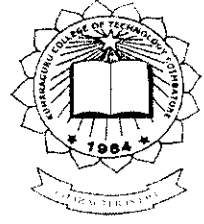
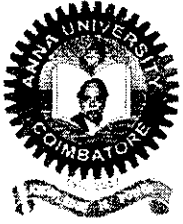


P-3450



**DESIGN OF HIGH SPEED RECURSION ARCHITECTURES FOR
WIRELESS COMMUNICATION SYSTEMS**

By

RA.ARAVINDH

Reg No: 0920106002

of

KUMARAGURU COLLEGE OF TECHNOLOGY, COIMBATORE.

(An Autonomous Institution affiliated to Anna University of Technology, Coimbatore)

COIMBATORE - 641049

A PROJECT REPORT

Submitted to the

FACULTY OF ELECTRONICS AND COMMUNICATION

ENGINEERING

*In partial fulfillment of the requirements
for the award of the degree*

of

MASTER OF ENGINEERING


IN

APPLIED ELECTRONICS

APRIL 2011


BONAFIDE CERTIFICATE


Certified that this project report titled “**DESIGN OF HIGH SPEED RECURSION ARCHITECTURES FOR WIRELESS COMMUNICATION SYSTEMS**” is the bonafide work of **Mr.RA.ARAVINDH (0920106002)** who carried out the project work(Phase one) under my supervision. Certified further, that to the best of my knowledge the work reported herein does not form part of any other project report of dissertation on the basis of which a degree or award was conferred on an earlier occasion on this or any other candidate.


PROJECT GUIDE
Mr.S.VENKATESH


HEAD OF THE DEPARTMENT
Dr. RAJESWARI MARIAPPAN, PhD

The candidate with **University Register No. 0920106002** was examined by us in Project Viva-Voce examination held on 21.04.2011


Internal Examiner


External Examiner

ACKNOWLEDGEMENT

A project of this nature needs co-operation and support from many for successful completion. In this regards, I am fortunate to express my heartfelt thanks to chairman **Padmabhusan Arutselvar Dr.N.Mahalingam B.Sc.,F.I.E.** and and Co-Chairman **Dr.B.K.Krishnaraj Vanavarayar B.Com,B.L.**,for giving this opportunity to pursue this course.

I would like to express my thanks and appreciation to the many people who have contributed to the successful completion of this project. First I thank, **Dr.J.Shanmugam, Ph.D**, Director, for providing me an opportunity to carry out this project work.

I would like to thank **Dr.S.Ramachandran, Ph.D**, Principal, who gave his continual support and opportunity for completing the project work successfully.

I express my sincere thanks to **Dr.Rajeswari Mariappan, Ph.D.**, Head of the Department of Electronics and Communication Engineering, who gave her continual support for me throughout this project.

I extend my heartfelt thanks to my internal guide **Mr.S.Venkatesh M.E.(Ph.D)**, Assistant Professor, for his ideas and suggestion, which have been very helpful for the completion of this project work.

In particular, I wish to thank and everlasting gratitude to the project coordinator **Ms.R.Latha M.E.(Ph.D)**, Associate Professor, Department of Electronics and Communication Engineering, for her expert counseling and guidance to make this project to a great deal of success.

Last, but not the least, I would like to express my gratitude to my family members, friends and to all my staff members of Electronics and Communication Engineering department for their encouragement and support throughout the course of this project.

ABSTRACT

Turbo code has an outstanding performance in industrial applications especially in wireless and satellite communications. Turbo decoders has great area efficiency but highly limited by the clock speed and the maximum number of iterations to be performed.

To facilitate iterative decoding, the Maximum A Posterior probability (MAP) algorithm has been widely used in Turbo decoding for its outstanding performance. Due to the recursive computations inherent with the MAP algorithm the conventional pipelining technique is not applicable for raising the effective processing speed. This project presents high speed recursion architectures for MAP-based Turbo decoders. Algorithmic transformation, approximation, and architectural optimization are incorporated in the proposed designs to reduce the critical path. Simulations show that neither of the proposed designs has observable decoding performance loss compared to the true MAP algorithm when applied in Turbo decoding.

Key Words: High-speed design, Maximum A Posterior probability (MAP) decoder, Turbo code, VLSI.

TABLE OF CONTENTS

CHAPTER NO	TITLE	PAGE NO
	Abstract	iv
	List of Figures	vii
	List of Tables	viii
	List of Symbols and Abbreviations	ix
1.	INTRODUCTION	1
1.1	Overview of the Project	1
1.2	Turbo Decoder	2
1.3	Software Used	4
2.	LOG - MAP ALGORITHM	5
2.1	MAP Algorithm Overview	5
2.2	LOG- MAP Algorithm Overview	8
2.3	Log-Map Turbo Decoder	9
3.	HIGH SPEED RECURSION ARCHITECTURE	11
3.1	Introduction	11
3.2	Recursive Architectures Overview	11
3.2.1	Traditional Architecture Arch-O	11
3.2.2	Radix-4 Architecture Arch-L	12
3.2.3	Advanced Radix-2 Architecture Arch-A	13
3.2.4	Improved Radix-4 Architecture Arch-B	18
3.2.5	Radix-8 Architecture Arch-C	20
3.2.6	Radix-16 Architecture Arch-D	21

4. SIMULATION RESULTS	22
4.1 Simulated results of Arch-O	22
4.2 Simulated results of Arch-L	24
4.3 Simulated results of Arch-A	26
4.4 Simulated results of Arch-B	28
4.5 Simulated results of Arch-C	30
4.6 Simulated results of Arch-D	34
5. CONCLUSION AND FUTURE WORK	36
REFERENCES	37
APPENDIX I	38
APPENDIX II	41

LIST OF FIGURES

FIGURE	TITLE	PAGE NO
Fig 1.1	A serial Turbo decoder architecture	2
Fig 1.2	Traditional SISO algorithm	3
Fig 2.1	Block diagram of MAP decoder	7
Fig 2.2	A Log-MAP Turbo decoder structure	9
Fig 3.1	Traditional architecture Arch-O	12
Fig 3.2	Radix-4 architecture Arch – L	13
Fig 3.3	Advanced Radix-2 architecture Arch-A	14
Fig 3.4	Correction function and its approximation	16
Fig 3.5	Carry- save structure	17
Fig 3.6	Structure of GLUT	17
Fig 3.7	Improved Radix-4 architecture Arch-B	19
Fig 3.8	Radix-8 architecture Arch-C	20
Fig 3.9	Radix-16 architecture Arch-D	21
Fig 4.1	Simulated results of Arch-O	23
Fig 4.2	Simulated results of Arch-L	25
Fig 4.3	Simulated results of Arch-A	27
Fig 4.4	Simulated results of Arch-B	29
Fig 4.5	Simulated results of Arch-C	33
Fig 4.6	Simulated results of Arch-D	35

LIST OF TABLES

TABLE NO	TITLE	PAGE NO
3.1	Proposed LUT approximation	18

LIST OF SYMBOLS AND ABBREVIATIONS

ACSO :	Add-Compare-Select-Offset
AWGN:	Additive White Gaussian Noise
BMU :	Branch Metric Unit.
ELUT :	Extended Look Up Table
GLUT :	Global Look Up Table
LLR :	Log Likelihood Ratio.
LUT :	Look-Up Table
MAP :	Maximum A Posterior Probability.
MUX :	Multiplexer
OACS :	Offset-Add-Compare-Select
RSC :	Recursive Systematic Convolutional.
SISO :	Soft-Input Soft-Output.
SOU :	Soft Output Unit.
α :	Forward metrics
β :	Backward metrics
γ :	Branch metrics
δ :	Difference metrics
L_{ex} :	Extrinsic information

CHAPTER 1

INTRODUCTION

1.1 OVERVIEW OF THE PROJECT

Error correction codes are an essential component in digital communication and data storage systems, wherein, Turbo code is one of the most optimal error correction codes. One key feature of Turbo code is the iterative process. However the iterative process leads to low throughput and long decoding latency. To overcome this, in this project the Turbo decoder is designed based on Maximum-A-Posterior probability (MAP) algorithm. This algorithm involves recursive computation of state metrics which leads to the design of high speed integrated circuit design.

In this project we will focus on Log-MAP based Turbo decoder design and will address high speed recursion architectures for Log-MAP decoders. Here several recursion architectures such as Radix-2, Radix-4, etc. are investigated. Algorithmic transformation, approximation, and architectural optimization are incorporated in the proposed designs to reduce the critical path. We then discuss the results of the experiments and highlight the shortcomings of the architecture and propose enhancements.

1.2 TURBO DECODER

A typical Turbo encoder consists of two recursive systematic convolutional (RSC) encoders and an interleaver between them. The source data are encoded by the first RSC encoder in sequential order while its interleaved sequence is encoded by the second RSC encoder. The original source bits and parity bits generated by two RSC encoders are sent out in a time-multiplexed way. Turbo codes work with large block size. In order to facilitate iterative decoding, the received data of a whole decoding block have to be stored in a memory.

A typical serial Turbo decoder architecture is shown in Figure 1.1. It has only one soft-input soft-output (SISO) decoder. Here probability MAP algorithm is employed for the SISO decoding.

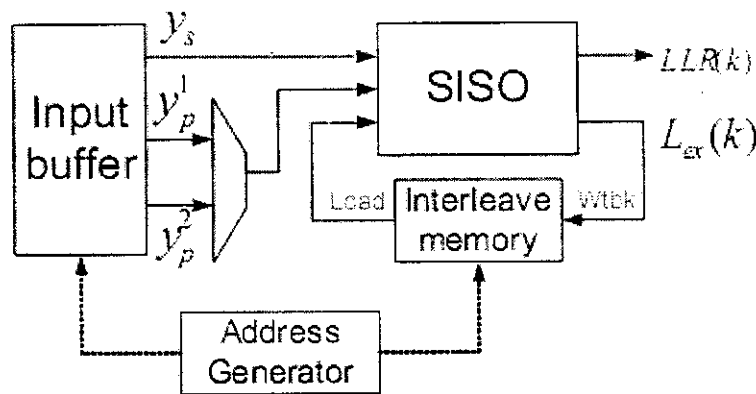


Figure 1.1: A serial Turbo decoder architecture

The serial Turbo decoder has two memories: one is for received soft symbols called the input buffer and the other is the interleaver memory to store extrinsic information. The SISO decoder takes soft inputs from the input buffer and the a priori information from the interleaver memory. It outputs the log likelihood ratio $LLR(k)$ and the extrinsic information $L_{ex}(k)$ for the k^{th} information bit in the decoding sequence. The extrinsic information is sent back as the new a priori information for next decoding. The data are loaded according to the current decoding sequence. The extrinsic information is loaded in sequential decoding phase while being loaded in the interleaved order at the interleaved decoding phase.

A high-level description of this traditional SISO algorithm is presented in Fig.1.2.

```
int SISO(){  
  
    /* Backward recursion of beta */  
    initialize_beta_at_end_block;  
    for (k1 = end_block; k1 > begin_block; k1--) {  
        beta[k1 - 1] = backward_recursion(beta[k1],input[k1]);  
    }  
  
    /* Forward recursion of alpha and intrinsic information */  
    initialize_alpha_at_begin_block;  
    for (k2 = begin_block; k2 < end_block; k2++) {  
        output[k2] = generate_out(alpha,beta[k2],input[k2]);  
        alpha = forward_recursion(alpha,input[k2]);  
    }  
    output[end_block] = f(alpha[end_block],beta[end_block],input[end_block]);  
}
```

Figure 1.2: Traditional SISO algorithm

1.3 SOFTWARE USED

1. Xilinx: It is a complete ECAD (electronic computer-aided design) application that provides platform for designing , testing and debugging of integrated circuits.
2. Model sim: ModelSim SE is a Windows-based simulation and debug environment that provides scalable HDL simulation solutions for a broad range of design sizes (both ASIC and FPGA) and complexities.

CHAPTER 2

LOG - MAP ALGORITHM

2.1 MAP ALGORITHM OVERVIEW

The MAP algorithm was first derived in the probability domain. The output of the algorithm is a sequence of decoded bits along with their reliabilities. This "soft" reliability information is generally described by the a posteriori probability (APP) $P(u|y)$. The MAP algorithm provides not only the estimated sequence, but also the probabilities for each bit that has been decoded correctly. Assuming binary codes are to be used, the MAP algorithm gives, for each decoded bit u_k in step k , the probability that this bit was $+1$ or -1 , given the received distorted symbol sequence $y_o^N = (y_0, y_1, y_2, \dots, y_N)$. This is equivalent to finding the likelihood ratio

$$\lambda_k = (P\{u_k = +1 | y_o^N\}) / (P\{u_k = -1 | y_o^N\}) \quad (1)$$

where $P\{u_k = i | y_o^N\}$, $i = +1, -1$ is the a posteriori probability (APP) of u_k .

Computation of $P\{u_k | y_o^N\}$, is done by determining the probability to reach a certain encoder state m after having received k symbols $y_o^{k-1} = (y_0, y_1, y_2, \dots, y_{k-1})$:

$$\alpha_k(m) = P\{m | y_o^{k-1}\} \quad (2)$$

and the probability to get from encoder state m' to the final state in step N with symbols y_{k+1}^N :

$$\beta_{k+1}(m') = P\{y_{k+1}^N | m'\} \quad (3)$$

The probability of the transition m to m' using the source symbol u_k , under knowledge of the received symbol y_k , is called γ_k :

$$\gamma_k(m, m', u_k) = P(m, m', u_k | y_k) \quad (4)$$

The probabilities $\alpha_k(m)$ and $\beta_{k+1}(m')$ are computed recursively over $\gamma_k(m, m', u_k)$ which are a function of the received symbols and the channel model as below:

$$\alpha_k(m') = \sum_{all\ m} \gamma_k(m, m', u_k) \alpha_{k-1}(m) \quad (5)$$

$$\beta_{k-1}(m) = \sum_{all\ m'} \beta_k(m') \gamma_k(m, m', u_k) \quad (6)$$

Knowing these values for each transition m to m' , the probability of having sent the symbol u_k in step k is the sum of all paths using the symbol u_k in step k . With $\phi(u_k)$ being the set of all transitions with symbol u_k , we can write

$$P\{u_k | y_0^N\} = \sum_{(m, m') \in \phi(u_k)} \gamma_k(m, m', u_k) \alpha_k(m) \beta_{k+1}(m') \quad (7)$$

Thus, from the above equations we can conclude that to evaluate the likelihood value of a decoded bit we require many additions and multiplications. The decoding complexity of the MAP algorithm has been reduced by operating it in the log domain. This technique was first used for the ISI channel. Taking the logarithm of $\alpha_k(m)$, $\beta_{k+1}(m')$, $\gamma_k(m, m', u_k)$ and λ_k values from the MAP algorithm we have:

$$A_k(m) = \ln \alpha_k(m) \quad (8)$$

$$B_{k+1}(m') = \ln \beta_{k+1}(m') \quad (9)$$

$$D_k(m, m', u_k) = \ln \gamma_k(m, m', u_k) \quad (10)$$

$$L_k(u_k) = \ln \lambda \quad (11)$$

Using the above equations, we rewrite (5) & (6) as follows:

Forward recursion : The recursion is initialized by forcing the starting state to state 0 and setting $A_0(0) = 0$ and $A_0(m) = -\infty$, $m \neq 0$.

$$A_k(m') = \ln \sum_{all\ m} \exp (D_k(m, m', u_k) + A_{k-1}(m)) \quad (12)$$

Backward recursion : The recursion is initialized by forcing the ending state to state 0 and setting $B_N(0) = 0$ and $B_N(m) = -\infty$, $m \neq 0$.

$$B_{k-1}(m) = \ln \sum_{all\ m'} \exp (B_k(m') + D_k (m, m', u_k)) \quad (13)$$

$$D_k(m, m', u_k) = \ln [((1/\sqrt{2\pi\sigma^2}) \exp [-(1/2\sigma)(y_m - y_m^{\wedge})^2]) P(u_k)] \quad (14)$$

where y_m , y_m^{\wedge} are the received signal and estimated signal over a Gaussian channel respectively, $P(u_k)$ is the a-priori probability of bit u_k and σ^2 is the noise variance. Soft-Output Calculation. The soft output, which is called the LLR, for each symbol at time k is calculated as

$$L_R(u_k) = \ln \left\{ \frac{ \sum_{m\ to\ m'\ uk = +1} (\exp [A_{k-1}(m) + B_k(m') + D_k (m, m', u_k)]) }{ \sum_{m\ to\ m'\ uk = -1} (\exp [A_{k-1}(m) + B_k(m') + D_k (m, m', u_k)]) } \right\} \quad (15)$$

The log-MAP decoder is divided into four major blocks. These are the branch metric calculator (BMC), the forward state metric calculator (FSMC), reverse state metric calculator (RSMC) and the log-likelihood ratio calculator (LLRC). To increase the speed of the decoder a parallel implementation is adopted in this project, that is all the SM's are calculated simultaneously. The block diagram of the architecture is shown in Fig.2.1.

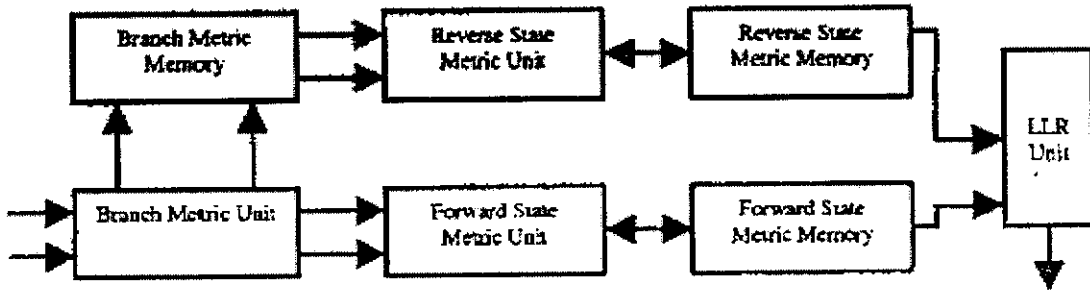


Figure 2.1 Block diagram of MAP decoder

The forward state metric unit uses the branch metrics obtained directly from the BMU to calculate the forward state metrics, and the backward state metric unit uses the reversed branch metrics from the branch metric storage to calculate the reverse state metrics.

The MAP algorithm, as described, requires the entire message to be stored before decoding can start. If the blocks of data are large, or the received stream continuous, this restriction can be too stringent; “on-the-fly” decoding using a sliding-window technique has to be used. Similar to the Viterbi algorithm, we can start the backward recursion from the “all-zero vector” B^0 (i.e., all the components of B^0 are equal to zero) with data $\{y_k\}$, k from n down to $n-L$. L iterations of the backward recursion allows us to reach a very good approximation of $g + B_{n-L}$ (where g is a positive additive factor). This additive coefficient does not affect the value of the LLR. In the following, we will consider that after L cycles of backward recursion, the resulting state metric vector is the correct one. This property can be used in a hardware realization to start the effective decoding of the bits before the end of the message. The parameter L is called the convergence length. For on-the-fly decoding of non-systematic convolutional codes, five to ten times the constraint length was found to lead only to marginal signal-to-noise ratio (SNR) losses. For turbo decoders, due to the iterative structure of the computation, an increased value of L might be required to avoid an error floor. In practice, the final value of L has to be determined via system simulation and analysis of the particular decoding system at hand.

2.2 LOG - MAP ALGORITHM OVERVIEW

To facilitate iterative decoding, Turbo decoders require soft-input soft-output decoding algorithms, among which the probability MAP algorithm is widely adopted. The MAP algorithm is commonly implemented in log domain, thus called Log-MAP. The Log-MAP algorithm involves recursive computation of forward state metrics (α metrics) and backward state metrics (β metrics). The log-likelihood-ratio is computed based on the two types of state metrics and associated branch metrics (γ metrics). Due to different recursion directions in computing α and β metrics, a straightforward implementation of Log-MAP algorithm will not only consume large memory but also introduce large decoding latency. The sliding window approach is proposed in which pre-backward recursion operation is introduced. Here the pre-backward recursion unit is denoted as β_0 and real backward recursion unit as β_1 .

In this project we have improved the processing speed by making the following approximations in the Log-MAP algorithm.

$$\max^*(\max^*(A,B), \max^*(C,D)) = \max^*(\max(A,B), \max(C,D)) \quad (1)$$

where

$$\max^*(A, B) = \max(A, B) + \log(1 + e^{-|A-B|}) \quad (2)$$

By applying this approximation the processing speed is improved approximately 40% and however, the hardware will be nearly doubled compared to the traditional ACSO architecture.



P-3450

CHAPTER 3

HIGH SPEED RECURSION ARCHITECTURES

3.1 INTRODUCTION

In MAP-based Turbo decoder structure the branch metrics unit (BMU) takes inputs from the receiver buffer and the interleaver memory. The outputs of BMU are directly sent to the pre-backward recursion unit (β_0 unit). The previously stored branch metrics for consecutive sliding windows are input to the forward recursion unit (α unit) and the effective backward recursion unit (β_1 unit), respectively. The soft output unit (SOU) that is used to compute the log likelihood ratio (LLR) and the extrinsic information (L_{ex}) takes inputs from the previously stored α metrics, the currently computed β metrics and the previously stored branch metrics (γ). The SOU starts to generate soft outputs after the branch metrics have been computed for the first two sliding windows. It can be observed that the high-speed bottleneck of a Log-MAP decoder lies in the three recursive computation units since both BMU and SOU can be simple pipelined for high speed applications.

3.2 RECURSIVE ARCHITECTURES OVERVIEW

It is known from Log-MAP algorithm that all the three recursion units have similar architectures. So we will design one of the three units (say α unit).

3.2.1 Traditional Architecture Arch-O:

The traditional design for computation is illustrated in Figure 3.1, where the ABS block is used to compute the absolute value of the input and the LUT block is used to implement a nonlinear function $\log(1 + e^{-x})$, where $x > 0$. For simplicity, only one branch (i.e., one state) is drawn. The over flow approach is assumed for normalization of state metrics as used in conventional Viterbi decoders.

It can be seen that the computation of the recursive loop consists of three multibit additions, the computation of absolute value and a random logic to implement the LUT.

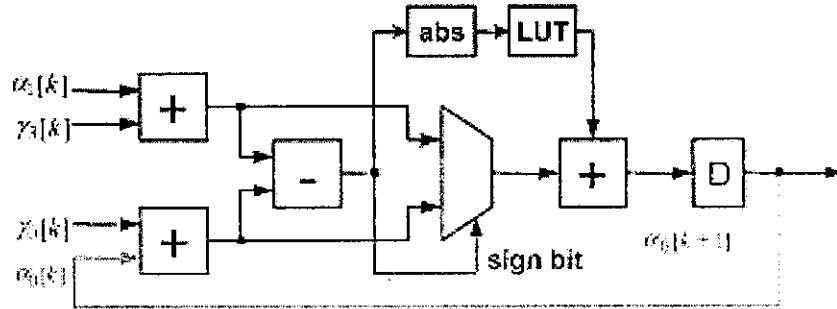


Figure 3.1 Traditional architecture Arch-O

As there is only one delay element in each recursive loop, the traditional retiming technique cannot be used to reduce the critical path.

3.2.2 Radix-4 architecture Arch-L:

In this architecture the Lucent Bell Labs proposed the following approximation for computation of $\alpha_0[k+2]$ is implemented

$$\alpha_0[k+2] \approx \max^*(\max(\alpha_0[k] + \gamma_0[k], \alpha_1[k] + \gamma_3[k]) + \gamma_0[k+1], \max(\alpha_2[k] + \gamma_2[k], \alpha_3[k] + \gamma_1[k]) + \gamma_3[k+1]). \quad (3)$$

The architecture to implement this approximation is shown in figure 3.2. However this approximation will have a 0.04-db performance loss compared to the original Log-MAP algorithm. Here the critical path consists of four multibit adder delays, one generalised LUT delay, and one 2:1 MUX delay. The LUT1 block includes absolute value computation and a normal LUT operation, the MAX block includes one subtractor and one 2:1 MAX.

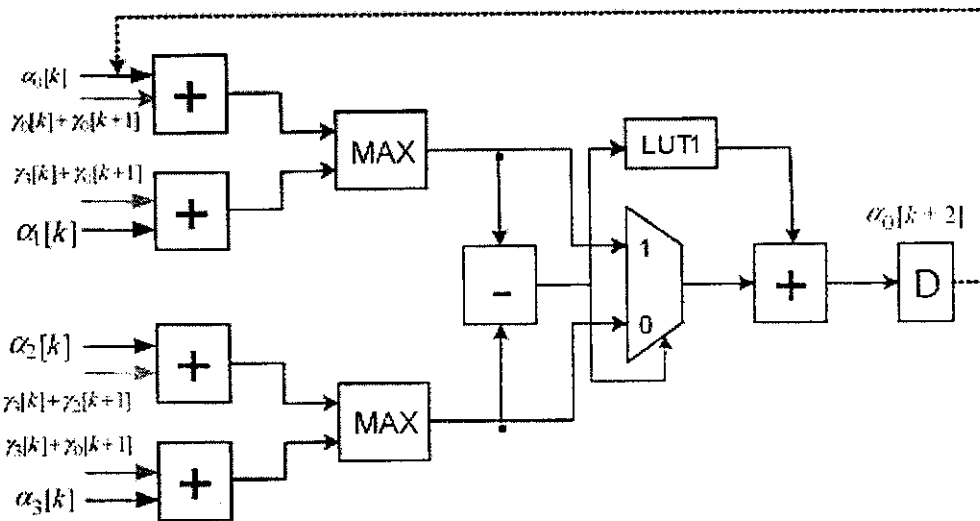


Figure 3.2 Radix-4 architecture Arch - L

3.2.3 Advanced Radix-2 architecture Arch - A:

Here we introduce a difference metric for each competing pair of states metrics so that we can perform the front-end addition and the subtraction operations simultaneously in order to reduce the computation delay of the loop. Second, we employ a generalized LUT (see GLUT in Fig. 3.3) that can efficiently avoid the computation of absolute value instead of introducing another subtraction operation. Third, we move the final addition to the input side as with the OACS architecture and then utilize one stage carry-save structure to convert a three-number addition to a two-number addition. Finally, we make an intelligent approximation in order to further reduce the critical path.

The following equations are assumed for the considered recursive computation shown in Fig. 3.3

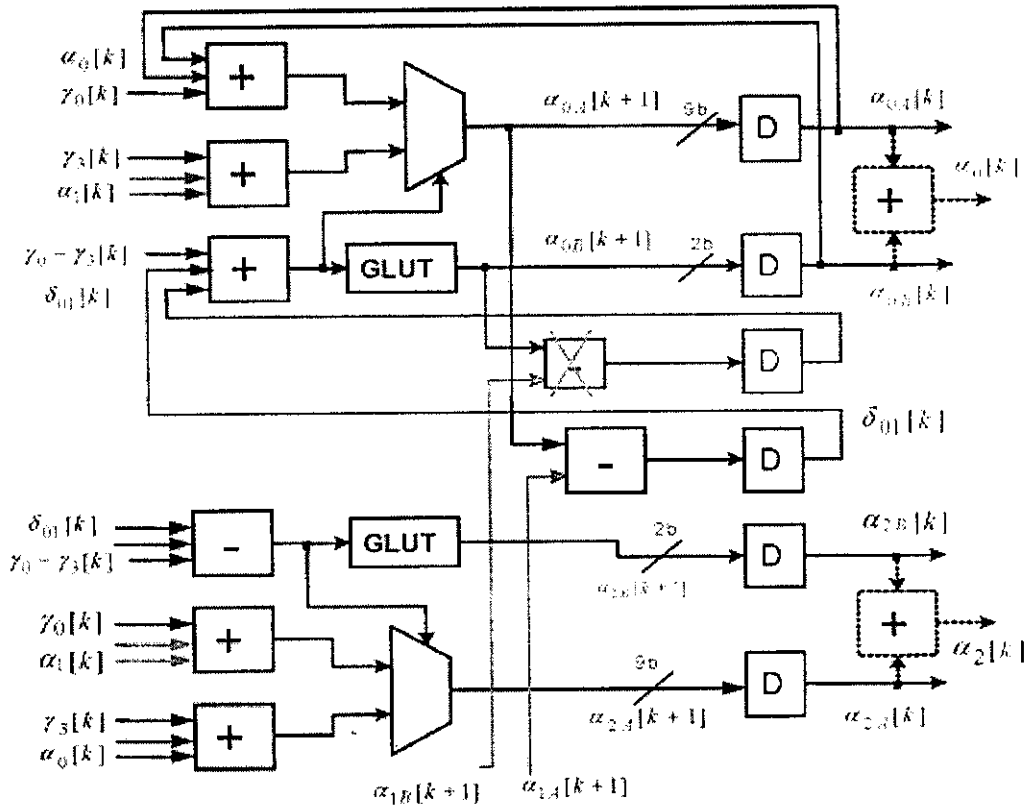


Figure 3.3 Advanced Radix-2 architecture Arch-A

$$\alpha_0[k+1] = \max^* (\alpha_0[k] + \gamma_0[k], \alpha_1[k] + \gamma_3[k])$$

$$\alpha_2[k+1] = \max^* (\alpha_0[k] + \gamma_3[k], \alpha_1[k] + \gamma_0[k]) \quad (4)$$

Where \max^* function is defined in (2).

In addition, we split each state metric into two terms as follows

$$\alpha_0[k] = \alpha_{0A}[k] + \alpha_{0B}[k]$$

$$\alpha_1[k] = \alpha_{1A}[k] + \alpha_{1B}[k]$$

$$\alpha_2[k] = \alpha_{2A}[k] + \alpha_{2B}[k] \quad (5)$$

Similarly, the corresponding difference metric is also split into the following two terms:

$$\begin{aligned}
\delta_{01}[k] &= \delta_{01A}[k] + \delta_{01B}[k] \\
\delta_{01A}[k] &= \alpha_{0A}[k] - \alpha_{1A}[k] \\
\delta_{01B}[k] &= \alpha_{0B}[k] - \alpha_{1B}[k]
\end{aligned} \tag{6}$$

In this way, the original add-and-compare operation is converted as an addition of three numbers, i.e.,

$$(\alpha_0 + \gamma_0) - (\alpha_1 + \gamma_3) = (\gamma_0 - \gamma_3) + \delta_{01A} + \delta_{01B} \tag{7}$$

Where $(\gamma_0 - \gamma_3)$ is computed by branch metric unit (BMU). In addition, the difference between the two outputs from two GLUTs, i.e., δ_{01B} in the figure 3.3 can be neglected. Since this small approximation does not cause any performance loss in Turbo decoding with either AWGN channels or Raleigh fading channels. If one competing path metrics (e.g., $p_0 = \alpha_0 + \gamma_0$) is significantly larger than the other one (e.g., $p_1 = \alpha_1 + \gamma_3$), the GLUT output will not change the decision anyway due to their small magnitudes. On the other hand, if the two competing path metrics are so close that adding or removing a small value output from one GLUT may change the decision (e.g., from $p_0 > p_1$ to $p_1 > p_0$), picking any one should not make big difference.

Upper Bounds for MAX^* : All the following upper bounds are derived from the definition of MAX^*

$$\text{MAX}^*(x,y) \geq \text{MAX}(x,y) \tag{8}$$

For practical implementation, one can notice that, due to the finite precision of the hardware implementation, the function $\ln(1 + e^{-|x-y|})$ gives a zero result as soon as is large enough. For example, if the values are coded in fixed precision with three binary places (a quantum of 0.125), then, $|x-y| > 2.5$ which gives $\ln(1 + e^{-|x-y|}) < 0.079$, thus it will be rounded to 0. In that case, the computation of the offset of the MAX^* operator can be performed with two pieces of information: a Boolean (for zero) that indicates if $|x-y|$ is above or equal to the first power of two greater than 2.5, i.e., four. If z is true, then the offset is equal to 0. If not, its exact value is computed with the five least significant bits of $|x-y|$. The maximum number is $\ln(2)$, which will

be quantized to 0.75, i.e., the width of the LUT is three bits for our example. An LUT is the most straight-forward way to perform this operation. In the general case, there is a positive value δ such that

$$\text{MAX}^*(x,y) = \text{MAX}(x,y) \quad \text{if } |x-y| > 2^\delta \quad (9)$$

We also have

$$\text{MAX}^*(x,y) \leq \text{MAX}(x,y) + \ln(2) \quad \text{if } x=y. \quad (10)$$

The correction function and its approximation is shown in the following figure 3.4

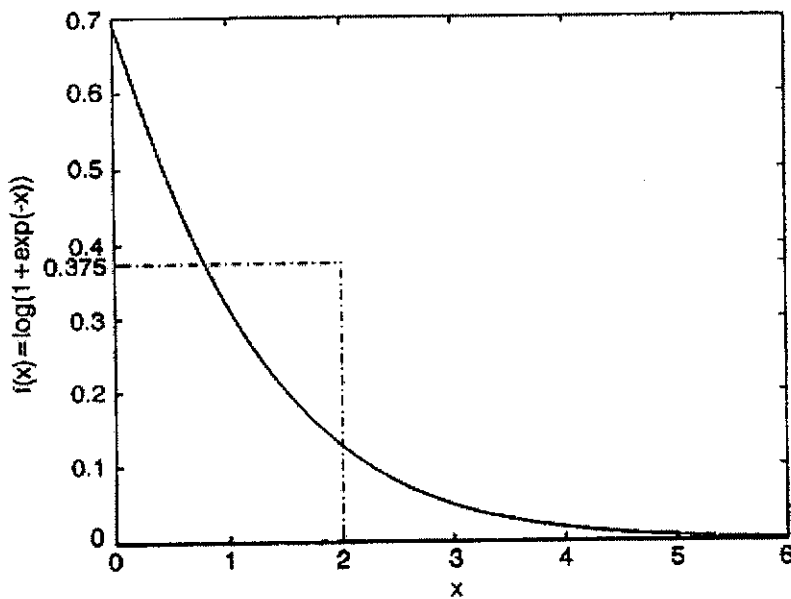


Figure 3.4 Correction function and its approximation

At the input side, a small circuitry shown in Fig. 3.4 is employed to convert an addition of three numbers to an addition of two numbers, where FA and HA represents full-adder and half-adder, respectively, XOR stands for exclusive OR gate, d0 and d1 correspond to the 2-bit output of GLUT. The state metrics and branch metrics are represented with 9 and 6 bits, respectively, in this example. The sign extension is only applied to the branch metrics. It should be noted that an extra addition operation (see dashed adder boxes) might be required to integrate each state metric before storing it into the memory.

The GLUT structure is shown in Fig. 3.5, where the computation of absolute value is eliminated by including the sign bit into two logic blocks, i.e., Ls2 and ELUT, where the Ls2 function block is used to detect if the absolute value of the input is less than 2.0, and the ELUT block is a small LUT with 3-bit inputs and 2-bit outputs.

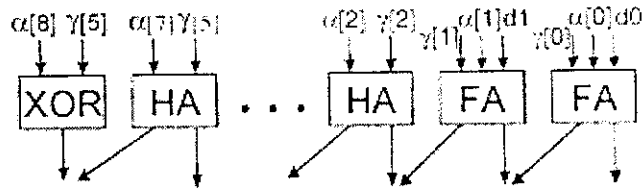


Figure 3.5 Carry- save structure

It can be derived that $Z = S' (b_7, \dots, +b_1 + b)' + S (b_7, \dots, b_1 b_0)$.

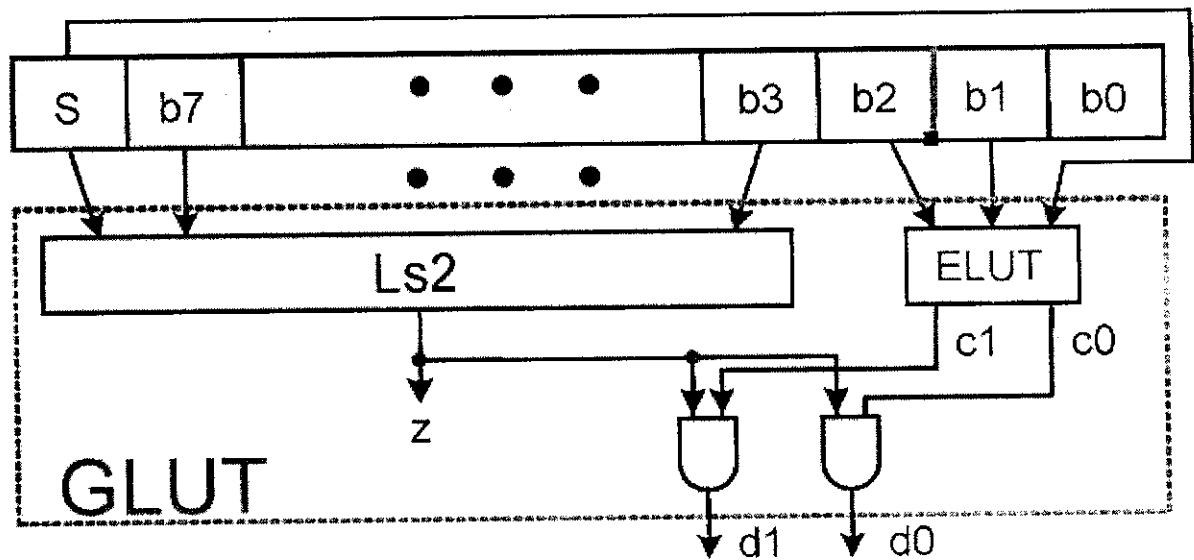


Figure 3.6 Structure of GLUT

The LUT approximation is described as follows

$$\text{If } |x| < 2; \quad f(x) = 3/8; \quad \text{else } f(x) = 0 \quad (11)$$

where x and $f(x)$ stand for the input and the output of the LUT, respectively. In this approach, we only need to check if the absolute value of the input is less than 2, which can be performed by the Ls2 block in Fig. 3.5. A drawback of this method is that its performance would be significantly degraded if only two bits are kept for the fractional part of the state metrics.

In our design, both the inputs and outputs of the LUT are quantized in four levels. The details are shown in Table 1. The inputs to ELUT are treated as a 3-bit signed binary number. The outputs of ELUT are ANDed with the output of Ls2 block. This means, if the absolute value of the input is greater than 2.0, the output from the GLUT is 0. Otherwise, the output from ELUT will be the final output. The ELUT can be implemented with combinational logic for high-speed applications. The computation latency is smaller than the latency of Ls2 block.

$ x $	0.0	0.50	1.0	1.50
$f(x)$	$3/4$	$2/4$	$1/4$	$1/4$

Table 3.1. Proposed LUT approximation

Therefore, the overall latency of the GLUT is almost the same as the previously discussed simplified method whose total delay consists of one 2:1 multiplexer gate delay and the computation delay of logic block Ls2.

After all the previous optimization, the critical path of the recursive architecture is reduced to two multibit additions, one 2:1 MUX operation, and 1-bit addition operation, which saves nearly two multibit adder delay compared to the traditional ACSO architecture.

3.2.4 Improved Radix-4 architecture Arch-B:

Here we will discuss the improved Radix-4 architecture. The computation of $\alpha_0[k+2]$ is given as follows

$$\begin{aligned}
 \alpha_0[k+2] &= \max * (\max(\alpha_0[k+1] + \gamma_0[k+1], \alpha_1[k+1] + \gamma_3[k+1])) \\
 &= \max * (\max * (\alpha_0[k] + \gamma_0[k], \alpha_1[k] + \gamma_3[k]) + \gamma_0[k+1]), \\
 &\quad \max * (\alpha_2[k] + \gamma_2[k], \alpha_3[k] + \gamma_1[k]) + \gamma_3[k+1]).
 \end{aligned} \tag{12}$$

where $\alpha_1[k+1] = \max * (\max(\alpha_2[k] + \gamma_2[k], \alpha_3[k] + \gamma_1[k]))$

The approximation is applied to the above equation as follows

$$\alpha_0[k+2] \approx \max (\max * (\alpha_0[k] + \gamma_0[k], \alpha_1[k] + \gamma_3[k]) + \gamma_0[k + 1], \max * (\alpha_2[k] + \gamma_2[k], \alpha_3[k] + \gamma_1[k]) + \gamma_3[k + 1]) \quad (13)$$

Turbo decoder employing this new approximation should have the same decoding performance as using (9). While directly implementing (10) does not bring any advantage to the critical path, we intend to take advantages of the techniques that we used in developing Arch-A. The improved Radix-4 architecture Arch-B is shown in Fig. 3.6.

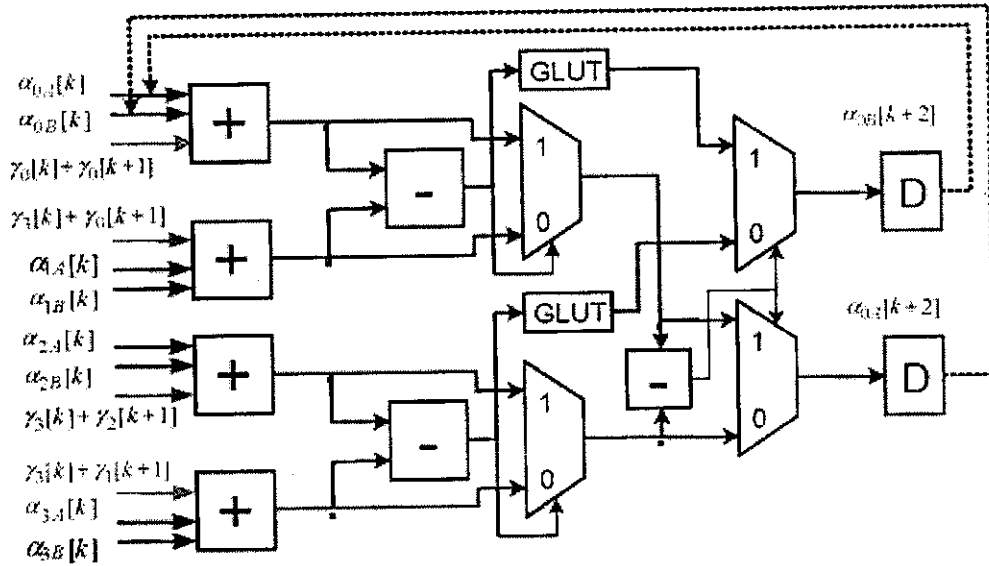


Figure 3.7 Improved Radix-4 architecture Arch-B

Here, we split each state metric into two terms and we adopt the same GLUT structure as we did before. In addition, a similar approximation is incorporated as with Arch-A. In this case, the outputs from GLUT are not involved in the final stage comparison operation. It can be observed that the critical path of the new architecture is close to a three multibit adder delay. To compensate for all the approximation introduced, the extrinsic information generated by the MAP decoder based on this new Radix-4 architecture should be scaled by a factor around 0.75.

3.2.5 Radix-8 architecture Arch-C:

Here the algorithm approximation of improved Radix-4 architecture ie., Arch –B is followed to form Radix-8 architecture Arch – C and the computation of $\alpha_0[k+3]$ is given follows.

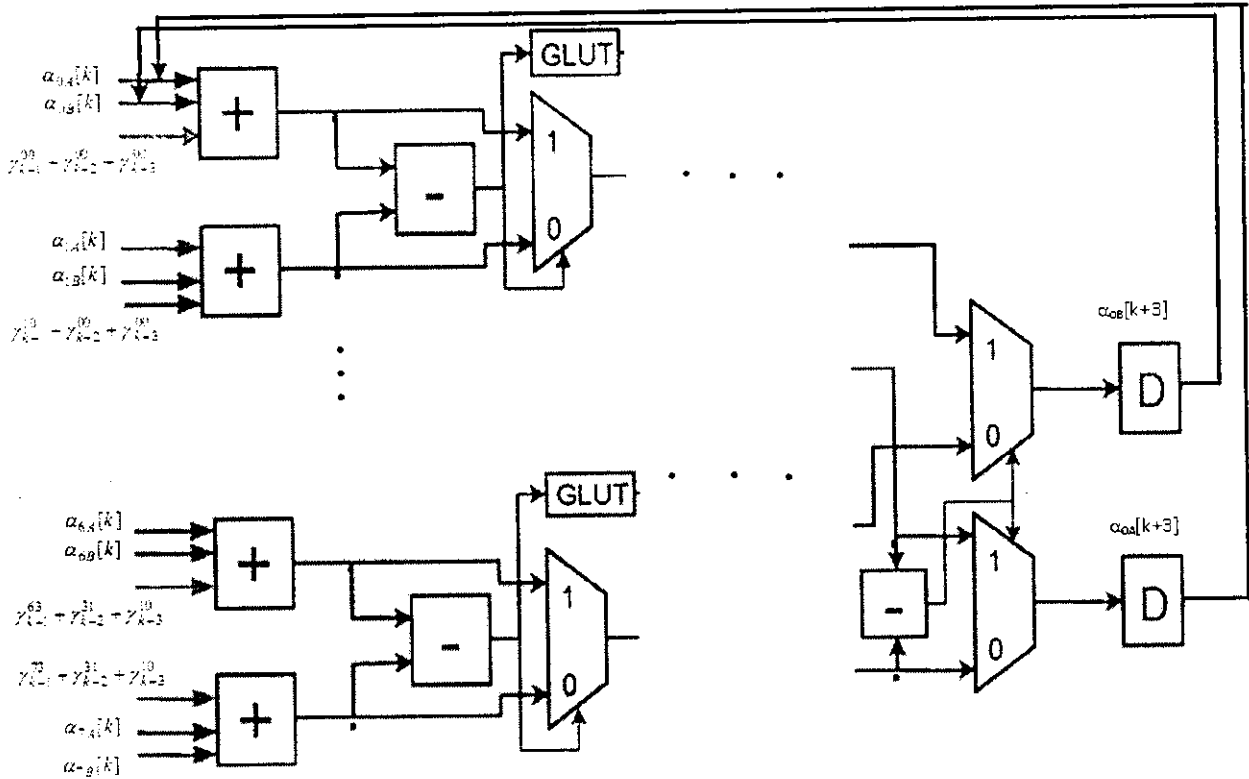


Figure 3.8 Radix-8 architecture Arch-C

$$\alpha_0[k+3] = \max \{ \max [\max^* (\alpha_0[k] + \gamma_{00}[k+1] + \gamma_{00}[k+2] + \gamma_{00}[k+3], \alpha_1[k] + \gamma_{10}[k+1] + \gamma_{00}[k+2] + \gamma_{00}[k+3]), \max^* (\alpha_2[k] + \gamma_{21}[k+1] + \gamma_{10}[k+2] + \gamma_{00}[k+3], \alpha_3[k] + \gamma_{31}[k+1] + \gamma_{10}[k+2] + \gamma_{00}[k+3])], \max [\max^* (\alpha_4[k] + \gamma_{42}[k+1] + \gamma_{21}[k+2] + \gamma_{10}[k+3], \alpha_5[k] + \gamma_{52}[k+1] + \gamma_{21}[k+2] + \gamma_{10}[k+3]), \max^* (\alpha_6[k] + \gamma_{63}[k+1] + \gamma_{31}[k+2] + \gamma_{10}[k+3], \alpha_7[k] + \gamma_{73}[k+1] + \gamma_{31}[k+2] + \gamma_{10}[k+3])] \} \quad (15)$$

3.2.5 Radix-16 architecture Arch-D:

Here the algorithm approximation of improved Radix-4 architecture ie., Arch-B is followed to form Radix-16 architecture Arch-D and is shown in figure 3.9

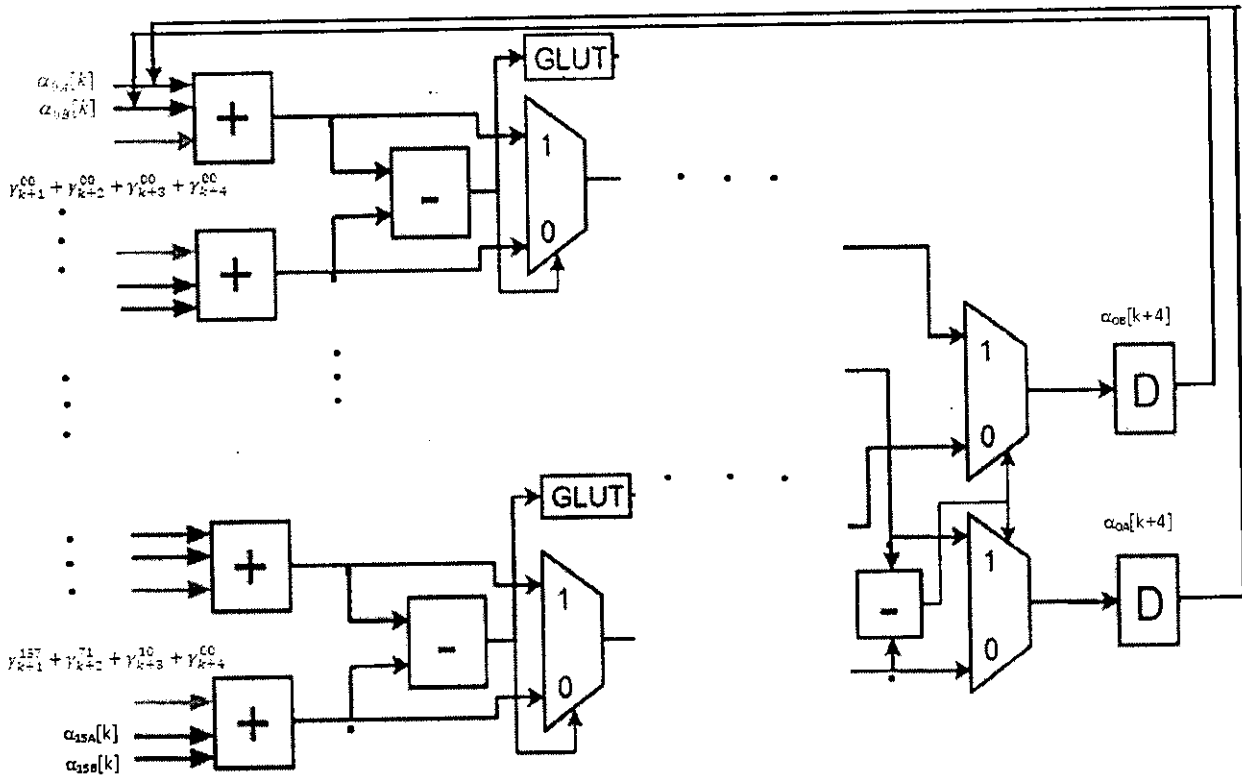


Figure 3.9 Radix-16 architecture Arch-D

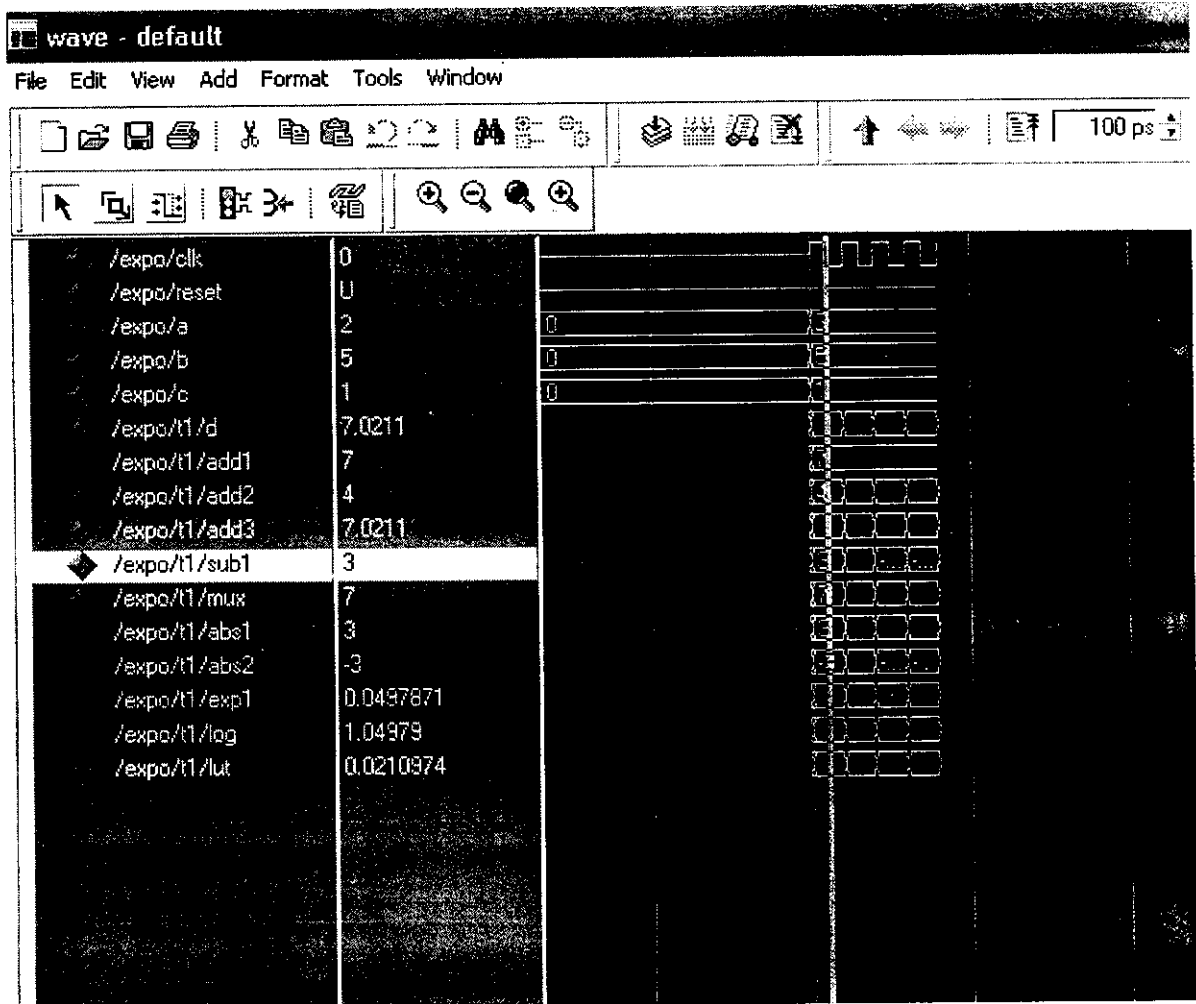
The equation for calculating the $\alpha_0[k+4]$ is similar to that of Arch-C except that we have four branch metric with respective delay components handled at a time.

CHAPTER 4

SIMULATION RESULTS

4.1 SIMULATED RESULTS OF ARCH-O

The simulated results of Arch-O are shown in the figure 4.1



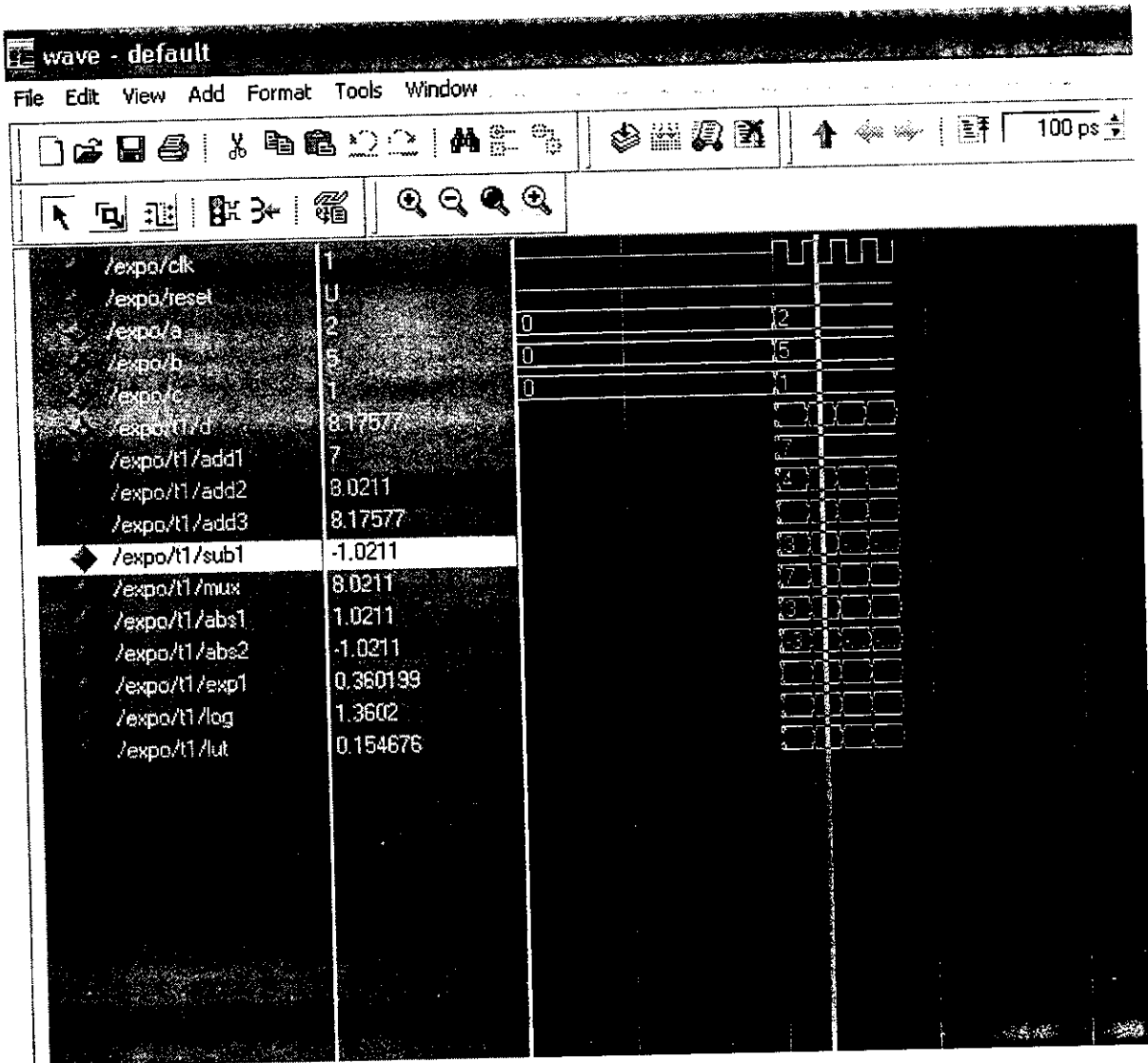
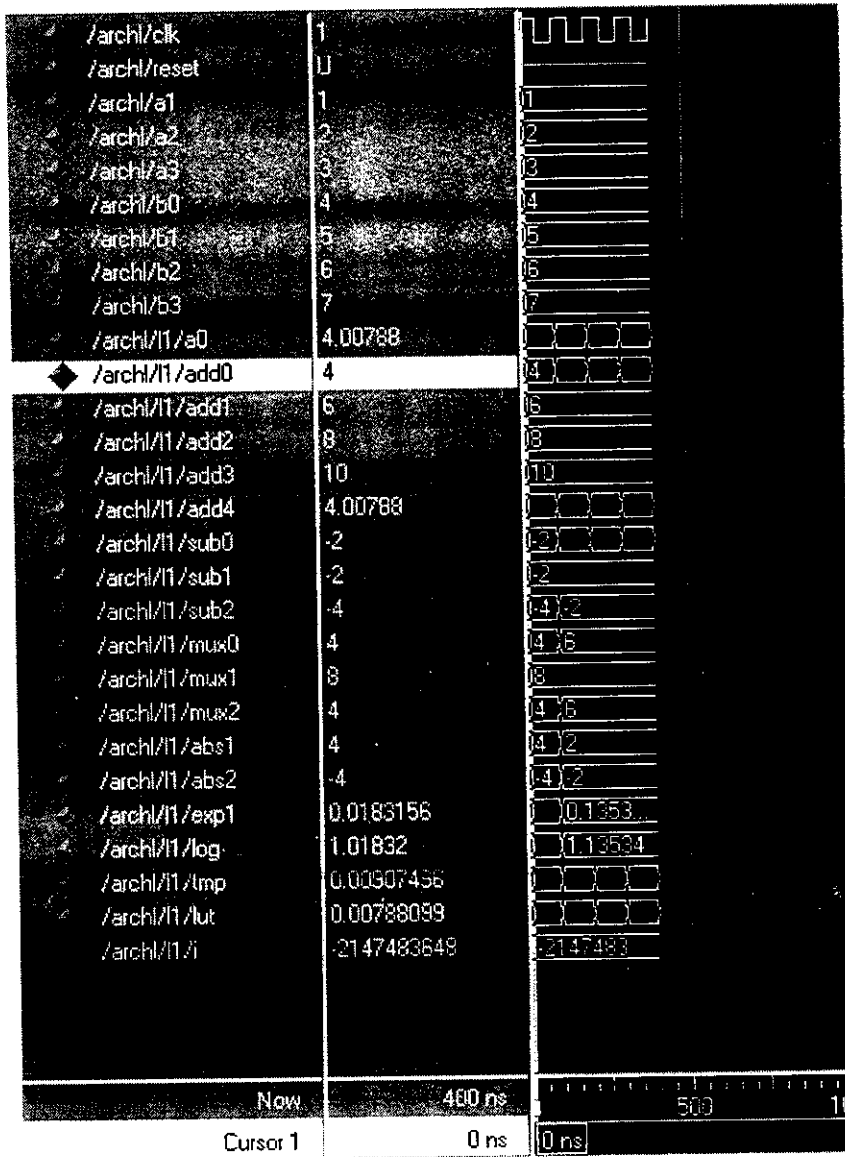


Figure 4.1 Simulated results of Arch-O

4.2 SIMULATED RESULTS OF ARCH-L

The simulated results of Arch-L are shown in the figure 4.2



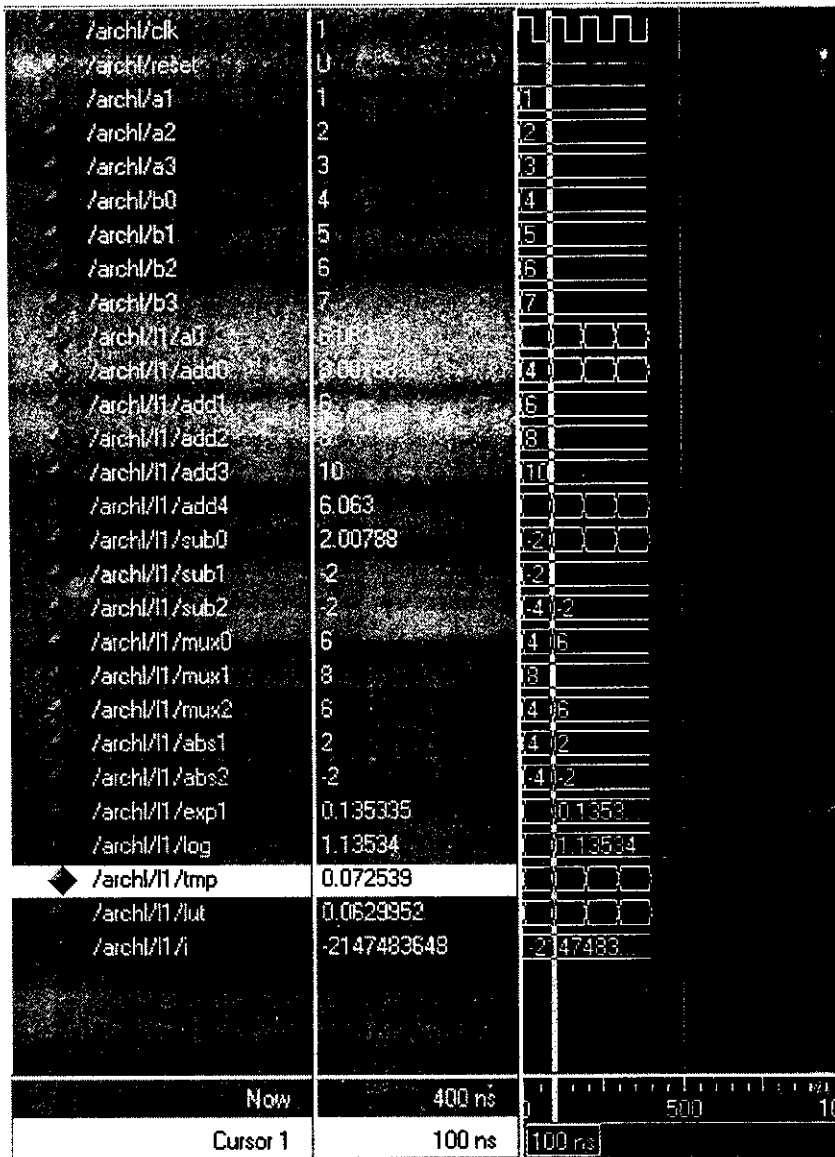
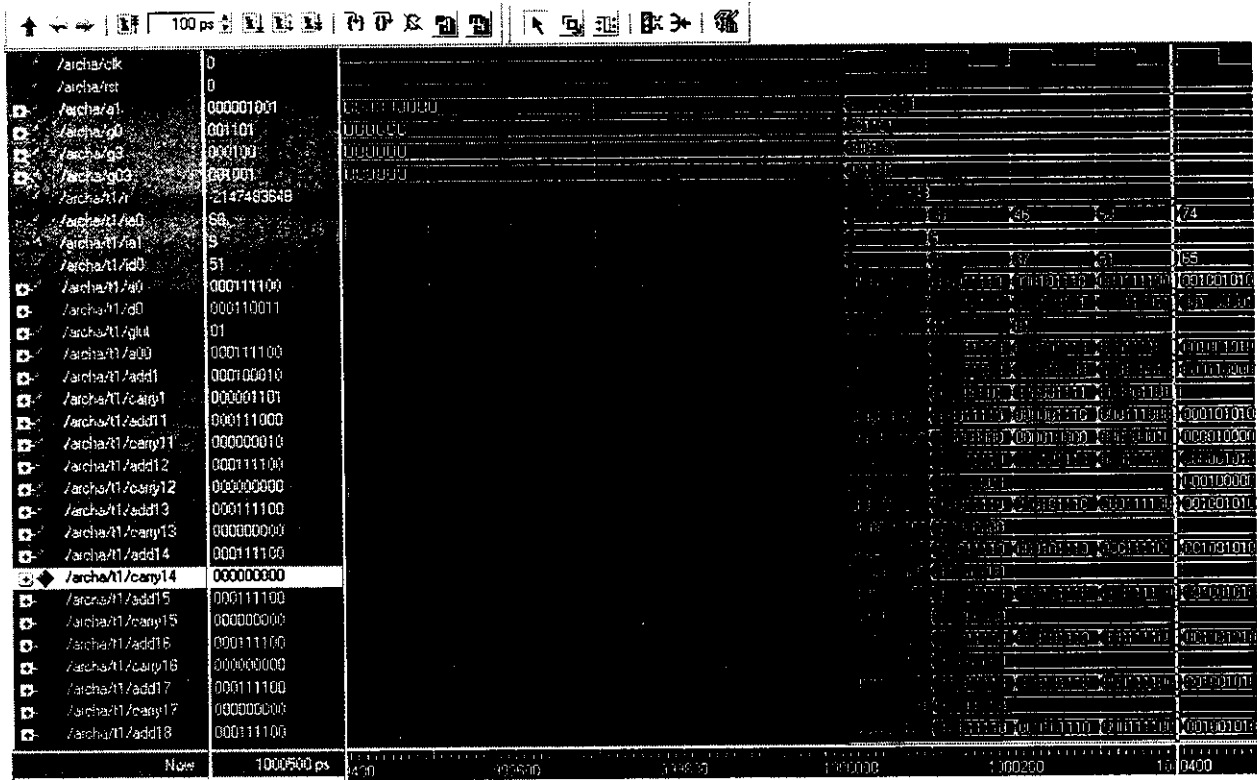


Figure 4.2 Simulated results of Arch-L

4.3 SIMULATED RESULTS OF ARCH-A

The simulated results of Arch-A are shown in the figure 4.3



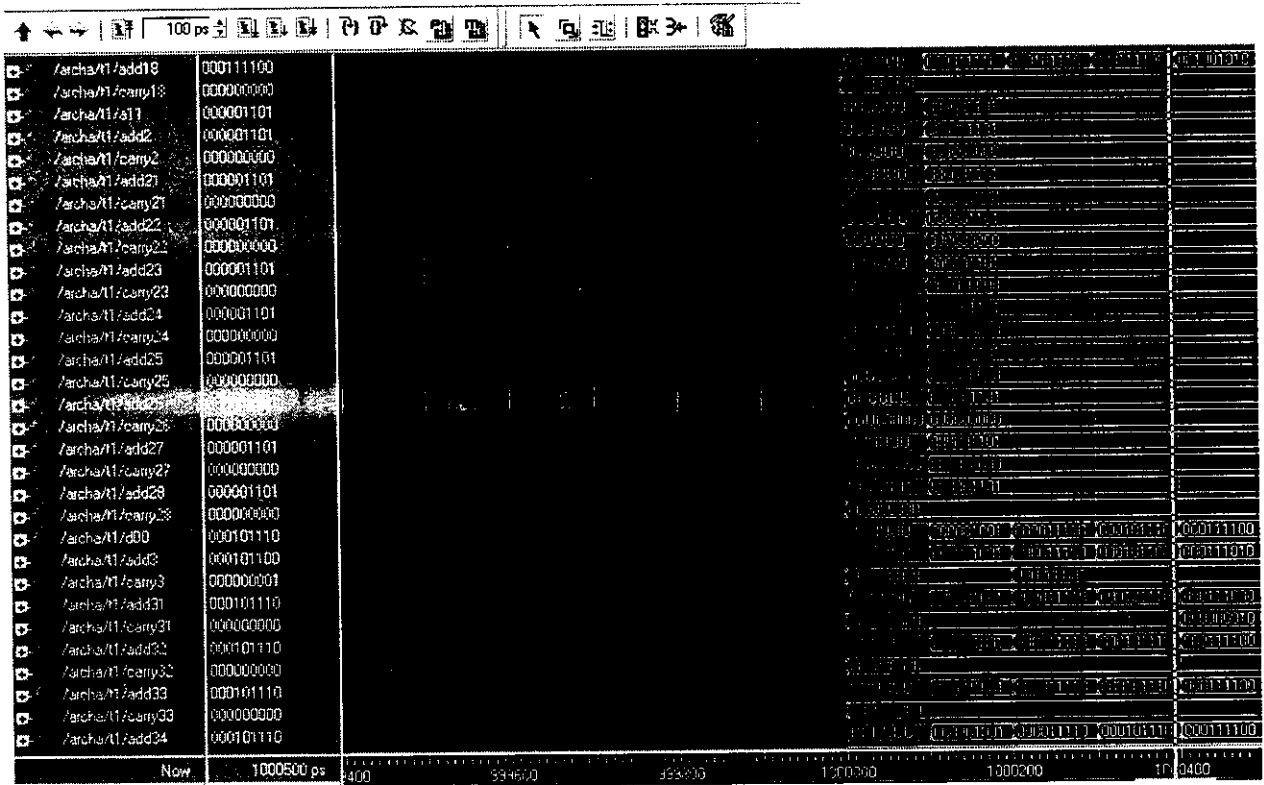
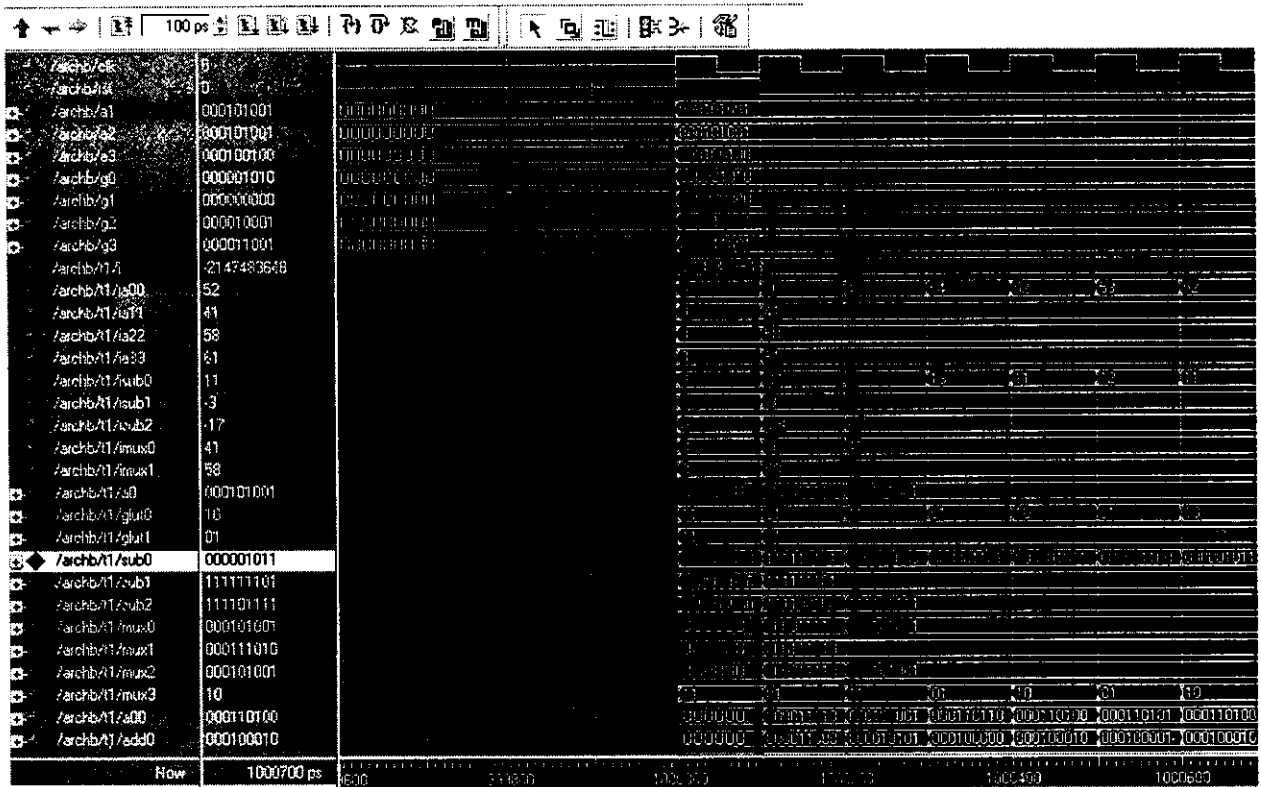


Figure 4.3 Simulated results of Arch-A

4.4 SIMULATED RESULTS OF ARCH-B

The simulated results of Arch-B are shown in the figure 4.4



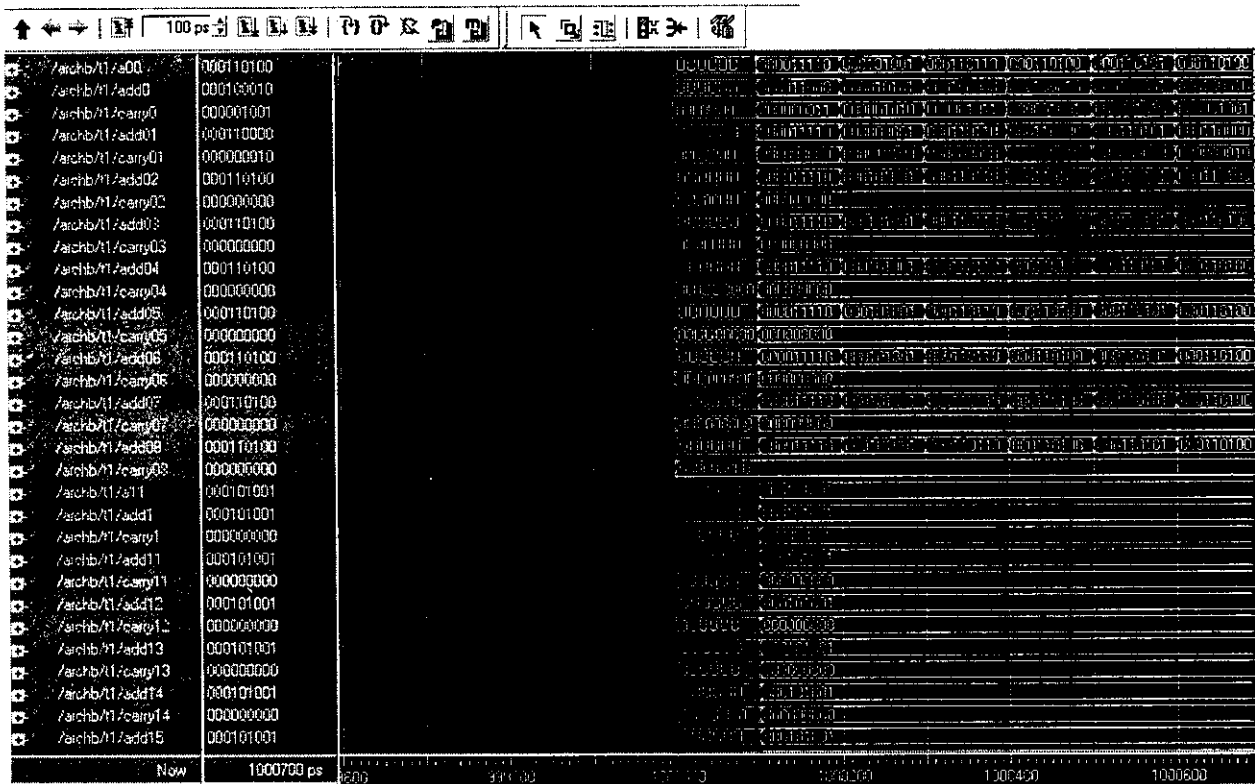
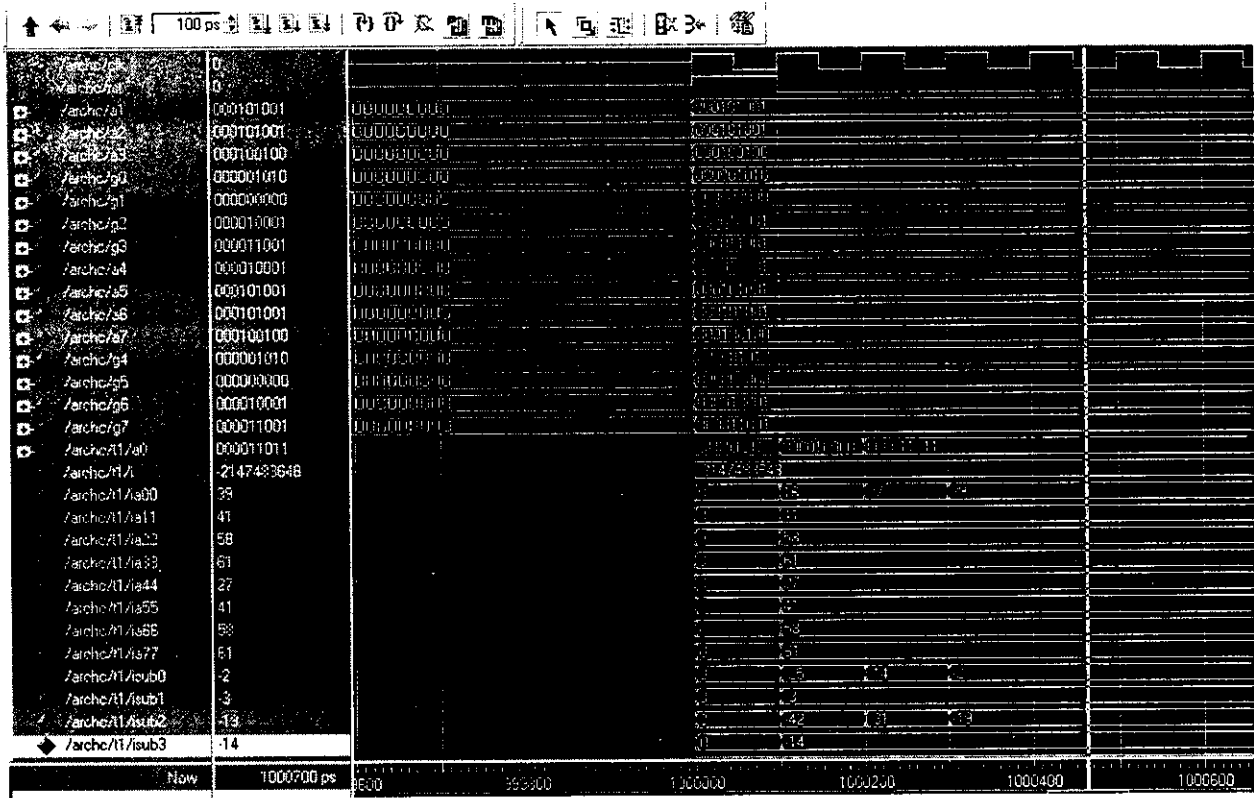


Figure 4.4 Simulated results of Arch-B

4.5 SIMULATED RESULTS OF ARCH-C

The simulated results of Arch-C are shown in the figure 4.5



Path	Value	Hex	Dec	Oct	Bin
/archc/1/sub3	14				
/archc/1/sub4	3				
/archc/1/sub5	31				
/archc/1/sub6	12				
/archc/1/mux0	39				
/archc/1/mux1	58				
/archc/1/mux4	27				
/archc/1/mux5	58				
/archc/1/mux2	39				
/archc/1/mux6	27				
/archc/1/glut0	01				
/archc/1/glut1	01				
/archc/1/glut2	10				
/archc/1/glut3	01				
/archc/1/sub0	111111110				
/archc/1/sub1	111111101				
/archc/1/sub2	111101101				
/archc/1/mux0	000100111				
/archc/1/mux1	000110110				
/archc/1/mux2	000100111				
/archc/1/mux3	01				
/archc/1/sub3	111110010				
/archc/1/sub4	111111101				
/archc/1/sub5	111100001				
/archc/1/mux4	000011011				
/archc/1/mux5	000111010				
/archc/1/mux6	000011011				
/archc/1/mux7	10				
/archc/1/sub6	000001100				
/archc/1/mux8	000011011				
/archc/1/mux9	10				

Now 1000700 ps

Path	Value	Hex	Dec	Oct	Bin
/archc/1/mux9	10				
/archc/1/add0	000100111				
/archc/1/add0	000010011				
/archc/1/cary0	000001010				
/archc/1/add01	000000111				
/archc/1/cary01	000010000				
/archc/1/add02	000100111				
/archc/1/cary02	000000000				
/archc/1/add03	000100111				
/archc/1/cary03	000000000				
/archc/1/add04	000100111				
/archc/1/cary04	000000000				
/archc/1/add05	000100111				
/archc/1/cary05	000000000				
/archc/1/add06	000100111				
/archc/1/cary06	000000000				
/archc/1/add07	000100111				
/archc/1/cary07	000000000				
/archc/1/add08	000100111				
/archc/1/cary08	000000000				
/archc/1/a11	000101001				
/archc/1/add1	000101001				
/archc/1/cary1	000000000				
/archc/1/add11	000101001				
/archc/1/cary11	000000000				
/archc/1/add12	000101001				
/archc/1/cary12	000000000				
/archc/1/add13	000101001				
/archc/1/cary13	000000000				
/archc/1/add14	000101001				
/archc/1/cary14	000000000				

Now 1000700 ps

Address	Hex	ASCII	Comment
/archc/1/cary14	00000000		
/archc/1/add15	000101001		
/archc/1/cary15	000000000		
/archc/1/add16	000101001		
/archc/1/cary16	000000000		
/archc/1/add17	000101001		
/archc/1/cary17	000000000		
/archc/1/add18	000101001		
/archc/1/cary18	000000000		
/archc/1/a22	000111010		
/archc/1/add2	000111000		
/archc/1/cary2	000000001		
/archc/1/add21	000111010		
/archc/1/cary21	000000000		
/archc/1/add22	000111010		
/archc/1/cary22	000000000		
/archc/1/add23	000111010		
/archc/1/cary23	000000000		
/archc/1/add24	000111010		
/archc/1/cary24	000000000		
/archc/1/add25	000111010		
/archc/1/cary25	000000000		
/archc/1/add26	000111010		
/archc/1/cary26	000000000		
/archc/1/add27	000111010		
/archc/1/cary27	000000000		
/archc/1/add28	000111010		
/archc/1/cary28	000000000		
/archc/1/a33	000111101		
/archc/1/add3	000111101		
/archc/1/cary3	000000000		

Now 1000700 ps

Address	Hex	ASCII	Comment
/archc/1/cary3	000000000		
/archc/1/add31	000111101		
/archc/1/cary31	000000000		
/archc/1/add32	000111101		
/archc/1/cary32	000000000		
/archc/1/add33	000111101		
/archc/1/cary33	000000000		
/archc/1/add34	000111101		
/archc/1/cary34	000000000		
/archc/1/add35	000111101		
/archc/1/cary35	000000000		
/archc/1/add36	000111101		
/archc/1/cary36	000000000		
/archc/1/add37	000111101		
/archc/1/cary37	000000000		
/archc/1/add38	000111101		
/archc/1/cary38	000000000		
/archc/1/a44	000011011		
/archc/1/add4	000011011		
/archc/1/cary4	000000000		
/archc/1/add41	000011011		
/archc/1/cary41	000000000		
/archc/1/add42	000011011		
/archc/1/cary42	000000000		
/archc/1/add43	000011011		
/archc/1/cary43	000000000		
/archc/1/add44	000011011		
/archc/1/cary44	000000000		
/archc/1/add45	000011011		
/archc/1/cary45	000000000		
/archc/1/add46	000011011		

Now 1000700 ps

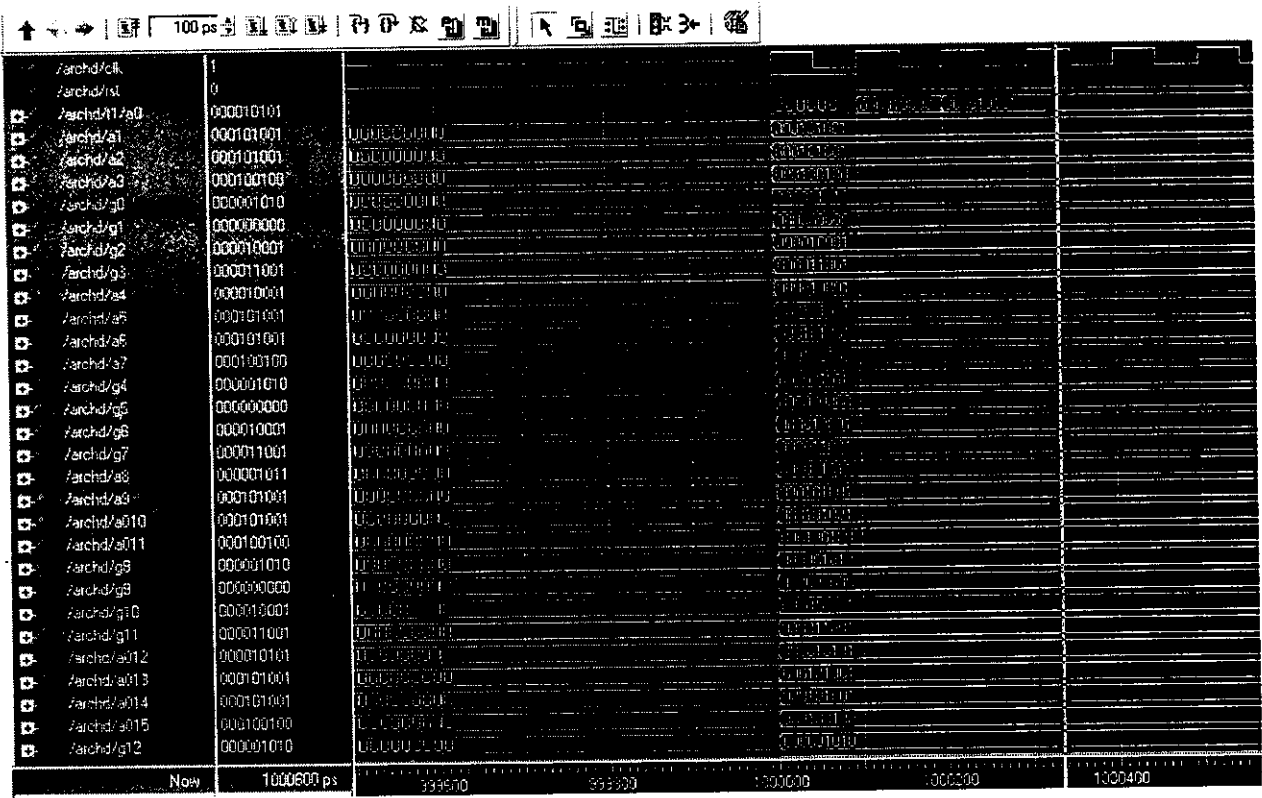
Address	Value
/archc/1/add46	000011011				
/archc/1/carry46	000000000				
/archc/1/add47	000011011				
/archc/1/carry47	000000000				
/archc/1/add48	000011011				
/archc/1/carry48	000000000				
/archc/1/add49	000000000				
/archc/1/add50	000101001				
/archc/1/add51	000101001				
/archc/1/carry51	000000000				
/archc/1/add52	000101001				
/archc/1/carry52	000000000				
/archc/1/add53	000101001				
/archc/1/carry53	000000000				
/archc/1/add54	000101001				
/archc/1/carry54	000000000				
/archc/1/add55	000101001				
/archc/1/carry55	000000000				
/archc/1/add56	000101001				
/archc/1/carry56	000000000				
/archc/1/add57	000101001				
/archc/1/carry57	000000000				
/archc/1/add58	000101001				
/archc/1/carry58	000000000				
/archc/1/add59	000111010				
/archc/1/add60	000111000				
/archc/1/carry60	000000001				
/archc/1/add61	000111010				
/archc/1/carry61	000000000				
/archc/1/add62	000111010				

/archc/1/add63	000111010				
/archc/1/carry63	000000000				
/archc/1/add64	000111010				
/archc/1/carry64	000000000				
/archc/1/add65	000111010				
/archc/1/carry65	000000000				
/archc/1/add66	000111010				
/archc/1/carry66	000000000				
/archc/1/add67	000111010				
/archc/1/carry67	000000000				
/archc/1/add68	000111010				
/archc/1/carry68	000000000				
/archc/1/add69	000111010				
/archc/1/carry69	000000000				
/archc/1/add70	000111010				
/archc/1/carry70	000000000				
/archc/1/add71	000111010				
/archc/1/carry71	000000000				
/archc/1/add72	000111010				
/archc/1/carry72	000000000				
/archc/1/add73	000111010				
/archc/1/carry73	000000000				
/archc/1/add74	000111010				
/archc/1/carry74	000000000				
/archc/1/add75	000111010				
/archc/1/carry75	000000000				
/archc/1/add76	000111010				
/archc/1/carry76	000000000				
/archc/1/add77	000111010				
/archc/1/carry77	000000000				
/archc/1/add78	000111010				
/archc/1/carry78	000000000				

Figure 4.5 Simulated results of Arch-C

4.6 SIMULATED RESULTS OF ARCH-D

The simulated results of Arch-D are shown in the figure 4.6



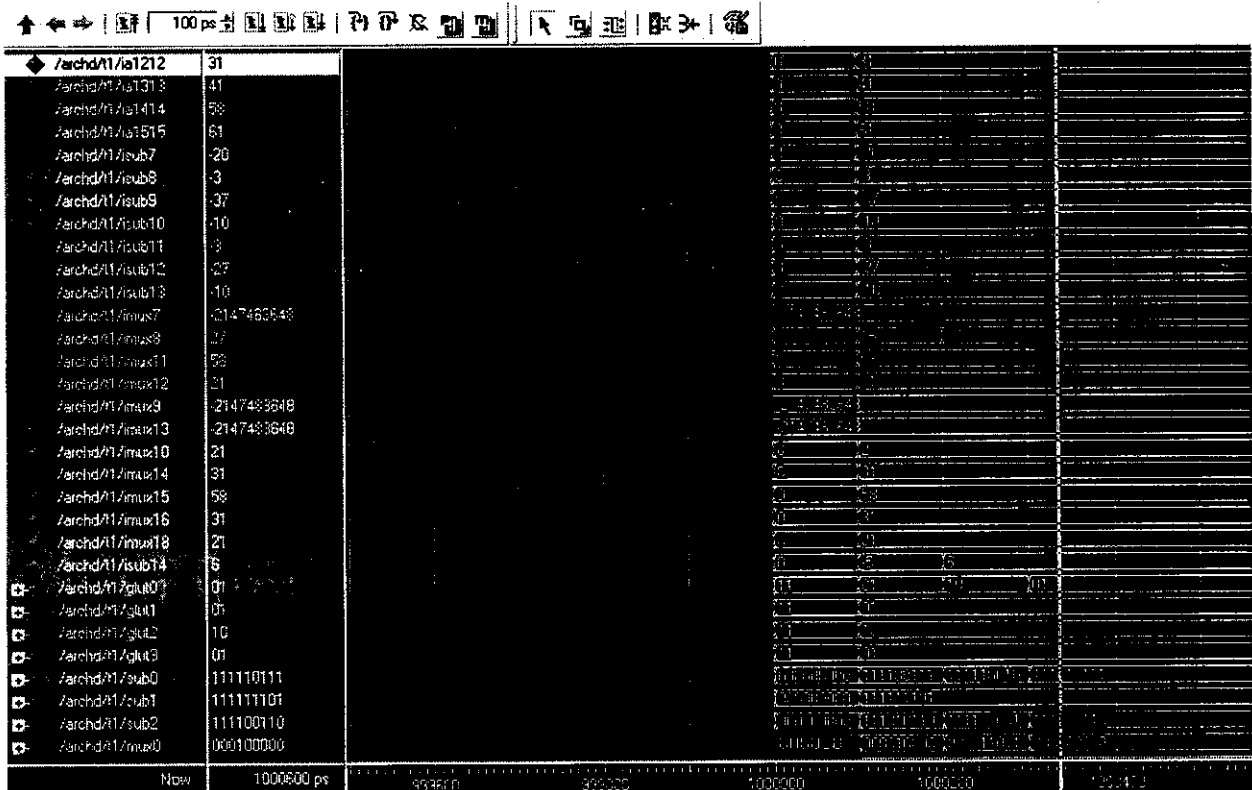
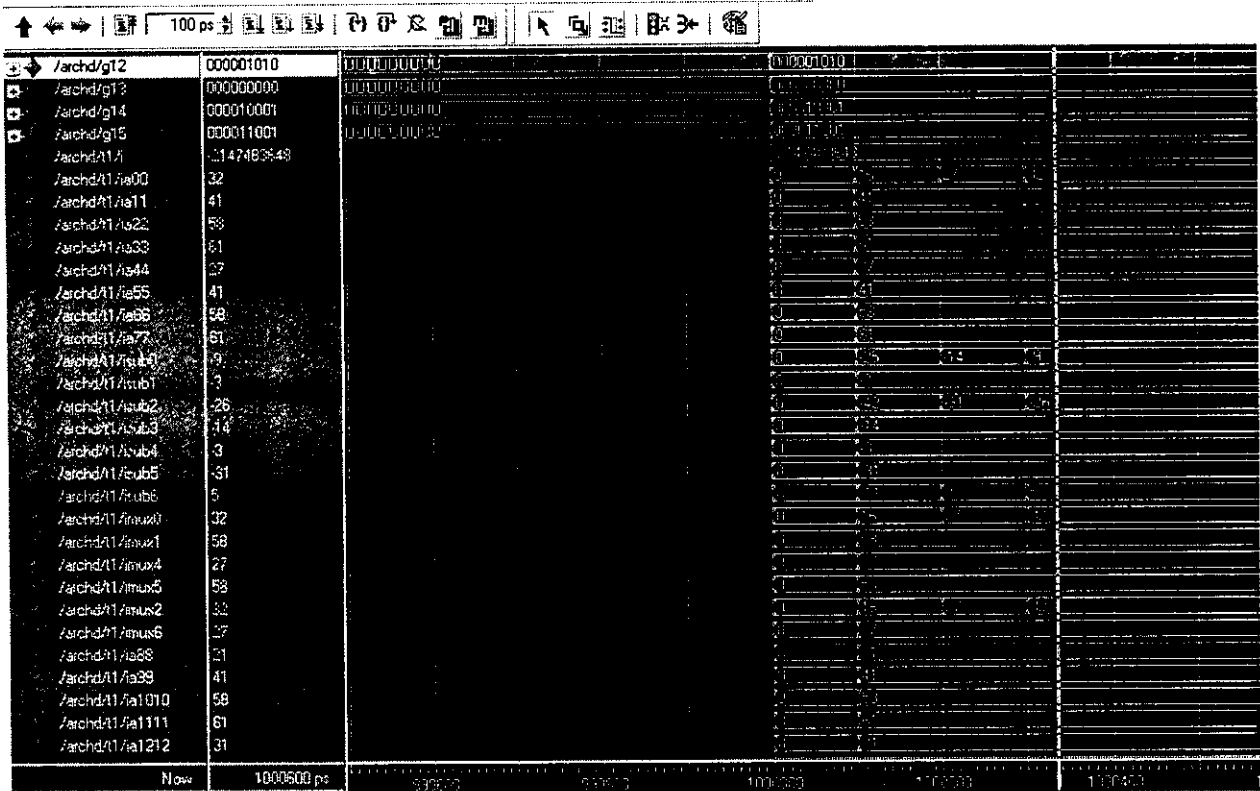


Figure 4.6 Simulated results of Arch-D

CHAPTER 5

CONCLUSION AND FUTURE WORK

In this work, we have designed the high speed recursion architecture Arch-C and Arch -D and obtained the simulated results of the same. Our future work will focus on designing other architectures with higher radices and compare their performances in terms of speed and area.

REFERENCES

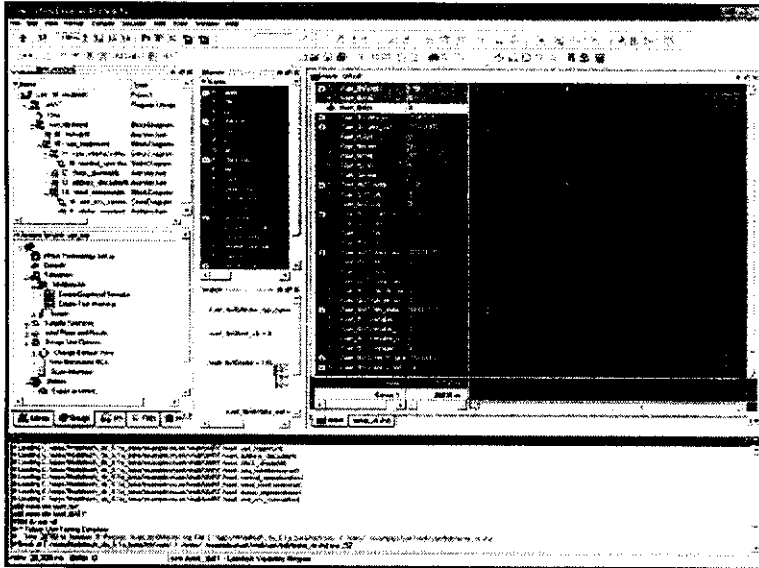
1. S.-J. Lee, N. Shanbhag, and A. Singer, "A 285-MHz pipelined MAP decoder in 0.18 m CMOS," *IEEE J. Solid-State Circuits*, vol. 40, no.8, pp. 1718–1725, Aug. 2005
2. P. Urard et al., "A generic 350 Mb/s Turbo codec based on a 16-state Turbo decoder." in *IEEE ISSCC Dig. Tech. Papers*, 2004, pp. 424–433.
3. E. Boutillon, W. Gross, and P. Gulak, "VLSI architectures for the MAP algorithm," *IEEE Trans. Commun.*, vol. 51, no. 2, pp. 175–185, Feb. 2003
4. M. Bickerstaff, L. Davis, C. Thomas, D. Garret, and C. Nicol, "A 24 Mb/s radix-4 Log-MAP Turbo decoder for 3 GPP-HSDPA mobile wireless," in *IEEE ISSCC Dig. Tech. Papers*, 2003, pp. 150–151.
5. T. Miyauchi, K. Yamamoto, and T. Yokokawa, "High-performance programmable SISO decoder VLSI implementation for decoding Turbo codes," in *Proc. IEEE Global Telecommun. Conf.*, 2001, pp. 305–309.

APPENDIX I

Design Creation to Realization

ModelSim Designer

D A T A S H E E T



ModelSim Designer combines easy to use, flexible creation with a powerful verification and debug environment.

The Easy to Use Solution for the FPGA Designer

ModelSim® Designer is a Windows®-based design environment for FPGAs. It provides an easy to use, advanced-feature tool at an entry-level price. The complete process of creation, management, simulation, and implementation are controlled from a single user interface, facilitating the design and verification flow and providing significant productivity gains.

ModelSim Designer combines the industry-leading capabilities of the ModelSim simulator with a built-in design creation engine. To support synthesis and place-and-route, ModelSim Designer is plug-in ready for the synthesis and place-and-route tools of your choice. Connection of these additional tools is easy, giving FPGA designers control of design creation, simulation, synthesis, and place-and route from a single cockpit.

A flexible methodology conforms to existing design flows. Designers can freely mix text entry of VHDL and Verilog code with graphical entry using block and state diagram editors. A single design unit can be represented in multiple views, and the management of compilation and simulation at all levels of abstraction is a single click away.

Major product features:

- Project manager, version control, and source code templates and wizards
- Block and state diagram editors
- Intuitive graphical waveform editor
- Automated testbench creation
- Text to graphic rendering facilitates analysis
- HTML Documentation option
- VHDL, Verilog, and mixed-language simulation
- Optimized native compiled architecture
- Single Kernel Simulator technology
- Advanced debug including Signal Spy™
- Memory window
- Easy-to-use GUI with Tcl interface
- Support for all major synthesis tools and the Actel, Altera, Lattice, and Xilinx place-and-route flows
- Built-in FPGA vendor library compiler
- Full support for Verilog 2001
- Windows platform support

Options: SWIFT Interface support, graphics-based Dataflow window, Waveform Compare, integrated code coverage, and Profiler (for more details on options, please see the ModelSim SE datasheet)

www.model.com/modelsimdesigner

**Mentor
Graphics**

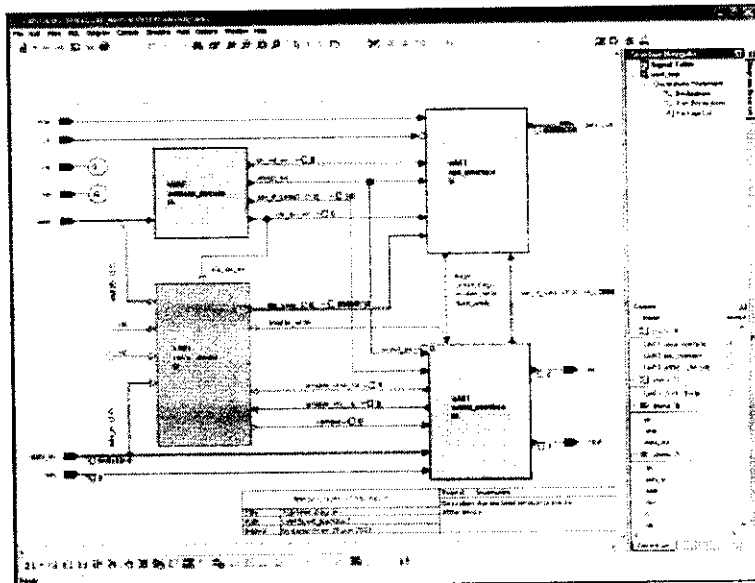
Implementation tasks are achieved through tight integration with the most popular FPGA synthesis and place-and-route tools in the industry. ModelSim Designer can either run these tools directly or externally, via scripts. Either way, the design data is maintained and used in a common and consistent way.

Project Manager

The flexible and powerful project manager feature allows easy navigation through a design in order to understand design content and execute all necessary tasks. The project manager provides easy access to all design data for each individual design unit, throughout the design process. Synthesis results, place-and-route results, back annotated netlists, and SDF files are associated with the relevant design unit. Project archiving is a few simple clicks away, allowing complete version control management of designs. The project manager also stores user-generated design documents, such as Word, PowerPoint, and PDF's.

Templates and Wizards

Creation wizards walk users through the creation of VHDL and Verilog design units, using either text or graphics. In the case of the graphical editor, HDL code is generated from the graphical diagrams created. For text-based design, VHDL and Verilog templates and wizards help engineers quickly develop HDL code without having to remember the exact language syntax. The wizards show how to create parameterizable logic blocks, testbench stimuli, and design objects. Novice and advanced HDL developers both benefit from timesaving shortcuts.



The Block Diagram editor is a convenient way to visually partition your design and have the HDL code automatically generated.

Intuitive Graphical Editors

ModelSim Designer includes block diagram and state machine editors that ensure a consistent coding style, facilitating design reuse and maintenance. These editors use an intuitive, graphical methodology that flattens the learning curve. This shortens the time to productivity for designers who are migrating to HDL methodologies or changing their primary design language.

Designers can automatically view or render diagrams from HDL code in block diagrams or state machines. When the code changes, the diagram can be updated instantly with its accuracy ensured. This helps designers understand legacy designs and aids in the debugging of current designs.

Automated Testbench Creation

ModelSim Designer offers an automated mechanism for testbench generation. The testbench wizard generates

VHDL or Verilog code through a graphical waveform editor, with output in either HDL or a TCL script. Users can manually define signals in the waveform editor or use the built-in wizard to define the waveforms. Either way, it is intuitive and easy to use and saves considerable time.

Active Design Visualization Enhances Simulation Debugging

During live simulation, design analysis capabilities are enhanced through graphical design views. From any diagram window, simulations can be fully executed and controlled. Enhanced debugging features include graphical breakpoints, signal probing, graphics-to-text-source cross-highlighting, animation, and cause analysis. The ability to overlay live simulation results in a graphical context speeds up the debug process by allowing faster problem discovery and shorter design iterations.

HTML Documentation

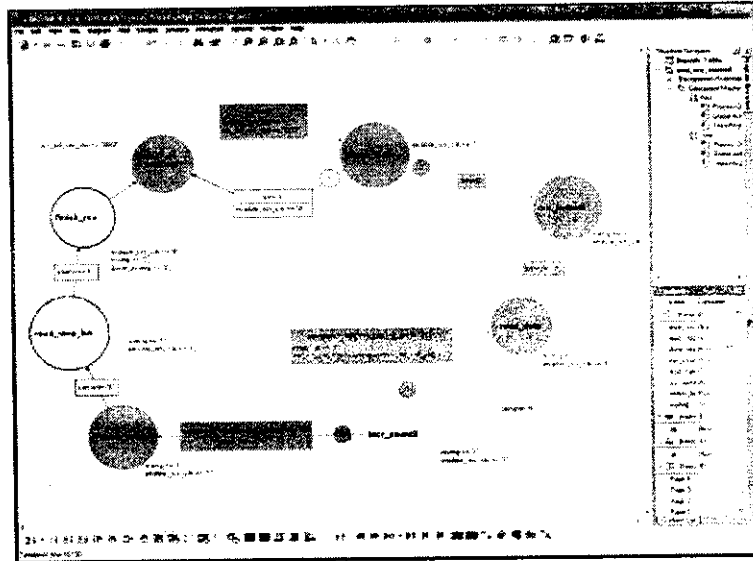
The HTML (HyperText Markup Language) Documentation option allows the user to output the complete design in HTML format. This allows the graphics, HDL, associative design files, and design hierarchy to be shared with anybody that uses an HTML browser. Design hierarchy and details can be viewed and navigated in any HTML browser, aiding communication and documentation.

Memory Window

The memory window enables flexible viewing and switching of memory locations. VHDL and Verilog memories are auto-extracted in the GUI, delivering powerful search, fill, load, and save functionality. The memory window allows pre-loading of memories, thus saving the time-consuming step of initializing sections of the simulation to load memories. All functions are available via the command line, making them available for scripting.

Intelligent GUI

An intelligently engineered GUI makes efficient use of desktop real estate. ModelSim Designer's intuitive layout of graphical elements (windows, toolbars, menus, etc.) makes it easy to step through the design flow. There are also wizards in place which help set up the design environment and make going through the design process seamless and efficient.



The State Diagram Editor uses interactive animation to track the simulation through the state machine and to check that all states were exercised.

Synthesis and Place-and-Route Integration

The industry's popular FPGA synthesis tools, such as Mentor Graphics Precision-RTL, and most FPGA vendor synthesis tools, can be integrated into ModelSim Designer with push button convenience. Actel Designer and Libero, Altera Quartus, Lattice ispLEVER, and Nilinx ISE place-and-route software are similarly integrated. Place-and-route results, together with SDF information, are automatically managed by ModelSim

Designer after the process is complete, making them ready for post-place-and-route, gate-level simulation.

FPGA Vendor Library Compiler

ModelSim Designer provides an intuitive mechanism to compile the necessary vendor libraries for post-place-and-route simulation. The compiler detects which FPGA vendor tools have been installed and compiles the necessary libraries as soon as the tool is launched, taking care of this step once and for all.

Visit our web site at www.model.com/modelsimdesigner for more information.

Copyright © 2005 Mentor Graphics Corporation.
Signal Spy is a trademark, and ModelSim and Mentor Graphics are registered trademarks of Mentor Graphics Corporation.
All other trademarks mentioned in this document are trademarks of their respective owners.

APPENDIX II



DS077-1 (v2.3) June 18, 2008

Spartan-IIE FPGA Family: Introduction and Ordering Information

Product Specification

Introduction

The Spartan[®]-IIE Field-Programmable Gate Array family gives users high performance, abundant logic resources, and a rich feature set, all at an exceptionally low price. The seven-member family offers densities ranging from 50,000 to 600,000 system gates, as shown in Table 1. System performance is supported beyond 200 MHz.

Features include block RAM (to 288K bits), distributed RAM (to 221,184 bits), 19 selectable I/O standards, and four DLLs (Delay-Locked Loops). Fast, predictable interconnect means that successive design iterations continue to meet timing requirements.

The Spartan-IIE family is a superior alternative to mask-programmed ASICs. The FPGA avoids the initial cost, lengthy development cycles, and inherent risk of conventional ASICs. Also, FPGA programmability permits design upgrades in the field with no hardware replacement necessary (impossible with ASICs).

Features

- Second generation ASIC replacement technology
 - Densities as high as 15,552 logic cells with up to 600,000 system gates
 - Streamlined features based on Virtex[®]-E FPGA architecture
 - Unlimited in-system reprogrammability
 - Very low cost
 - Cost-effective 0.15 micron technology
- System level features
 - SelectRAM[™] hierarchical memory:
 - 16 bits/LUT distributed RAM
 - Configurable 4K-bit true dual-port block RAM

- Fast interfaces to external RAM
- Fully 3.3V PCI compliant to 64 bits at 66 MHz and CardBus compliant
- Low-power segmented routing architecture
- Dedicated carry logic for high-speed arithmetic
- Efficient multiplier support
- Cascade chain for wide-input functions
- Abundant registers/latches with enable, set, reset
- Four dedicated DLLs for advanced clock control
 - Eliminate clock distribution delay
 - Multiply, divide, or phase shift
- Four primary low-skew global clock distribution nets
- IEEE 1149.1 compatible boundary scan logic
- Versatile I/O and packaging
 - Pb-free package options
 - Low-cost packages available in all densities
 - Family footprint compatibility in common packages
 - 19 high-performance interface standards
 - LVTTTL, LVCMOS, HSTL, SSTL, AGP, CTT, GTL
 - LVDS and LVPECL differential I/O
 - Up to 205 differential I/O pairs that can be input, output, or bidirectional
 - Hot swap I/O (CompactPCI friendly)
- Core logic powered at 1.8V and I/Os powered at 1.5V, 2.5V, or 3.3V
- Fully supported by powerful Xilinx[®] ISE[®] development system
 - Fully automatic mapping, placement, and routing
 - Integrated with design entry and verification tools
 - Extensive IP library including DSP functions and soft processors

Table 1: Spartan-IIE FPGA Family Members

Device	Logic Cells	Typical System Gate Range (Logic and RAM)	CLB Array (R x C)	Total CLBs	Maximum Available User I/O ⁽¹⁾	Maximum Differential I/O Pairs	Distributed RAM Bits	Block RAM Bits
XC2S50E	1,728	23,000 - 50,000	16 x 24	384	182	83	24,576	32K
XC2S100E	2,700	37,000 - 100,000	20 x 30	600	202	86	38,400	40K
XC2S150E	3,888	52,000 - 150,000	24 x 36	864	265	114	55,296	48K
XC2S200E	5,292	71,000 - 200,000	28 x 42	1,176	289	120	75,264	56K
XC2S300E	6,912	93,000 - 300,000	32 x 48	1,536	329	120	98,304	64K
XC2S400E	10,800	145,000 - 400,000	40 x 60	2,400	410	172	153,600	160K
XC2S600E	15,552	210,000 - 600,000	48 x 72	3,456	514	205	221,184	288K

Notes:

1. User I/O counts include the four global clock/user input pins. See details in Table 2, page 5

© 2003-2008 Xilinx, Inc. All rights reserved. XILINX, the Xilinx logo, the Brand Window, and other designated brands included herein are trademarks of Xilinx, Inc. All other trademarks are the property of their respective owners.

DS077-1 (v2.3) June 18, 2008
Product Specification

www.xilinx.com

General Overview

The Spartan-IIE family of FPGAs have a regular, flexible, programmable architecture of Configurable Logic Blocks (CLBs), surrounded by a perimeter of programmable Input/Output Blocks (IOBs). There are four Delay-Locked Loops (DLLs), one at each corner of the die. Two columns of block RAM lie on opposite sides of the die, between the CLBs and the IOB columns. The XC2S400E has four columns and the XC2S600E has six columns of block RAM. These functional elements are interconnected by a powerful hierarchy of versatile routing channels (see Figure 1).

Spartan-IIE FPGAs are customized by loading configuration data into internal static memory cells. Unlimited reprogramming cycles are possible with this approach. Stored values in these cells determine logic functions and interconnections implemented in the FPGA. Configuration data can be read from an external serial PROM (master serial mode), or written into the FPGA in slave serial, slave parallel, or Boundary Scan modes. Xilinx offers multiple types of low-cost configuration solutions including the Platform Flash in-system programmable configuration PROMs.

Spartan-IIE FPGAs are typically used in high-volume applications where the versatility of a fast programmable solution adds benefits. Spartan-IIE FPGAs are ideal for shortening product development cycles while offering a cost-effective solution for high volume production.

Spartan-IIE FPGAs achieve high-performance, low-cost operation through advanced architecture and semiconductor technology. Spartan-IIE devices provide system clock rates beyond 200 MHz. In addition to the conventional benefits of high-volume programmable logic solutions, Spartan-IIE FPGAs also offer on-chip synchronous single-port and dual-port RAM (block and distributed form). DLL clock drivers, programmable set and reset on all flip-flops, fast carry logic, and many other features.

Spartan-IIE Family Compared to Spartan-II Family

- Higher density and more I/O
- Higher performance
- Unique pinouts in cost-effective packages
- Differential signaling
 - LVDS, Bus LVDS, LVPECL
- $V_{CCINT} = 1.8V$
 - Lower power
 - 5V tolerance with external resistor
 - 3V tolerance directly
- PCI, LVTTTL, and LVCMOS2 input buffers powered by V_{CCO} instead of V_{CCINT}
- Unique larger bitstream

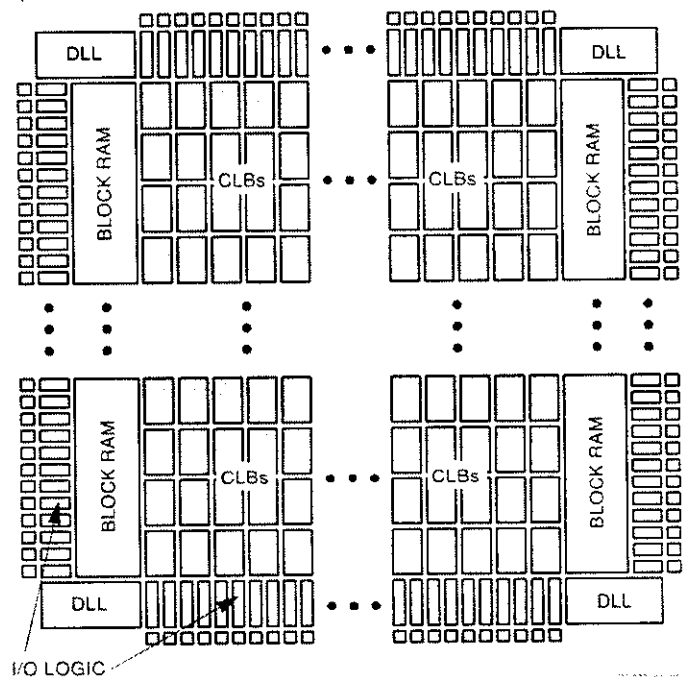


Figure 1: Basic Spartan-IIE Family FPGA Block Diagram

Architectural Description

Spartan-IIE FPGA Array

The Spartan[®]-IIE user-programmable gate array, shown in Figure 3, is composed of five major configurable elements:

- IOBs provide the interface between the package pins and the internal logic
- CLBs provide the functional elements for constructing most logic
- Dedicated block RAM memories of 4096 bits each
- Clock DLLs for clock-distribution delay compensation and clock domain control
- Versatile multi-level interconnect structure

As can be seen in Figure 3, the CLBs form the central logic structure with easy access to all support and routing structures. The IOBs are located around all the logic and memory elements for easy and quick routing of signals on and off the chip.

Values stored in static memory cells control all the configurable logic elements and interconnect resources. These values load into the memory cells on power-up, and can reload if necessary to change the function of the device.

Each of these elements will be discussed in detail in the following sections.

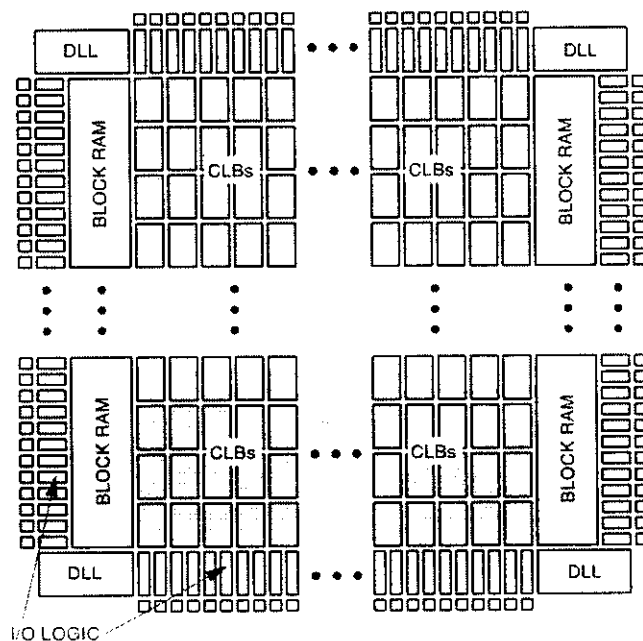
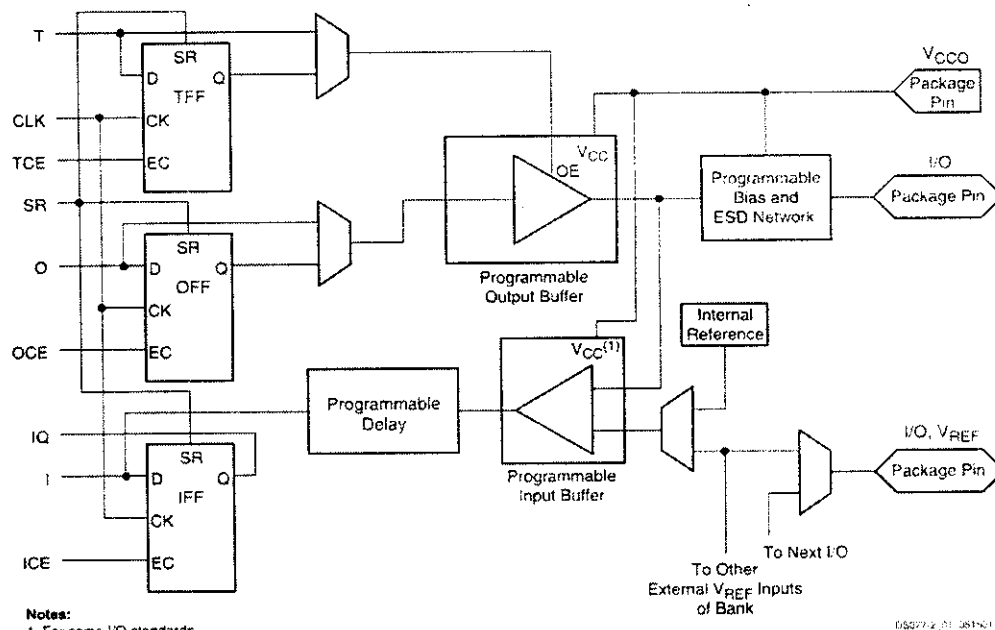


Figure 3: Basic Spartan-IIE Family FPGA Block Diagram



Notes:
1. For some I/O standards.

DS077-2 (v2.3) 051501

Figure 4: Spartan-IIE Input/Output Block (IOB)

Table 3: Standards Supported by I/O (Typical Values)

I/O Standard	Input Reference Voltage (V_{REF})	Input Voltage (V_{CC0})	Output Source Voltage (V_{CC0})	Board Termination Voltage (V_{TT})
LVTTL (2-24 mA)	N/A	3.3	3.3	N/A
LVC MOS2	N/A	2.5	2.5	N/A
LVC MOS18	N/A	1.8	1.8	N/A
PCI (3V, 33 MHz/66 MHz)	N/A	3.3	3.3	N/A
GTL	0.8	N/A	N/A	1.2
GTL+	1.0	N/A	N/A	1.5
HSTL Class I	0.75	N/A	1.5	0.75
HSTL Class III	0.9	N/A	1.5	1.5
HSTL Class IV	0.9	N/A	1.5	1.5
SSTL3 Class I and II	1.5	N/A	3.3	1.5
SSTL2 Class I and II	1.25	N/A	2.5	1.25
CTT	1.5	N/A	3.3	1.5
AGP	1.32	N/A	3.3	N/A
LVDS, Bus LVDS	N/A	N/A	2.5	N/A
LVPECL	N/A	N/A	3.3	N/A

Input/Output Block

The Spartan-IIE FPGA IOB, as seen in Figure 4, features inputs and outputs that support a wide variety of I/O signaling standards. These high-speed inputs and outputs are capable of supporting various state of the art memory and bus interfaces. The default standard is LVTTL. Table 3 lists several of the standards which are supported along with the required reference (V_{REF}), output (V_{CC0}) and board termination (V_{TT}) voltages needed to meet the standard. For more details on the I/O standards and termination application examples, see XAPP179, "Using SelectIO Interfaces in Spartan-II and Spartan-IIE FPGAs."

The three IOB registers function either as edge-triggered D-type flip-flops or as level-sensitive latches. Each IOB has a clock signal (CLK) shared by the three registers and independent Clock Enable (CE) signals for each register.

In addition to the CLK and CE control signals, the three registers share a Set/Reset (SR). For each register, this signal can be independently configured as a synchronous Set, a synchronous Reset, an asynchronous Preset, or an asynchronous Clear.

A feature not shown in the block diagram, but controlled by the software, is polarity control. The input and output buffers and all of the IOB control signals have independent polarity controls.

Optional pull-up and pull-down resistors and an optional weak-keeper circuit are attached to each user I/O pad. Prior to configuration all outputs not involved in configuration are forced into their high-impedance state. The pull-down resistors and the weak-keeper circuits are inactive, but inputs may optionally be pulled up. The activation of pull-up resistors prior to configuration is controlled on a global basis by the configuration mode pins. If the pull-up resistors are not activated, all the pins will float. Consequently, external pull-up or pull-down resistors must be provided on pins required to be at a well-defined logic level prior to configuration.

All pads are protected against damage from electrostatic discharge (ESD) and from over-voltage transients. After configuration, clamping diodes are connected to V_{CC0} for LVTTTL, PCI, HSTL, SSTL, CTT, and AGP standards.

All Spartan-II^E FPGA IOBs support IEEE 1149.1-compatible boundary scan testing.

Input Path

A buffer in the IOB input path routes the input signal directly to internal logic and through an optional input flip-flop.

An optional delay element at the D-input of this flip-flop eliminates pad-to-pad hold time. The delay is matched to the internal clock-distribution delay of the FPGA, and when used, assures that the pad-to-pad hold time is zero.

Each input buffer can be configured to conform to any of the low-voltage signaling standards supported. In some of these standards the input buffer utilizes a user-supplied threshold voltage, V_{REF} . The need to supply V_{REF} imposes constraints on which standards can be used in close proximity to each other. See I/O Banking.

There are optional pull-up and pull-down resistors at each input for use after configuration.

Output Path

The output path includes a 3-state output buffer that drives the output signal onto the pad. The output signal can be routed to the buffer directly from the internal logic or through an optional IOB output flip-flop.

The 3-state control of the output can also be routed directly from the internal logic or through a flip-flop that provides synchronous enable and disable.

Each output driver can be individually programmed for a wide range of low-voltage signaling standards. Each output buffer can source up to 24 mA and sink up to 48 mA. Drive strength and slew rate controls minimize bus transients. The default output driver is LVTTTL with 12 mA drive strength and slow slew rate.

In most signaling standards, the output high voltage depends on an externally supplied V_{CC0} voltage. The need to supply V_{CC0} imposes constraints on which standards

can be used in close proximity to each other. See I/O Banking.

An optional weak-keeper circuit is connected to each output. When selected, the circuit monitors the voltage on the pad and weakly drives the pin High or Low to match the input signal. If the pin is connected to a multiple-source signal, the weak keeper holds the signal in its last state if all drivers are disabled. Maintaining a valid logic level in this way helps eliminate bus chatter.

Because the weak-keeper circuit uses the IOB input buffer to monitor the input level, an appropriate V_{REF} voltage must be provided if the signaling standard requires one. The provision of this voltage must comply with the I/O banking rules.

I/O Banking

Some of the I/O standards described above require V_{CC0} and/or V_{REF} voltages. These voltages are externally supplied and connected to device pins that serve groups of IOBs, called banks. Consequently, restrictions exist about which I/O standards can be combined within a given bank.

Eight I/O banks result from separating each edge of the FPGA into two banks (see Figure 5). The pinout tables show the bank affiliation of each I/O (see Pinout Tables, page 53). Each bank has multiple V_{CC0} pins which must be connected to the same voltage. Voltage requirements are determined by the output standards in use.

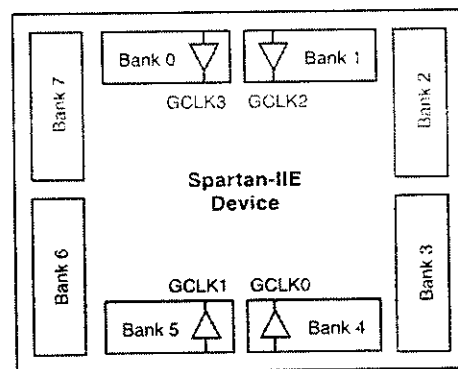


Figure 5: Spartan-II^E I/O Banks

In the TQ144 and PQ208 packages, the eight banks have V_{CC0} connected together. Thus, only one V_{CC0} level is allowed in these packages, although different V_{REF} values are allowed in each of the eight banks.

Within a bank, standards may be mixed only if they use the same V_{CC0} . Compatible standards are shown in Table 4. GTL and GTL+ appear under all voltages because their open-drain outputs do not depend on V_{CC0} . Note that V_{CC0}

is required for most output standards and for LVTTTL, LVCMOS, and PCI inputs.

Table 4: Compatible Standards

V _{CCO}	Compatible Standards
3.3V	PCI, LVTTTL, SSTL3 I, SSTL3 II, CTT, AGP, LVPECL, GTL, GTL+
2.5V	SSTL2 I, SSTL2 II, LVCMOS2, LVDS, Bus LVDS, GTL, GTL+
1.8V	LVCMOS18, GTL, GTL+
1.5V	HSTL I, HSTL III, HSTL IV, GTL, GTL+

Some input standards require a user-supplied threshold voltage, V_{REF}. In this case, certain user-I/O pins are automatically configured as inputs for the V_{REF} voltage. About one in six of the I/O pins in the bank assume this role.

V_{REF} pins within a bank are interconnected internally and consequently only one V_{REF} voltage can be used within each bank. All V_{REF} pins in the bank, however, must be connected to the external voltage source for correct operation.

In a bank, inputs requiring V_{REF} can be mixed with those that do not but only one V_{REF} voltage may be used within a bank. The V_{CCO} and V_{REF} pins for each bank appear in the device pinout tables.

Within a given package, the number of V_{REF} and V_{CCO} pins can vary depending on the size of device. In larger devices, more I/O pins convert to V_{REF} pins. Since these are always a superset of the V_{REF} pins used for smaller devices, it is possible to design a PCB that permits migration to a larger device. All V_{REF} pins for the largest device anticipated must be connected to the V_{REF} voltage, and not used for I/O.

Table 5: I/O Banking

Package	TQ144, PQ208	FT256, FG456, FG676
V _{CCO} Banks	Interconnected as 1	8 independent
V _{REF} Banks	8 independent	8 independent

See Xilinx® Application Note [XAPP179](#) for more information on I/O resources.

Hot Swap, Hot Insertion, Hot Socketing Support

The I/O pins support hot swap — also called hot insertion and hot socketing — and are considered CompactPCI Friendly according to the PCI Bus v2.2 Specification. Consequently, an unpowered Spartan-IIIE FPGA can be plugged directly into a powered system or backplane without affecting or damaging the system or the FPGA. The hot swap functionality is built into every XC2S150E, XC2S400E, and XC2S600E device. All other Spartan-IIIE devices built after Product Change Notice [PCN2002-05](#) also include hot swap functionality.

To support hot swap, Spartan-IIIE devices include the following I/O features.

- Signals can be applied to Spartan-IIIE FPGA I/O pins before powering the FPGA's V_{CCINT} or V_{CCO} supply inputs.
- Spartan-IIIE FPGA I/O pins are high-impedance (i.e., three-stated) before and throughout the power-up and configuration processes when employing a configuration mode that does not enable the preconfiguration weak pull-up resistors (see Table 11, page 22).
- There is no current path from the I/O pin back to the V_{CCINT} or V_{CCO} voltage supplies.
- Spartan-IIIE FPGAs are immune to latch-up during hot swap.

Once connected to the system, each pin adds a small amount of capacitance (C_{IN}). Likewise, each I/O consumes a small amount of DC current, equivalent to the input leakage specification (I_L). There also may be a small amount of temporary AC current (I_{HSP0}) when the pin input voltage exceeds V_{CCO} plus 0.4V, which lasts less than 10 ns.

A weak-keeper circuit within each user-I/O pin is enabled during the last frame of configuration data and has no noticeable effect on robust system signals driven by an active driver or a strong pull-up or pull-down resistor. Undriven or floating system signals may be affected. The specific effect depends on how the I/O pin is configured. User-I/O pins configured as outputs or enabled outputs have a weak pull-up resistor to V_{CCO} during the last configuration frame. User-I/O pins configured as inputs or bidirectional I/Os have weak pull-down resistors. The weak-keeper circuit turns off when the DONE pin goes High, provided that it is not used in the configured application.

Configurable Logic Block

The basic building block of the Spartan-IIE FPGA CLB is the logic cell (LC). An LC includes a 4-input function generator, carry logic, and storage element. The output from the function generator in each LC drives the CLB output or the D input of the flip-flop. Each Spartan-IIE FPGA CLB contains four LCs, organized in two similar slices: a single slice is shown in Figure 6.

In addition to the four basic LCs, the Spartan-IIE FPGA CLB contains logic that combines function generators to provide functions of five or six inputs.

Look-Up Tables

Spartan-IIE FPGA function generators are implemented as 4-input look-up tables (LUTs). In addition to operating as a function generator, each LUT can provide a 16 x 1-bit synchronous RAM. Furthermore, the two LUTs within a slice can be combined to create a 16 x 2-bit or 32 x 1-bit synchronous RAM, or a 16 x 1-bit dual-port synchronous RAM.

The Spartan-IIE FPGA LUT can also provide a 16-bit shift register that is ideal for capturing high-speed or burst-mode data. This mode can also be used to store data in applications such as Digital Signal Processing.

Storage Elements

Storage elements in the Spartan-IIE FPGA slice can be configured either as edge-triggered D-type flip-flops or as level-sensitive latches. The D inputs can be driven either by function generators within the slice or directly from slice inputs, bypassing the function generators.

In addition to Clock and Clock Enable signals, each slice has synchronous set and reset signals (SR and BY). SR forces a storage element into the initialization state specified for it in the configuration. BY forces it into the opposite state. Alternatively, these signals may be configured to operate asynchronously.

All control signals are independently invertible, and are shared by the two flip-flops within the slice.

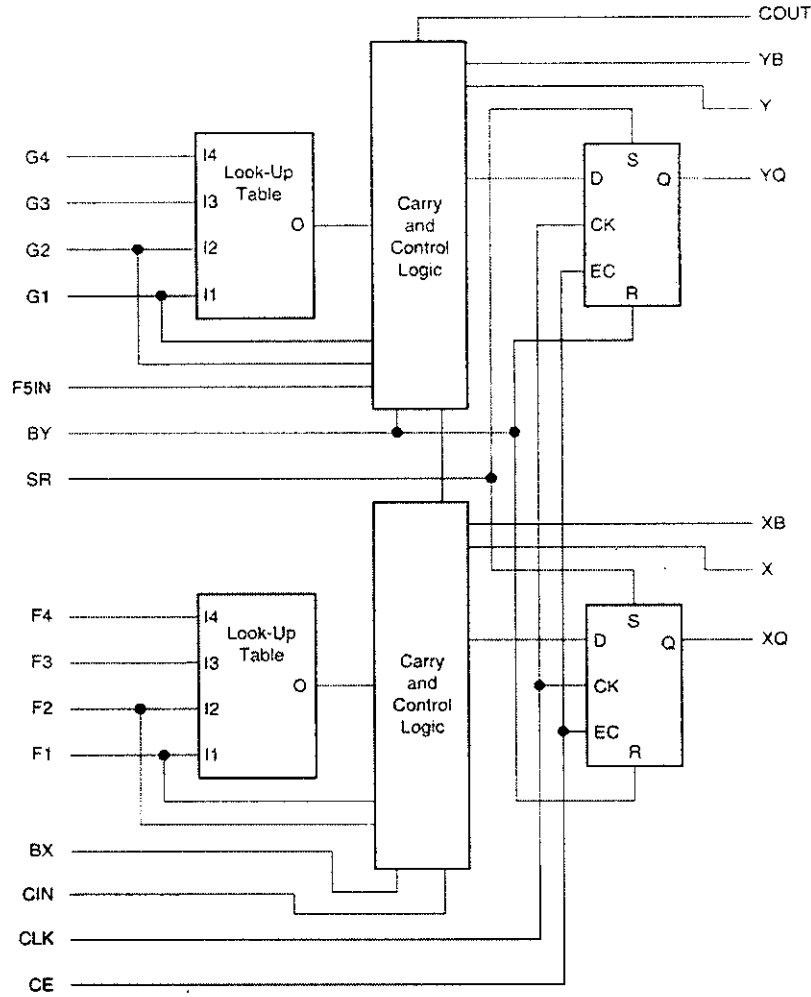
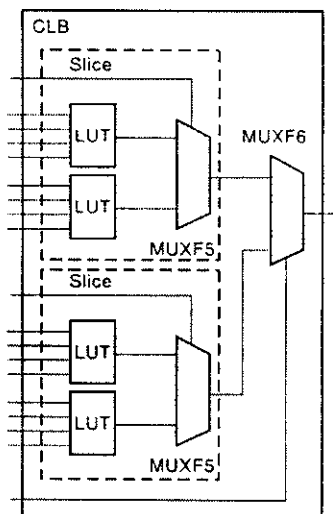


Figure 6: Spartan-IIe CLB Slice (two identical slices in each CLB)

Additional Logic

The F5 multiplexer in each slice combines the function generator outputs (Figure 7). This combination provides either a function generator that can implement any 5-input function, a 4:1 multiplexer, or selected functions of up to nine inputs.

Similarly, the F6 multiplexer combines the outputs of all four function generators in the CLB by selecting one of the two F5-multiplexer outputs. This permits the implementation of any 6-input function, an 8:1 multiplexer, or selected functions of up to 19 inputs.



DS077-2_05-111501

Figure 7: F5 and F6 Multiplexers

Each CLB has four direct feedthrough paths, one per LC. These paths provide extra data input lines or additional local routing that does not consume logic resources.

Arithmetic Logic

Dedicated carry logic provides capability for high-speed arithmetic functions. The Spartan-IIE FPGA CLB supports two separate carry chains, one per slice. The height of the carry chains is two bits per CLB.

The arithmetic logic includes an XOR gate that allows a 1-bit full adder to be implemented within an LC. In addition, a dedicated AND gate improves the efficiency of multiplier implementations.

The dedicated carry path can also be used to cascade function generators for implementing wide logic functions.

BUFTs

Each Spartan-IIE FPGA CLB contains two 3-state drivers (BUFTs) that can drive on-chip busses. The IOBs on the left and right sides can also drive the on-chip busses. See Dedicated Routing, page 17. Each Spartan-IIE FPGA BUFT has an independent 3-state control pin and an independent input pin. The 3-state control pin is an active-Low enable (T). When all BUFTs on a net are disabled, the net is High. There is no need to instantiate a pull-up unless desired for simulation purposes. Simultaneously driving BUFTs onto the same net will not cause contention. If driven both High and Low, the net will be Low.

Block RAM

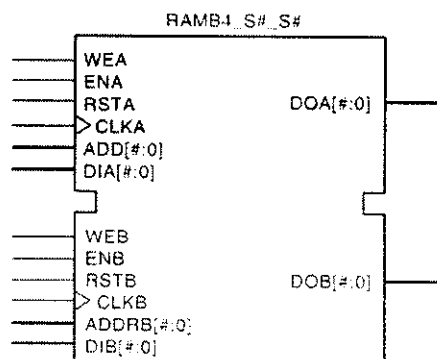
Spartan-IIE FPGAs incorporate several large block RAM memories. These complement the distributed RAM Look-Up Tables (LUTs) that provide shallow memory structures implemented in CLBs.

Block RAM memory blocks are organized in columns. Most Spartan-IIE devices contain two such columns, one along each vertical edge. The XC2S400E has four block RAM columns and the XC2S600E has six block RAM columns. These columns extend the full height of the chip. Each memory block is four CLBs high, and consequently, a Spartan-IIE device 16 CLBs high will contain four memory blocks per column, and a total of eight blocks.

Table 6: Spartan-IIE Block RAM Amounts

Spartan-IIE Device	# of Blocks	Total Block RAM Bits
XC2S50E	8	32K
XC2S100E	10	40K
XC2S150E	12	48K
XC2S200E	14	56K
XC2S300E	16	64K
XC2S400E	40	160K
XC2S600E	72	288K

Each block RAM cell, as illustrated in Figure 8, is a fully synchronous dual-ported 4096-bit RAM with independent control signals for each port. The data widths of the two ports can be configured independently, providing built-in bus-width conversion.



DS077-2_05-111501

Figure 8: Dual-Port Block RAM