme has to be built. As Fig.3.20 shown, a "ghost region" is set for each sub-block which is a part of extra memory stores the grid points on the edge of other sub-block. MPI libraries provide the function to transfer or exchange a chunk of data among processors. All of these processors can work jointly by exchanging data in the ghost region. Inside each sub-block, the processor just runs the sequential program of reusing a single rice computing unit.

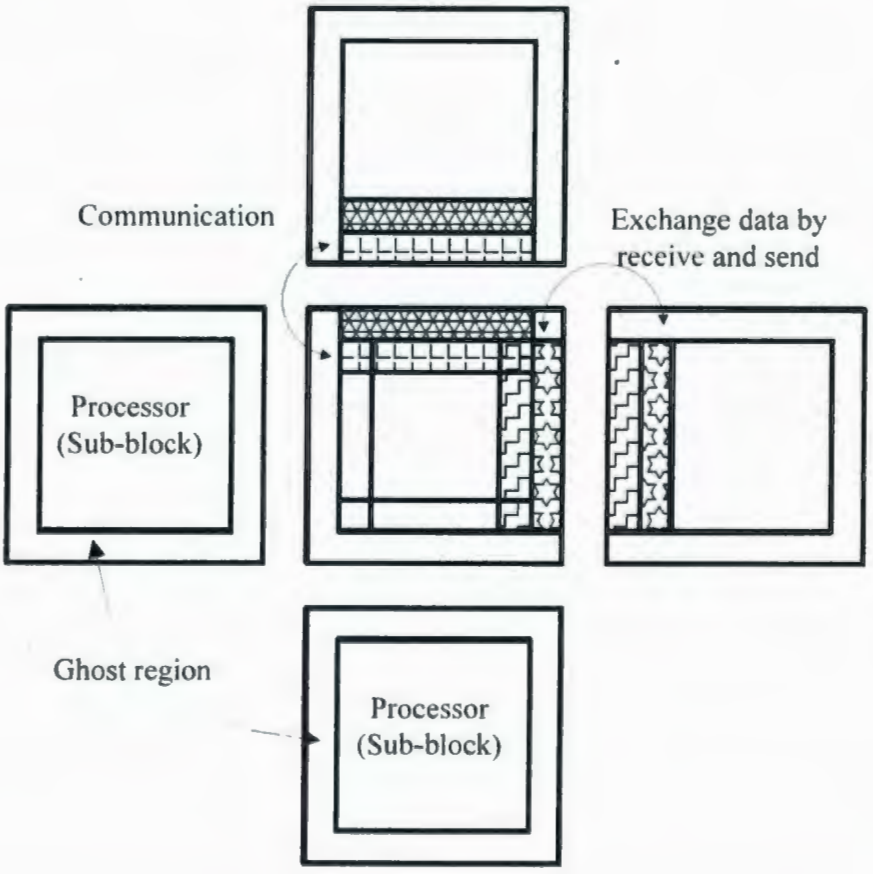


Figure 3.20: Ghost region and communications inter-processor.

Generally speaking, the parallel programming increases the computing speed. The modification of the algorithm into this fully parallel version makes the MPI programming become a feasible solution for our algorithm.

Chapter 4

Hardware Design and Implementation

After developing the algorithm, verifying it, hardware implementation issues are discussed in this chapter.

4.1 Motivation

The motivation to develop such a system is to verify the feasibility of hardware implementation of the algorithm, and build a prototype for future commercial use. FPGA (Field Programmable Gate Array) is chosen as the implementation platform. FPGAs are usually slower than their ASIC (Application Specific Integrated Circuit) counterparts, cannot handle as complex a design, and draw more power. But their advantages include a shorter time to market, ability to reprogram in the field to fix bugs, and lower non-recurring engineering costs. Sometimes, the designs are developed on regular FPGAs and then migrated into a fixed version that more resembles an ASIC. From another perspective, comparing to the general purpose processor or DSP so-

lution. FPGA has more design flexibility because fixed instruction set programming pattern in CPU's which is decided by the processor architecture. Moreover, with FPGA, designer can parallel many operations into the same clock cycle to exert the potential of this hardware platform.

The system can be designed as a coprocessor communicating with CPU of a PC through PCI or PCIe interface[35]; or it can be an independent system working jointly with peripherals to complete various tasks[36]. Our project focuses on the implementation of core logic to realize the algorithm steps described in section 2.3, and proposes a system frame which can be easily extended from a demo system to a platform that can run a real geophysics model by adding in larger memory capacity. The peripheral issues such as I/O design, data visualization device design and high performance memory structure design can be further decided in the future commercialization process by the professionals. In this project, algorithm verification and feasibility evaluation are the main tasks.

4.2 FPGA Design and Implementation Flow

FPGA design, as an efficient way of prototyping a digital system, has been in development for many years, and the technology evolution is still going on. There is a relatively mature methodology proposed for FPGA design and implementation procedures. It can be illustrated with flow charts. Before introducing this design flow, let's first overview the profile of the FPGA platform chosen for the project.

Profile of Virtex-5

Xilinx is one of the major FPGA production companies. They provide powerful and flexible FPGA chips and IDE(integrated development environment) that can satisfy

a wide range of applications. For this project, LXT platform of Virtex-5 family from Xilinx company is used. Virtex-5 is the newest high-end FPGA production of Xilinx[37]. It adopts the 65 nm copper CMOS process technology; contains the most advanced, high performance, optimal utilization logic fabric; and also includes many hard-IP system level blocks, such as 36 Kbit block RAM (on chip memory), advanced DSP48E slice (high performance arithmetic unit), enhanced clock management tiles (CMT) and advanced configuration options. These features mentioned brought many conveniences to our project design.

Xilinx FPGA design flow

For its FPGA production series, Xilinx proposes the design flow as shown in Fig. 1.1[38].

The first step is design entry, in which designers can describe their design in the tools such as schematic editor, HDLs (Verilog and VHDL) or Impulse C[1] [39]. Thus, if using a HDL for text-based entry, the HDL file has to be synthesized into EDIF file; or if using the Xilinx Synthesis Technology (XST) GUI, it is synthesized into NGC files.

In the implementation step, user can assign constraints to the design entry ac-

[1]There is an increasing trend toward using FPGAs as hardware accelerators for high performance computing, just as we do in this thesis. This kind of applications typically begins their lives as software model, and then manually rewritten and implemented in hardware using VHDL or Verilog. This manual conversion of software algorithms to hardware is a process that can be long and tedious in extreme; hence, there is a strong demand for more rapid paths to working hardware. There is also a strong desire to avoid later redesigns of that hardware to reflect software algorithm updates. In this context, some researchers and companies put their efforts on the research to efficiently map algorithms and applications written for traditional microprocessors into arbitrary logic gates and registers combined with somewhat higher level logic structures. Impulse C is one production from this kind of research. It is closer to software programming language, rather than a HDL (Hardware Description Language) such as System C. The design in this high level language can be compiled into the form that is acceptable to the FPGAs with certain compiler. Finally, they try to realize the goal that the FPGA devices, when programmed using appropriate methods, are not so different from other non-traditional computing platforms, such as DSPs. This research becomes a hot topic in recent years. [39]

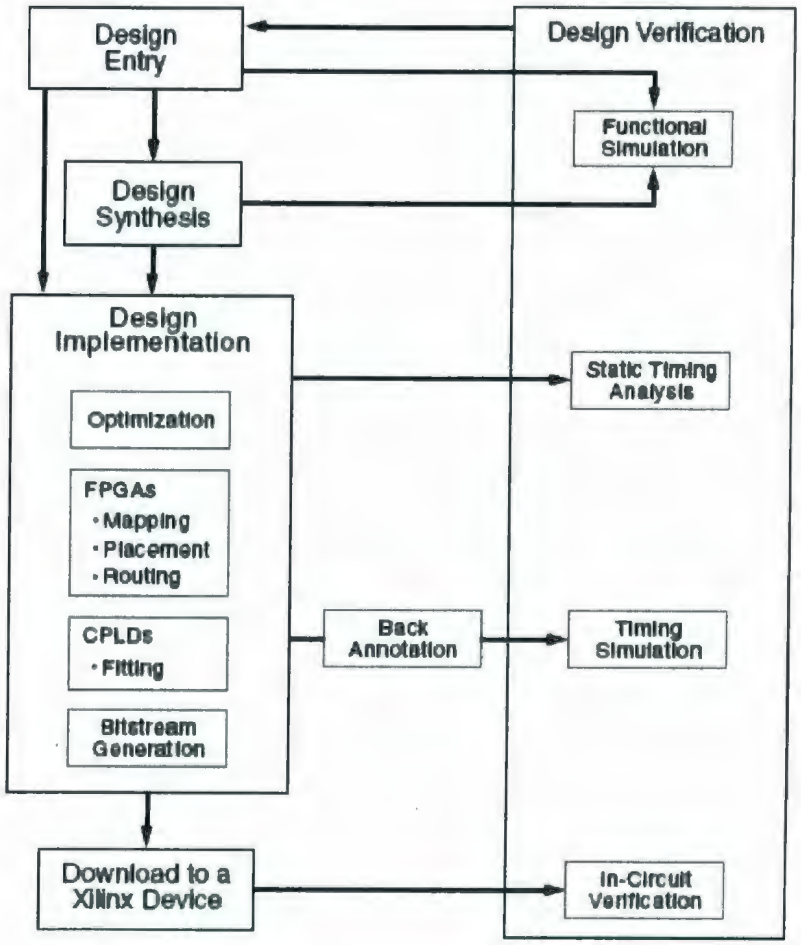


Figure 4.1: Xilinx design flow. Reference from [7].

cording to timing and area design expectations. The logic design are then passed through the steps of mapping, placement and routing, to be mapped into a specific architecture (as Virtex-5 in our case). Accordingly, the development tools convert the logical design file format, such as EDIF, that created in the design entry and synthesis stage into a physical file format. Finally, the file can be compiled into bit streams that can be loaded into FPGA chips directly.

Simulation and verification in the flow

In the whole process from design entry to bit stream generation, the design verification is a step taken to make sure the design is correct processed in each step. In this flow, several simulation and verification measures are applied. For example, the function simulation is applied to verify the logic correctness of the design entry; the post-simulation make sure the design with constraints can be correctly mapped into certain architecture. In digital design, simulation and verification are as important as the design and implementation themselves.

Detailed sub-steps in the major steps of design flow

Furthermore, a detailed flow chart that illustrates the steps and corresponding output of each step using Xilinx IDE is shown in Fig.4.2 [38].

The several major steps such as design entry, synthesis, implementation and verification actually include particular sub-steps in them. The initial design entry is added with more and more constraints when passed through all these sub-steps becomes the final physical implementation. The details of all these procedures involve the knowledge in a broad area of IC (integrated circuit) technology, and can be very complicated, even dreadful for an inexperienced designer. However, before mastering all the details of these implementation options to optimize the design performance

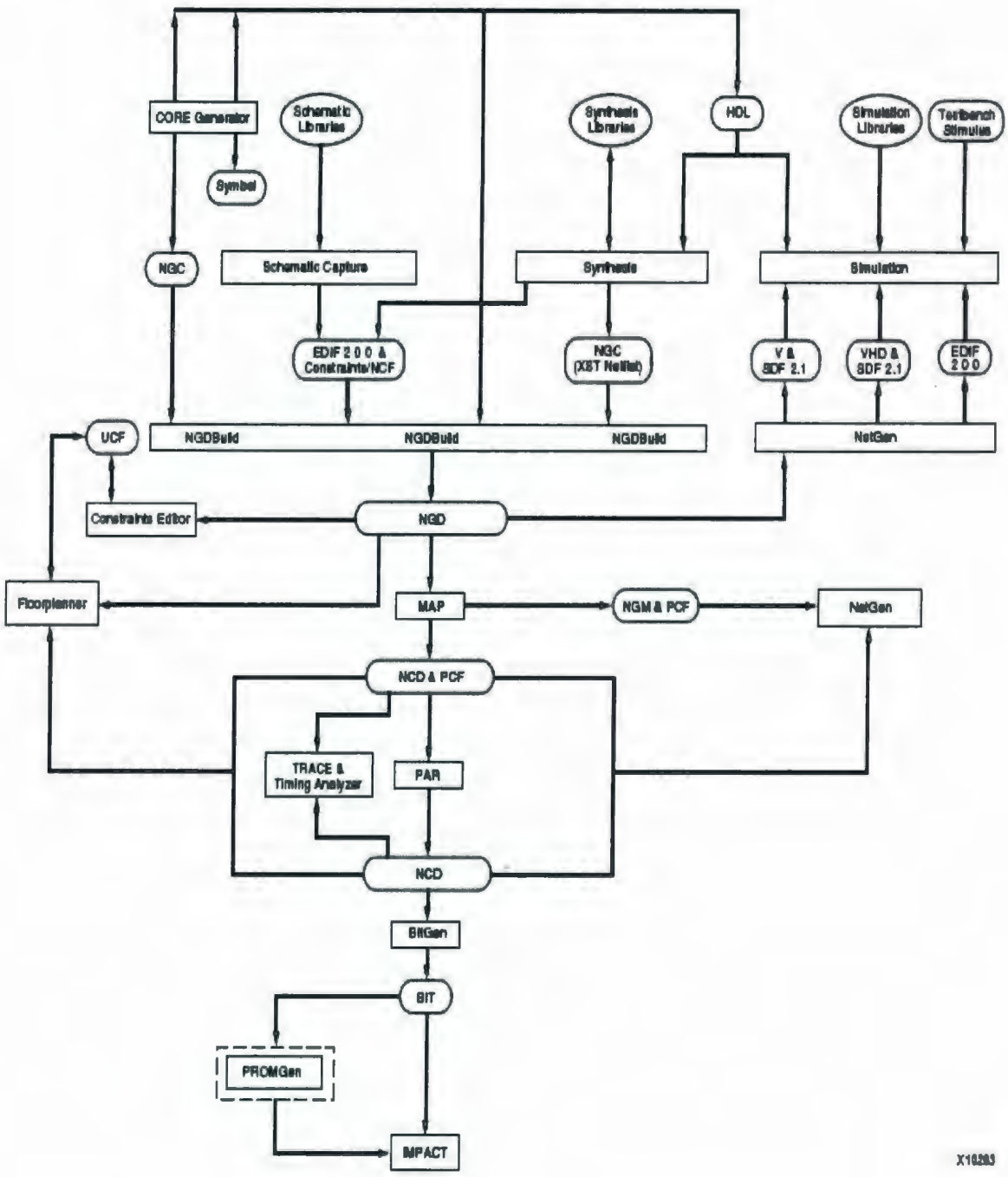


Figure 4.2: Xilinx FPGA design flow using Xilinx ISE. Reference from [7].

into maximum, we can apply some moderate optimization choices to achieve a system prototype.

4.3 System Design (Top Level Block Diagram)

No matter in industry or academia, HDLs (Verilog and VHDL) are the dominant ways to describe a digital design; they are essential to achieve design goals, because of their explicit and precise description and control over hardware implementation. However, the design itself is the most essential part to the algorithm, rather than the tools to describe it. From this section, the design will be presented stage by stage without involving HDL details.

The discussion of software design in previous sections has gone through the major components of the system and the data flow transferring between the modules. Hardware design is different from the soft programming that it does not have an implicit execution order of the statements. All the events in circuit are actually triggered simultaneously all the time. The way to make modules working in an order is to design and implement FSMs to control the timing. To avoid over-complex FSMs, to make the design clear and reliable, two strategies are adopted:

Modularize the system (partition the system into functional blocks, each module can complete a single task and has its own implementation non-related to other modules. All the modules communicate with each other through clearly defined module interfaces);

Explicitly use combinational components (such as basic gates, decoders, and multiplexers) to realize control functions as much as possible.

According to this basis, a system structure is proposed, whose top level diagram can be shown as in Fig. 1.3.

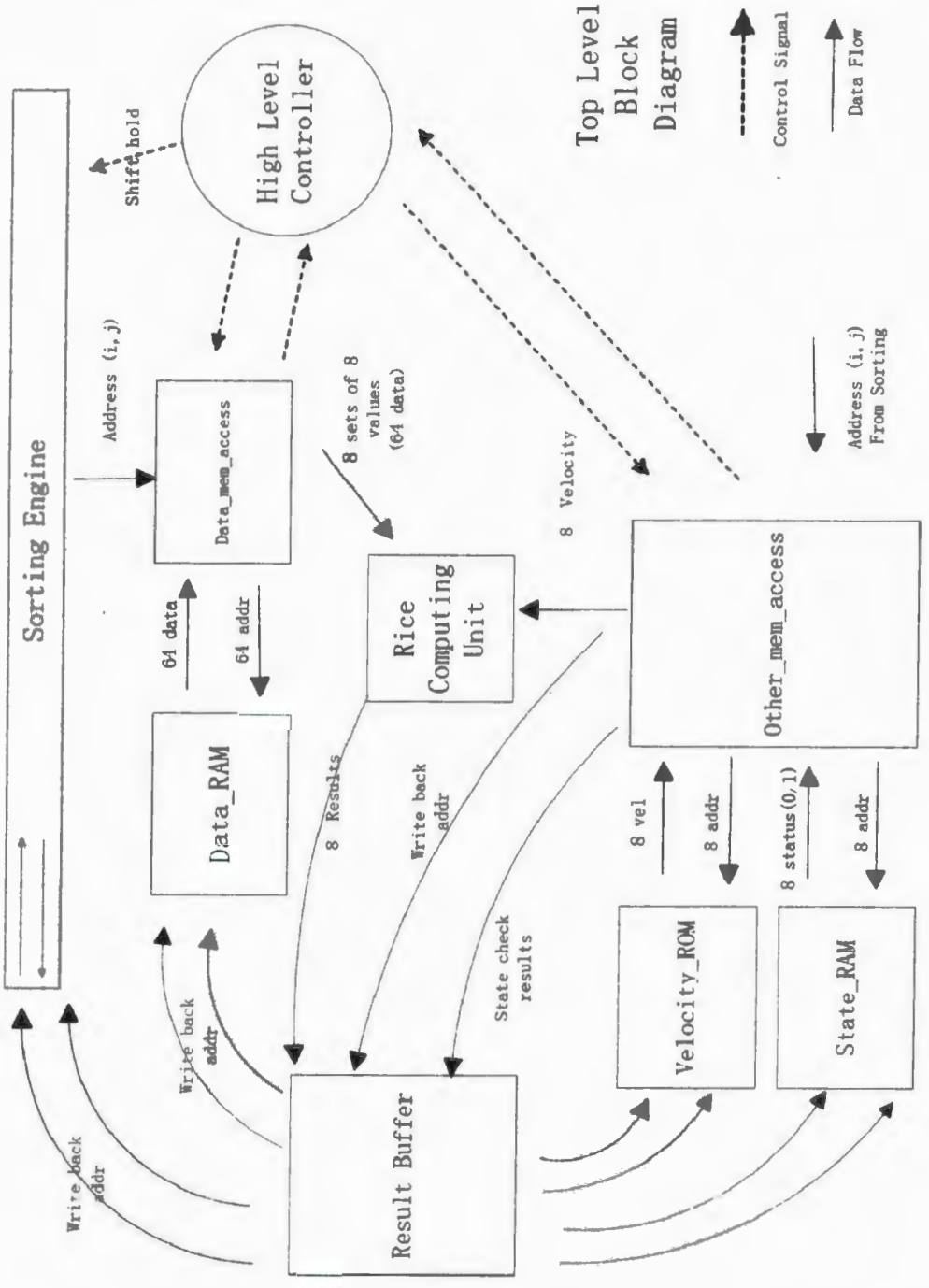


Figure 4.3: Top level system diagram.

The system in this diagram has similar structure with the one has been introduced in software simulation as shown in Fig.3.2. The major modules in the system are computing unit, sorting engine, storage modules (RAMs and ROMs), memory access modules, and result buffer. The working flow of the system is like this: at the beginning of a computing iteration, sorting engine release the coordinates of the minimum travel time point in the sorting heap. This information is sent to "data memory access" module, and the module calculates the coordinates of the 25 points in the vicinity (see section 3.1) and fetch the data block. In the similar way, the module called "other memory access" can fetch the 8 velocity values and 8 flag bit information from velocity ROM and state RAM which is the memory stores the flag bits array. Computing unit receives the data from memory access modules, and computes the 8 new travel time values on the neighbor points surrounding the starting point. Before sending to result buffer, the 8 results would be tested in the computing unit to be tagged as a "good[2]" result or "bad" result. From result buffer, the "good" results would be written back to corresponding memory and the sorting engine. Thus, a new round of computing is ready to begin. In the system, timing and order of this working flow is maintained by the "high level controller" which is a FSM sending control signals to all of other functional modules.

In Fig 4.3 the data flow and control flow are denoted with wide and fine arrows respectively. To further introduce this system, the design and implementation of each functional module would be introduced one by one in the following sections.

[2]The concept "good" can be found in section 2.2

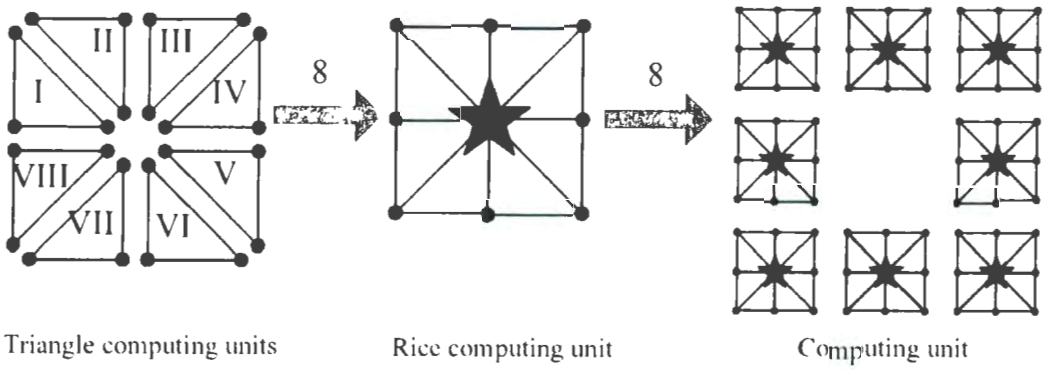


Figure 4.4: Hierarchical structure of computing unit.

4.4 Computing Unit

The first module to present is the computing unit. It is the core of the system: most arithmetic operations in the algorithm are carried out in this module. According to the section 3.2.1, the computing unit has a hierarchical structure (see Fig.4.4).

The elementary component is "triangle computing unit". 8 triangle computing units compose a rice computing unit. Furthermore, 8 rice computing units are combined together to form a computing unit which can process the data on a block of 25 grid points, this makes a rather large single chip solution. For FPGA's, to do all eight the design has to be split across several chips.

VHDL design usually follows the methodology of top-down design and bottom-up implementation. After partitioning the system into modules and specifying certain function into a module, the module would be built from basic logic components.

4.4.1 Triangle computing unit

In this way, let's first look at the triangle computing unit. The mathematics for this module has already been explained in section 3.2.1. Due to the precision requirements

s sign bit - 0 = positive, 1 = negative
e biased exponent (8-bits) = true exponent + 7F (127 decimal).
f fraction - the first 23-bits after the 1. in the significand.

Figure 4.5: Single precision floating point number representation in IEEE 754: Standard for Binary Floating-Point Arithmetic. Figure is from [8].

of practical geophysics application (we need a number representation scheme that can represent numbers in a wide range), the calculation is best to be done in the precision of "single-precision floating-point number". The fixed point implementation has less representation range: it may satisfy the requirement of small-scale travel time computation, but it is lack of expansibility to solve a 3D problem which requires wide range and sufficient significant digits in number representation.

Floating point number

The bit layout of a single precision floating point representation is shown in Fig. 4.5

There are three portions to represent a number in this format: sign bit; exponent field; and significand field which is actually the fractional part (also called mantissa). Single-precision floating point format can represent the number from $N_{min} = 2^{-126} \approx 1.2 \times 10^{-38}$ (including subnormal numbers) to $N_{max} = 2^{128} \approx 3.4 \times 10^{38}$, and the machine epsilon is $\epsilon = 2^{-23} \approx 1.2 \times 10^{-7}$ [8]. This floating point scheme has advantages in range and precision of number representation; however, it is not as straightforward as fixed point representation in its arithmetic operations, especially, the division operation and square root operation. For these two operations, the direct "shift and restoring" method that is the mimic of the paper and pencil method can be

complicated in digital implementation; whereas, the numerical iterative methods are better. Furthermore, the inverse problem to evaluate reciprocal is solved by applying Newton-Raphson iteration[40] [41]. In this way, the expensive division operation can be converted to several relative cheaper operations using addition and multiplication.

Through the work in chapter 2, the formulae such as Eq.2.8, Eq.2.12, and Eq.2.13 have been transformed to avoid expensive operations. There is only one square root operation left in these three equations, and all other operations only includes addition, subtraction and multiplication (no division). For the square root operation, we develop a method in which a look up table is used jointly with a third order Newton-Raphson iteration formula.

Xilinx Floating Point Arithmetic Core

In the project, the arithmetic functions of triangle computing unit is implemented with Xilinx floating point cores[42]. Xilinx provides a core generator system which can generate and deliver a library of parameterizable IP cores optimized for Xilinx FPGAs. With this system, user can create high density, high performance designs in less time. This core generator system is integrated into Xilinx ISE WebPack IDE. Users can access this generator tool seamlessly in the ISE development flow.

Not all of the IP cores from Xilinx are free. In most cases, users can freely run function simulation on the designs which involve Xilinx IPs; but have to purchase licenses to use these IPs in post-simulation and FPGA implementation. Fortunately, the floating point arithmetic IP cores from Xilinx are open source for both simulation and implementation. There are abundant operations included in this floating point collection, such as addition/subtraction, multiplication, division, square root, comparison and fixed point to floating point conversion. From GUI (graphical user interface), the parameters of the cores can be easily customized, which include the

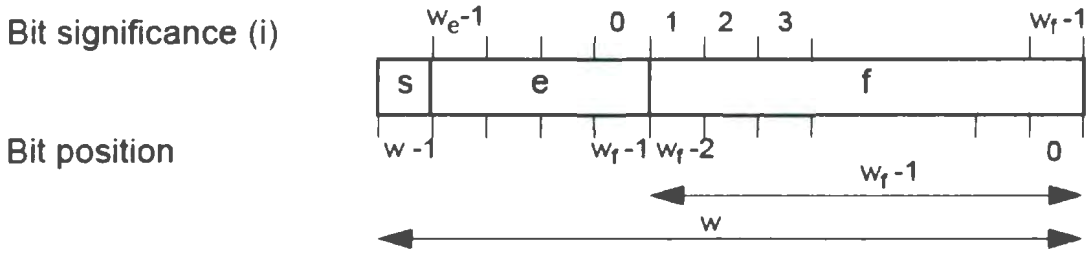


Figure 4.6: Xilinx floating point format. W_e is the width of exponential field of the number, and W_f is the width of fractional field.

input/output width, control signals and basic building elements[3].

The floating point format in this Xilinx core is not only fully compatible with IEEE 754 single precision and double precision floating point standard; user can also establish their own extended floating point format by customizing the width of exponential field and fractional field of the number (see Fig.4.6). After customization, the generator tool can generate corresponding cores in the form of low level circuit description file with HDL interface.

Xilinx Dedicated Arithmetic Unit: Dsp48

In the customization of Xilinx floating point core, there is an option that allow user to decide whether to use Dsp48 slice (see Fig.4.7) to build the floating point operation modules. All of these floating point cores can be built with standard logic slice (slice LUTs and slice registers); however, the Dsp48E slice gives designers a better choice when building arithmetic functions.

The DSP slice is designed for DSP applications and large arithmetic operations.

[3]The basic building block is slice. A slice contains LUTs and registers. Generally speaking, the slice LUTs are used to form combinational circuit; and the slice registers are used for sequential circuit synthesis. To learn the basic FPGA architecture, such as CLB(XILINX VIRTEX Configurable Logic Block), slice, i/o resources, memory and clocking resources. Please refer to [13] [14] and [15] for details.

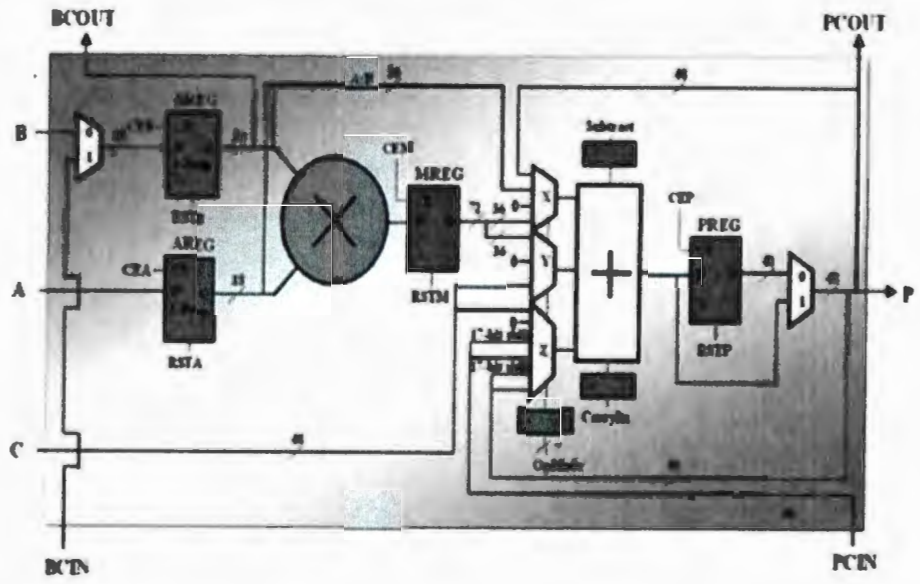


Figure 4.7: DSP48 Slice Has a 2's Complement Multiplier 18x18 and 48-Bit Accumulator.

DSP designers who traditionally use the FPGA fabric for arithmetic applications will find that much of their job is done for them internally by this Dsp48 block. All they need to do is configure the block by using operation mode inputs from the GUI of IDE, which control the flow of data in the block. The DSP48 slice has a 2's complement multiplier 18x18. It also has a 3-input, 48-bit adder/subtractor, which can be used to create several different arithmetic operations. Cascade pins are included to support complex functions with no speed penalty. This allows user to implement larger arithmetic operations by linking multiple slices together. There are optional pipeline registers at several points within the DSP48 slice to maximize performance. Most users are targeting this resource with the core generator software tool which can largely simplify the configuration process: user can follow the configuration guide of the core generator, and the software tool will generate the low level files and provide

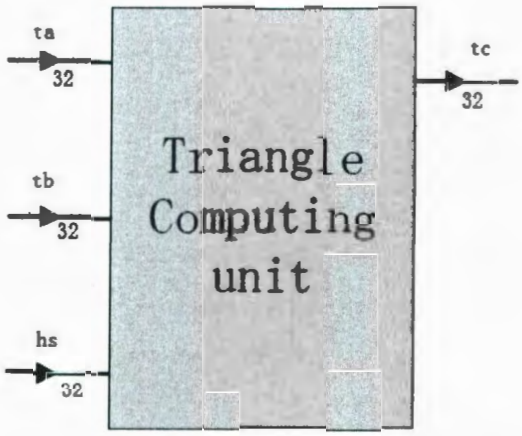


Figure 4.8: I/O specification of triangle computing unit.

HDL interface. Application of DSP48 slice can save the standard slice resource in the FPGA, and achieve a higher performance on computing.

Design Comparison

In RTL (Register Transfer Level), the interface of triangle computing unit can be defined as shown in Fig.4.8.

All the inputs for computation of Eq.2.8, Eq.2.12, and Eq.2.13 are t_a, t_b and hs[4] These inputs are all represented in 32 bit single precision floating point format. The output from this unit is the travel time value t_c.

Design I

In Fig.4.9, there is one design for this triangle computing unit, in which each operation in the formulae is implemented with one floating point core. This is a fully parallel implementation that all the operations can be processed concurrently. There is no

[4] hs is the product of grid spacing h and slowness s. This product can be pre-calculated and saved in velocity array.

reuse of the operation modules and no internal sequential processing.

In Fig.4.9, "7f800000" is the hexadecimal representation of a 32 bit binary vector, which stands for the value "+ Infinity" in single precision floating point format. The circuit first makes a trial of the two inputs t_a and t_b. If either of these two is "+ Infinity", the result will be directly assigned as "+ Infinity" by setting the control signals of "result mux". The input signals pass comparison and arithmetic modules in certain logic and final result is output from "result mux".

Xilinx floating point cores provide powerful features that user can control the resource usage according to specific application by customizing "latency" and "cycles per operation" options:

1. The parameter "latency" describes the number of cycles between an operand input and result output. The latency of the operators can be set between 0 and a maximum value. The maximum latency of the divide and square root operations is $fraction\ width + 4$, and for compare operation it is three cycles. Different specification of latency results in different low level implementation with different resource usage: shorter latency means higher usage of the slice LUTs in realizing the logic; in contrary, pipeline implementation and module reuse consumes more slice registers and cause longer latency.

2. "Cycles per operation (rate)" describes the minimum number of cycles that must elapse between inputs. A value of one allows operands to be applied on every clock cycle, and results in a fully-parallel circuit. A value greater than one enables hardware reuse. The number of slices consumed by the core reduces as the number of cycles per operation is increased. A value of two approximately halves the number of slices used. A fully sequential implementation is obtained when the value is equal to $fraction\ width + 1$ for the square root operation.

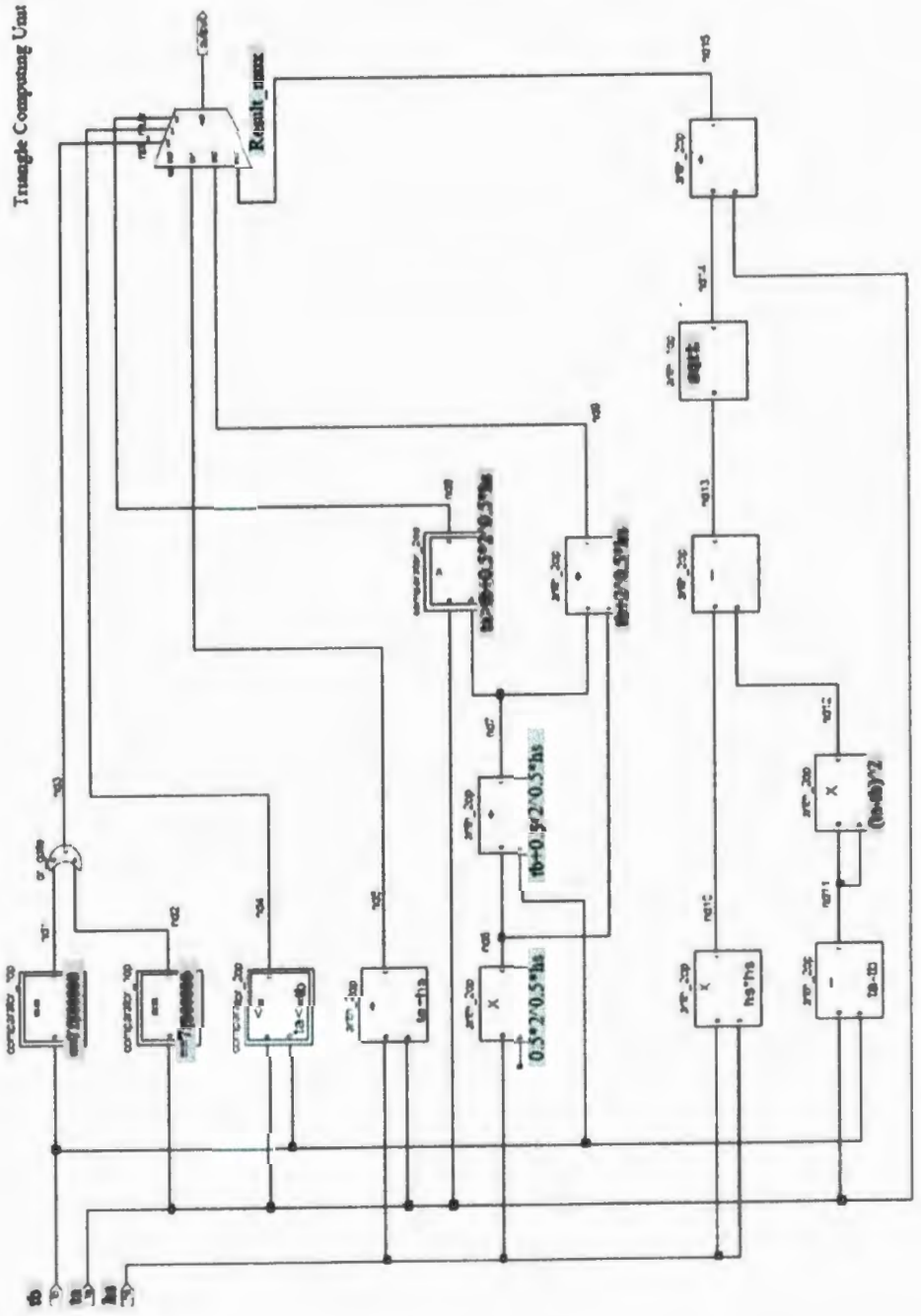


Figure 4.9: Fully parallel design. In the diagram, each block represents a floating point operation core. The operation is marked off on the block. The block with double lines on the edge is comparison operation module; the single line edge block is arithmetic operation module.

and *fraction width* + 2 for the divide operation.

In our design, the latency of all the modules are set to zero, and cycles per operation is set to one. In this way, a fully parallel design is achieved: this design is logic clear without latency; and because cycles per operation is one, the triangle computing unit can be reused in every clock cycle. However, because the lack of register and pipelining, the critical path is long and therefore decreases the theoretical maximum working frequency. At this point, the optimization potential of the circuit design can be the topic for future research work.

Design II

Fig.4.10 shows the other design of the "triangle computing unit" which includes pipelining technology. Several stages of pipeline registers are set in before and after the modules. In this way, the critical path in the combinational circuit is shortened, and the maximum working frequency can be increased. With pipelining, batch of inputs can be continuously fed into the computing unit. It is good for computing long data streams.

The pipeline design can be flexible. The design in Fig.4.10 is only one of several options. By setting different "latency" and "cycles per operation" parameters inside each floating point operation module, the pipeline structure in Fig.4.10 can be changed. For example, increase the latency of the modules in the same pipeline stage together, several stages shown in Fig.4.10 can be incorporated into one stage, and the design is changed. There will be a lot of interesting work can be done to try different combinations of the parameters to achieve better performance for the computing unit.

In perspective of FPGA technology, because there are so many registers in FPGAs,

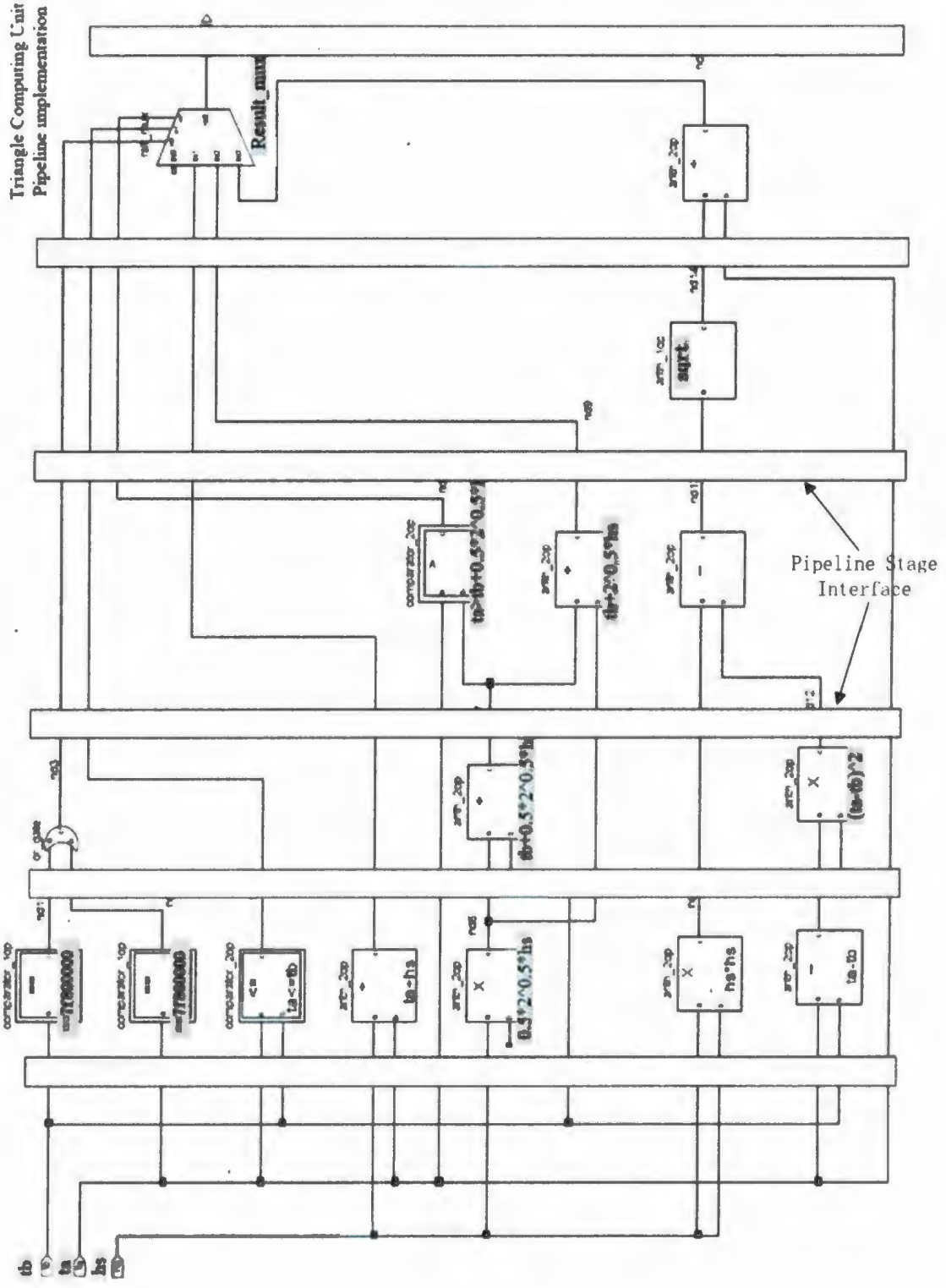


Figure 4.10: Pipeline design.

pipelining is usually an effective way to increase performance. Moreover, Xilinx FPGA has the SRL (Shift Register LUT) primitive[45] which can be used as programmable delay elements (or No Operation, NOP). The SRL provide a good way to add delay to balance pipelines.

Design III

The design of reusing the modules is shown in Fig.4.11. The goal of this design is to minimize the number of cores used in the system. There is only one module for each arithmetic operation, and the operations of the formulae have to be done through the modules sequentially. To schedule the inputs into the operation modules in a correct order, a FSM can be implemented. Instead, a counter can be simply used jointly with a decoder, to issue correct control signals to the input multiplexers. The immediate results are saved in the immediate registers. The whole system runs for several clock cycles as a round to generate one correct result, and the counter is restarted at the beginning of each round.

This design is economic in resource usage. It is meaningful if designer wants to implement a lot of triangle computing units. Nevertheless, there are too many clock cycles to generate one result, the computing efficiency is low. Moreover, it is complicated to design the counter and decoder to schedule the operations.

At this stage, based on the use of Xilinx floating point core, three design solutions are proposed. They are fully parallel design, fully pipelined design and module reused design. Each of them has the advantages and disadvantages. The optimal design should be the one trade off among these three methods.

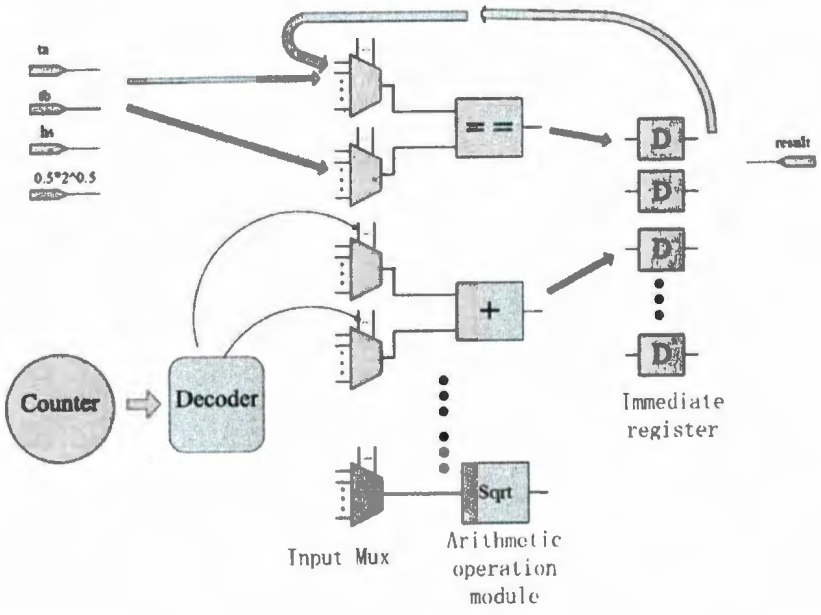


Figure 4.11: Module reuse.

Synthesis Result

In the project, the first design (the fully parallel one) is finally adopted. This is because the Virtex-5 FPGA cannot contain more than 5 triangle computing units based on our estimation. To implement the computing unit as described in Fig. 4.4, a triangle computing unit has to be reused for 64 times in a computing round. In this way, the fully parallel design is the most straightforward and reliable choice (it only takes one cycle to complete each triangle computing); moreover, in our implementation, Dsp48E slice is adopted to build the operation blocks.

The device utilization summary after design systhesis can be shown in Fig. 4.12

From synthesis, the maximum combinational path delay in the circuit is 83.755 ns in which logic delay is 46.646 ns (55.7%) and route delay is 37.109 ns (44.3%).

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	3	28,800	1%
Number used as Latch-thrus	3		
Number of Slice LUTs	2,515	28,800	8%
Number used as logic	2,510	28,800	8%
Number using O6 output only	2,102		
Number using O5 output only	58		
Number using O5 and O6	380		
Number used as exclusive route-thru	5		
Number of route-thrus	63	57,600	1%
Number using O6 output only	63		
Slice Logic Distribution			
Number of occupied Slices	871	7,200	12%
Number of LUT Flip Flop pairs used	2,515		
Number with an unused Flip Flop	2,512	2,515	99%
Number with an unused LUT	0	2,515	0%
Number of fully used LUT-FF pairs	3	2,515	1%
IO Utilization			
Number of bonded IOBs	129	180	26%
Specific Feature Utilization			
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number of DSP18Es	18	18	37%
Number of RPM macros	6		
Total equivalent gate count for design	21,731		
Additional JTAG gate count for IOBs	6,192		

Figure 4.12: Device utilization summary.

Usually, the result after PAR[5] (placing and routing) process can be 30% to 35% worse than the logic synthesis result[7].

Synthesis and Implementation Steps in Xilinx FPGA Design Flow

As the flow shown in Fig.4.2,there are two major steps between design entry and programming file generation process: synthesis and implementation.

During synthesis, synthesizer[6] analyzes the HDL code and attempts to infer specific design building blocks or macros (such as Muxes, RAMs, adders, and subtracters) for which it can create efficient technology implementations. Finite state machine (FSM) recognition is also part of the HDL synthesis step. To create the most efficient implementation, synthesizer uses a target optimization goal, to determine which of several FSM encoding algorithms to use. User can control the HDL synthesis step using constraints.

After synthesis, the design implementation process comprises the following steps:

1. Translate process merges all of the input netlists and design constraints, and outputs a Xilinx native generic database (NGD) file, which describes the logical design reduced to Xilinx primitives.

2. The Map process maps the logic defined by an NGD file into FPGA elements, such as CLBs and IOBs. The output design is a native circuit description (NCD) file that physically represents the design mapped to the components in the Xilinx FPGA.

---

[5]In placing and routing, the placer maps logic from the design into specific locations in the target FPGA chip; the router assigns logical nets to physical wire segments in the FPGA that interconnect logic cells. For the knowledge of design flow, please refer to section 1.2.[7] According to the 60-10 rule: if the route delay is higher than 10 percent (or more), it is more likely to improve the timing by raising the routing effort level.

[6]The major FPGA synthesizers for Xilinx FPGA include Synplify and Precision from third part, and XST from Xilinx.

3. The Place and Route process takes a mapped NCD file, places and routes the design to produces an NCD file that is used as input for bitstream generation in following steps.

From each of these steps, timing reports can be generated:

Synthesis report is the first place performance estimates are provided. The estimate is not very accurate at this early stage, but the estimate can be an indicator of whether synthesis results are good enough to proceed to the next step.

The Post-Map Static Timing Report is useful because it is based on the Xilinx timing constraints, and this report shows detailed descriptions of the longest paths covered by each constraint.

Post-Place and Route Static Timing Report is the report including accurate routing delays. For accurate final timing information of the design, user should refer to this PAR timing report.

4.4.2 Rice Computing Unit

Design Comparison

A rice computing unit contains 8 triangle computing units, and it selects the minimum from the 8 results. The ideal implementation is to build 8 triangle computing units, and make them work in parallel. However, as mentioned in last section, our FPGA cannot hold this many triangle computing units, so a single unit is reused for 8 times in the design.

There are different designs for the rice computing unit. Fig. 4.13 shows the case of using 8 triangle computing units working in parallel (this may become true in future, when we have higher density platform and optimized triangle computing unit). The results coming out of 8 triangle computing units are sent into a comparator

network. Through 3 stages of 7 comparators, the minimum among the 8 results is found out. This circuit can be built in pure combinational logic or be implemented with pipelining (pipeline implementation is not shown in Fig. 4.13).

Fig. 4.14 shows the design of reusing only one triangle computing unit. There is a controller in the system, which is actually a counter. The multiplexers switch the right input signals into the triangle computing unit ports according to the control signal decoded from the counter. Meanwhile, there is only one comparator in the system. At the first work cycle, the intermediate register loads in the first results. Afterwards, the value in this intermediate register is compared to the following incoming values, and the smaller value is loaded into the intermediate register. In the 8th clock cycle the minimum is found out and stored in the intermediate register.

This design saves logic resources by reuse. Comparing to the cost of building eight triangle computing units, the slice expense on the controller, comparator, and multiplexers are acceptable.

Synthesis Result

After synthesis, the device utilization summary of the design is shown in Fig. 4.15. The minimum period is 86.713 ns in which logic delay is 45.385 ns (53.2%) and route delay is 41.328 ns (47.7%). The post-map static timing result[7] is 91.266 ns for the achievable best case delay.

Design Constraints

Besides HDL design in the design entry step, user can specify different types of constraints to help improve the design performance through the synthesis and im-

[7] this timing report is also generated using estimated delay information; for accurate numbers, please refer to the post Place and Route timing report.

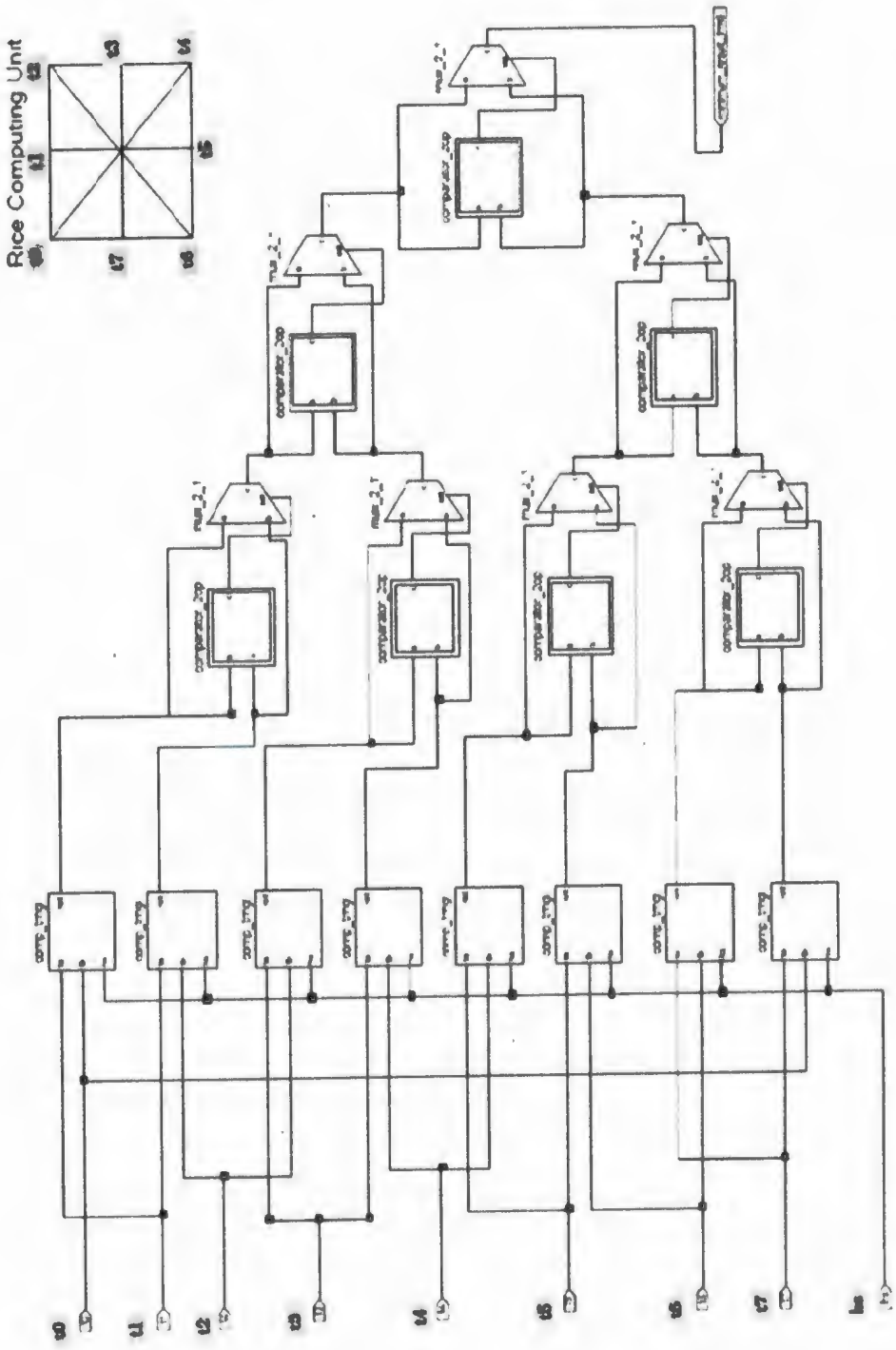


Figure 4.13: Design with 8 "triangle computing unit" in parallel.

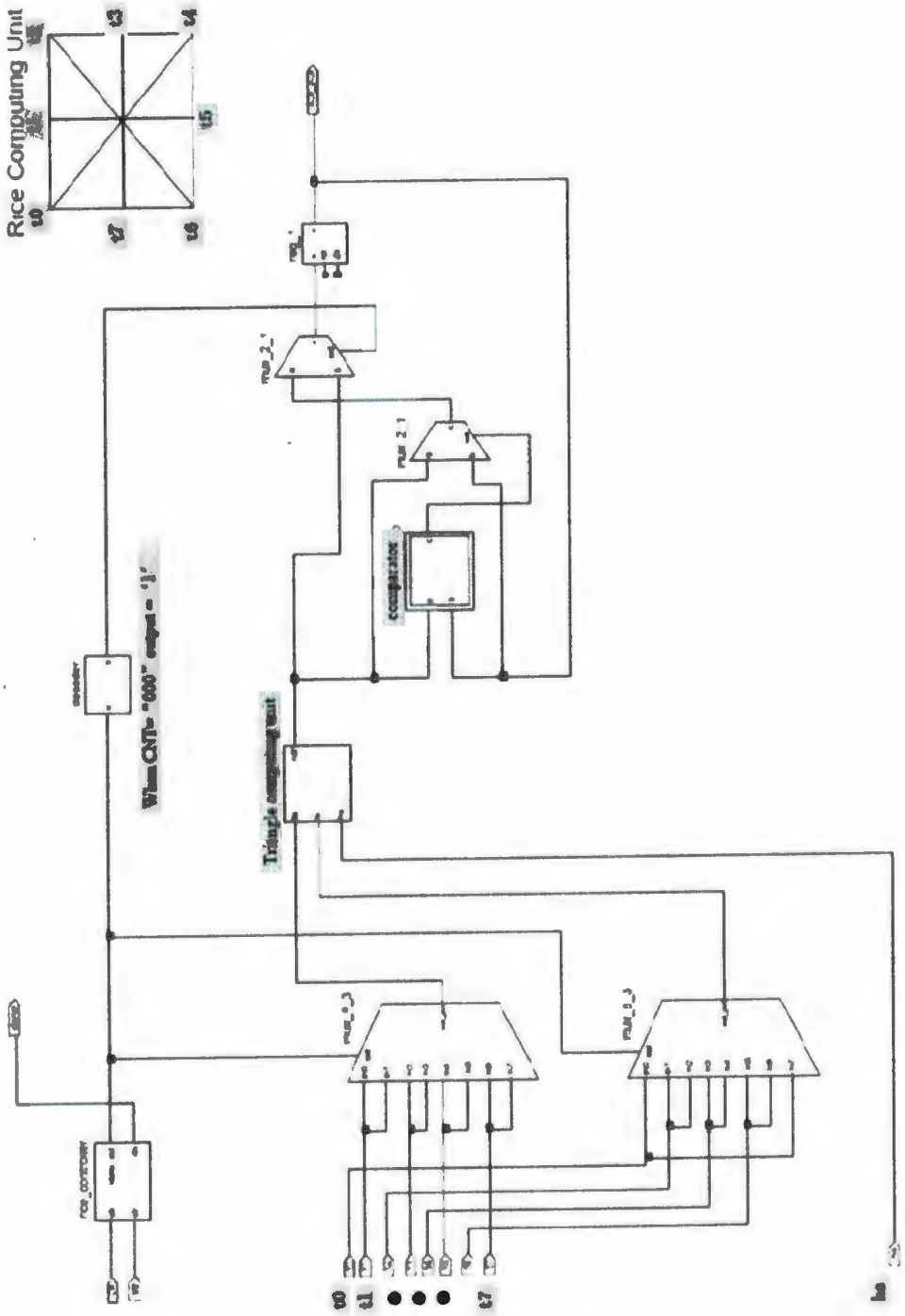


Figure 4.14: Design of using only one "triangle computing unit".

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	40	28,800	1%
Number used as Flip Flops	37		
Number used as Latch-thrus	3		
Number of Slice LUTs	2,745	28,800	9%
Number used as logic	2,742	28,800	9%
Number using O6 output only	2,303		
Number using O5 output only	58		
Number using O5 and O6	381		
Number used as exclusive route-thru	3		
Number of route-thrus	61	57,600	1%
Number using O6 output only	61		
Slice Logic Distribution			
Number of occupied Slices	1,074	7,200	14%
Number of LUT Flip Flop pairs used	2,745		
Number with an unused Flip Flop	2,705	2,745	98%
Number with an unused LUT	0	2,745	0%
Number of fully used LUT-FF pairs	40	2,745	1%
Number of unique control sets	5		
IO Utilization			
Number of bonded IOBs	323	480	67%
Specific Feature Utilization			
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number of DSP48Es	18	48	37%
Number of RPM macros	6		
Total equivalent gate count for design	26,626		
Additional JTAG gate count for IOBs	15,504		

Figure 4.15: Device utilization summary for "rice computing unit".

plementation processes. Each type of constraint serves a different purpose and is recommended under different circumstances.

Synthesis constraints instruct the synthesis tool to perform specific operations. When using XST, synthesis constraints control how XST processes and implements FPGA resources, during the HDL synthesis and low level optimization steps. Synthesis constraints also allow register duplication control and fanout control during global timing optimization.

Placement constraints can be specified for each type of logic element, such as flip-flops, ROMs and RAMs, FMAPs, BUFTs, CLBs, IOBs, I/Os, and global buffers in FPGA designs.

Timing Constraint and PAR

Timing constraints can affect the design performance. The FPGA implementation tools do not attempt to find the place and route that will obtain the best speed. Instead, the implementation tools try to meet user's performance expectations. In other words, timing constraints do not optimize the design or change the netlist in any way, rather they only improve the placement and routing of the design. Performance expectations are communicated with timing constraints. Timing constraints improve the design performance by placing logic closer together so shorter routing resources can be used.

The implementation tools can actually do a good job of placing and routing the design without using timing constraints. Without time constraints, the logic is grouped closely to provide a good internal frequency and minimize clock skew by the tools. Likewise, the in/out pins have a logic grouping if the design has no pin assignments.

If the design goal is to achieve higher frequency, timing constraints should be added to instruct the routing tools to generate the circuit of shorter critical path.

Sometimes, the routing effort level should be set higher to drive the tools to spend more CPU time to generate better result. If the design is aimed to achieve area efficiency (make the design take less resources as possible), floorplan editor can be used by the designer to directly lay out the netlists to access the placement work manually.

Such as our work in this thesis, the prototyping system at this stage is not required to achieve a timing or area optimization. The default strategy from Xilinx can be used, which provides a balanced optimization of performance results versus runtime.

The placing and routing result of this computing unit is 102.116 ns for the achievable best case delay.

4.5 Sorting Engine

In section 3.2.2, the importance of the sorting engine to our algorithm has been indicated. After analyzing several commonly used sorting algorithms, a sorting method which is based on insertion sorting is proposed. The sorting engine should have the capability to hold and sort hundreds of elements in its sorting cells. Each element has three fields: travel time value, x index and z index (if writing x and z indexes into one vector, there are two fields). In every computing round, when a new element is sent into the sorting engine, the engine should sort all the elements according to their value field, and always return the index fields of the element contains the minimum travel time value. In RTL, the I/O interface of the sorting engine can be defined as shown in Fig. 1.16.

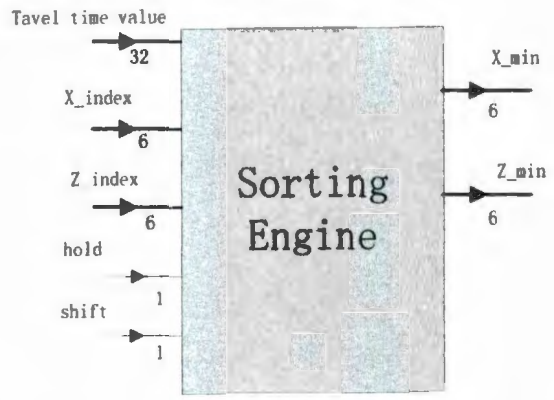


Figure 4.16: I/O specification of sorting engine.

4.5.1 Sorting Cell

In our method, the sorting engine is designed as an array of sorting cells (see Fig. 4.17). A single cell is connected to its right and left neighbors. The adjacent sorting cells can pass their own travel time value field and index fields to each other. All the sorting cells are connected to the control signal buses and data input buses. The buses are linked from the input port, and can carry the control and data to each cell efficiently.

The structure of each sorting cell can be shown in Fig. 4.18.

There are three registers in each sorting cell to store the travel time and indexes. In front of the registers are input multiplexers. The multiplexer switches the inputs into the register according to control signals.

The four data inputs of the multiplexer are "from left" (the value from left neighbor), "insert" (the value from input data bus), "hold" (the value saved in current register will be fed back to itself in next clock cycle to complete the "hold" function.) and "shift out" (the value from right neighbor).

The multiplexer has four control inputs. Two of them come from the control input

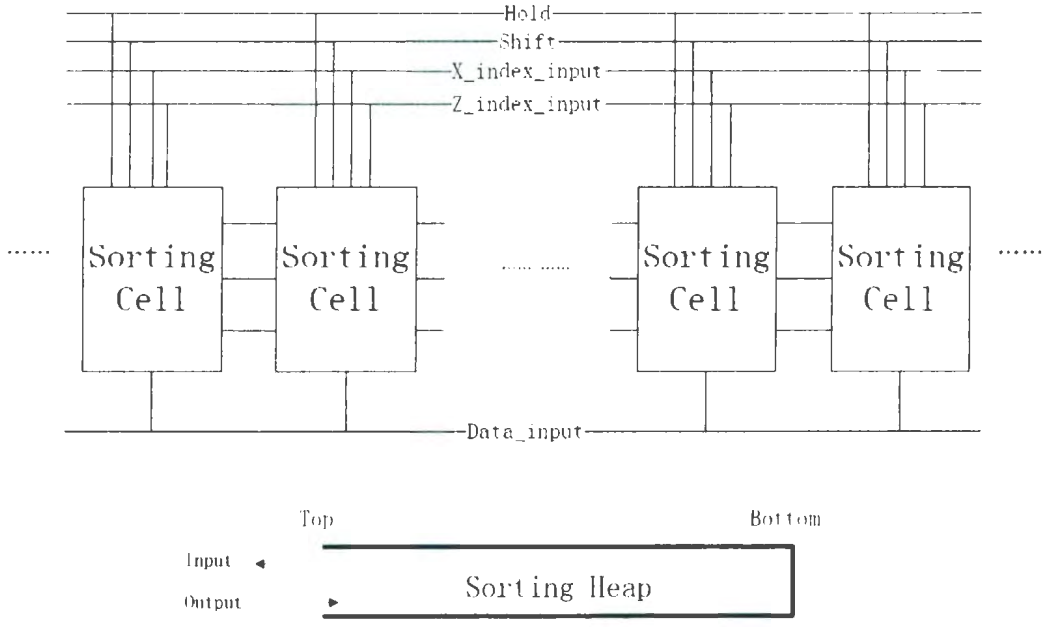


Figure 4.17: Sorting cell array.

bus "hold" and "shift". The other two come from the result of the comparators: one is from the result of the comparator in the current sorting cell; the other is from the comparator of left neighbor cell. The comparator compares the travel time saved in the data register of current sorting cell and the travel time brought in by the data input bus.

Fig. 4.19 shows the relevancy between control signal combinations and their corresponding outputs.

To further understand how the sorting engine works, let's look at the following example (see Fig. 4.20). First assume that the input control signals "hold" and "shift" are '0'. The data bus brings in an incoming travel time value 18. The compare results and the control bit combinations are shown in Fig. 4.20 (1). The sorting cells action according to control combinations, and the input travel time value is inserted into the right position as shown in Fig. 4.19 (2).

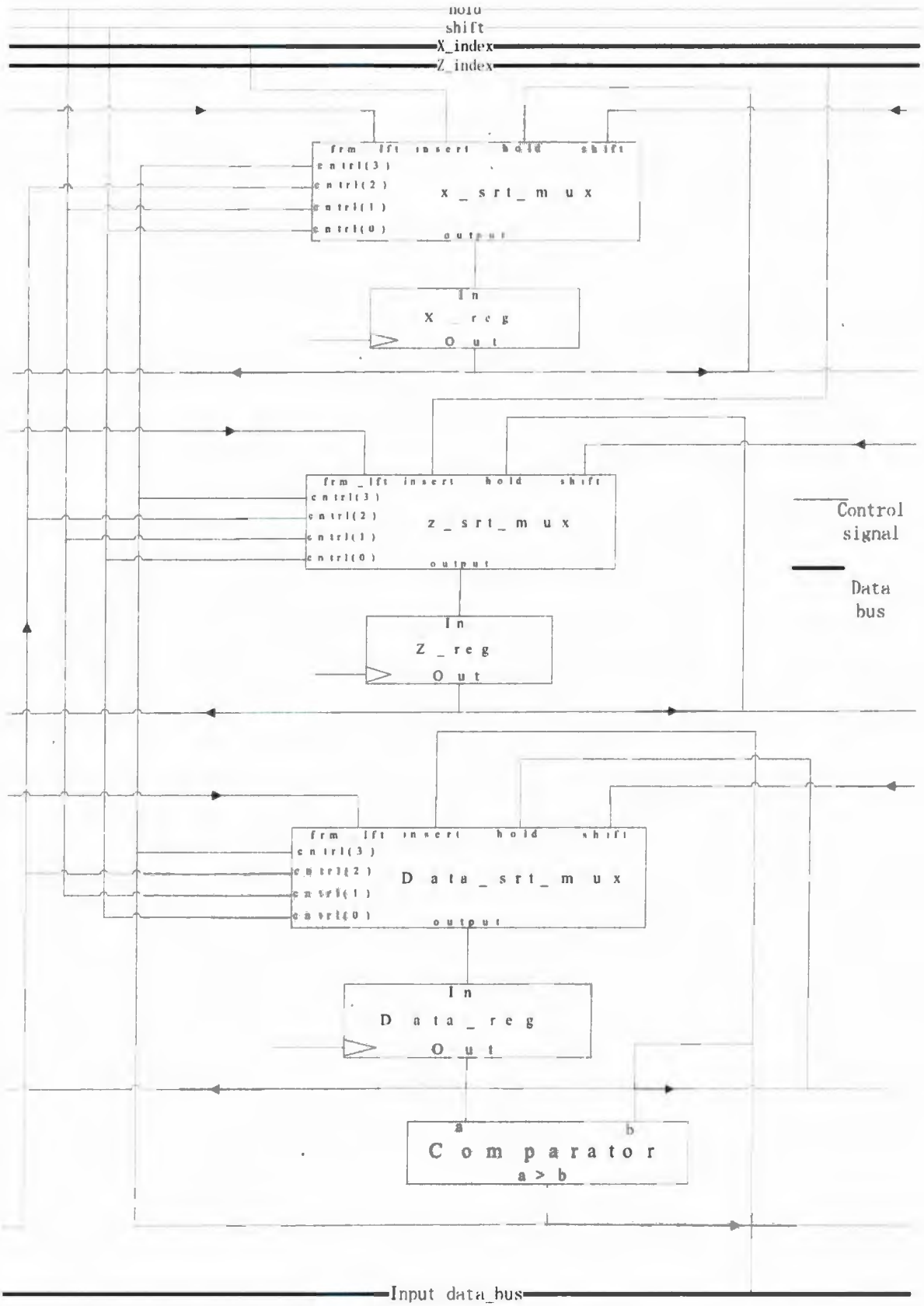


Figure 4.18: Sorting cell.

Control input bits of MUX				Output is from which input port of the MUX	Corresponding operation of the sorting engine
0	1	2	4		
0	0	1	1	from_left	Travel time in current cell is larger than the input value. The travel time in two neighbor cells are also larger than the input value. Thus, the cell sends its values in each field to the right neighbor, and receives the values from left neighbor.
0	0	0	1	insert	Travel time in current cell is larger than the input value, and travel time in left neighbor is smaller than the input value. Thus, all the cells at right side of current cell (including current cell) move their values to the right neighbors. The input travel time is inserted into current cell.
0	0	0	0	hold	Travel time values in current cell and left neighbor cell are smaller than input value. Thus, current cell should hold current value by feeding the value back to input.
0	1	0	0	hold	Control input is ``hold". All the sorting cells feed current values back to the cell inputs.
1	0	0	0	shift	All the sorting cells move their values to the left neighbors. The sorting engine pops out a result.
Other combinations				hold	Simply take all other input combination as ``hold".

Figure 4.19: Control signal combinations and outputs.

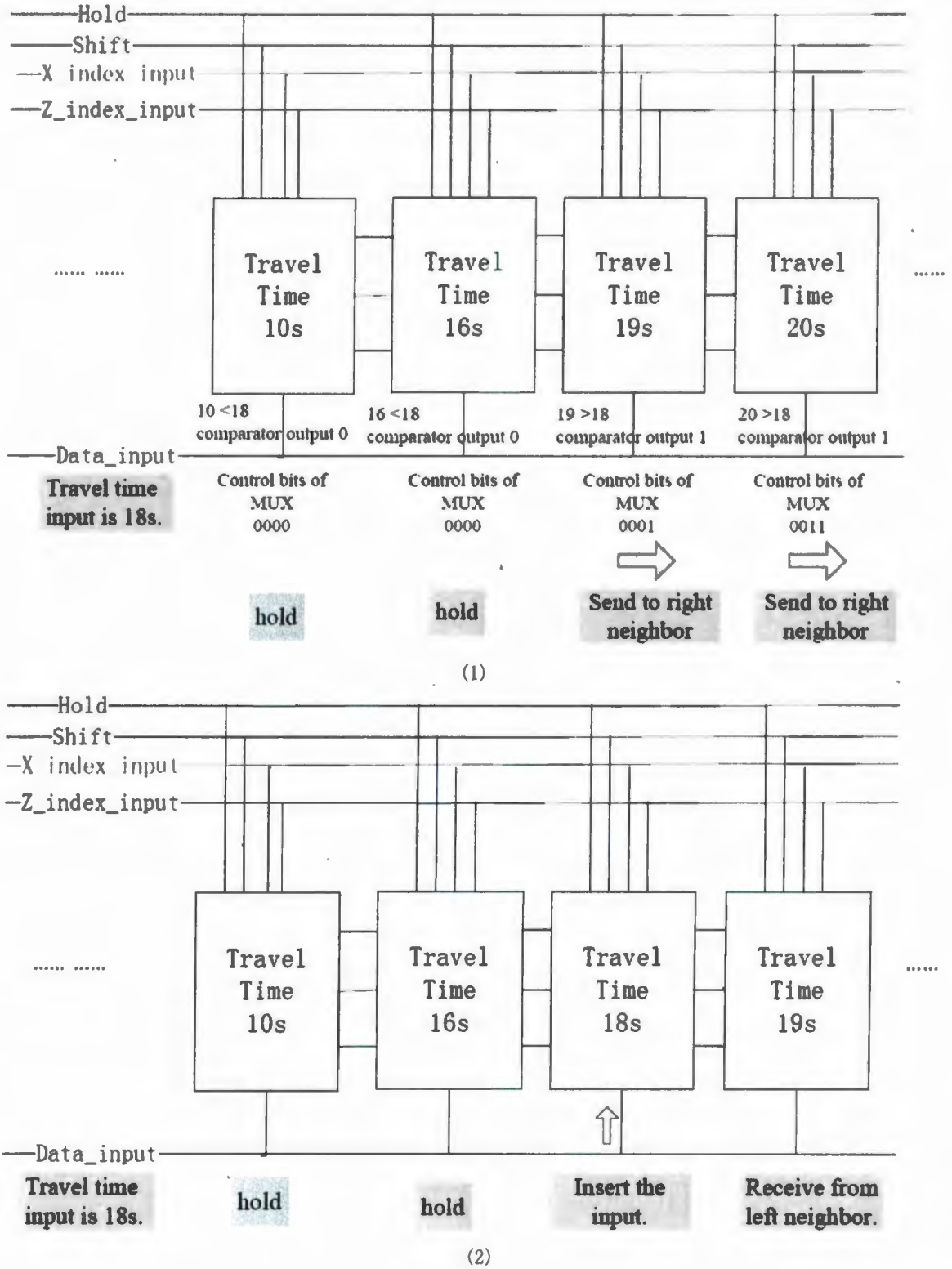


Figure 4.20: An example of sorting engine working.

In this way, the values in the sorting engine array are always kept in an order from minimum to maximum. By judging the pattern of compare results from current cell and neighbor cell, the incoming travel times are inserted into right positions in the array.

4.5.2 Low Skew Routing Resource for Bus Implementation

As the structure of sorting cell array in Fig.4.17 suggests, the number of sorting cells can be changed according to the application. The structure is extendable; more sorting cells can be easily added on for a larger computing task. However, if the array contains a large number of the sorting cells, the sorting engine would consume more on-chip resource, and be placed into a large area on the FPGA die. Thus, how to make sure the control and data buses transmit the control and data signals to each sorting cell without serious time skews becomes a problem. The distribution skew and long critical path keeps the working frequency low, and impairs the design performance.

To minimize the skew, "big and fat wire" is needed. The global buffers (or BUFGs) are such primary low-skew routing resource on Xilinx FPGA, which is used to distribute clocks or other control signals across the entire device [15]. The control signal buses of the sorting engine for the signal "hold" and "shift" can be routed with these resource.

However, the low-skew routing resource such as BUFGs is sparse. The FPGA used in our project does not have many BUFGs to route our data buses. On the one hand, we pin our hope on a future ASIC solution and tolerate the low performance prototyping in the current FPGA implementation.

Still, there are design rules and experience, such as I/O layout guidelines and

data bus layout guidelines. which can be followed to improve performance. Simply speaking. the guideline tell us that most FPGAs are designed such that their I/O pins are placed in columns on the left and right edge of the device. For example, in Fig.4.21, there is a FPGA has its I/O pins separated into three columns:

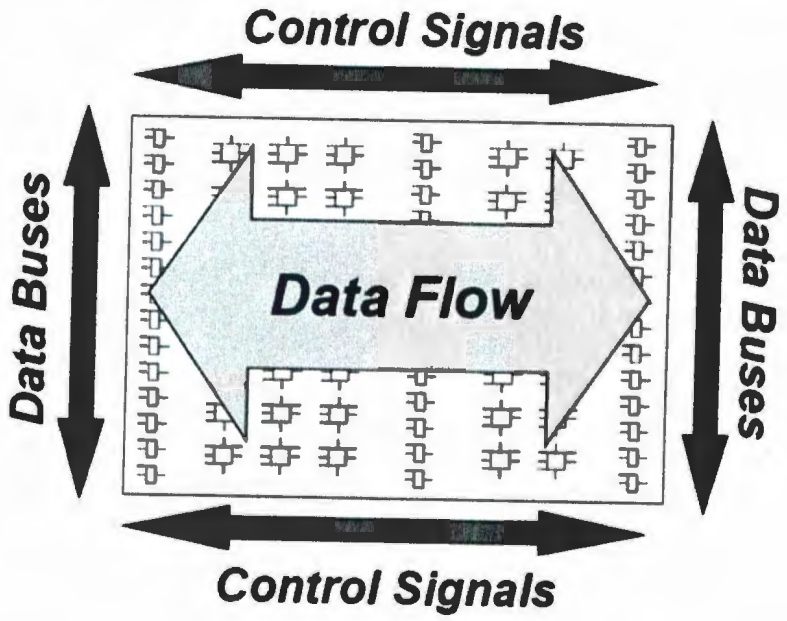


Figure 4.21: Layout rules.

The columns on the left and right edge are designed for data I/O. The center column of I/O pins is designed for dedicated clock pins. Users should plan on assigning clocks and secondary control signals such as global reset signal to the middle column I/O pins. This will ensure minimal routing delays to these global buffers and the DCMs (Digital Clock Manager). Data signals can be assigned to any other I/O pin (in the middle or on the two edges). The user may want to plan on having target logic assigned to an area near the source input pins and the destination output pins. Typical data paths flow left to right, or right to left (in the transverse direction). Although possible, critical nets do not usually route vertically. This implies that

users should try to plan critical data paths such that the source and destination pins do not travel very far vertically. This will help improve the system speed by shortening the internal routing delays of data paths. To summarize these, some layout rules can be illustrated as Fig.1.21 shown. The implementation tools usually do a better job when following these rules.

4.5.3 Synthesis Result

After synthesis, the device utilization summary of a single sorting cell can be shown in Fig. 1.22. The minimum period is 5.126 ns (maximum frequency: 195.072 MHz) in which logic delay is 1.084 ns (21.1%) and route delay is 4.042 ns (78.9%). The PAR timing result is 5.493 ns for the achievable best case delay.

The device utilization summary of 100 sorting cells can be shown in Fig. 1.23. This sorting engine including 100 sorting cells takes more than a half LUTs resources of the FPGA chip.

The minimum period is 5.126 ns (maximum frequency: 195.072 MHz) in which logic delay is 1.298 ns (22.2%) and route delay is 4.550 ns (77.8%). The PAR timing result is 6.486 ns for the achievable best case delay. The results are actually not too bad at this stage; the distribution skew is not that serious at the scale of hundreds of sorting cells.

4.6 Memory System

The travel time computing is carried out on finite difference grids. In the system, there are memories to store the grid arrays. These memory components and corresponding memory access management modules together compose the memory system of the travel time computing engine. This memory system manipulates the data flow

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	18	28,800	1%
Number used as Flip Flops	18		
Number of Slice LUTs	153	28,800	1%
Number used as logic	153	28,800	1%
Number using O6 output only	152		
Number using O5 and O6	1		
Slice Logic Distribution			
Number of occupied Slices	87	7,200	1%
Number of LUT Flip Flop pairs used	153		
Number with an unused Flip Flop	105	153	68%
Number with an unused LUT	0	153	0%
Number of fully used LUT-FF pairs	18	153	31%
Number of unique control sets	1		
IO Utilization			
Number of bonded IOBs	198	180	11%
Specific Feature Utilization			
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		

Figure 4.22: Device utilization summary for a single sorting cell.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	4,800	28,800	16%
Number used as Flip Flops	4,800		
Number of Slice LUTs	15,212	28,800	52%
Number used as logic	15,212	28,800	52%
Number using O6 output only	15,112		
Number using O5 and O6	100		
Slice Logic Distribution			
Number of occupied Slices	5,928	7,200	82%
Number of LUT Flip Flop pairs used	15,212		
Number with an unused Flip Flop	10,412	15,212	68%
Number with an unused LUT	0	15,212	0%
Number of fully used LUT-FF pairs	4,800	15,212	31%
Number of unique control sets	1		
IO Utilization			
Number of bonded IOBs	100	180	55%
Specific Feature Utilization			
Number of BUFG/BUFGCTRLs	2	32	6%
Number used as BUFGs	2		

Figure 4.23: Device utilization summary for sorting engine with 100 cells.

through each function module and storage module, organizes all the parts working jointly to form the datapath of the travel time computing system.

4.6.1 System Overview

Components

The RTL diagram of the memory access logic can be shown in Fig. 4.24

There are three massive memory modules in the system, including two RAMs (travel time RAM and flag bit RAM) and one ROM (velocity ROM, which is read only. Contents of the ROM is pre-defined and unchanged during the computing). Besides the massive memory, there are five pieces of small size memory which are used as buffers (one is used in "data_input_buffer" module, three are used in "other_input buffer" module, and the other one is used in the "result_buffer" module). The memory system can be divided into three parts with five modules:

1. The first part includes "Data_addr_gen" module and "Data input_buffer" module. "Data_addr_gen" module generates the indexes of a block of 25 grid points in the travel time array (please refer to Fig.3.1). "Data_input_buffer" is the buffer to save the 25 travel time values as input to the computing unit (see Fig.3.1). These two modules in this part fetch the data block according to the 25 addresses generated by "Data_addr_gen", and saves the data into "Data_input_buffer", once the control signal is received from controller, the "Data_input_buffer" can send the 25 travel time values to the computing unit.

2. The second part of memory system includes "other_mem_addr_gen" module and "other_mem_input_buf" module. This part is used to accomplish the function of memory access to velocity ROM and flag RAM (other than travel time RAM.

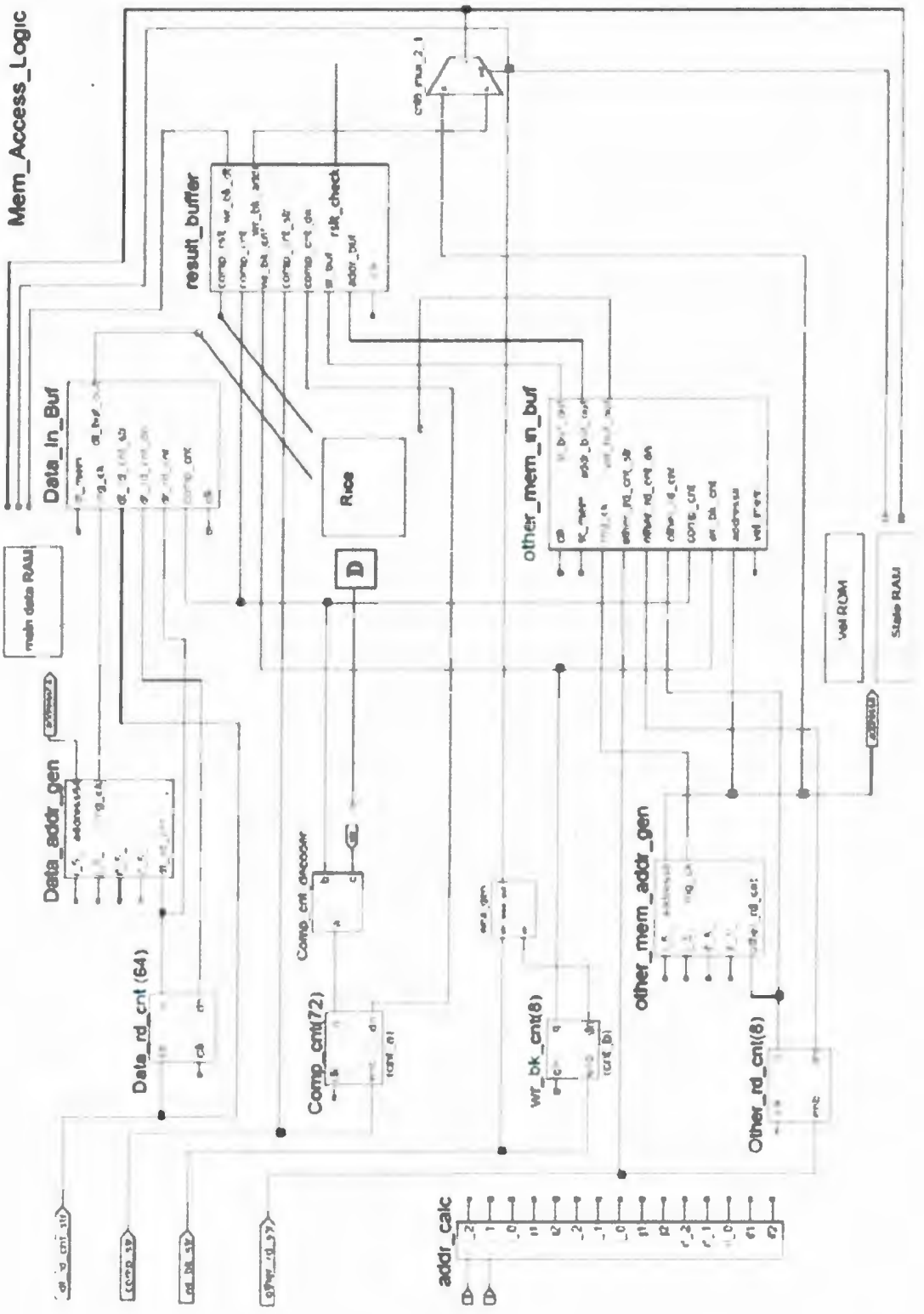


Figure 4.24: RTL diagram of memory system.

so called "Other mem_addr gen"). "Other mem addr gen" generates the addresses of 8 grid points in velocity array and flag bit array. These 8 points are the neighbor points of the starting point (see Fig.3.1). "Other mem input _buf" module has two pieces of buffer memory in it, which store the values fetched from velocity array and state flag bit array. The values in this buffer will be fed to computing unit as inputs when receiving the "computing start" signal.

3. The third part of the memory system is the result buffer. It has one buffer memory to save the result coming out of computing unit. After a round of computing, the results are sent back to travel time RAM, state flag bit RAM and sorting engine from this result buffer module.

Working Flow

In Fig. 4.24, at beginning of each computing iteration, the index i (x) and j (z) of minimum travel time points from sorting engine is sent to the module "addr_calc" which calculates the indexes of the rows and columns adjacent to minimum travel time point, such as $i-2$, $i-1$, $i+1$, $i+2$, $j-2$, $j-1$, $j+1$ and $j+2$. Meanwhile, "addr_calc" determines whether these indexes are valid (the index cannot be out of range as a number smaller than 0 or larger than the data array size), and there are flag bit output matching with each index to indicate its validity (whether it is out of range). For example, "if1" is used to notify whether "i1" (represents the index of $i+1$) is valid. The "addr_calc" module is implemented with combinational circuit. These resulting indexes from "addr calc" are then sent to "Data_addr gen" and "Other mem_addr_gen" to generate the memory addresses.

There are counters combined with decoders to control the operation of the mod-

ules. In Fig. 4.24, there is a "Data_rd cnt", which is a 64 counter. When "Data rd cnt" is started, it is counting from 0 to 63. in this process, the "Data addr_gen" and "Data input buffer" work together to save 64 travel time values into the data buffer. The 64 values are different combinations of the 25 travel time values in the data block fetched from memory as Fig. 4.25 illustrates. The values are organized in the order as shown in Fig. 4.25 because the computing unit can directly receive 8 values in a row to compute them.

Similarly, there is a "Other_rd cnt", which is a 8-counter, used to control the operation of reading 8 velocity values and 8 state flag bits from corresponding memory components into "Other_mem_input_buf".

After these two reading operations, computing counter "Comp cnt" is started: the travel time values in the "Data input_buffer", and velocity values and flag bits in "Other_mem_input buf" are fed into computing unit; at the output port of computing unit, the results are continuously coming out, and further written into "result buffer". The computing counter has 72-counts, because in the computing unit we reuse a rice computing unit for 8 times as shown in Fig. 4.25. Each rice computing in our design takes 9 clock cycles (8 clock cycles are for 8 triangle computing, and one more clock cycle is for output buffering.)

When computing finishes, the 8 results are all stored in "result buffer". There is a write back counter "wr_bk_cnt", reading the result out one by one back to travel time RAM, state flag bit RAM and sorting engine.

Generally speaking, there are three main steps in the memory system working flow, which are controlled by three counters and corresponding decoders. The three steps are reading, computing and writing back. The "start" signals of these counters are issued from external controller.

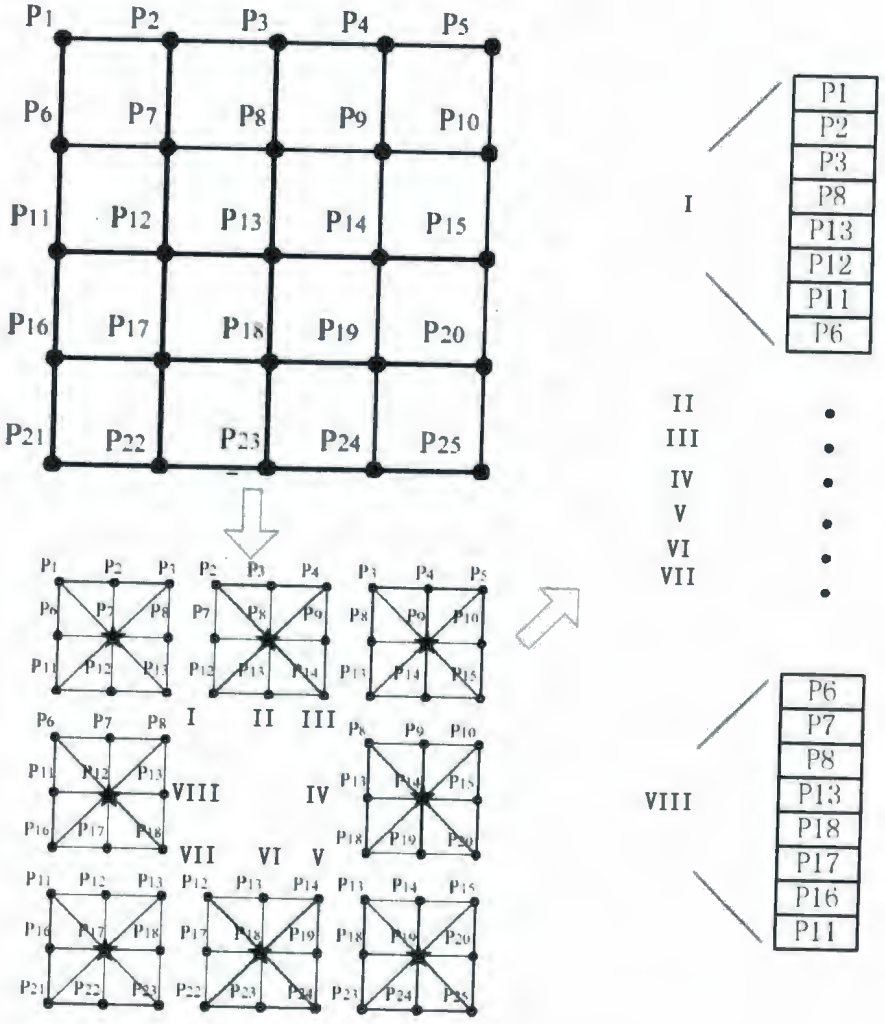


Figure 4.25: Data layout in the memory.

4.6.2 Xilinx Memory Solution

Before presenting the design of each module in the memory system, it is better to look at the technology of the memory implementation. Xilinx provides several memory solutions for different applications.

Block RAM

In Virtex-5 FPGA, there are a large number of 36 *Kbit* block RAMs as shown in Fig. 4.26 [46]. Each 36 *Kbit* block RAM contains two independently controlled 18 Kb RAMs. These block RAMs (also known as BRAM) are dedicated hardware resources on chip. They can be cascadable to enable a deeper and wider memory implementation, with a minimal timing penalty. From Xilinx core generator tool, user can easily implement the components such as embedded dual/single port RAM modules, ROM modules, synchronous FIFOs, and data width converters, using the block memory modules. For our Virtex-5 XC5VLX50 FPGA, there are 48 36 *Kbit* block RAMs on the chip, and 1728 *Kbits* memory in total.

In our design, the travel time data array, velocity array and "Data_input_buf" are implemented with BRAMs. The three memory blocks mentioned are all large enough for this kind of BRAM implementation. Furthermore, the memory block created from BRAMs has the other feature that, the read port and write port can be in different width. For example, in our "Data_input_buf" implementation, through the Xilinx core generator tool, the write port of the buffer is set to be 32 *bit* wide which is the width of a single precision floating point number; while the read port is set to be 256 *bit* that is the width of 8 single precision floating point numbers (this is because the input for computing unit is 8 single precision floating point numbers). Moreover, there is RAM-based shift register that can provide a very efficient multi-bit wide shift

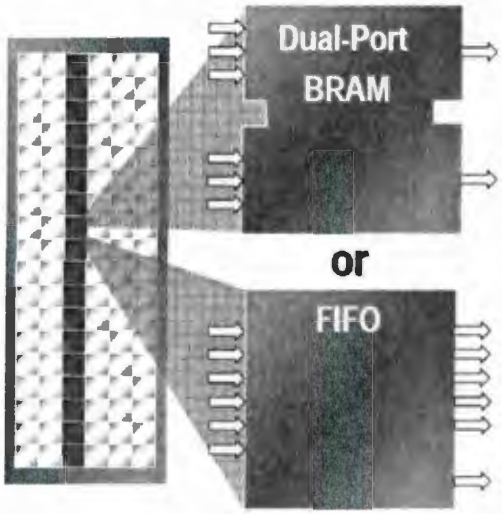


Figure 4.26: Block RAM.

register for use in FIFO-like applications or as a delay line or time skew buffer. The fixed length shift registers and variable length shift registers are built from BRAM primitives without using any of the slice resource [47].

Distributed RAM

Distributed RAM[48] is the other memory solution. It is another functionality of the LUT. Each LUT can be programmed to implement a tiny 16-bit RAM. For large size memory, block RAM should be always considered over the distributed RAM. because LUTs can also be used for combinatorial logic, which BRAM are dedicated resources only for memory implementation. For our Virtex-5 XC5VLX50 FPGA. there are 7200 Virtex-5 slices (each Virtex-5 slice contains four LUTs and four flip-flops). and 480 *Kbit* distributed memory can be generated at maximum.

In our design. the "velocity_input_buffer" (contains eight 32 *bit* vector). "flag_bit_input_buffer" (contains 8 bits) and "result_buffer" (contains eight 32bit vectors) are all small size buffers. the distributed memory is good solution for these applications.

Memory Interface Generator

MIG (Memory Interface Generator)[49] is a tool used to generate memory interfaces for Xilinx FPGAs. MIG generates Verilog or VHDL RTL design files, UCF constraints, and script files. The script files are used to run synthesis, MAP, and PAR for the selected configuration. The tool takes inputs such as the memory interface type, FPGA family, FPGA devices, frequencies, data width and memory mode register values from the user through a graphical user interface (GUI). The tool generates RTL, SDC, UCF, and document files as output. By building up memory interface, user can take advantage of the on-board memory resource. In this way, it provides solution for extra large memory use.

4.6.3 Module implementation

There are 5 modules in the memory system to accomplish "data_memory_access" and "other_memory_access" functions.

Data memory address generator

The structure of the "Data_addr_gen" module can be shown in Fig.4.27. This module receives the indexes from the "addr_calc" module, which are the row indexes related to i (x) and column indexes related to j (z). There are two multiplexers whose control ports are connected to a counter decoder. The decoder issues control signals to multiplexers, and the multiplexers select right index combinations to form memory addresses. Meanwhile, the same counter decoder issues signals to control the other two multiplexers to generate corresponding flag bit for each address to indicate its validity (whether the address is out of range). In this way, the outputs of this module are travel time memory addresses and corresponding flag bits along with the addresses.

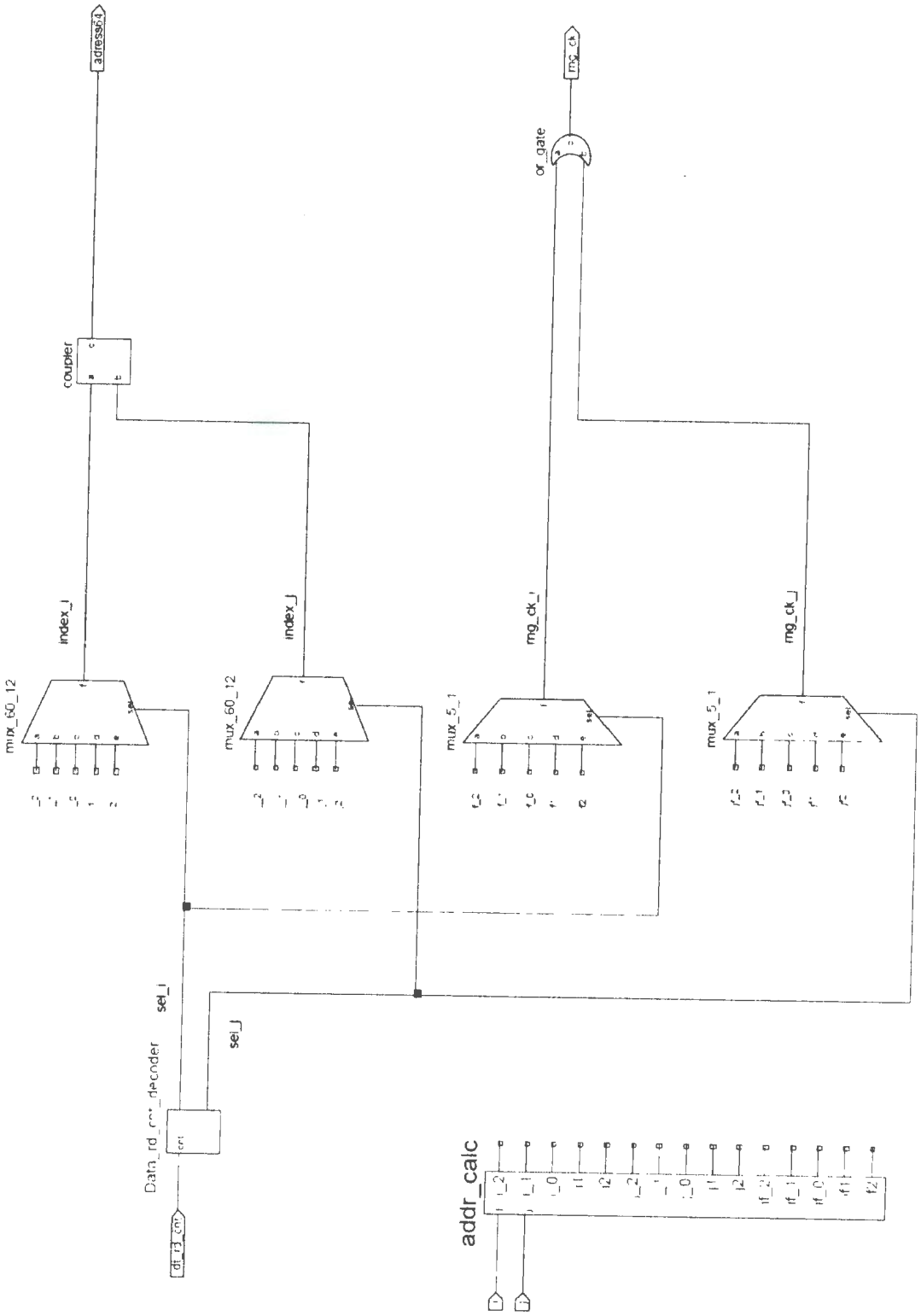


Figure 1.27: Design of data memory address generator.

Other memory address generator

The structure of the "Other_mem addr_gen" module which generates the addresses for 8 neighbor point velocity values and 8 state flag bits can be shown in Fig.4.28.

The design of this module is similar to the one in "Data_addr_gen". The difference is that there are only 8 items to generate, so the design can be smaller. The state flag bits generated in this module are used to indicate whether the 8 neighbor points of the starting point are computed or not. These 8 grid points will be assigned the 8 rice computing results (these 8 grid points are notified by "start" marks as in Fig.3.1).

Data memory input buffer

The addresses generated by "Data addr_gen" are sent to travel time memory to fetch the 25 data block. The fetched data is saved in "Data input buffer" module. The structure of this module can be shown in Fig.4.29.

In the module, the buffer memory is a piece of dual port memory created from block RAMs. As mentioned before, the input port and output port of this memory are different in width.

Before sending into the input port of data buffer, the travel time values fetched from travel time memory are first flowed through a multiplexer. The control port of this multiplexer is linked to the range check result from "Data_addr_gen" module. If any address is invalid (out of address range), the multiplexer will select "7f800000"[5] as the travel time value for this addresses.

The "wea" signal in Fig.4.29 means write enable. In our design, a module called "wea_gen" is used to generate a level signal as write enable signal from two pulse

[5]Which is the Hex representation of a binary vector representing positive infinity in single precision floating point format.

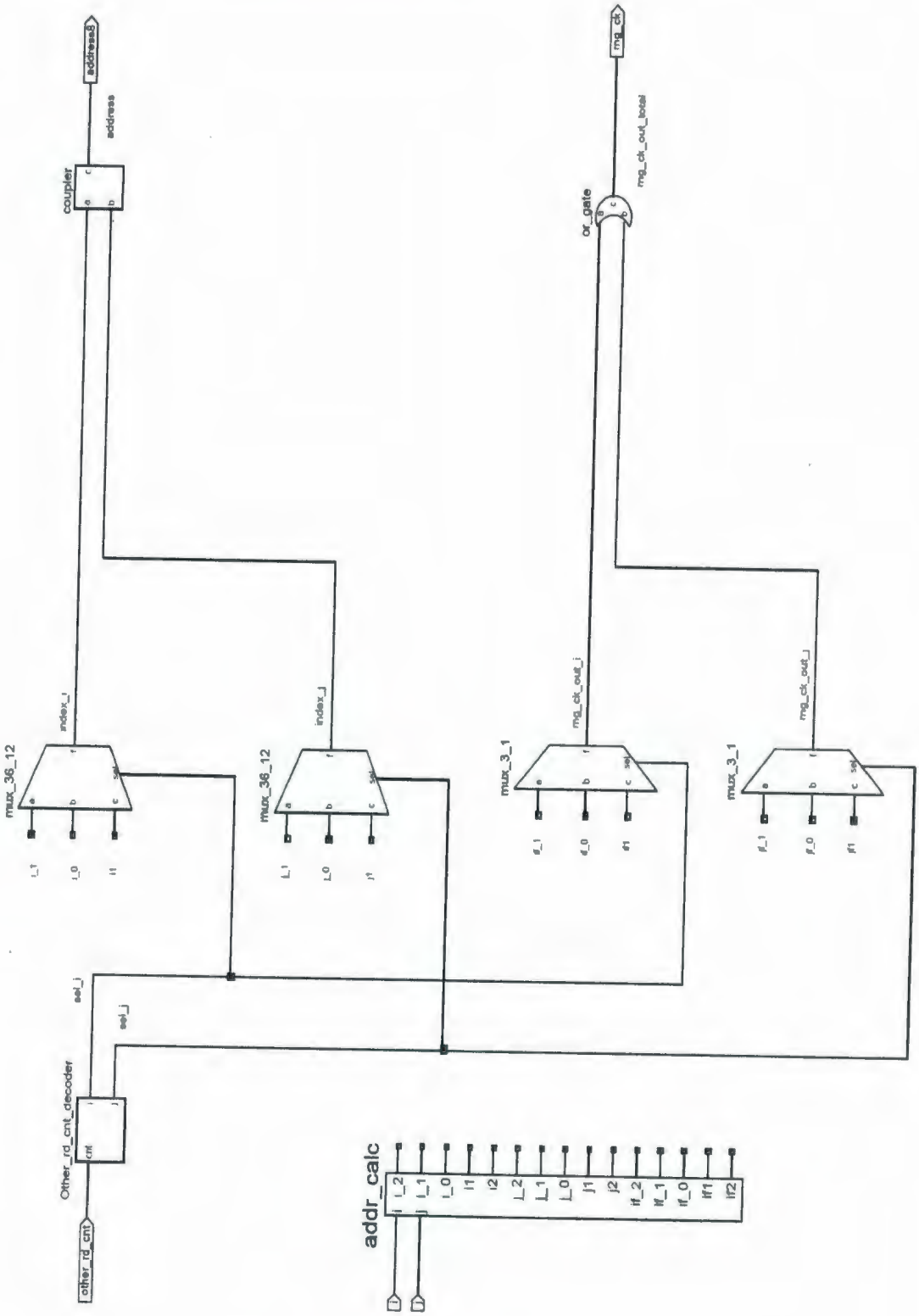


Figure 4.28: Design of other memory address generator.

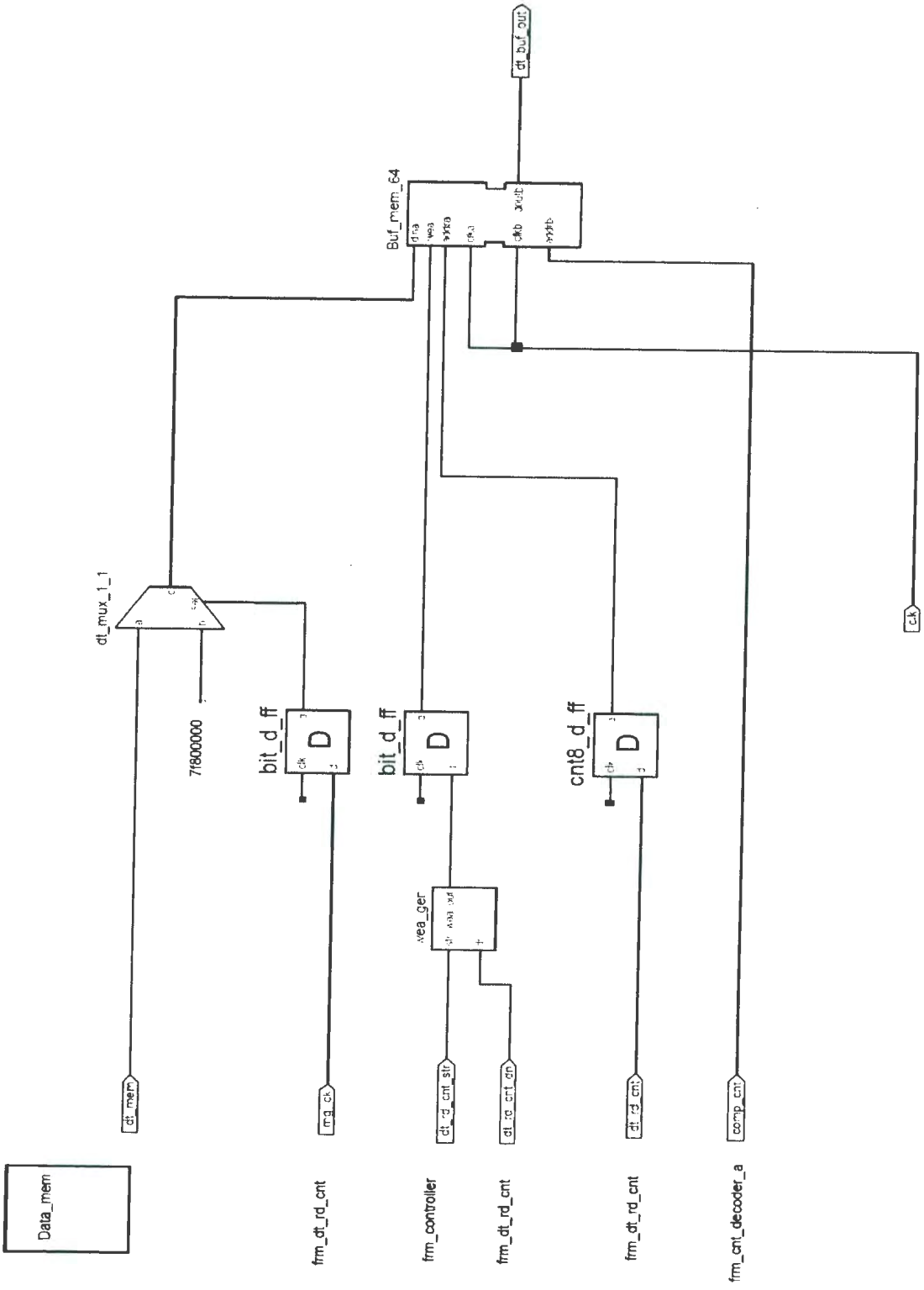


Figure 4.29: Design of data memory input buffer.

control signal inputs. The write address port of the buffer memory is connected to the "Data rd_cnt" counter shown in Fig. 4.24. The read address port is linked to "comp cnt decoder"; when computing starts, the travel time values are read out and sent to the rice computing unit.

In Fig.4.29, it may be noticed that there are delay registers embedded in the circuit. The reason to use these delays is that the block RAM memory and distributed RAM take different number of clock cycles to complete a write operation. The distributed RAM is created from slice LUT resources, so the content can be immediately read out without clock cycle delay. However, the minimum delay for block RAM is one clock cycle. The Xilinx core generator has the option to incorporate delay registers inside the memory as needed.

Other memory input buffer

Similar to data memory input buffer, "other_mem input buffer" module has the structure can be shown in Fig.4.30.

In this module, there are three pieces of buffer memories to save the state flag bits, travel time memory addresses of the 8 result points and 8 velocity values of them.

The flag bits saved in "Buf mem st" are decided by two factors: whether the grid point in the travel time array is in the computed set or not; whether the address of that grid point is out of range. These two factors are incorporated together by the "or gate", and the results are saved into the buffer.

The 8 addresses of the result points saved in "Buf_mem addr" are prepared for future write back operation.

The input velocity values flow through the multiplexer before being saved into the buffer memory. If the corresponding flag bit indicate the grid points are not a

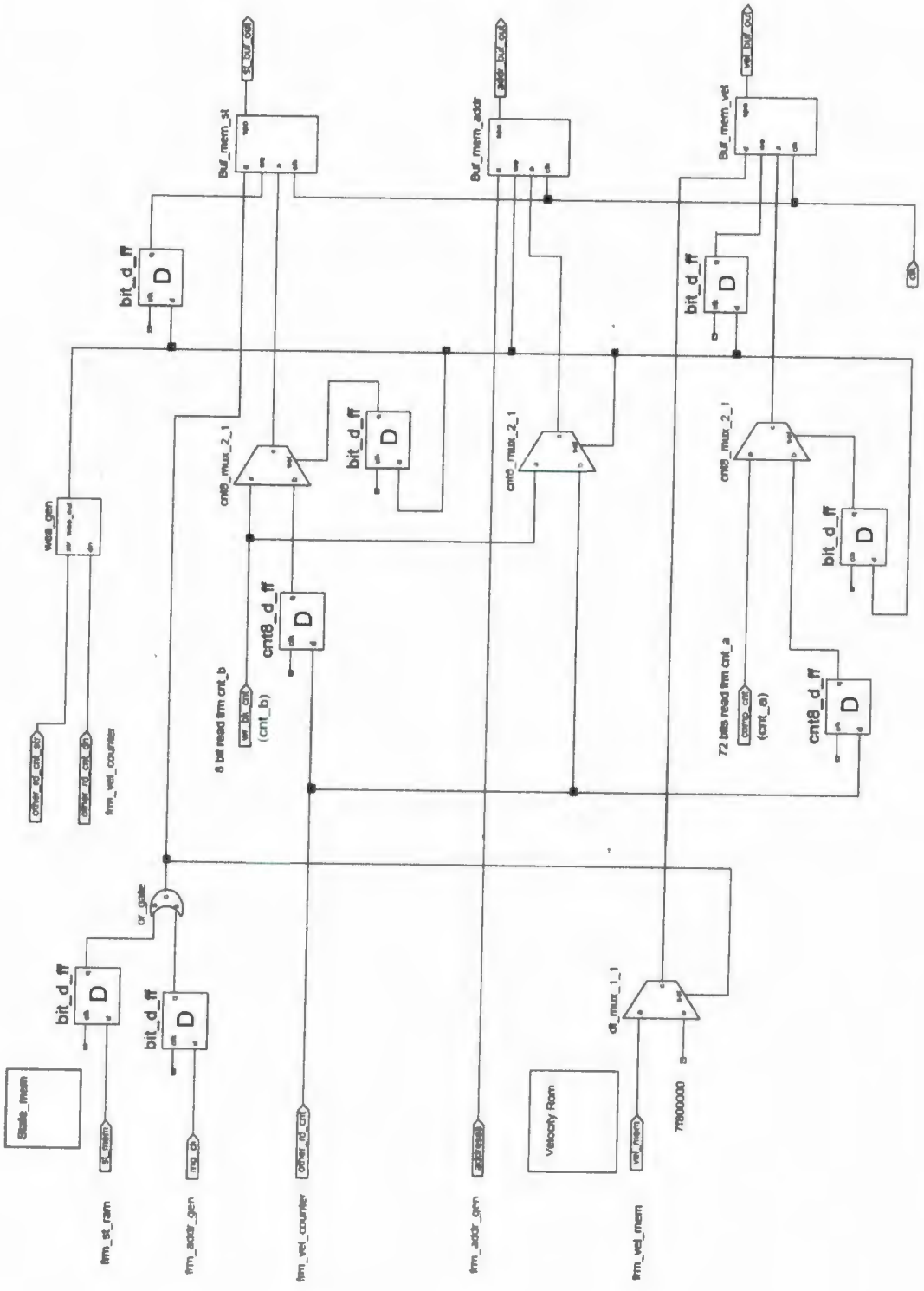


Figure 4.30: Design of other memory input buffer.

"good"[9] one, the velocity value is set to be "7f800000".

Moreover, for each of these 3 buffers, there are two address inputs. One of the two addresses is for read operation and the other is for write operation. They are sent to a multiplexer. The write enable signal "wea" for the buffer memory is connected to the control port of the multiplexer. When the "wea" signal is "on", the buffer memory are switched to write mode, meanwhile the multiplexer output the write addresses.

Result buffer

The structure of "result_buffer" can be shown in Fig.4.31.

The design style of this module is consistent with other modules. There is an address multiplexer can switch between read address input and write address input.

The "equ_ck" unit check whether the result from computing unit is equal to "7f800000". Because if any input of the rice computing is "7f800000" (infinity), the result will be "7f800000" (infinity); furthermore, "7f800000" (infinity) always means an invalid input (invalid inputs can be the case that the grid point is out of range, or the travel time on that grid point is uncomputed). No matter in which case, if the result is not "good", this result will not be written back to the corresponding address. On contrary, during write operation, all the "bad" results will be placed into a same special memory address, for example, the first or the last address in the memory.

4.6.4 Synthesis Result

The device utilization summary of the memory access system (including computing unit and all three memories) can be shown in Fig. 4.32. The minimum period is 88.201 ns (maximum frequency: 11.337 MHz) in which logic delay is 17.011 ns

[9] the meaning of "good" is explained in the section 2.2.

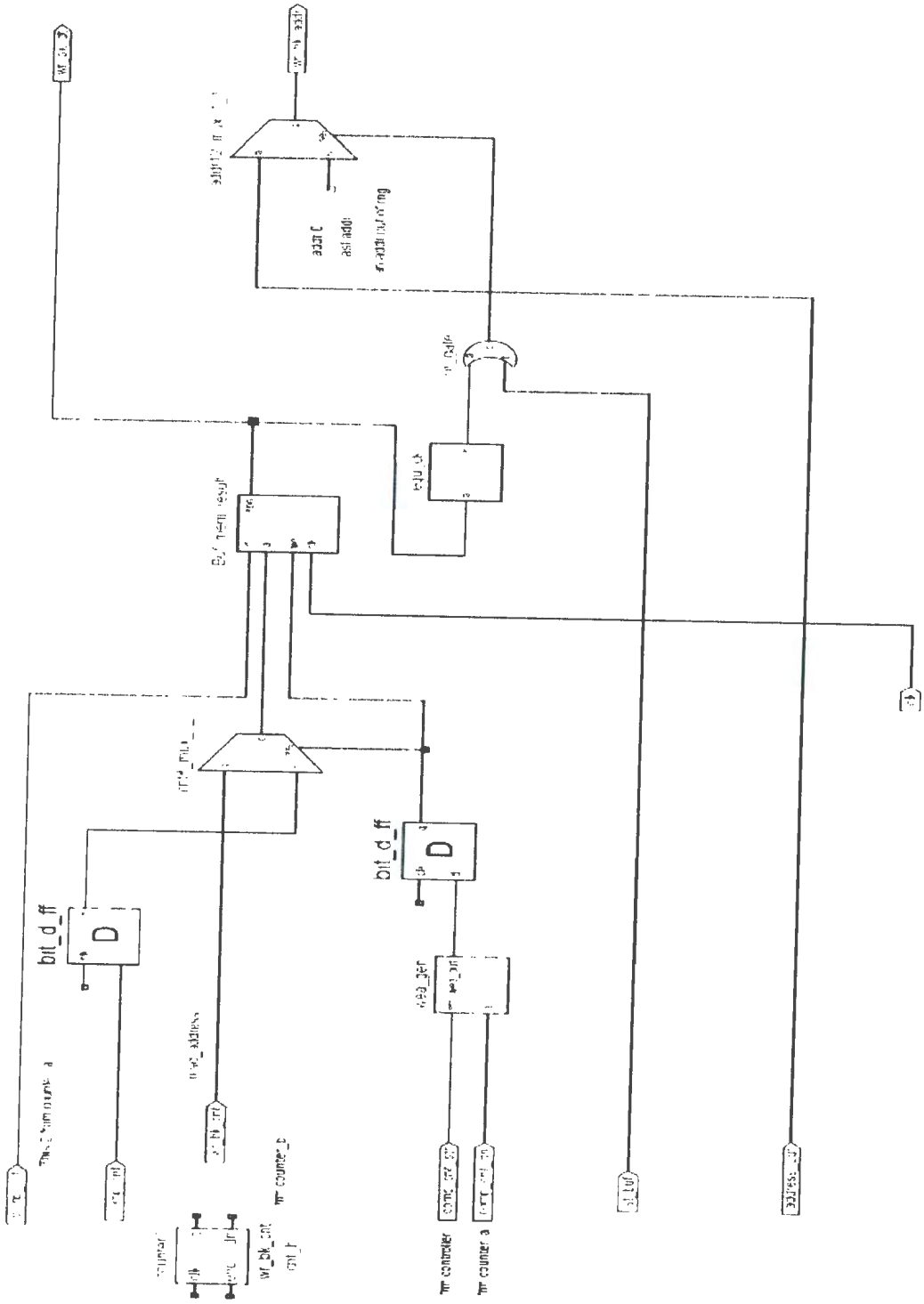


Figure 4.31: Design of result buffer.

(53.3%) and route delay is 11.163 ns (16.7%). The PAR timing result is 99.365 ns for the achievable best case delay.

4.7 Top Level Design

After building up all the modules, the last step is to assemble all these modules together to make them work together. Fig. 4.3 shows the block diagram of the system design, in which the data flow and control flow are illustrated. In Fig. 4.33, there is the top level implementation of the system. Comparing to Fig. 4.3, modules including memory, buffers, access control logic and computing unit are shown into the upper level "memory access" module. This module realizes the data path of the algorithm as shown in Fig. 4.24.

4.7.1 Controller Design

The FSM of the controller can be designed as shown in Fig.4.34. When the "start" signal is sent to controller, the FSM (Finite State Machine) of the controller begins to run. It initializes the sorting engine by loading the travel time data of seed points into sorting cells. An initialization counter is used to count the initialization cycles. Processes of "reading", "computing" and "writing back" are started in turn by the "start" signals which are pulse signals issued from the controller FSM. The computing stops and FSM goes back to "idle" state, when the iteration counter reaches the number which is set before hand.

Device Utilization Summary			
Slice Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	121	28,800	1%
Number used as Flip Flops	118		
Number used as Latch-thrus	3		
Number of Slice LUTs	3,127	28,800	10%
Number used as logic	3,003	28,800	10%
Number using O6 output only	2,564		
Number using O5 output only	58		
Number using O5 and O6	381		
Number used as Memory	121	7,680	1%
Number used as Single Port RAM	121		
Number using O6 output only	101		
Number using O5 and O6	20		
Number used as exclusive route-thru	3		
Number of route-thrus	61	57,600	1%
Number using O6 output only	61		
Slice Logic Distribution			
Number of occupied Slices	1,186	7,200	16%
Number of LUT Flip Flop pairs used	3,146		
Number with an unused Flip Flop	3,025	3,146	96%
Number with an unused LUT	19	3,146	1%
Number of fully used LUT-FF pairs	102	3,146	3%
Number of unique control sets	40		
IO Utilization			
Number of bonded IOBs	62	480	12%
Specific Feature Utilization			
Number of BlockRAM/FIFO	16	60	26%
Number using BlockRAM only	16		
Total primitives used			
Number of 36k BlockRAM used	16		
Total Memory used (KB)	576	2,160	26%
Number of BUFG/BUFGCTRLs	1	32	3%
Number used as BUFGs	1		
Number of DSP48Es	18	48	37%
Number of RPM macros	6		
Total equivalent gate count for design	2,144,182		
Additional JTAG gate count for IOBs	2,976		

Figure 4.32: Device utilization summary for memory access system.

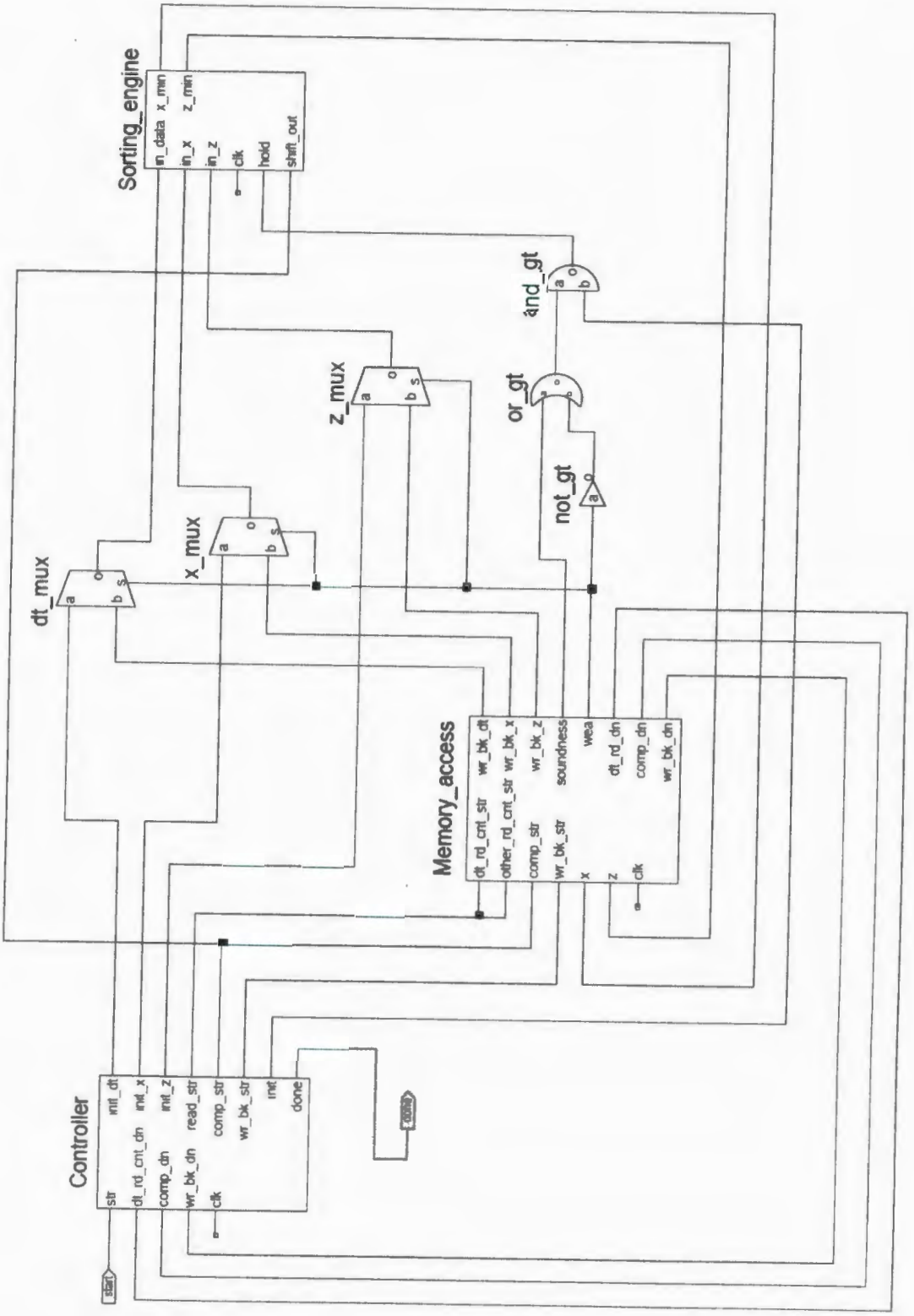


Figure 4.33: Top level diagram.

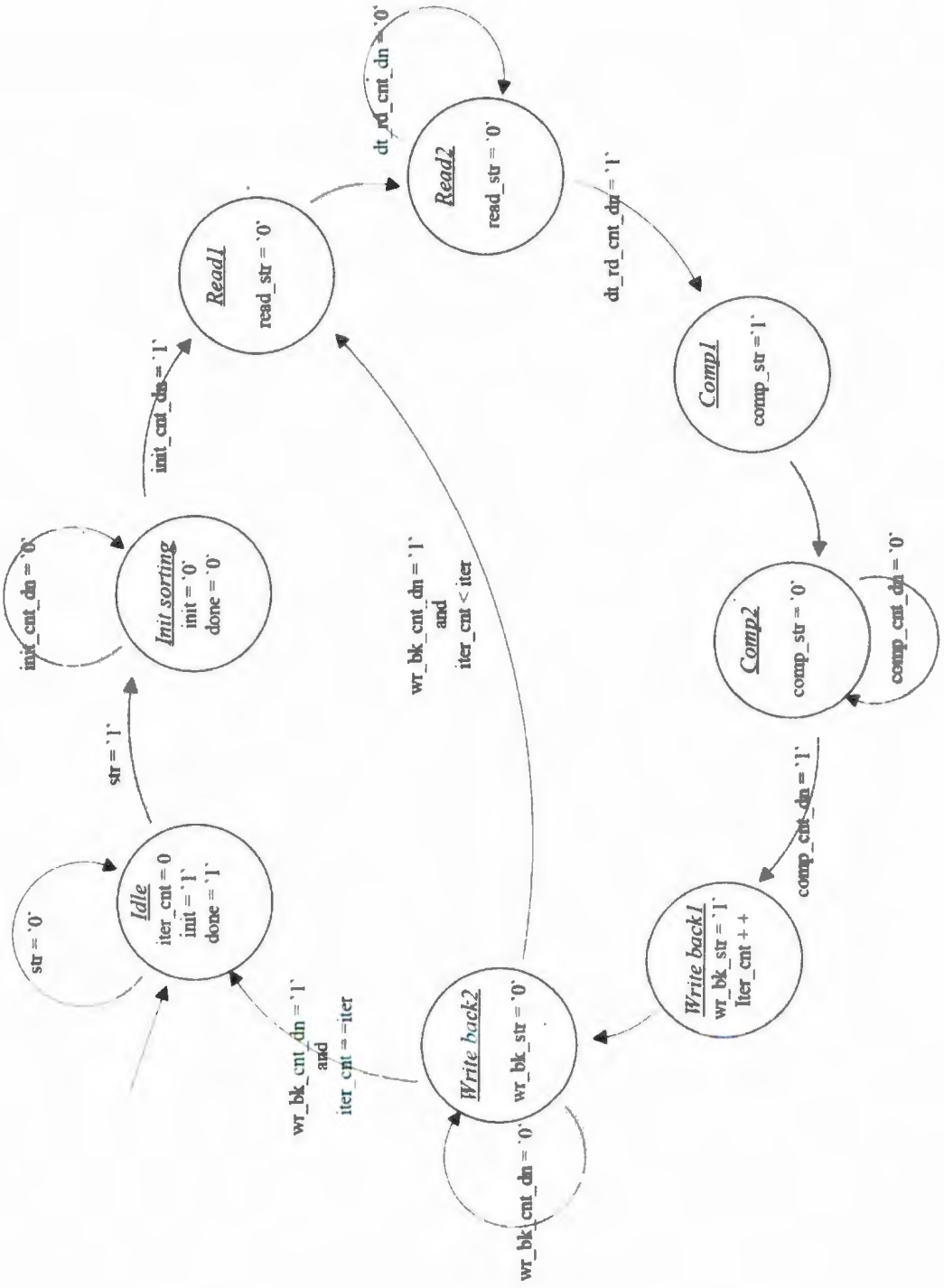


Figure 4.34: Top level controller.

4.7.2 Synthesis Result

To implement a travel time engine with 170 sorting cells in the sorting engine, the device utilization summary of the whole system as shown in Fig.4.35 can be listed in Fig.4.35. This is too small for real 2D and 3D problems. The length of the sorting engine is $O(n)$ for 2D and $O(n^2)$ for 3D, where n is the largest number of grid cells in any dimension.

Device Utilization Summary (estimated values)			
Logic Utilization	**Used**	**Available**	**Utilization**
Number of Slice Registers	216	28800	0%
Number of Slice LUTs	28219	28800	98%
Number of fully used LUT-FF pairs	205	28290	0%
Number of bonded IOBs	1	180	0%
Number of Block RAM/FIFO	16	60	26%
Number of BUFG/BUFGCTRLs	2	32	6%
Number of DSP48Es	18	48	37%

Figure 4.35: Device utilization summary for travel time engine.

In our travel time engine system, the limit of travel time array size is decided by the amount of block RAM resource on the Virtex-5 FPGA chip. The size of sorting engine is limited by the LUT resources. As listed in Fig.4.35, it consumes around 98% LUT resource on the chip to implement a travel time engine which has 170 sorting cells in its sorting engine. This is almost an upper limit for the design size on a single Virtex-5 FPGA chip.

From synthesis, the estimated minimum clock period is 1514.212 ns (maximum frequency: 0.66 MHz) in which logic delay is 264.569 ns (17.5%) and route delay is 1249.643 ns (82.5%).

Chapter 5

Summary and Conclusions

5.1 Conclusions

The research in our group is mainly concerned with Computational Geophysics, Reservoir Characterization and Seismic Modeling/Migration problems in oil and gas exploration. To accelerate computing speed and improve the processing efficiency of seismic migration process in subsurface survey and oil exploration, we developed a new methodology including a novel algorithm and a corresponding software/hardware implementation frame to carry out the travel time computing on a reconfigurable digital circuit platform.

Through the survey of seismic migration, as well as other seismic data processing techniques in the oil and gas industry, the focus is put on the seismic travel time computation problem for Kirchhoff migration. The first chapter reviewed geophysics fundamentals, explaining concerning concepts and summarizing the significance of our research topic. Moreover, in the first chapter, the two major methods to calculate seismic travel times, ray tracing method and eikonal equation solving method, are introduced. The fast marching method that solves eikonal equations in finite differ-

ence grids is reviewed. We propose our novel least-time path fast marching method in the second chapter, through combining the advantages of ray tracing method, eikonal equation solving method and fast marching method, mainly based on the theory of Vidale and Sethian. Mathematics of the method is presented in two aspects: the local scheme of extrapolation and the inductive scheme. High level design of the algorithm is proposed: the algorithm parallelism and 3D extension are discussed.

In chapter 3, software simulation is run for verification of the least-time path fast marching method, and the simulation result is presented. Based on the software simulation, a parallel version of the algorithm is proposed. The program design for multi-processor programming with MPI technique is discussed.

The major hardware system design work is written in chapter 4. The chapter begins with the analysis of the necessity and feasibility of the design of a digital system to implement the algorithm onto a FPGA chip. Then, following the FPGA design flow and the top-down design methodology, our design is introduced stage by stage from top level block diagram to the RTL structure map of each module. At the same time, the feature of the FPGA platform we chose, the methodology and technique for FPGA synthesis, and implementation are discussed based on our design practice. The synthesis results of the design are compared and analyzed.

We also summarize our experience of choosing our research direction. We invent a new algorithm to improve the data processing in the oil and gas industry. We propose the parallel program solution of the algorithm in software. We also raise the idea of building a application specific computing chip to gain the maximum parallelism and acceleration. The project has been an exciting experience of creative academic research.

5.2 Future Work

This thesis is the first dissertation on topic of **Least Time Path Fast Marching Method** from our research group. It is the initial work, and the approach of using a FPGA solution has been demonstrated. There are several suggested points can be taken as the direction for future research work:

1. As mentioned in chapter4, because the limitation of technology and design capability, implementation of the sorting engine presented in this thesis is not optimal. Some of the modules can be refined with more testing efforts to find out optimized parameters, for instance, the performance of computing unit can be improved through more experiments of the Xilinx floating point cores. Furthermore, adopting advanced technology in the future, such as abundant on-chip global low skew routing resources and high-speed external memory interface, will largely facilitate the design of sorting engine and the memory system.

2. The parallelism potential of this travel time engine can be further examined. The concept discussed in previous chapters, such as network computing on chip and FPGA array should be tried out in the future research. Our algorithm is best suited to be implemented onto a FPGA array, and the applicability on this platform actually determines the success of the method.

3. In future, cross-platform comparison among different implementation solutions on this travel time computing problem will be an interesting research topic. The various implementation platforms include sequential programming on single CPU, MPI programming on multiple processor cluster, GPU[1] programming,

[1]GPU is a dedicated graphics rendering device for a personal computer, workstation, or game console. Modern GPUs are very efficient at manipulating and displaying computer graphics, and

cell[2] processor programming and reconfigurable computing on FPGA.

1. This travel time engine can be commercialized. Furthermore, the product can be designed as a platform that would fit the application of a class of finite difference computing methods with similar structure, in which only a small amount of modification is required on the reconfigurable chips for each algorithm.

5. Besides technology and implementation issues, there are some more work can be done to improve the algorithm itself. Due to the increasing demand on 3D data processing capability from oil and gas industry, the seismic migration algorithms are evolving to their 3D version, the same trend as on our algorithm.

The above issues either are currently under investigation or will be addressed in the future research.

their highly parallel structure makes them more effective than general-purpose CPUs for a range of complex algorithms. The research of taking advantage of its parallel structure and floating point processing capability to solve scientific computing problems becomes a hot topic in these days. There are some researchers in our group are now engaged in this field.

[2]Cell is a microprocessor architecture jointly developed by Sony, Toshiba, and IBM. Cell combines a general-purpose Power Architecture core of modest performance with streamlined coprocessing elements which greatly accelerate multimedia and vector processing applications, as well as many other forms of dedicated computation. We have some researchers in this field as well.

References

[1] R. Mc Quillin. *An Introduction to Seismic Interpretation*. 1st ed. London: Graham & Trotman, 1984.

[2] H. Sun. "Wavepath migration for depth imaging and velocity analysis." Doctor of Philosophy, Department of Geology and Geophysics, The University of Utah, Utah, Dec 2001.

[3] D. A. Waltharn. "Two-point ray tracing using fermats principle." *Geophysical Journal*, vol. 93, pp. 575–582, 1988.

[4] E. I. Vinje, V. and H. Gjostdal. "Traveltime and amplitude estimation using wavefront construction." *Geophysics*, vol. 58, pp. 115–1166, 1993.

[5] J. E. Vidale. "Finite-difference calculation of travel times." *Bulletin of the Seismological Society of America*, vol. 78, pp. 2062 -2076, Dec 1988.

[6] A. D. Isabelle Lecomte, Havar Gjoystdal and O. C. Pedersen. "Improving modelling and inversion in refraction seismics with a first-order eikonal solver." *Geophysical Prospecting*, vol. 48, pp. 137–151, 2000.

[7] Xilinx. *Xilinx Integrated Software Environment (ISE) Help*. Xilinx, may 2007.

[8] M. L. Overton. *Numerical Computing with IEEE Floating Point Arithmetic*. Philadelphia: Society for Industrial and Applied Mathematics, 2001.

[9] X. Zhang and R. P. Bording. "Travel time engine for migration." *NECEC 2006 Proceedings, IEEE Newfoundland and Labrador*, Nov 2006. [Online]. Available: http://necec.engr.mun.ca/ocs2006/viewrecord.php?id=21

[10] S. H. Gray, J. Etgen, J. Dellinger, and D. Whitmore, "Seismic migration problems and solutions," *Geophysics*, vol. 66, no. 5, pp. 1622–1640, 2001.

[11] T. S. Rappaport, *Theory of Seismic Imaging*, 2nd ed. Heidelberg: Springer Berlin / Heidelberg, 1995.

[12] P. Plasterie and D. Chagalov, "Wave equation versus kirchhoff pre-stack depth migration algorithms - an australian case study," *AESC Proceedings, Australian Earth Science Convention Melbourne 2006*, Jun 2006.

[13] J. Um and C. H. Thurber, "A fast algorithm for two-point seismic ray tracing," *Bulletin of the Seismological Society of America*, vol. 77(3), pp. 972–986, 1987.

[14] B. R. Julian and D. Gubbins, "Three-dimensional seismic ray tracing," *Geophysics*, vol. 43, pp. 95–114, 1977.

[15] V. Cerveny, "Seismic ray theory," *Journal of Seismology*, vol. 7, p. 543, Nov 2003.

[16] V. Pereyra, "Two-point ray tracing in general 3d media," *Geophysical Prospecting*, vol. 40(3), pp. 267–287, Apr 1992.

[17] A. L. Vesnaver, "Ray tracing based on fermat's principle in irregular gridsl," *Geophysical Prospecting*, vol. 44, pp. 741–760, 1996.

[18] R. Coman and D. Gajewski, "Traveltime computation by wavefront-orientated ray tracing," *Geophysical Prospecting*, vol. 53, pp. 23–36, jan 2005.

[19] S. H. Gray and W. P. May, "Kirchhoff migration using eikonal equation traveltimes," *Geophysics*, vol. 59, no. 5, pp. 810–817, may 1994.

[20] J. A. Sethian and A. M. Popovicia, "3-d traveltime computation using the fast marching method," *GEOPHYSICS*, vol. 64, no. 2, pp. 516–523, Mar-Apr 1999.

[21] J. A. Sethian, *Level set method*, 1st ed., ser. Cambridge Monographs on Applied and Computational Mathematics, R. V. K. P. G. Ciarlet, A. Iserles and M. H. Wright, Eds. Berkeley, Califonia: Cambridge University Press, 1996.

[22] J. A. Sethian and A. Vladimirsky, "Fast methods for the eikonal and related hamiltoncjacobi equations on unstructured meshes," *Proceedings of the National Academy of Sciences*, vol. 97, no. 11, p. 5699C5703, may 2000.

[23] S. F. Maria Cameron and J. Sethian, "Seismic velocity estimation and time to depth conversion of time-migrated images," *SEG/New Orleans 2006 Annual Meeting*, p. 3066C3070, 2006.

[24] S. B. F. M. K. Cameron and J. A. Sethian, "Seismic velocity estimation from time migration velocities," Oct 2006.

[25] Xilinx, *Embedded Processor Block in Virtex-5 FPGAs*. Xilinx, may 2008.

[26] ——, *Embedded System Tools Reference Manual*. Xilinx, may 2008.

[27] ——, *Platform Specification Format Reference Manual*. Xilinx, may 2008.

[28] L. Benini and G. DeMicheli, "Networks on chip: A new soc paradigm," *IEEE Computer*, vol. 35, no. 1, pp. 70–78, jan 2002.

[29] A. Jantsch and H. Tenhunen, *Networks on Chip*. Kluwer Academic Publishers, Feb 2003.

[30] M. v. I. J. R. David M. Lewis, David R. Galloway and P. Chow, "The transmogrifier-2: A 1 million gate rapid-prototyping system," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 6, no. 2, pp. 188–198, Jun 1998.

[31] J. R. K. C. G. P.-M. Paul Chow, Soon Ong Seo and I. Rahardja, "The design of an sram-based field-programmable gate array, part i: Architecture," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 2, pp. 191–197, Jun 1999.

[32] – –, "The design of an sram-based field-programmable gate array, part ii: Circuit design and layout," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 3, pp. 321–330, Sep 1999.

[33] J. D. A.M. Trullemans, R. Ferreira and J. Legat, "A multi-fpga system for prototyping power conscious algorithms," *15th Design of Circuits and Integrated Systems Conference (DCIS 2000)proceeding*, pp. 41–46, Nov 1999.

[34] M. J. Quinn, *Parallel Programming in C with MPI and OpenMP*. McGraw-Hill Companies, Inc. 2005, no. Aug.

[35] L. M. He, C. and C. W. Sun, "Accelerating seismic migration using fpga-based coprocessor platform," *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM04)Proceeding*, pp. 207–216, 2004.

[36] C. He, C. Sun, M. Lu, and W. Zhao, "Prestack kirchhoff time migration on high performance reconfigurable computing platform." [Online]. Available: http://faculty.cs.tamu.edu/zhao/papers/conf/2005/0511-IEAM-HSLZ.pdf

[37] Xilinx, *Virtex-5 family overview*. Xilinx, may 2007.

[38] ——, *Development system reference guide*. Xilinx, may 2007.

[39] D. Pellerin and S. Thibault, *Practical FPGA Programming in C*. Prentice Hall Professional Technical Reference, 2005.

[40] T. L. Milo*s D. Ercegovac, *Digital arithmetic*. San Francisco, CA: Morgan Kaufmann Publishers, 2004.

[41] J. J. F. Cavanagh, *Digital computer arithmetic : design and implementation*, ser. McGraw-Hill computer science series. McGraw-Hill, 1984.

[42] Xilinx, *Xilinx Floating-Point Operator v3.0*. Xilinx, may 2007.

[43] ——, *Xilinx Basic FPGA Architecture*. Xilinx, may 2007.

[44] ——, *Xilinx Virtex-5 FPGA User Guide*. Xilinx, may 2007.

[45] ——, *Virtex-5 Libraries Guide for HDL Designs*. Xilinx, Apr 2008.

[46] ——, *Block Memory Generator v2.6*. Xilinx, Oct 2007.

[47] ——, *RAM-based Shift Register v9.0*. Xilinx, jul 2007.

[48] ——, *Distributed Memory Generator v3.3*. Xilinx, Apr 2007.

[49] ——, *Xilinx Memory Interface Generator(MIG) User Guide*. Xilinx, Sep 2007.