FORTH-ICS / TR-304 March 2002

Fast Parallel Comparison Circuits for Scheduling

Kostas G. I. Harteros[1,2]

Abstract

Per-flow queueing and sophisticated schedulers are an important mechanism for providing Quality of Service (QoS) guarantees in networks. Most advanced scheduling algorithms rely on a common computational primitive: the priority queue. Priority queues can be built efficiently using heap data structures, when the set of elements in the queue varies slowly. However, if this set of eligible flows changes arbitrarily fast, new hardware structures are needed to support high-speed operation; in this work we develop such circuits. We use a binary tree of comparators, which locates the minimum in an arbitrary set of elements. The comparison operation is studied extensively and a 2-element comparator, which is the main block of the binary tree, is designed. We developed an innovative organization for the tree, where signals are propagated across each 2-element comparator as well as the tree levels, at the same time; in this way, the delays of the individual comparators and the delays of the tree levels are placed in parallel, rather than in series. This binary tree of comparators is the heart of a weighted-round-robin scheduler that we designed. Our designs are desicribed in synthesisable Verilog (HDL); in addition, designs were described in C code for verification purposes. Synthesis results are presented in terms of delay, power, and area, for a 0.18 CMOS process.

[1]ICS-FORTH, P.O. Box 1385, GR 711 10 Heraklion, Crete, Hellas. E-mail: harteros@ics.forth.gr
[2]Department of Computer Science, University of Crete, Heraklion, Crete, Hellas.

Acknowledgments

I would like to thank all those who helped me throughout my work. First of all I would like to thank professor Manolis Katevenis who gave the idea for the whole work; I would like to thank him both for his priceless suggestions and ideas during my work, and for trying to reveal and communicate to me the way one should think while designing and in general while working on scientific research.

I would like to greatly thank professor Dionysis Pnevmatikatos for his helpful remarks on synthesis, and professors Apostolos Traganitis and Evangelos Markatos who participated for the evaluation of this work. Also, I thank George Kornaros and Christos Sotiriou for teaching me CAD Tools.

I would like to thank Nikos Chryssos for many hours of working together and for the discussions about this work, for his critical suggestions and observations. Also, I thank Pavlos Robogiannakis for corrections on this document and Tassia Rissanou for the transparencies of the presentation of this work.

I would like to thank Grigoris Gikas, Xeni Asimakopoulou, and Antonis Danalis for their help in my first steps in the Computer Science Department.

I would like to thank Europractice and the University of Crete for providing many of the CAD tools used and the Technology Libraries. Also I would like to thank the University of Crete and the Foundation for Research and Technology Hellas for the funding provided.

Finally I thank my family, George, Irene, Vassilis and Voula, for their help and support throughout my studies.

Contents

1 Scheduling for QoS in Advanced Networks **1**

1.1 Scheduling Disciplines 3

1.2 Choices in designing disciplines 4

1.3 Approaches to Service Schedules 5

1.4 Contributions of This Work 6

2 Priority Queue Implementations in Hardware **8**

2.1 Weighted Round Robin 8

2.2 Related Work in Priority Queue Algorithms implemented in Hardware . 12

2.3 Cases where Heap Data Structures are inappropriate 12

2.4 Algorithms for Minimum Value Determination in a non-sorted Set of Elements 14

2.4.1 Array and Row Architectures 14

2.4.2 Traditional Binary Tree Architecture 17

2.4.3 Parallel Operation of multiple Tree Levels 19

3 Comparator Circuits and Trees **22**

3.1 The Binary Comparator 23

3.1.1 Definitions 23

3.1.2 The 2 Bit Comparator Cell 23

3.1.3 Coding 24

3.1.4 The Circuit 27

3.2 The 2 Element Comparator 30

3.2.1 The Ripple Comparator 30
3.2.2 The Square Root Carry Select Comparator 31
3.2.3 The Carry Look Ahead Comparator 36
3.2.4 Conclusions for the 2 Element Comparator 39
3.3 The Binary Tree Comparator 40
3.3.1 The Binary Three with Ripple Comparators. 41
3.3.2 The Binary Three with Carry Select Comparators. 42
3.3.3 The Binary Three with Carry Look Ahead Comparators. 45

4 Design of the Scheduler **47**
4.1 Operations 47
4.2 Tasks and Interface 48
4.3 Element Representation and Wrap-around 49
4.4 Pipelining 51
4.5 Building the Datapath 52
4.6 Adding more Features 57
4.7 Optimizations for Fan-out 59
4.8 Economizing on Power 60
4.9 Conclusions on Datapath Design 61

5 Design Flow **62**
5.1 Technology used for Synthesis 63
5.2 Binary Tree of Comparators 63
5.2.1 Delay Results 64
5.2.2 Area Results 65
5.2.3 Power Results 69

6 Conclusions **71**

A Pattern Generator **73**

B Synthesis Scripts **74**

List of Figures

2.1 The WRR scheduling with 5 flows. There are two cases presented: 1. The smooth scheduling where the services are spread in time. 2. The bursty scheduling where the services are gathered. 9

2.2 The work-conserving and the non-work-conserving options. 10

2.3 Reinsert eligible flows. 10

2.4 Example of WRR algorithm. 11

2.5 The array architecture for the comparison of N elements, k-bit each. Each bit slice (row) compares N bits and gives one bit of the winner M. The columns are the bits of each element n_i. 15

2.6 (a) The architecture of the cell that enables the element to participate in the comparison and propagates the disable signal to the next lower significant bit. (b) The coding of the cell. 16

2.7 The row architecture for the comparison of N elements, k-bit each. 17

2.8 (a) The structure of the circuit is a binary tree with 8 leaves (N = 16). Every block is a 2 element comparator. The delay of the circuit is O(logN) = 4. The advantage of this scheme is that comparisons of different elements are taking place in parallel. (b) Arithmetic example for the determination of the total minimum. 18

2.9 The sequential events of the 16 element comparison. The active cells are indicated with red color. Black color pinpoints cells that have completed the comparison. White ones have not yet started. In instance V, the MSB of Min is ready after 4 cells delay. . . 20

3.1 The cell which the 2_elements_comparator consists of. The C_1 and C_2 signals will be encoded to represent the values of *the choice*. Table at the right shows the possible values of C_1 and C_2 as a function of the bits of the element. Soon, we will see the dependence of $C_{1,i}$, $C_{2,i}$ on $C_{1,i-1}$, $C_{2,i-1}$. 24

3.2 The Boolean equations for the carries and the result. 28

3.3 a) The AND-OR implementation of the circuit. b) The AND-OR_INVERT implementation. . . . 28
3.4 Inversion of signals for the cell$_i$. . . . 29
3.5 Four-bit ripple-carry comparator: topology 30
3.6 Four-bit ripple-carry comparator: circuit implementation with inverse signals 31
3.7 Four-bit carry select comparator: topology. . . . 32
3.8 The cs-cell and the table recording the functionality. . . . 32
3.9 The Boolean equations for the carries. . . . 32
3.10 The cs-cell circuit. Inversion of signals is used to minimize the delay. . . . 34
3.11 a) The linear comparator. b) The square root comparator. . . . 35
3.12 The carry select comparator's configuration after the critical path delay optimization. 36
3.13 The carry look ahead implementation of the 2 element comparator. . . . 38
3.14 The optimized carry look ahead implementation of the 2 element comparator for k=4. 38
3.15 The ripple, carry Look ahead and carry select delays for 2-element comparator. . . . 39
3.16 Propagation delay for the three 2-element comparator structures in the binary tree. . 41
3.17 Propagation delays of the binary tree structure with carry select comparators. . . . 42
3.18 A 5-level binary tree with carry select comparators. Only one comparator per level is shown for simplicity. The hybrid comparators are shown. The delays(with blue) of the signals are matched. . . . 44
3.19 A 3-level binary tree with carry look ahead comparators.Only one comparator per level is shown for simplicity. . . . 46

4.1 Managing wrap around. The elements values can be only one color. . . . 50
4.2 The top block of the N-flow WRR scheduler. . . . 50
4.3 Forwards in the pipeline. . . . 51
4.4 The datapath of the WWR scheduler: a first approach. . . . 53
4.5 An example of locating the address of the winner element. . . . 54
4.6 The datapath of the WWR scheduler 58
4.7 An example of reducing the fanout of the root. 2-element comparators from Level-1 to Level-3 have fan-out 2. . . . 59

4.8 The power consumption after updating one element. The active 2-element comparators at this cycle are highlighted. 61

5.1 The general design flow followed 63

5.2 Delay comparison of 2-element comparator circuit. 64

5.3 Delay comparison for binary tree with 8-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 65

5.4 Delay comparison for binary tree with 16-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 66

5.5 Delay comparison for binary tree with 24-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 66

5.6 Area comparison for binary tree with 8-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 67

5.7 Area comparison for binary tree with 16-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 68

5.8 Area comparison for binary tree with 24-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 68

5.9 Power comparison for binary tree with 8-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 69

5.10 Power comparison for binary tree with 16-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 70

5.11 Power comparison for binary tree with 24-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select. 70

List of Tables

3.1 Operation of 2-element comparator. 23

3.2 The truth table of output m_i and *the choice*. 25

3.3 The coding used for the carries of the cell. The number at the top left corner is the serial number of the coding. 26

3.4 The truth table of inputs/outputs based on the coding that is used for the carries of the cell. 27

3.5 The Karnaugh maps for the carries of *the choice*. 28

3.6 The truth table of inputs/outputs of the carry select cell. 33

3.7 The Karnaugh maps for the carries of *the choice*. 34

4.1 The possible states for a flow in the scheduler. 55

4.2 Apply the out_flag to the initial condidtion of the 2-element comparator. 55

Chapter 1

Scheduling for QoS in Advanced Networks

The Internet connects tens of millions of computers around the world, allowing them to exchange messages and share resources. It is a loose collection of networks organized into a multilevel hierarchy using a wide variety of interconnection technologies. At the lowest level, ten and hundred computers may be connected to each other, and to a *router*, by al local area network (LAN) or by a modem. Computer networks allow users to share resources such as printers, file systems, long distance trunks, and sites on the World Wide Web. Sharing, however, automatically introduces the problem of *contention* for the shared resource. Given a set of resource requests in the service queue, a server uses a *scheduling discipline* to decide which request to serve next. Scheduling disciplines are important because they are the key to fairly sharing network resources and to providing performance-critical applications, such as telephony and Video On Demand.

Most experts agree that future networks will carry at least two types of applications. Some applications are relatively insensitive to the performance they receive from the network (e.g. FTP). The performance requirements of such applications are *elastic*: they can *adapt* to the resources available. Such applications are called *best-effort applications*, because the network promises them only to attempt to deliver their packets, without guaranteeing them any particular performance bound. Besides best_effort applications networks are expected to carry traffic from applications that *do* require a bound on performance(e.g. Voice). Those applications demand a *guarantee* of service quality from the network. *Guaranteed-service* applications require the network to serve resources on their behalf. This feature is called a *Quality of Service (QoS)*.

The performance received by these applications depends primarily on the scheduling

discipline present at each multiplexed server along the connection's path from a source to the destination. These servers are typically the ones scheduling packets at each output link in switches or routers. The provision of QoS guarantees requires isolation among the multiple traffic flows, for each one to get its own behaviour characteristics and service level. A prerequisite for good isolation is that traffic belonging to different flows be placed on different queues i.e. *per-flow queueing*. At each output queue, a server uses a scheduling discipline to choose which ready packet to transmit next, and control access to output queue buffers. The server can allocate different *mean delays* to different connections by its choice to service order. It can allocate different *bandwidths* to connections by serving at least a certain number of packets from a particular connection in a given time interval. Finally, it can allocate different *loss rates* to connections by giving them more or fewer buffers. Thus, to build a network that performs QoS, scheduling disciplines are required to support per-flow delay, bandwidth and loss bounds.

A scheduling discipline must satisfy four sometimes-contradictory requirements

- Ease of implementation (for both guaranteed-service and best-effort connections)
- Fairness and protection (for best-effort connections)
- Performance bounds (for guaranteed-service connections)
- Ease and efficiency of admission control (for guaranteed-service connections)

Each scheduling discipline makes a different trade-off among these requirements. Depending on the situation, some of these requirements may be more important than others. The "best" choice, therefore, depends on the applicable binding constraints. There are four principal degrees of freedom

- The number of priority levels
- Whether each level is work-conserving or non-work-conserving
- The degree of aggregation of connections within a level
- Service order (scheduling policy) within a level.

In a high speed network, a server's scheduler may need to pick the next packet for transmission every time a packet departs, which can be once every few nanoseconds. Thus, it has a very little time to make a decision. A scheduling discipline for such a

network should require inexpensive and easy implementation in terms of hardware instead of software. With modern VLSI technology the implementation of a complicated scheduling algorithm is possible. The bounding constraint instead is mainly the memory required to maintain *scheduling state* (such as pointers to packet queues, and memory about the service already received by a flow) and the time required to access this state. These problems are discussed in [IK01].

1.1 Scheduling Disciplines

As mentioned earlier a scheduling discipline must satisfy four sometimes-contradictory requirements. Those are analyzed in the following paragraphs [Kes97].

In a high speed network, a server's scheduler may need to pick the next packet for transmission every time a packet departs, which can be once every few nanoseconds. Thus, it has a very little time to make a decision. A scheduling discipline for such a network should require inexpensive and easy implementation in terms of hardware instead of software. With modern VLSI technology the implementation of a complicated scheduling algorithm is possible. The binding constraint instead is mainly the memory required to maintain *scheduling state* (such as pointers to packet queues, and memory about the service already received by a flow) and the time required to access this state. These problems are discussed in [KKVK97] and [NK01].

A scheduling discipline allocates share of the link capacity and output queue buffers to each connection it serves. An allocation at a switch is called *fair* if it satisfies the *max-min fair share* criterion. Fairness is an intuitively desirable property of a scheduling discipline serving best-effort connections. For guaranteed-service connections, which should pay the network operator a fee in proportion to their resource usage, fairness is not a concern.

The third major requirement of a scheduling discipline is that it should allow a network operator to guarantee arbitrary per-connection performance bounds. An operator can guarantee performance bounds, for a connection only by reserving some network resources. The user agrees that its traffic will remain within certain bounds and the operator guarantees that the network will meet the connections performance requirements. To specify and guarantee performance requirements, the measurement of the connection's performance have to be more precise. It can be expressed either *deterministically* or *statistically*. The former is refers to explicit defined bounds for every packet (e.g. 10s end-to-end delay) and the latter to the probability for the set of the packets to meet the bounds(e.g. 0.99 of the total number of packets will meet the bound). Four common

performance parameters are widely used in the literature: *bandwidth, delay, delay-jitter* and *loss*.

A scheduling discipline should permit easy admission control. A switch controller should be able to decide, given the current set of connections, whether it is possible to meet a new connection's performance bounds without jeopardizing the performance of existing connections. Moreover the scheduling discipline should not lead to network underutilization.

1.2 Choices in designing disciplines

As mentioned earlier there are four principal degrees of freedom in designing a scheduling discipline [Kes97].

In a priority scheduling scheme, each connection is associated with a priority level. Priority allows a scheduler to give packets at higher priority level a lower mean queueing delay at the expense of packets at lower priority. A scheduler can have arbitrary priority levels according to the number of classes the network operator wants to support.

The necessity of per-flow queueing lead to the priority queue, one of the most fundamental data structures. It is the basis for a class of queue scheduling algorithms that are designed to provide a relatively simple method of supporting differentiated service classes. In classic priority queue, packets are first classified by the server and then placed into different priority queues.

A work-conserving scheduler is idle only when there is no packet awaiting service. In contrast, a non-work-conserving scheduler may be idle even if it has packets to serve. There are a number of arguments for and against the two schemes. Work-conserving scheduler maximizes the utilization of the output bandwidth but in some cases it can create jitter for congested flows. Non-work-conserving scheduler underutilizes the output bandwidth but the behaviour of the traffic is predictable and easier to manage.

Another degree of freedom in the design of scheduling disciplines is the degree to which individual connections are aggregated in deciding their service order. Usually, the scheduler has a small set of classes and every class has a large set of flows. It provides different qualities of service to different classes, while flows within the same class share the same service quality. The latter scheme can change introducing subclasses in the same class performing hierarchical QoS. The latter scheme provides protection to the flows from each other in a class.

1.3 Approaches to Service Schedules

Assume N flows (or queues) served by a scheduler with N priority levels [Kat01]. Each flow (or queue) is associated with one priority.

The first and simplest discipline is to use *strict priorities*. The scheduler serves the eligible flow (queue) with the highest priority. If this flow (queue) is ineligible is serves the directly lower priority flow (queue),and so on. A starvation issue is associated with static priorities. If the priority $i \in N$ is not policed or regulated and becomes "persistent" (i.e. always has a non-empty, eligible queue), then all levels below i will be starved. The solution is to ensure that all levels but the last one are policed or regulated. The implementation is feasible with a priority enforcer/encoder chain of elements with a ripple signal. To speed-up the ripple signal, analogous ideas to carry look ahead or carry select can be used. A tree of OR-gates detects the presence of eligible entries among N in time $\log_2$N.

The opposite approach is to use *round robin (RR)* disciplines. The scheduler now does not always start the iteration from the flow with the highest priority as previously. Instead, the flow with the "highest priority" is different at every iteration in a round robin series. An issue of inserting the new eligible flow is raised with this approach which is discussed in [Kat01]. Various implementations are proposed for this algorithm: i) The first one uses a programmable priority encoder/enforcer which is programmed to begin searching for the first eligible flow (or queue) from different starting point. ii) The second uses two static priority circuits removing the programmability of the previous implementation [GM99].

One middle discipline-compared to the two formers- is the *Weighted Round Robin (WRR)* service schedule. Every flow (or queue) is associated with a weight, which is translated in two terms: *Flow Service Interval (FSI)* and *Next Service Time (NST)*. The former is inverse proportional to the weight associated with the flow (or queue) and the latter is the time (measured in internal system units) that the flow (or queue) is scheduled to be served in the future. Flows (or queues) are receiving performance by the network according to their weight, in terms of bandwidth, latency and loss. No starvation issue is raised. In order to perform WRR, a priority queue data structure is needed. The implementation of that structure is the issue this work is dealing with. Priority queues implementing WRR disciplines will be discussed in the next chapter.

Sorting has been a favourite subject in computer science research for the past few decades. Since the advent of VLSI technology, specialized sorting circuits have been pro-

posed and analyzed [Tho83]. However, the complete ordering of the given numerical elements is often not needed. Many practical applications, such as QoS scheduler, use a *priority queue* which only require determining the minimum (or maximum) value. The **problem discussed** in this report is the implementation of sophisticated scheduling algorithm at high speed, when there are many thousands of contending flows, using priority queues to perform QoS.

Priority queues with only few tens of entries or with priority numbers drawn from a small menu of allowable values are easy to implement e.g. by using combinational priority encoder circuits [KaSM97]. However, for priority queues with many thousand entries and with values drawn from a large set of allowable numbers, *heap* or *calendar queue* data structures must be used. The advantages of the former structure over the latter are presented in [IK01], which uses a heap tree, with a pipelined heap manager. The idea in this implementation is to keep the minimum (or maximum) element always at the top of the tree, in order to find it in O(1) time. One disadvantage of the *heap* is the difficulty in inserting and removing many elements from the tree. This cannot be done appropriately with the heap implementations: only one operation can be executed per cycle in the structure in one-cycle implementations. In contrary, one operation can be executed per level of the heap per cycle in pipeline implementations.

1.4 Contributions of This Work

The **approach proposed** in this work is to find the minimum (or maximum) in a set of arbitrary (non-sorted) elements using set of parallel comparators. Thus, the number of elements, which participate at every comparison, is easily changeable at every iteration of the algorithm. An element can participate or not according to a permission given by the scheduler(implemented in one bit).

The main **contribution** of this work is the introduction of a novel comparator organization, where the binary tree of comparators is such that signals propagate *in parallel* within the bits of each comparator *and* accross the levels of the tree. In this way, the delays of the individual comparators and the delays of the tree levels are placed in parallel, rather than in series. Those comparators are discussed in Section 3 . The binary tree of comparators is the heart of a weighted-round-robin scheduler that we designed. Our design is in synthesisable Verilog form. Synthesis results of the design show a clock period of 6ns, for a 256-flow scheduler, in a 0.18um CMOS ASIC technology.

The rest of the work is organized as follows. Section 2 talks about WRR and already

known implementations of priority queues, in hardware. Section 2.4 discusses two algorithms for finding the minimum (or maximum) element in a set of arbitrary (non-sorted) elements. The first one is already known, while the second algorithm was proposed by Manolis Katevenis at the begining of this work. In Section 3 , the comparison operation is studied, and three implementations for binary trees of comparators are examined in detail: ripple-carry comparators as proposed by Manolis Katevenis, carry select, and carry look ahead comparators as proposed by the author. In Section 4 , the scheduler datapath using carry select comparators is designed and optimized. Finally, Section 5 presents the synthesis results for the binary tree and its delay, area, and power consumption figures.

Chapter 2

Priority Queue Implementations in Hardware

Previous sections explained the need for per-flow queueing in order to provide advanced QoS in future high speed networks. To be effective, per-flow queueing needs a good scheduler. Priorities is a first important mechanism; usually a few levels of priority suffice, so this mechanism is easy to implement. *Aggregation* (hierarchical scheduling) is a second mechanism: first choose among a number of flow aggregates, then choose a flow within the given aggregate. Some levels of the hierarchy contain few aggregates, while others may contain thousands of flows; this work concerns the latter levels. One of the hardest scheduling disciplines are those belonging to the WRR family.

2.1 Weighted Round Robin

Using per-flow queueing, incoming packets from different flows are stored in different queues. As mentioned earlier each flow is associated with a weight. Flows with large weight should receive better performance than those with small one. Flow Service Interval FSI_i of flow i is inverse proportional to the corresponding weight w_i. Examples of WRR are shown in Figure 2.1 .

The first example shows a quite smooth schedule minimizing bursty transmissions from the switch. The implementation of this scheme is difficult with priority circuits, because it is difficult to manage the eligibility bits. If circular linked lists are used, the re-insertion in multiple positions is difficult. If a set of eligible flows varies slowly, the schedule can be computed *off-line*. On the other hand, if the set varies fast (or weights change often), a

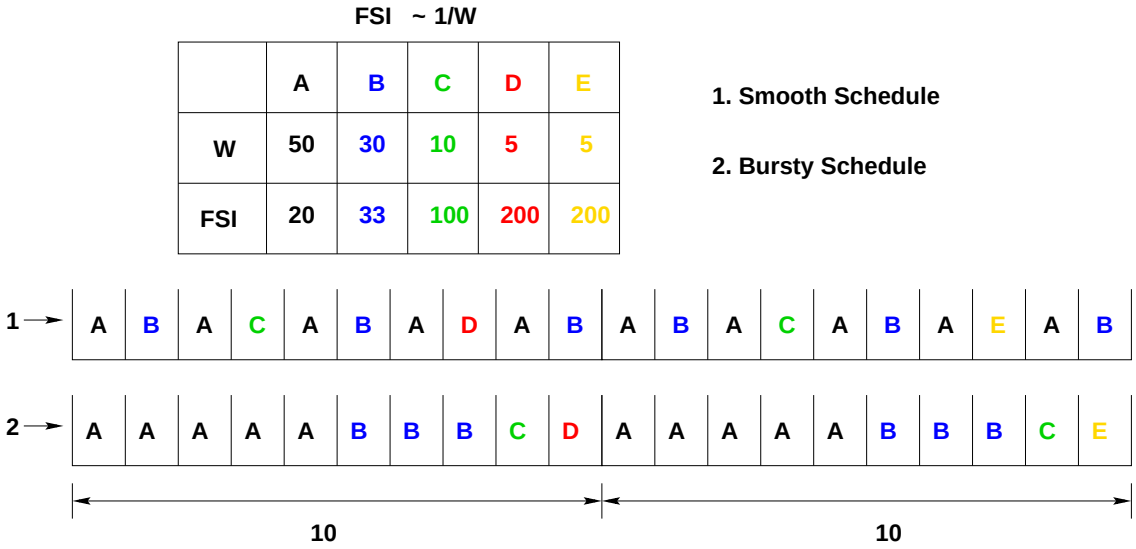


Figure 2.1: The WRR scheduling with 5 flows. There are two cases presented: 1. The smooth scheduling where the services are spread in time. 2. The bursty scheduling where the services are gathered.

priority queue is used to recompute schedule *on line*. The second example shows a bursty approach of the WRR service.The implementation is feasible with RR. In every visit to the flow, number of packets (bytes, words) are used according to flow's weight.

Priority queues in a WRR schedule are defined according to the next statements.

- Maintain a varying set of eligible flow
- Associate NST with each of them
- Find and serve the (eligible) flow that has the *minimum* (earliest) NST
- Reschedule for future time the flow served

The issue of real or virtual time in priority queues arises, associated with work-conserving or non-work-conserving disciplines. A non-work-conserving scheduler defines a real time. The NST of the flow is measured in terms of the latter. Scheduler can be idle for a period of time, if no flow is rescheduled to be served at this period. In contrast to that, a work-conserving scheduler defines a "virtual time". After serving one flow, the pointer of virtual time "jumps" to the next minimum NST in order to serve the corresponding flow immediately. An example is shown in Figure 2.2 . This work assumes work-conserving scheduling using virtual time clock.

Another matter of great importance is where in the time to reinsert a flow that becomes eligible. Assume that t_v is the virtual time and $t_{NST,i}$ is the NST of the flow i. There are two situations that will be discussed and drawn in Figure 2.3 .

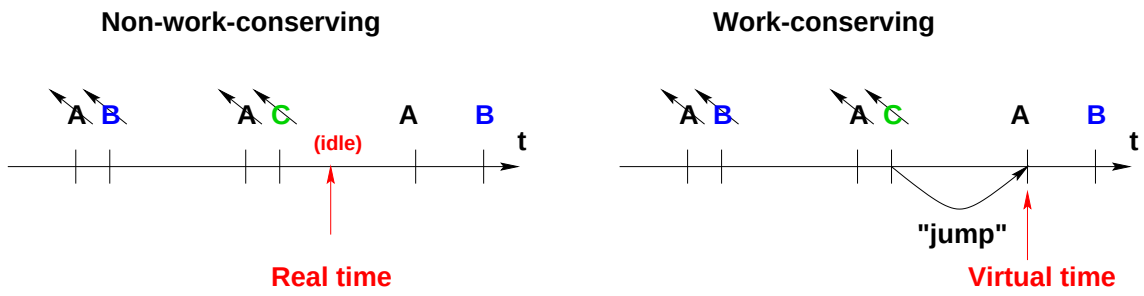


Figure 2.2: The work-conserving and the non-work-conserving options.

1. The virtual time (t_v) exceeds the NST of the reinserted flow i $t_{NST,i}$ ($t_v > t_{NST,i}$). Flow i must reinsert at the "time point" where it was scheduled to be served and **not earlier**. If the latter happens, flow i can take advantage over the others by being consecutively eligible and ineligible, and always reinserted into schedule before the $t_{NST,i}$.

2. The virtual time (t_v) is behind the NST of the reinserted flow i ($t_v < t_{NST,i}$). Flow i must reinsert at "current time" and **not earlier**. If the latter happens, it will be served against other flows.

Many algorithms are dealing with reinsertion and computation of NST discussed in [Zha95]. These issues account for the differences among the weighted fair queueing algorithm and its variants [Kes97, ch9].

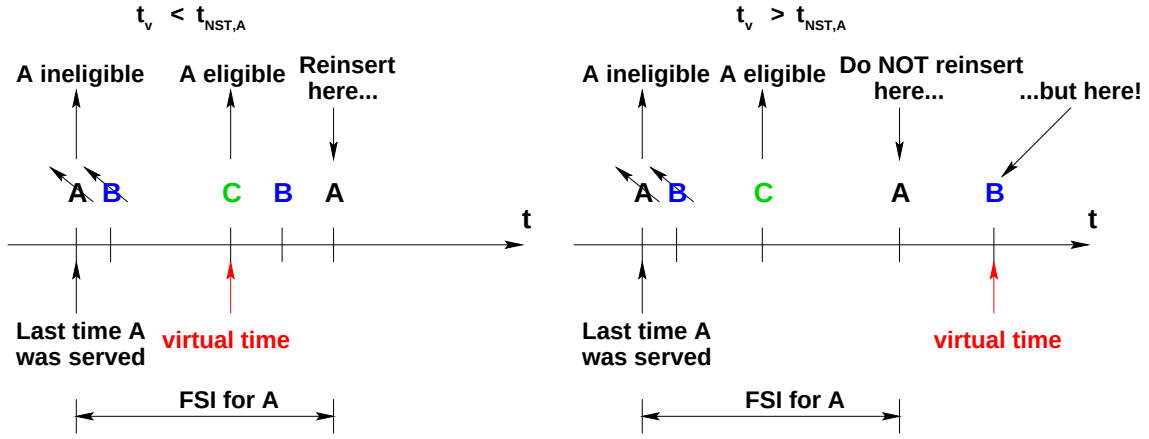


Figure 2.3: Reinsert eligible flows.

This is the time to present the WRR algorithm with an example, in order to understand the sequence of events. Figure 2.4 presents eight flows, A through H. Five of them are currently active (non-empty queues). The scheduler must serve the active flows in an order such that the service received by each active flow is proportional to its weight

factor. "Current" time is 150 and flow A is going to be scheduled for service, according to its FSI to "future" time 183. Also, the other four flows are scheduled to be served in "future" times: D at t=155, H at t=158, G at t=162 and B at t=170. The flow to be served earlier is the one that has the earliest, i.e. minimum scheduled NST. In this example, this is flow A. This flow remains active after its head packet is transmitted, so it has to be rescheduled. The FSI of A is 33 added to current "time" 150: $t_{NST,A}$= 150 + 33 = 183. Flow A will not be served again until the current "time" reaches 183. Continuing, "time" advances to 155 and flow D is served and rescheduled to be served again in "time" 205. If flow C will not receive any new packet, after "time" 662 it will be removed from schedule.

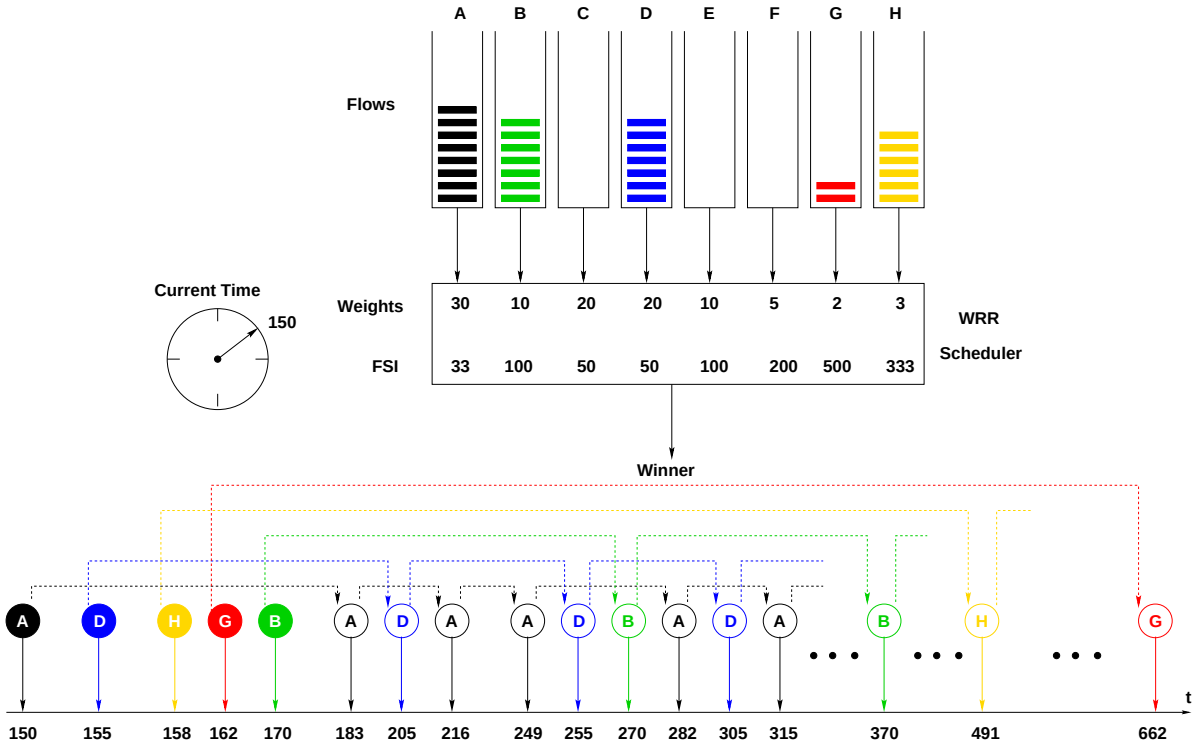


Figure 2.4: Example of WRR algorithm.

The scheduler operates by keeping track of a NST number of each active flow. In each step, the minimum of these numbers must be found and updated if the flow remains active or removed from schedule if the flow becomes inactive. When a new packet of an inactive flow arrives, that flow has to reinsert into the schedule.

2.2 Related Work in Priority Queue Algorithms implemented in Hardware

Priority queue can be implemented in hardware in many different ways. all of them are based in keeping track of a list of numbers. The simplest implementations of priority queues are discussed in [MRS00]. There, priority queues are used for *per-packet* queueing. The model of the switch considered is characterized by shared buffer space, and output queueing, with separate priority queue serving each output link. The per-packet queueing has a prohibitive cost in hardware for large number of packets N. The latter causes fan-in and fan-out problems to the structures introduced in that work (binary tree of comparators, FIFO priority and shift array). The last structure introduced (hybrid systolic/shift array) tries to solve most of those problems. This implementation is similar to ‘Aa *linked list*. Thus, a power issue is raised; at every insertion or deletion many elements have to move from their position in the list, which is a power-consuming task for large N. In addition, an Other approaches are the *heap* and the *calendar queue* data structures. Discussion about those data structures can be found n [IK01], where a pipelined heap manager is introduced. Every level of the tree is one stage of pipeline. The elements of the heap are stored in 2-port SRAMS and the operations performed at every stage of the pipeline are: read, compare and write. The design scales well with the number of elements. Insertions to the structure are performed from the root element. Two operations are implemented: *insert* and *delete*. Consecutive deletes have one stall cycle between them. In contrary to that, the scheduler presented in this work can insert and delete as many flows as necessary, in the same cycle.

The above mentioned implementations are using data structures to keep the elements sorted. Instead, this work deals with algorithms for the minimum value determination in a non-sorted set of elements, presented in Section 2.4 . Before that, the environment of the scheduler will be defined and the problem with the heap will be explained.

2.3 Cases where Heap Data Structures are inappropriate

A scheduler is a critical part of a switch. It is responsible for the order the switch serves flows and expresses the policy of the network administrator. The scheduler interacts with the Queue Manager. The scheduler receives inputs about the eligible/ineligible flows from

the Queue Manager, and responds with the next flow to be served by the queue manager. In addition, the scheduler receives backpressure information from the flow control. All these are expressed as *ready bits*. There is one ready bit for every flow. One bit suffices to describe the two states of a flow: eligible/ineligible.

The first approximation to the eligibility of a flow is the following: Every flow having a non-empty queue is eligible. A flow becomes ineligible if the connection ends or if all the packets in the queue are served. A closer look at the switch uncovers more situations where a flow may be ineligible. The first one has to do with a problem called memory interleaving. The physical memory where packets are stored is usually SDRAM. Header packets from different flows may be stored in the same SDRAM bank. However, consecutive accesses to the same bank need "wait" cycles, for the necessary precharging of the bank. Thus, flows having packets in the just accessed bank become ineligible for the next iteration of the scheduler.

A second reason to make flows ineligible is backpressure. The cause of backpreassure signals can be internal or external to the switch. Assume a buffered crossbar is used in the switch [CK02]. One scheduler is used at each input, which sends packets to the buffers at the crosspoints. If the buffers at one crosspoint are filled, backpressure signals are sent to every flow of that input having packets for that output (crosspoint). Those flows should be ineligible for that input scheduler. Furthermore, there are schedulers at each output. These accept many packets arriving simultaneously at the corresponding crosspoints (inputs to those schedulers). Thus, many flows may become eligible at the same time.

Next consider a switching fabric rather than a crossbar. Even though those elements are blocking or non-blocking, a link through them could be congested. Thus, backpressure signals must sent to flows having packets for this link. All of these flows must become ineligible at the same time. Another situation concerns backpressure in a switch level. A "hot-spot" could occur somewhere in the network, where many flows send packets there. This could cause backpressure signals form the "hot-spot" switch to previous switches. Those make many flows ineligible at the same time. As soon as, the problem is solved to the overloaded switch, the previously ineligible flows should become eligible again.

Thus, many flows may change state at the same cell time. As a result, many flows may be inserted to or removed from the priority queue. The number of elements in the set (which actually is the priority queue) may vary fast. Thus, it is difficult for the heap to keep track with fast changes in the number of its elements. The solution to that problem is a *brute force* approach: use a circuit capable to quickly find the minimum in an arbitrary

(non-sorted) set of elements. In addition, simultaneously insertions and deletions should be performed easily. An already proposed solution and the one proposed in this work are discussed in the next section.

2.4 Algorithms for Minimum Value Determination in a non-sorted Set of Elements

The problem of minimum determination is formally defined as follows. Given a set of N numerical elements n_1,n_2,...n_N, it is desired to produce a number $M \in N$ such that $M \geq n_i$ ($M \leq n_i$) ($N \geq i \geq 1$). The following discussion assumes that each numerical element n_i is represented by k-bit unsigned binary number ($n_{i,k-1}$,$n_{i,k-2}$,$n_{i,k-3}$,...$n_{i,0}$). Some **already known solutions** are presented and discussed in the section *array and row architecture* and **our contribution** is presented in the *binary tree architecture.*

2.4.1 Array and Row Architectures

The *array architecture* proposed in [VM93] determines the *maximum* value of N numerical elements. With a small change, the same circuit can determine the *minimum* value. The circuit performing the latter operation is shown in Figure 2.5 . This architecture is organized as a k-by-N array of cells. Each cell is labelled by its row and column indexes according to its position in the array. The m inputs in column (i) of this array provide the bits of an element. The N modules in a row (j) construct a bit slice of the N elements. Every disable signal (*the choice*) in the first row (j = *k - 1*) is set to zero, which means that all the elements can participate in the comparison. Each bit slice performs parallel operations to determine one bit M(j) of the minimum value. Enable signals of bit slice j announce *the choice* to the bit slice *j-1*. The elements that do not match with the output at the current bit position are removed from further comparison, by setting the correspondence disable signal to ace. Afterwards, the next bit slice starts the comparison.

At the end of the process, if n_l is the minimum of the set $n_1, n_2, \ldots n_N$, *the choice* (final) will be represented with 1 zero and N-1 aces. The 1 zero will indicate the originating column of the minimum in l position. In other words, this ace (or aces) indicates the winner (or winners) element of the comparison. The **invariant condition** that must be satisfied for disable signals is the following: the j bit slice can change the bits of *the choice* only from zero to ace and not the opposite way. No change from ace to zero is permitted, because previous bit slices decided to extract the element, by setting the disable

signal to ace. The disable signals are ripple, thus the signals M(j) are not ready until the disable(j) signals are ready. The M signals are stabilized sequentially, from the Most Significant Bit to the Least Significant Bit.

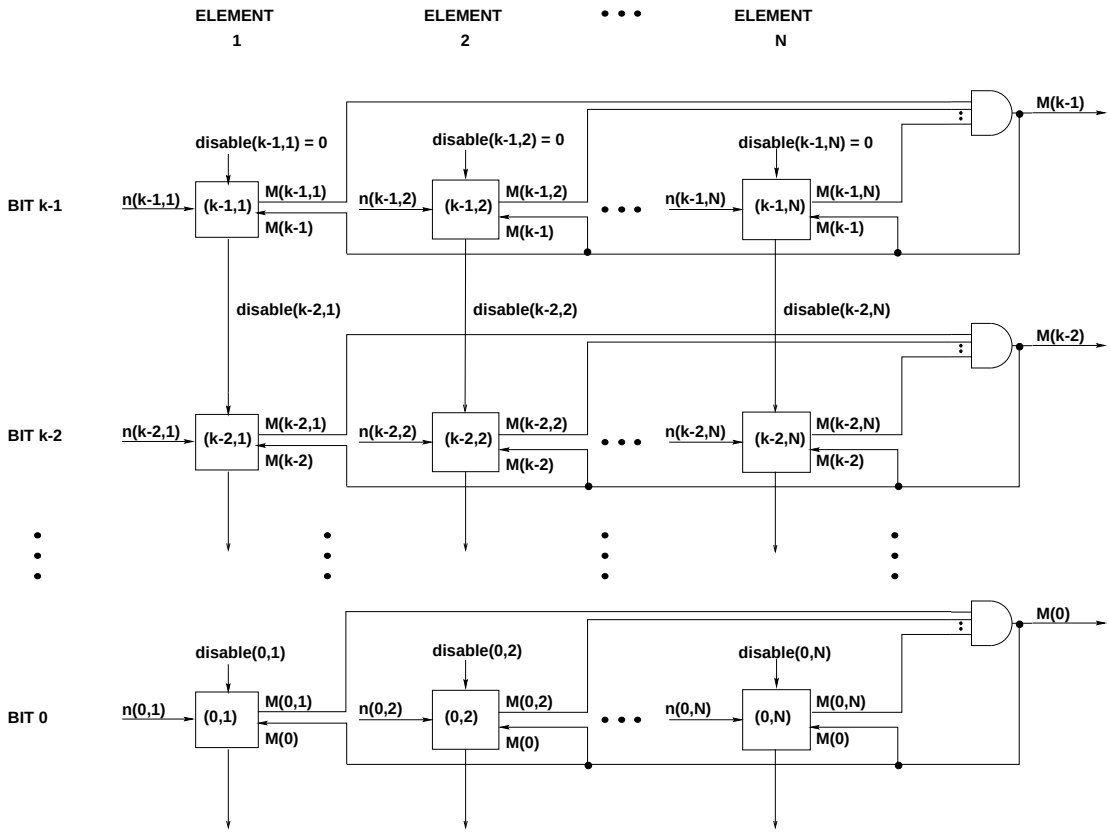
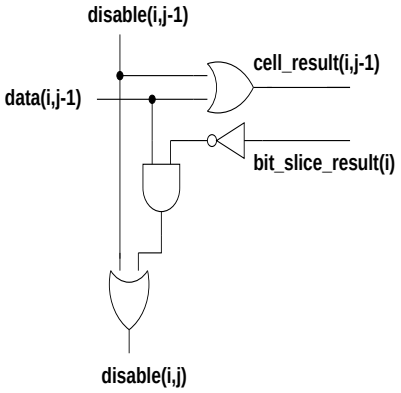


Figure 2.5: The array architecture for the comparison of N elements, k-bit each. Each bit slice (row) compares N bits and gives one bit of the winner M. The columns are the bits of each element n_i.

The architecture of the cell is shown in Figure 2.6 . The disable signal permits the data to evaluate the output M of the N-input AND gate of each bit slice. This M signal returns back to the cell, in order to evaluate the disable signal for the next bit slice.

The circuit has two major disadvantages, which may lead the promising architecture to poor delay performance. The first one is the large fan-in of the N-input AND gate. The AND gate takes as input the result of every cell at each bit slice, resulting to a fan-in proportional to the number of elements. The other disadvantage is the fan-out of that gate. The AND gate drives back every cell at each bit slice, giving a fan-out proportional to the number of elements (as in fan-in). This significant delay is in the critical path and becomes the bottleneck to the design performance. The computational complexity of this algorithm is not O(k) as mentioned in [VM93] but depends on the number of the compared elements. The dependency is generated by the large fan-in-fan-out AND



Inputs			Outputs	
$disable_{j-1}$	bit slice $result_{j-1}$	$data_{j-1}$	cell $result_j$	$disable_j$
0	0	0	0	0
0	0	1	1	1
0	1	0	x	x
0	1	1	1	0
1	0	0	1	1
1	0	1	1	1
1	1	0	1	1
1	1	1	1	1

Figure 2.6: (a) The architecture of the cell that enables the element to participate in the comparison and propagates the disable signal to the next lower significant bit. (b) The coding of the cell.

gate. The implementation of that gate will determine the computational complexity of the algorithm. One possible implementation is a tree of AND gates, which gives $O(\log_2 N)$ complexity. An increment $O(\log_2 N)$ to the complexity is added by the large fan-out of the AND gate according to the next solution. A balanced tree of buffers with gradually increasing load is added to the output of such an AND gate. The calculation, that is M and the choice (final), is finished in one cycle time. The area and the consumed power are increasing linearly with N and m for the cells and logarithmically for the AND gates.

The *row architecture* based to the previous paper proposed in [VR94]. It uses one bit slice from array architecture and some storage elements. The circuit is shown in Figure 2.7 . The comparison of N elements can be done in m cycles. The comparison starts with the most significant bits of every element at the first cycle. The most significant bit of the result M, appears at the output of the AND gate. At the next cycle, it will be stored in an m-bit left shift register, while the next most significant bit appears at the output of the AND gate. After m+1 cycles the minimum of N elements will be available in the m-bit left shift register and the 1 zero at the output of the D flip-flops will indicate the origin of the minimum. Like the array architecture, the computational complexity of the procedure presented is dependent on the number of elements. The complexity of the algorithm is O(k). In contrast to the array architecture, the area depends only on the number of elements and not on the number of bits in the element. Thus, it is approximately k times smaller than that of the array architecture, plus the area of the D flip-flops and the k-bit shift register. Unfortunately, the consumption of power increases with respect to the array architecture. The number of calculations performed by the two circuits is the same but the row architecture must evaluate the sequential parts of the

circuit (register and flip-flops) in every cycle.

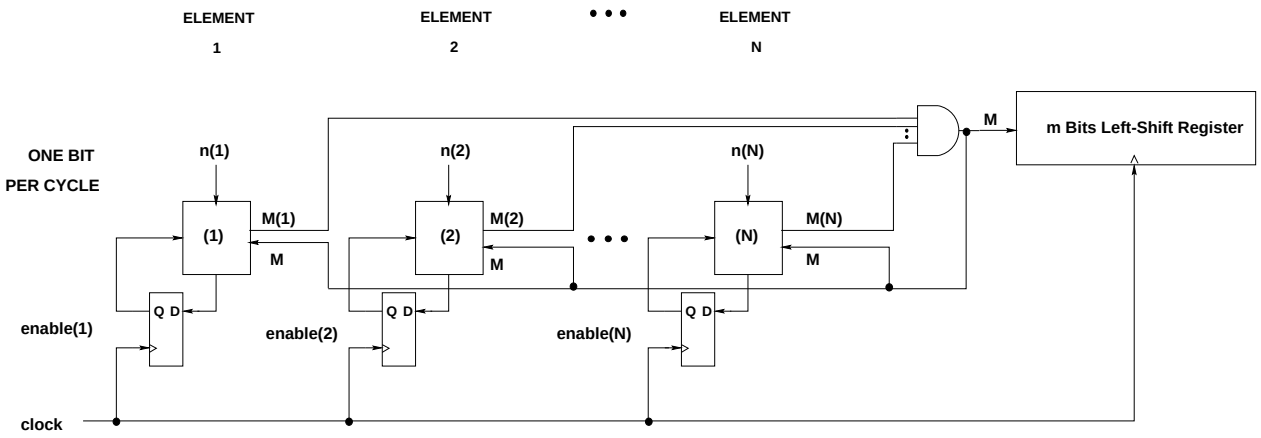


Figure 2.7: The row architecture for the comparison of N elements, k-bit each.

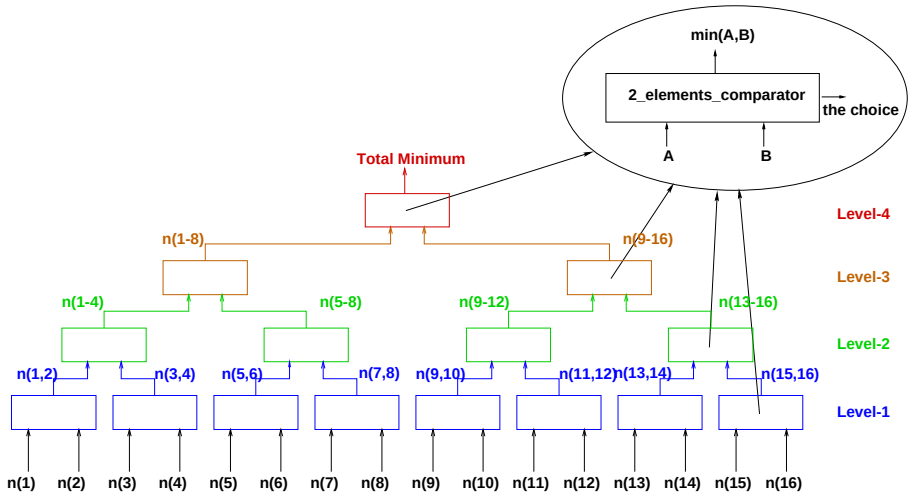
The disadvantages mentioned for the array architecture record to the row architecture. The large fan-in-fan-out AND gate remains and becomes the bottleneck to the delay of the circuit.

2.4.2 Traditional Binary Tree Architecture

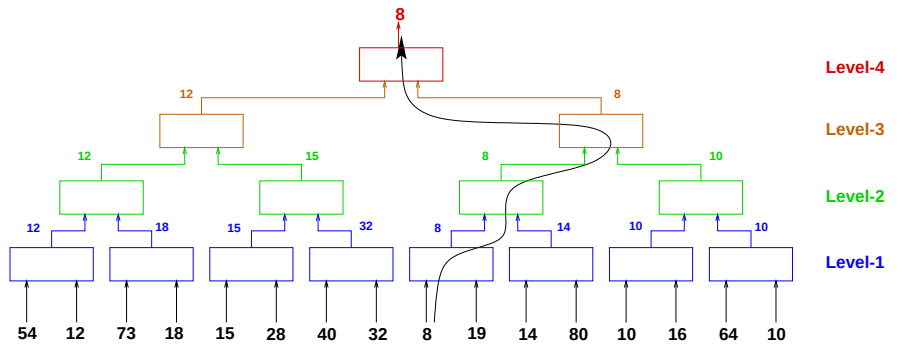
In order to overcome the significant delay of the previous algorithms, first we can use a *binary tree of comparators*, as shown in Figure 2.8 . Each node is a 2-element comparator. Elements are presented to the tree leaves and are compared in pairs in parallel. Assume that in O(1) times we can find the minimum (maximum) of two elements. Thus, the bottom level performs N/2 calculations in O(1) time. The result of each calculation is presented at the outputs of the 2- element comparator. Subsequently, the next upwards tree level performs N/4 calculations in parallel. It compares in pairs the results of the lower level, in O(1) time. The previous process continues sequentially in all levels, until the total minimum (maximum) is calculated by the root comparator. The root has always the minimum (maximum) of the set of N elements. The fact that the two elements comparator forwards one of its inputs to the output, gives the opportunity to an element to travel through the nodes of the tree towards the root. Its fate is to win at every comparison and become the total minimum (maximum) or to be defeated at a certain node by another element. The loser's path stops at that node.

The implementation for N = 16 is shown at Figure 2.8 . The block that appears in every node of the binary tree is called *2_element_comparator*. The function that executes is the min(A, B) (or max(A,B)). It presents at the output the winner, which is one of

the two elements (A, B) lying at the inputs. The other output of the block is one bit that indicates which of the two input elements is the winner. This output will be used to denote the origin of the final winner element. The elements are inserted at the leaves of the tree (Level-1). All the 2_element_comparator blocks at Level-1 start calculating for the minimum (maximum) simultaneously.



(a)



(b)

Figure 2.8: (a) The structure of the circuit is a binary tree with 8 leaves (N = 16). Every block is a 2 element comparator. The delay of the circuit is O(logN) = 4. The advantage of this scheme is that comparisons of different elements are taking place in parallel. (b) Arithmetic example for the determination of the total minimum.

As soon as all the Level-1 2_element_comparator blocks finish their calculation, the winners are present at the corresponding outputs. For example, the winner n(1,2), which is one of (n1, n2) is moving one level up to the tree. There it is compared with the winner n(3,4), which is one of (n3, n4). The procedure continues with Level-2 2_element_comparators taking the torch from those at Level-1 to execute the comparison. A numerical example is presented in Figure 2.8 for the determination of the total

minimum. The winner element (8) travels from the leaves through the nodes, indicated by the curve, to the root. The complexity of calculation of the proposed binary tree algorithm is equal to the height of the corresponding binary tree. Thus, for a set of N numbers, it results to O (log2N). The complexity of this architecture, like the previous one, depends logarithmically on the number of elements. The area and power are also logarithmic functions of N.

The binary tree forces N to be power of two. If N = 2k, then 2k-1 comparators must be used. In fact, all the values of N can be used by performing the calculation with a non-balanced binary tree. The delay in such architecture is dominated by the longest path, which resides at the heavy sub-tree. For example if N = 129, we can use 127 comparators in a binary tree calculating the minimum (maximum) among 128 elements at $O(\log_2 128) = 7$. The result of the previous sequence is compared with the remaining element, adding one more level of delay resulting to N=128 comparators. This scheme is actually a non-balanced binary tree with 8 levels. The overall delay is $O(\log_2 N) = 8$, where $N = 2^8 = 256$. Thus we could compare N elements, tolerating the same delay, with N$\in$ {129, 130 ...256}, by just adding more 2-element comparators at the light sub-tree. This is a general conclusion to which all the sets $\{N_1, N_2 \ldots N_n\}$, $2^k < N_i \leq 2^{k+1}$are being subjected. The cost of area increases linearly as the number of elements grows. The evolution of technology overcomes this problem, making the delay to be the bottleneck rather than the area of a circuit.

2.4.3 Parallel Operation of multiple Tree Levels

The inovation of this work is to start the comparison at Level-(i+1), before the end of the comparison at Level-(i). The above is possible by starting the comparison of two elements from the most significant bits (MSBs), sequentially comparing all the bits, one-by-one. This approach to the comparison operation is different from the traditional implemented in comparator circuits; based on the subtraction operation, traditional comparators ripples the carries form the LSBs to the MSBs. Continuing with our algorithm, as soon as the bits from the MSBs comparison at Level-(i) are generated, are given to Level-(i+1). Thus, the latter starts the comparison before the end of the former. The procedure continues as described and ends as soon as the calculation reaches the least significant bit (LSB) of the last Level. The sequence of events and the propagation of the results in the binary tree is described in the next paragraph, in order to calculate the delay of the circuit.

Assume that we have N=2k elements, each one having k bits (n_{k-1},... n_1,n_0). In ad-

dition, it is assumed that all 2_element_comparators at Level-0 (the leaves of the binary tree) start the comparison simultaneously, because all the elements are available at the same time. R is the result of every 2_element_comparator.

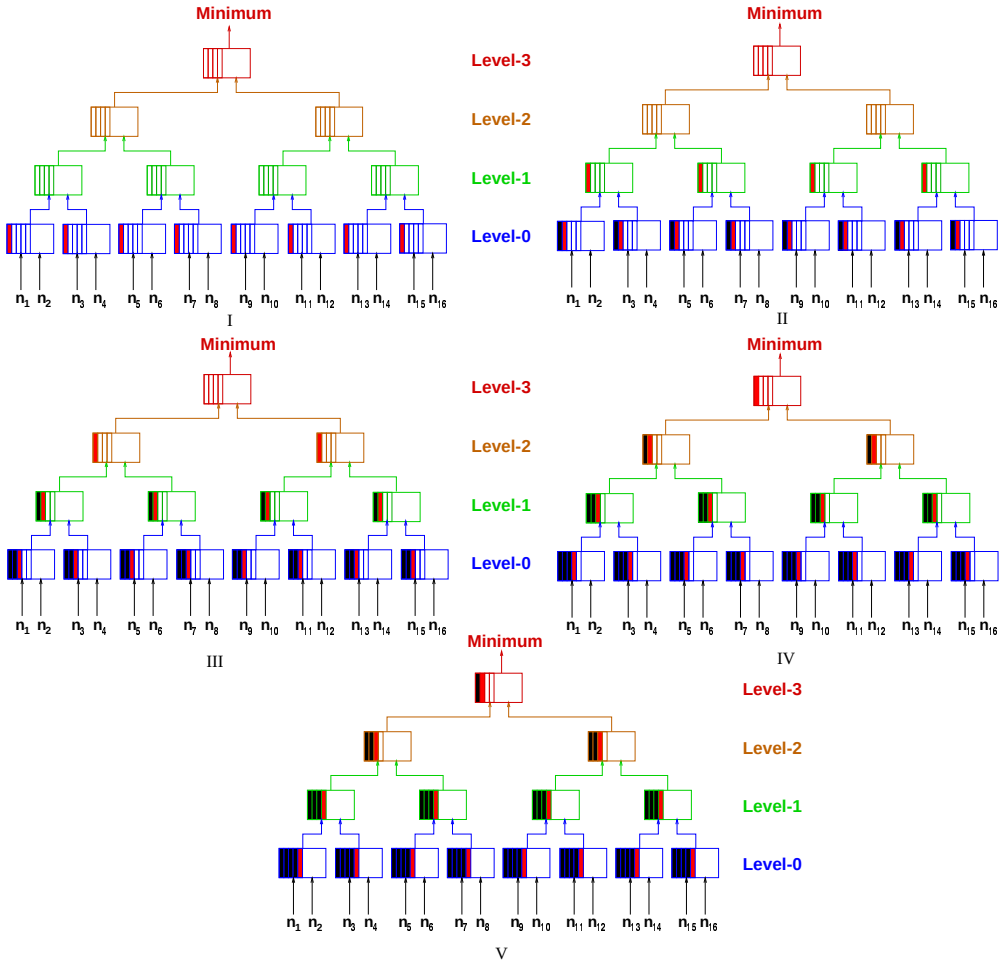


Figure 2.9: The sequential events of the 16 element comparison. The active cells are indicated with red color. Black color pinpoints cells that have completed the comparison. White ones have not yet started. In instance V, the MSB of Min is ready after 4 cells delay.

Every bullet describes events taking place in parallel, while sequential bullets describe sequential events.

- The 2_element_comparators at Level-(0) generate the $R_{k-1,0}$ bits (the second index shows the level that generates the bit) , which are the result of the comparison by pairs of n_{k-1} 2-MSBs. These bits are propagated immediately to Level-(1) of the binary tree.

- The 2_element_comparators at Level-(0) start the comparison of the n_{k-2} bits. The resulting $R_{k-2,0}$ bits are propagated to Level-1 of the binary tree. The

2_element_comparators at Level-(1) start the comparison of $R_{k-1,0}$ bits, since the inputs are ready. The generated $R_{k-1,1}$ bits are propagated to Level-2 of the binary tree.

- The 2_element_comparators at Level-(0) start the comparison of the n_{k-3} bits. The resulting bit $R_{k-3,0}$ is propagated to Level-(1) of the binary tree. The 2_element_comparators at Level-1 start the comparison of the $R_{k-2,0}$ bits. The generated $R_{k-2,1}$ bits are propagated to Level-(2) of the binary tree. The 2_element_comparators at Level-(2) start the comparison of the $R_{k-1,1}$ bits. The resulting bits $R_{k-1,2}$ are propagated to Level-(3) of the binary tree.

- The sequence of events is terminated, as soon as the 2_element_comparator at Level n (the root of the tree) calculates the $R_{0,n}$ bit, which is the last bit of the winner element.(n=$\log_2$N)

Figure 2.9 shows the sequence described earlier for N = 16. In instance I (refers to the first bullet), the first cell of every Level-0 2_element_comparator, which is marked with red color compare the 2 MSBs bits. In instance II (refers to the second bullet), every Level-1 2_element_comparator compare the 2 MSBs and every Level-0 2_element_comparator have already proceeded to the next cell (e.g. comparing the next MSBs). In instance IV, it is observed that the Level-3 2_element_comparator has provided the MSB of minimum after 4 cells delay.

Chapter 3

Comparator Circuits and Trees

The speed of the arithmetic elements often dominates the overall system performance. A careful design optimization is required. It rapidly becomes obvious that the design task is not straightforward. There exist multiple equivalent logic and circuit topologies, each of which has its own benefits and disadvantages in terms of delay, power and area. The optimization at only one design level leads to inferior designs.

The basic unit of the binary tree is the 2_element_comparator block. This arithmetic block accepts two numbers as inputs and presents the minimum of them at the output. The operation executed by this block is similar to the *addition* operation. The addition operation is probably the mostly used arithmetic operation. Careful optimization of the adder is of outmost importance. This optimization can proceed either at the logic or circuit level. Typical logic-level optimizations try to rearrange the Boolean equations so that a faster or smaller circuit is obtained. There are multiple design implementations examined for the addition operation such as *carry look ahead* or *carry select*[Rab96]. Circuit optimizations, on the other hand, manipulate transistor sizes and circuit topology to optimize speed. The same ideas that led to the implementation of fast adders will be followed for the implementation of fast comparators.

At the following sections a short presentation of fundamentals of comparator circuit is provided, followed by *ripple*, *carry select* and *carry look ahead* implementation of the comparator circuit.

3.1 The Binary Comparator

3.1.1 Definitions

The 2-element comparator is sliced into basic cells that compares the 2-bits comparison. Assume two k-bit numbers A ($a_{k-1}a_{k-2}\ldots a_0$) and B ($b_{k-1}b_{k-2}\ldots b_0$) to be the inputs and one k-bit number M ($m_{k-1}m_{k-2}\ldots m_0$) to be the output of the 2-element comparator. According to the operation executed by the comparator block the corresponding number (A or B) will appear at the output. Table 3.1 records the possible operations and outputs.

The decision is taken once from the 2-bit comparator cell that first found unequal bits of the elements A and B. This decision and can no longer change from the next significance cells following in the comparison chain. The nature of the comparison operation forces the

Inputs	Operation	Outputs
A$\geq$ B	Maximum	M = A
A$\geq$ B	Minimum	M = B

Table 3.1: Operation of 2-element comparator.

calculation to start from the MSBs. Thus, the cell that operates the 2-MSBs comparison must advertise *the choice* to the cell that compares the two next significance bits. *The choice* is the analogous to the carry of the addition operation. Three states can occur for *the choice*: "A equals B", "A is the minor", "B is the minor". Three states can be coded with two bits. Thus, *the choice* consists of two signals C_1 and C_2.

One **invariant** condition, which must be satisfied across the 2-element comparator, is the following:

2-bit comparator cells after the one that took the decision on behalf of A or B, cannot further change the value of the choice.

3.1.2 The 2 Bit Comparator Cell

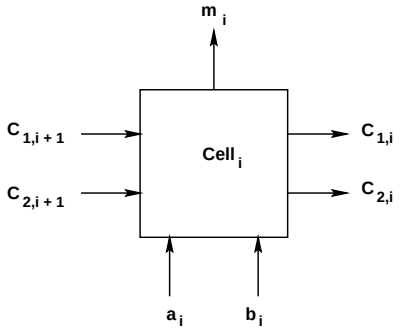
As mentioned earlier the basic block of the 2-element comparator is the cell that operates the 2-bit comparison. A k-bit 2-element comparator is consisted of k such cells. Every cell executes the same operation that includes two steps:

1. It produces *the choice* to be advertised to the cell that compares the two next sig-

nificance bits, including the information from *the choice* calculated by the previous cell.

2. It produces the m_i bit of the result M, which is one of the current a_i or b_i bits (of A or B) and presents it at the output, using the value of *the choice* produced by the current cell.

The cell that operates the 2-bit comparison and the possible values of the choice are shown in Figure 3.1 . In table in Figure 3.1 , the contribution of the previous value of *the choice*



a_i	b_i	*the choice*
0	0	A equals B
0	1	A is the minor/B is the major
1	0	A is the major/B is the minor
1	1	A equals B

Figure 3.1: The cell which the 2_elements_comparator consists of. The C_1 and C_2 signals will be encoded to represent the values of *the choice*. Table at the right shows the possible values of C_1 and C_2 as a function of the bits of the element. Soon, we will see the dependence of $C_{1,i}$, $C_{2,i}$ on $C_{1,i-1}$, $C_{2,i-1}$.

to the determination of current value was not taken into account. The *initial condition* of the signals $C_{1,k}$ and $C_{2,k}$ is "A equals B". Those are preserved and propagated along the comparator's sequential cells, as long as the condition $a_i = b_i$ is valid for every pair of bits at the inputs. As soon as the initial condition becomes invalid ($a_i \neq b_i$), it is changed and the new value of *the choice* is one of the others presented in table in Figure 3.1 . According to the **invariant** condition, this new value of *the choice* can no longer be changed by the remaining next significance cells. It is preserved along the 2-element comparator, until the end of comparison. Table 3.2 shows the relation between the inputs and outputs of $cell_i$ as evident of the above analysis. The first 2 rows of Table 3.2 specify the state, in which one of the previous cells has taken the decision and changed the initial values of C_1 and C_2. Those values are preserved and propagated through the cell unchanged. The initial condition "A equals B" arriving at the inputs of the current cell in the four last rows of Table 3.2 can change if and only if $a_i \neq b_i$.

3.1.3 Coding

There is another issue to deal with before the specifications of the cell are defined precisely and the design become feasible. The coding of the signals of *the choice* must be done

Inputs			Outputs	
The choice$_{i+1}$	a_{i+1}	b_{i+1}	*The choice*$_i$	m_i
A is the minor/major	x	x	A is the minor/major	a_i
B is the minor/major	x	x	B is the minor/major	b_i
A equals B	0	0	A equals B	0
A equals B	0	1	A is the minor/B is the major	0/1
A equals B	1	0	A is the major/B is the minor	0/1
A equals B	1	1	A equals B	1

Table 3.2: The truth table of output m_i and *the choice*.

carefully, because it influences the performance of the total design in terms of time, power and area. The necessity for coding arises because the possible states that may occur during the comparison of 2 elements are 3 and the total states described by 2 bits are 4. The remaining state will be "don't care". This is a problem of 4 discrete objects to be placed in 4 positions. Thus, if E is the number of different combinations, it is given by the formula

$$E = 4! = 1 \cdot 2 \cdot 3 \cdot 4 = 24$$

This is the total number of codings. Different codings produces different circuits, which must be examined in order to choose the one with the best features. The choice will be done with criteria the minimum delay, power consumption and minimum area.

The existence of **two symmetries** reduces the number of different circuits. The **first symmetry** is between the two input bits of cell a_i and b_i. The mutual transference of the two input bits gives the same circuit for every coding. This results to a total number of circuits equal to 12. The **second symmetry** is between the two carries C_1 and C_2. For every coding, the mutual transference of the two carries gives the same circuit. Two out of 12 remaining codings are complementary and do not obey to that symmetry. In contrast to that, the other 10 are subject to the latter. Consequently, the number of different circuits is reduced to 7.

Generally, the information propagated by C_1 (or C_2) depends on the previous values of both carries C_1 and C_2. This could make the values of both carries $C_{1,i+1}$ and $C_{2,i+1}$ requisite to the generation of the new $C_{1,i}$ (or $C_{2,i}$). Some codings partially question the previous acceptance by performing the calculation of $C_{1,i}$ using only the value of $C_{1,i+1}$. Further reduction is achieved with the assistance of an observation about the values of *the choice* that selects A and B to be the winner. Faster circuits are produced if those

values are not complementary. Thus, the value of *the choice* that selects A must not be complementary to the value that selects B.

There are two families constructed of the 7 remaining circuits: the *first* uses complementary signals for the value of *the choice*, after decision for the winner is taken. The *first* family's Boolean equation for C_2 is produced by Boolean equation for C_1, if indexes of the variables are mutually transferred ($1 \leftrightarrow 2$). Thus, the circuit that calculates C_2 is identical to the circuit that calculates C_1. In addition to that, the generation of the carry $C_{1,i}$ (for the i-th bit) uses both the values of $C_{1,i+1}$ and $C_{2,i+1}$ (of the (i+1)-bit). This is also the case for $C_{2,i}$. In simple words for the generation of s new carry, the values of *both* previous carries are needed. In contrast to that, the *second* family uses non-complementary values of *the choice* that selects A and B to be the winner. This feature gives an interesting result: the evaluation of $C_{1,i}$ (or $C_{2,i}$) depends only on the value of $C_{1,i+1}$ (or $C_{2,i+1}$). Therefore, if $C_{1,i}$ (or $C_{2,i}$) depends on $C_{1,i+1}$ (or $C_{2,i+2}$) then $C_{2,i}$ (or $C_{1,i}$) depends on both $C_{1,i+1}$ and $C_{2,i+1}$. It will be proved later that if the cells are cascaded the delay of both carries concludes to be 1 level of logic. Concluding, the second family produces faster circuits than the first one, because less amount of logic is used. In addition to that result, all codings that belonging to the second family produce the one circuit, which will be used for the implementation of the 2-bit comparator cell.

After the previous discussion, the coding chosen for the cell is shown in Table 3.3 . Useful observations can be registered for this coding. Initially, the renaming of the carries

2	a_i	b_i	*the choice*
	0	0	A equals B
	0	1	A is the minor/B is the major
	1	0	A is the major/B is the minor
	1	1	A equals B

Table 3.3: The coding used for the carries of the cell. The number at the top left corner is the serial number of the coding.

would be useful for the rest of the report. Based on the values of *the choice*, it is observed that C_2 is zero (0) in the case that the winner is not defined yet and one (1) in the case the winner is found. Thus, C_2 can be renamed to *found*. Carry C_1 is zero (0) when "A is the winner" and one (1) in the case that "B is the winner". Thus, C_1 indicates which is the winner and it can be renamed to *choose*.

3.1.4 The Circuit

The second family has provided the proper coding for the 2-bit comparator cell. This coding is applied to the cell$_i$ and the truth table for the inputs and the outputs is shown in Table 3.4 . The latter is used for Boolean optimization with the assistance of Karnaugh maps shown in Table 3.5 . Boolean equations are shown Figure 3.2 . It is often useful from an implementation perspective to define *choose*$_i$ and *found*$_i$ as functions of some intermediate signals gc_i (generate choose$_i$) and gf_i (generate found$_i$). If gc$_i$ = 1 choose$_i$ will be generated else the previous value will be propagated. If gf$_i$ =1, the found$_i$ will be generated, else the previous value will be propagated. The result m$_i$ is defined by

Inputs					Outputs		
The choice$_{i+1}$	choose$_{i+1}$	found$_{i+1}$	a$_{i+1}$	b$_{i+1}$	choose$_i$	found$_i$	*The choice*$_i$
A equals B	0	0	0	0	0	0	A equals B
	0	0	0	1	0	1	A is the minor/B is the major
	0	0	1	0	1	1	B is the minor/A is the major
	0	0	1	1	0	0	A equals B
A is the minor/ B is the major	0	1	0	0	0	1	A is the minor/ B is the major
	0	1	0	1	0	1	
	0	1	1	0	0	1	
	0	1	1	1	0	1	
don't care	1	0	0	0	x	x	don't care
	1	1	0	1	x	x	
	1	1	1	0	x	x	
	1	0	1	1	x	x	
B is the minor/ A is the major	0	1	0	0	1	1	B is the minor/ A is the major
	0	1	0	1	1	1	
	0	1	1	0	1	1	
	0	1	1	1	1	1	

Table 3.4: The truth table of inputs/outputs based on the coding that is used for the carries of the cell.

equations according to the functionality of the cell (minimum or maximum finding). The polarity of *choose*$_i$ can be used, the equation stands for the finding of either minimum or maximum. Notice that gc_i and gf_i are only functions of a$_i$ and b$_i$ and do not depend on the carries. The circuits for the implementation of cell$_i$, which are based on the optimized Boolean equations, are shown in Figure 3.3 .

$choose_i$				
	00	**01**	**11**	**10**
00	0	0	0	1
01	0	0	0	0
11	1	1	1	1
10	x	x	x	x

$found_i$				
	00	**01**	**11**	**10**
00	0	1	0	1
01	1	1	1	1
11	1	1	1	1
10	x	x	x	x

Table 3.5: The Karnaugh maps for the carries of *the choice*.

AND-OR	AND-OR-INVERT
$choose_i = choose_{i+1} + \overline{found_{i+1}} \cdot gc_i$	$choose_i = \overline{(\overline{choose_{i+1}} \cdot found_{i+1} + gc_i)}$
$found_i = found_{i+1} + gf_i$	$found_i = \overline{(\overline{found_{i+1}} + gf_i)}$
Where $gc_i = a_i \cdot \overline{b_i}$ and $gf_i = a_i \otimes b_i$	Where $gc_i = \overline{a_i} + b_i$ and $gf_i = \overline{(a_i \otimes b_i)}$

RESULT
$m_i = choose_i \cdot b_i + \overline{choose_i} \cdot a_i$ Seek the minor
$m_i = \overline{choose_i} \cdot b_i + choose_i \cdot a_i$ Seek the major

Figure 3.2: The Boolean equations for the carries and the result.

The number of gates used for $cell_i$ is 8 for AND-OR and 9 for AND-OR-INVERT including the multiplexer that evaluates m_i. The delay from $found_{i+1}$ to $found_i$ is one gate-level. The AND-OR implementation of $cell_i$ is hiding more than one level of logic per gate. OR and AND gates are iimplementes by NOR and NAND gates with inverters in front. Thus, each of them has two gate-level delay. Reduction to this delay can be achieved by using inversion of the polarity of signals. $Cell_{i+1}$ is followed by $cell_i$ with inverted signals in relevant to $cell_{i+1}$. The result is a design consisting NANDs and NORs, which has one gate-level delay per 2-bit comparator for $choose_i$ and $found_i$. Figure 3.1.4

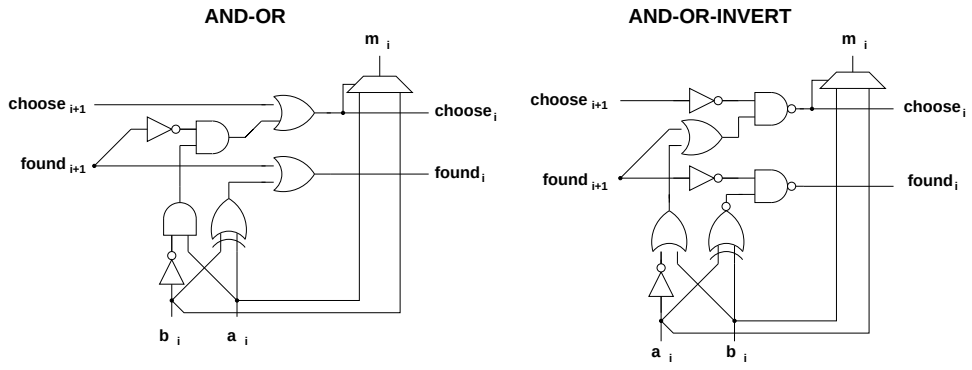


Figure 3.3: a) The AND-OR implementation of the circuit. b) The AND-OR_INVERT implementation.

shows the mentioned optimization. The number of gates for each iimplementation of the

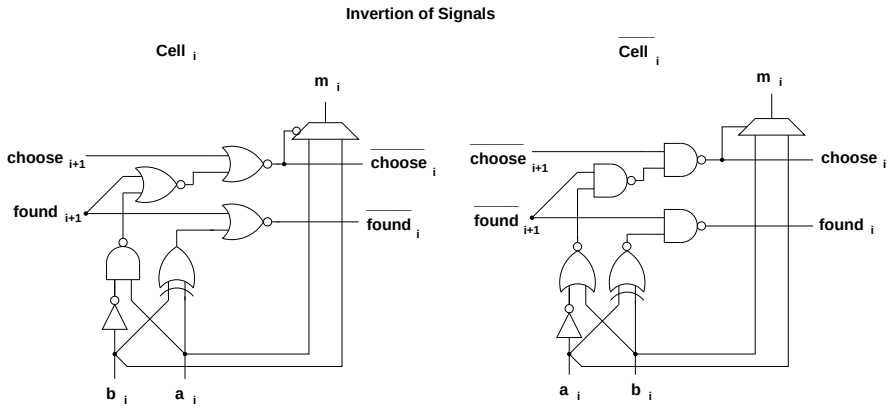


Figure 3.4: Inversion of signals for the cell$_i$.

cell$_i$ is 7. The delay of found$_i$ is one gate-level. The same is for choose$_i$ for cascaded cells, This will be proved in following sections. In addition, the fan-out and the fan-in of all the gates in the cell is less or equal to 2.

3.2 The 2 Element Comparator

3.2.1 The Ripple Comparator

Usually the elements A and B have more than 1 bit. A k-bit comparator can be constructed by cascading k 2-bit comparator circuits in series, connecting *the choice*$_{i+1}$ to the circuit that calculates *the choice*$_i$. This configuration is called ripple-carry comparator since the carry bits "ripples" from one stage to the other.The initial value of carries is "A equals B". The delay through the circuit depends upon the number of logic stages that must be traversed and is a function of the applied input signals. For some output signals no rippling effect occurs at all (e.g. the MSBs), while for others *the choice* has to ripple all the way from the MSB to the LSB. The propagation delay of such a structure (also called *the critical path*) is defined as the worst-case delay over all possible input patterns. In Figure 3.2.1 a 4-bit ripple carry comparator is shown.

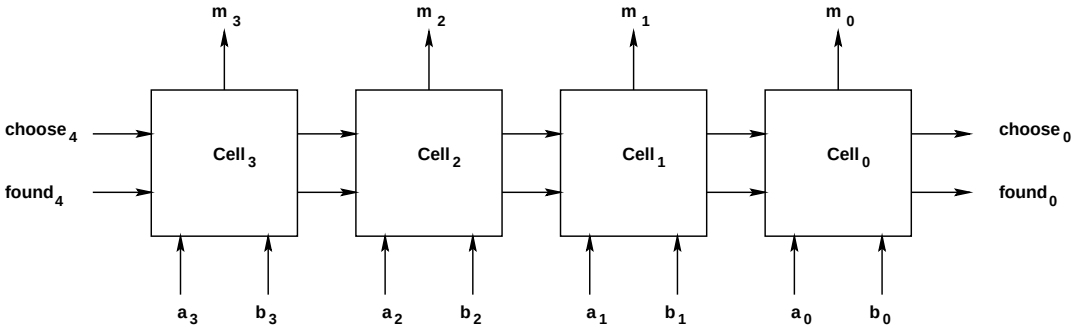


Figure 3.5: Four-bit ripple-carry comparator: topology

In the case of ripple-carry comparator, the worst-case delay happens when the input MSB cell carries are propagated all the way to the LSB. Thus, the delay is proportional to the number of bits in the input words (k) and is approximated by

$$t_{comparator} \approx t_g + k \cdot t_{carries} + t_{mux}$$

where t_g is the delay for the *gc* and *gf* signals to be evaluated, $t_{carries}$ is the delay for the carries to be propagated through one cell and t_{mux} is the delay for the m_0 signal to be stabilised. The dominant factor in equation is $k \cdot t_{carries}$ of carries.

The delay of the 2-element comparator circuit can be reduced by using the inverted signal cells which decrease the delay of the choose carry. Figure 3.6 shows the circuit for k = 4. The enhancement comes from the fact that the cells are cascaded. At the cell that compares the MSBs $found_4$ has 2 gate-levels delay

in contrast to $choose_4$, which suffers from 3-gate-levels delay. This is only true for

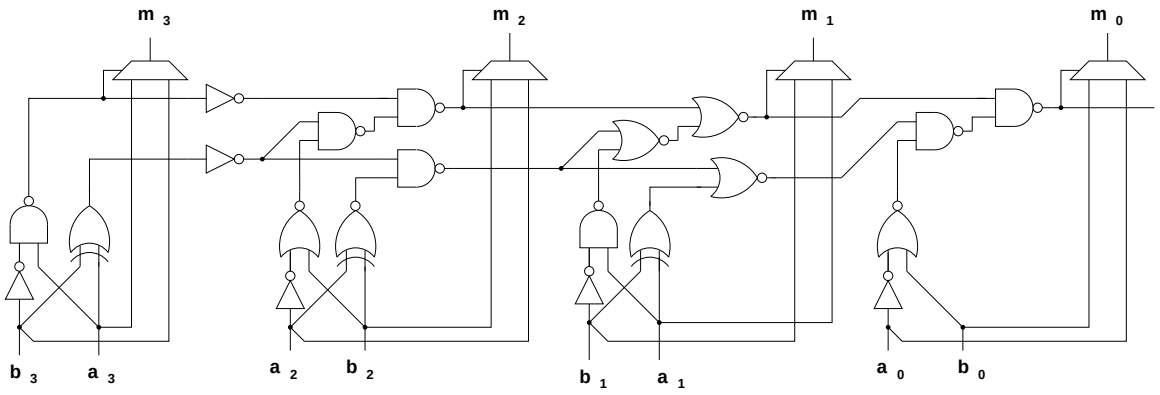


Figure 3.6: Four-bit ripple-carry comparator: circuit implementation with inverse signals

the first cell. At the next significane cell, the output of the gate that accepts as inputs the previous $found_4$ and the current gc_3, is ready simultaneously with the output of the gate that produces $choose_4$ of the previous cell. Thus, $choose_3$ suffers only of 1 gate-level delay. This is the case for both carries until the end of calculation. The overall delay is approximately 1 gate-level/cell, for each carry.

$$t_{comparator} = t_g + (k + 1) \cdot t_{carry} + t_{mux}$$

The propagation delay of the ripple-carry comparator is linearly proportional to number of bits. This property becomes increasingly important when designing comparators for wide datapaths. When designing ripple-carry 2-element comparator, it is far more important to optimize $k \cdot t_{carries}$ than t_{mux} (or t_g) since the latter has only minor influence on the total value of $t_{comparator}$.

3.2.2 The Square Root Carry Select Comparator

In ripple-carry comparator, every 2-bit comparator cell has to wait for the incoming carries before an outgoing carry can be generated. One way to get around this linear dependency is to use the idea of *carry select* adders [Rab96]. The nature of addition operation enforces the carry to be propagate from the LSB to the MSB. We adapted the idea of carry select to the comparison operation where the carries are propagated form the MSBs to the LSBs. The carry select comparators prepare the carries, as the elements were equal. Once the real value of the incoming carry is known, the correct result is selected with a simple *carry select cell (cs-cell).*

Consider a chain of four 2-bit comparator blocks, calculating bits i+3 to i. Instead of waiting for the arrival of *the* $choice_{i+4}$, the comparison starts, as if all the previous bits {k-1,$\cdots$,i+4} were equal. When *the* $choice_{i+4}$ finally settles, either this or *the* $choice_i$

is selected. The selection is done according to the value of *the choice*$_{i+4}$. If the latter indicates the winner,

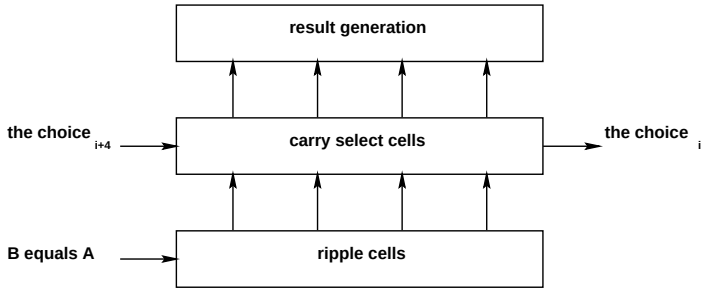
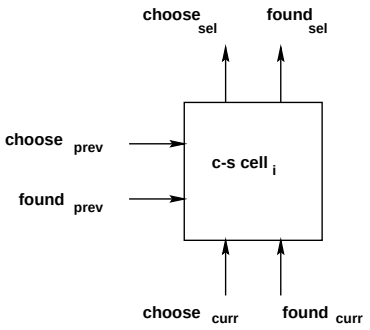


Figure 3.7: Four-bit carry select comparator: topology.

carries are discarded. Otherwise, the incoming carries are discarded. The situation reveals a short of priority, based on the invariant condition of the 2-element comparator. Figure 3.7 shows the case described. The cs-cell (carry select cell) is responsible for the proper selection of carries. The functionality is defined by Figure 3.8 , where the priority property of the cs-cell is shown.



the choice$_{prev}$	*the choice*$_{curr}$	*the choice*$_{sel}$
A equals B	A equals B	A equals B
A equals B	A is the minor/ B is the major	A is the minor B is the major
A is the minor/ B is the major	x	A is the major/ B is the minor

Figure 3.8: The cs-cell and the table recording the functionality.

The truth table is shown in Table 3.6 . Based on that, optimization with Karnaugh maps is performed and the Boolean equations are shown in Figure 3.9 .

AND-OR	AND-OR-INVERT
$\text{choose}_{\text{sel}} = \text{choose}_{\text{prev}} + \overline{\text{found}_{\text{prev}}} \cdot \text{choose}_{\text{curr}}$	$\text{choose}_{\text{sel}} = \overline{(\overline{\text{choose}_{\text{prev}}} \cdot \text{found}_{\text{prev}} + \overline{\text{choose}_{\text{curr}}})}$
$\text{found}_{\text{sel}} = \text{found}_{\text{prev}} + \text{found}_{\text{curr}}$	$\text{found}_{\text{sel}} = \overline{(\overline{\text{found}_{\text{prev}}} + \overline{\text{found}_{\text{curr}}})}$

Figure 3.9: The Boolean equations for the carries.

Similarity of cs-cell's equations with those of the 2-bit comparator cell is observed, as shown in Figure 3.10 .

Inputs				Outputs	
choose_{prev}	found_{prev}	choose_{curr}	found_{curr}	choose_{sel}	found_{sel}
0	0	0	0	0	0
0	0	0	1	0	1
0	0	1	0	x	x
0	0	1	1	1	1
0	1	0	0	0	1
0	1	0	1	0	1
0	1	1	0	x	x
0	1	1	1	0	1
1	0	0	0	x	x
1	0	0	1	x	x
1	0	1	0	x	x
1	0	1	1	x	x
1	1	0	0	1	1
1	1	0	1	1	1
1	1	1	0	x	x
1	1	1	1	1	1

Table 3.6: The truth table of inputs/outputs of the carry select cell.

Finally, the inversion property is used in order to achieve lower delay propagation of the signals. Although only the $choose_{sel}$ is used for the generation of the M, the $found_{sel}$ must be generated too, to be used by the next significance cs-cell. Cascaded cs-cells result to one gate delay for both of the carries.If $choose_{curr}$ is ready early the $choose_{sel}$ results to one gate delay. The critical path is that of $choose_{sel}$ (Figure 3.10).

Consider a case of 20-bit 2-element carry-select comparator constructed by cascaded 4-bit carry select modules. As mentioned earlier the propagation delay of every cell is 1/gate-level/bit. The worst-case arrival times of the signals with respect to the time the input is applied at the different network nodes are marked and annotated in Figure 3.11 (top). This analysis demonstrates that the critical path of the comparator ripples through the cs-cells network of the subsequent stages. From inspection of the circuit, a first-order model of the worst-case propagation delay of the module can be derived.

$$\mathrm{t_{comparator}} = \mathrm{t_g} + (\mathrm{k}) \cdot \mathrm{t_{carry}} + (\tfrac{\mathrm{k}}{\mathrm{K}}) \cdot \mathrm{t_{cs-cell}} + \mathrm{t_{mux}}$$

Where k and K represent the total number of bits, and the number of bits per stage,

$choose_{sel}$				
	00	**01**	**11**	**10**
00	0	0	1	x
01	0	0	0	x
11	1	1	1	x
10	x	x	x	x

$found_{sel}$				
	00	**01**	**11**	**10**
00	0	1	1	x
01	1	1	1	x
11	1	1	1	x
10	x	x	x	x

Table 3.7: The Karnaugh maps for the carries of *the choice*.

respectively. The carry delay through a single 4-bit module is proportional to the length of that stage, or equals k·t_{carry}

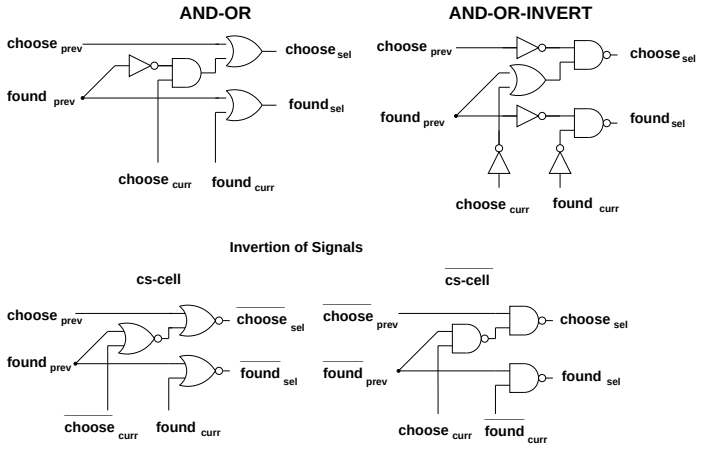


Figure 3.10: The cs-cell circuit. Inversion of signals is used to minimize the delay.

The propagation delay of the comparator is once again linearly proportional to k. The reason for this linear behaviour is that *the choice* still has to ripple through all stages.

The next topology Figure 3.11 (bottom) illustrates a major improvement to the delay of the comparator shown in Figure 3.11 (top). Investigating the linear carry select comparator, one striking opportunity comes to light. Consider the cs-cells of the last comparator stage. The inputs to those cs-cells are *the choice* of the ripple cells of the current block and *the choice* of the previous stage comparison cs-cells. A major mismatch between the signal arrival times can be observed (delays are shown in parenthesis). *The choice* of the current comparison block is stable long before *the choice* of the previous block arrives. It makes sense to equalize the delay through both paths. This can be achieved by progressively adding more bits to the subsequent stages in the 2-element comparator, requiring more time for the generation of the carry signals. For instance, the first stage can compare 2 bits, the second contains 3, the third has 4 and so forth, as demonstrated in Figure 3.11 .The annotated arrival times show that this comparator topology is faster

than the linear organization. It can be observed that the discrepancy in arrival times at the cs-cell nodes has been eliminated.

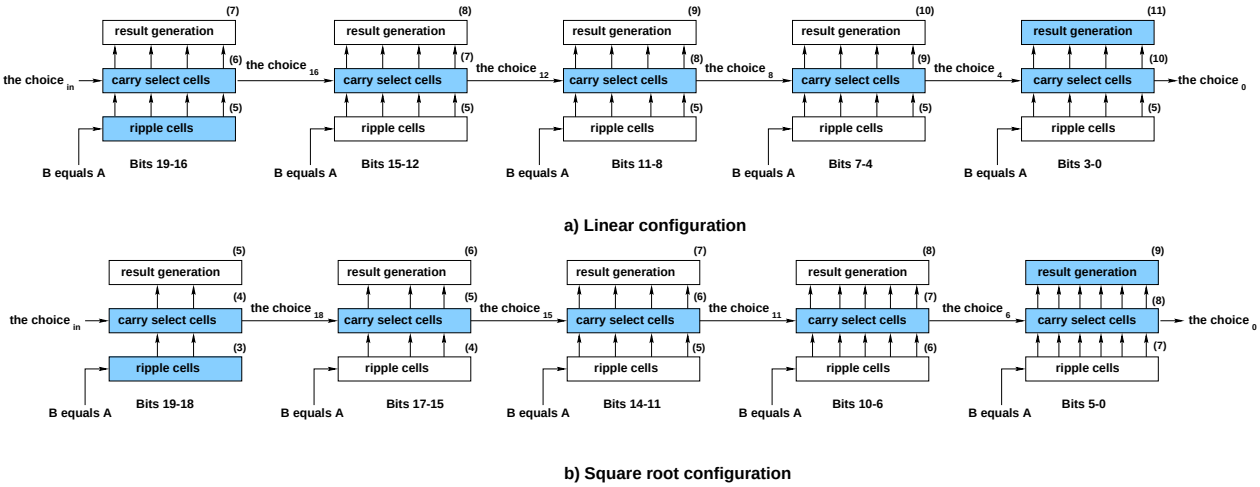


Figure 3.11: a) The linear comparator. b) The square root comparator.

In effect, the simple trick of making the comparator stages progressively longer results in a structure with sublinear delay characteristics. This is illustrated by the following analysis. Assume that an k-bit comparator contain P stages, and the first stage compares K bits. An additional bit is added to each subsequent stage. The following relation then holds:

$$\begin{aligned} k &= K + (K+1) + (K+2) + \ldots + (K+P-1) = \\ &= K \cdot P + \frac{P \cdot (P-1)}{2} = \frac{P^2}{2} + P \cdot (K - \frac{1}{2}) \end{aligned}$$

If K$\ll$k, the first term dominates and the above equation is simplified

$$k \cong \frac{P^2}{2} \Rightarrow P = \sqrt{2 \cdot k}$$

Thus, the delay can be expressed as

$$t_{comparator} = t_g + K \cdot t_{carry} + (\sqrt{2 \cdot k}) \cdot t_{cs-cell} + t_{mux}$$

The delay is proportional to $\sqrt{k}$ for large comparators (K$\ll$k), or $t_{comparator} = O(\sqrt{k})$.

The square root carry select comparator circuit is susceptible to optimization, at a closer look. The initial value of *the choice* usually is not provided and the carry select cells of the initial bock are unnecessary. The carries to the next significance block are the output of the last ripple cell of the initial block. The result generation block is taking the carries directly from the ripple cells. Another issue is the linearly increasing fan-out to the output of cs-cells as more blocks are added to the comparator chain. This is also the case for carry select adders. The idea is to transfer the fan-out from the path that announces *the choice* to the next significance block, to the path that produces the result.

The latter is not critical and can undertake the mentioned delay. The implementation of these ideas is illustrated in Figure 3.12 .

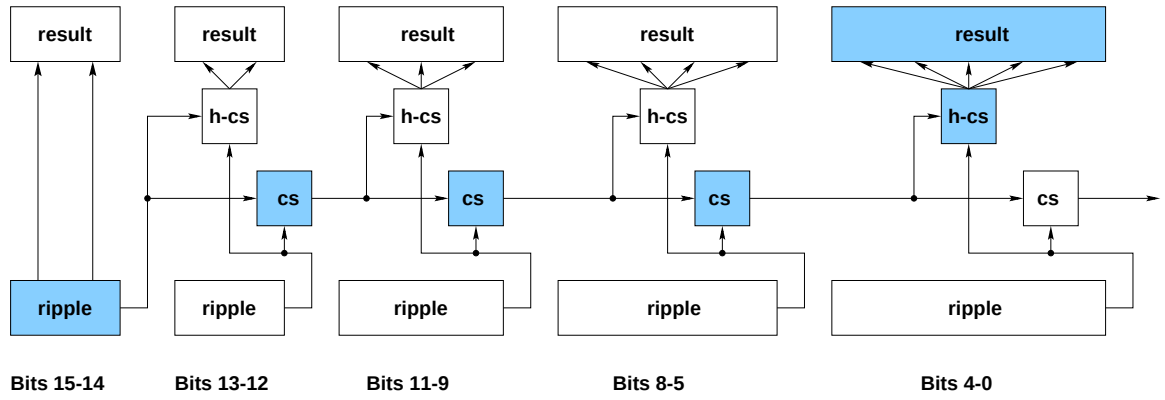


Figure 3.12: The carry select comparator's configuration after the critical path delay optimization.

The *h-cs* (stands for half carry select) blocks shown in Figure 3.12 are cs-blocks without the gate that calculates the carry *found*. The latter is not needed for the evaluation of the results. The fan-out of the carry select blocks residing on the critical path is constant and equal to 2. For large values of k (k>64) the fan-out of the h-cs cells is significant. One solution is to use a tree of buffers from the driving cell to the driven cells. This structure has a logarithmic delay.

The equation for the delay now needs to be redefined. The following relation holds

$$k = K + K + (K+1) + (K+2) + \ldots + (K+P-1) =$$
$$= K \cdot (P+1) + \frac{P \cdot (P-1)}{2} = \frac{P^2}{2} + P \cdot (K - \frac{1}{2}) + K$$

Solving the above equation for P,

$$P^2 + (2 \cdot K - 1) \cdot P + 2 \cdot (K - k) = 0$$

the following result occurs

$$P = \frac{-(2 \cdot K - 1) + \sqrt{(2 \cdot K - 1)^2 + 8 \cdot (k - K) 2}}{}$$

Thus, the delay for the carry select comparator can be expressed as previously

$$t_{comparator} = t_g + K \cdot t_{carry} + P \cdot t_{cs-cell} + (\log_2 P) \cdot t_{mux}$$

If K$\ll$k the result is the same as the pre-optimized configuration.

$$P = \sqrt{2 \cdot k}$$

3.2.3 The Carry Look Ahead Comparator

When designing fast comparators, it is essential to get around the rippling effect of the carries that is still present in carry select comparator. The *carry look ahead* principle

offers a possible way to do so. As stated before the following relation holds for each of the carries

$$\text{choose}_\text{i} = \text{choose}_{\text{i}+1} + \overline{\text{found}_{\text{i}+1}} \cdot \text{gc}_\text{i}$$
$$\text{found}_\text{i} = \text{found}_{\text{i}+1} + \text{gf}_\text{i}$$

Where $\text{gc}_\text{i} = \text{a}_\text{i} \cdot \overline{\text{b}_\text{i}}$ and $\text{gf}_\text{i} = \text{a}_\text{i} \otimes \text{b}_\text{i}$ The dependency between choose_i and choose_{i+1} can be eliminated by expanding choose_i. This is also the case for found_i. For the first 2 bits

$$\text{choose}_{\text{N}-1} = \text{choose}_\text{N} + \overline{\text{found}_\text{N}} \cdot \text{gc}_{\text{N}-1}$$
$$\text{found}_{\text{N}-1} = \text{found}_\text{N} + \text{gf}_{\text{N}-1}$$
$$\text{choose}_{\text{N}-2} = \text{choose}_\text{N} + \overline{\text{found}_\text{N}} \cdot (\text{gc}_{\text{N}-1} + \overline{\text{gf}_{\text{N}-1}} \cdot \text{gc}_{\text{N}-2})$$
$$\text{found}_{\text{N}-2} = \text{found}_\text{N} + \text{gf}_{\text{N}-1} + \text{gf}_{\text{N}-2}$$

The general forms for the two carries are for the vth bit

$$\text{choose}_\text{v} = \text{choose}_\text{N} + \overline{\text{found}_\text{N}} \cdot (\textstyle\sum_{\text{i}=\text{N}-1}^{\text{v}} (\prod_{\text{k}=\text{i}+1}^{\text{v}+1} \overline{\text{gf}_\text{k}}) \cdot \text{gc}_\text{i})$$
$$\text{found}_\text{v} = \text{found}_\text{N} + (\textstyle\sum_{\text{i}=\text{N}-1}^{\text{v}} \text{gf}_\text{i})$$

Observe that $\text{choose}_N = 0$ and $\text{found}_N = 0$. Thus, the equations take the close form

$$\text{choose}_\text{v} = \textstyle\sum_{\text{i}=\text{N}-1}^{\text{v}} (\prod_{\text{k}=\text{i}+1}^{\text{v}+1} \overline{\text{gf}_\text{k}}) \cdot \text{gc}_\text{i}$$
$$\text{found}_\text{v} = \textstyle\sum_{\text{i}=\text{N}-1}^{\text{v}} \text{gf}_\text{i}$$

Those general forms can be used for the implementation of v-bit comparator. For every bit, the carries are independent of the previous carry bits. The ripple effect has been effectively eliminated, and therefore the comparison time should be independent of the number of bits. This is not true, because of delay's hidden dependencies on the number of bits. The constant comparison time is wishful thinking and the delay is at least linear with the number of bits. One problem arises from the large fan-out of the gates that evaluate gf_i and gc_i. The delay of those gates increases, since the propagation delay of a gate is proportional to its load. Another problem arises from the large fan-in of the gates that evaluates the corresponding carries. Finally, the area of the implementation grows progressively with k. Therefore the carry look ahead structure is only useful for small values of k.

The resultant circuit can be further optimized. The value of the carry *found* is not used for the calculation of the result, but only for that of *choose*. In addition, the circuit is designed with NAND and XOR gates. The optimized circuit is shown in Figure 3.14 .

The fan-in of the NAND gates is linearly increased with k. This problem can be solved by replacing a k-input AND gate with a binary tree of AND gates. The delay of this structure is increased logarithmically with k. Another problem is the linearly increased fan-out of the XOR and NOR gates, resulting a significant load to the driving gate. This

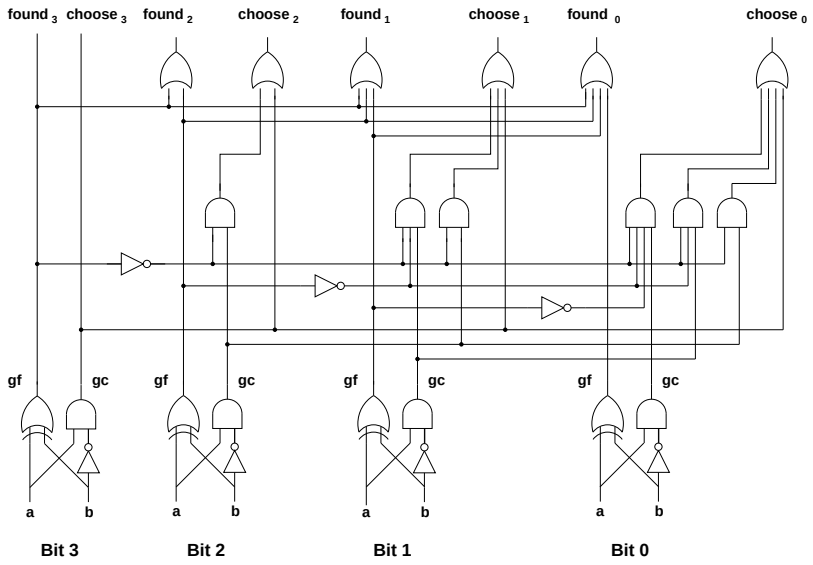


Figure 3.13: The carry look ahead implementation of the 2 element comparator.

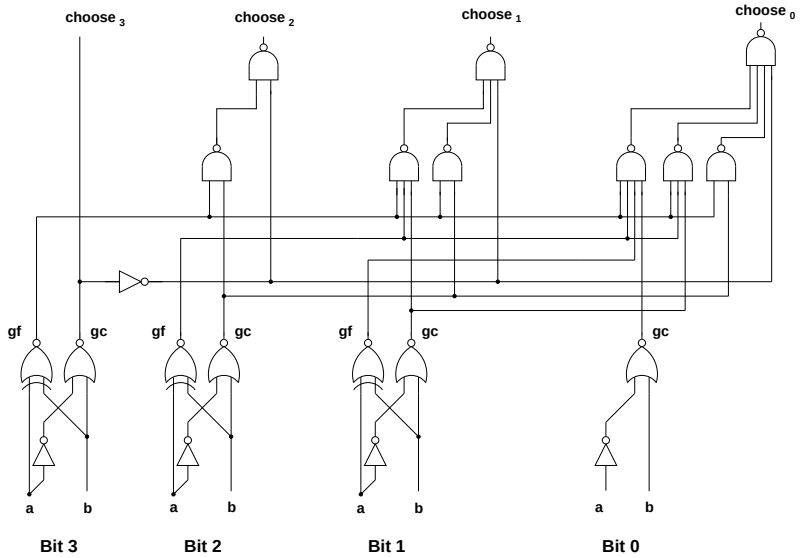


Figure 3.14: The optimized carry look ahead implementation of the 2 element comparator for k=4.

problem can be solved by using tree of buffers from the latter to the driving loads. This can lead to logarithmically increased delay as function of k, which can be expressed as

$$t_{comparator} = t_g + 2 \cdot (\log_2 k) \cdot t_{gate} + t_{mux}$$

The above equation shows that the delay of the carry look ahead version of the 2-element comparator is logarithmic function of k.

3.2.4 Conclusions for the 2 Element Comparator

The discussion of comparators is by no means complete. Due to its impact on the performance of computational structures, the design of fast adder circuits has been the subject of many publications. The optimizations introduced in those publications can be directly used for the implementation of fast comparators.

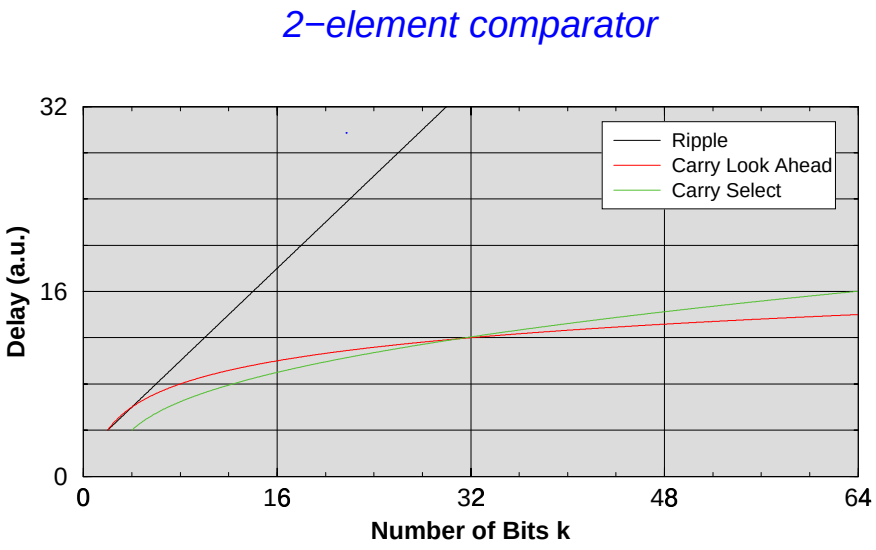


Figure 3.15: The ripple, carry Look ahead and carry select delays for 2-element comparator.

A proper coding for the 2-bit comparator cell is introduced and optimized. Three versions of a 2-element comparator are proposed here. Delay optimizations are proposed and the performance of the circuits is enhanced. This is a good time to compare the delay of the various versions of the 2-element comparator. This can be done easily by plotting in the same figure the functions of the delays with the proper approximations. Setting

$$t_{cs_cell} = t_{gate} = t_{mux} = t_g = t_{carry} = 1$$

the equations for the delay are (K=2 for carry select)

$$t_{cla} = 2 \cdot (\log_2 k) + 2$$

$$t_{cs} = \frac{\sqrt{8 \cdot k - 7}}{2} + \log_2(-3 + \sqrt{8 \cdot k - 7}) + 0.5$$

The plotted equations are shown in Figure 3.15 . Detailed examination of the carry select circuit shows that it is faster than that of the carry look ahead for elements shorter than 28bits.

3.3 The Binary Tree Comparator

The 2-element comparator circuit was designed in order to be the basic block of the binary tree of comparators. The binary tree algorithm, the structure and the propagation of the signals are described in Section 2.4.2 . In this section, the delay of the binary tree of comparators is examined, using the 2-element comparators designed in the previous section. The carry select comparator will be redesigned in order to match the delays of the signals propagated in the binary tree. In contrary, there is nothing to be done with the other topologies.

The delay of comparing N elements by using a binary tree structure is not independent on the number of compared elements. The complexity of the algorithm as mentioned earlier is $\log_2$N. The calculation for the minimum (or maximum) of an N-element set can be executed by placing to every node one 2-element comparator.The advantage of this topology is that if the compared elements are doubled the total delay will be inreased by the delay of one 2-element comparator. In contrary to the delay the area of the circuit is growing linearly with the number of elements. This is also the case for the power.

The property of parallelism is used and all the levels of the binary tree are performing calculations in parallel. In order to take full advantage of this property and for the maximum utillization of the circuit, the arrivals of the signals must be equalized. The ripple comparator is the configuration that by nature equalizes the delays of propagated signals. This is due to the fact that the comparison operation for this topology is performed bit by bit. The results are produced sequentially at every comparator in all levels. Figure 2.9 presented in Section 2.4.2 is the proof of the previous statements. The disadvantage of this configuration is the linearly dependent delay of the 2-element comparator.

Investigating the 2-element comparator cell is necessary for the derivation of the delay equation of the binary tree. The outputs of every cell_i are not stable simultaneously. The carries are ready earlier than the result because the latter uses one of the carries for its calculation. In addition, the path along the 2-element for calculating the carries is 1/gate delay per 2-bit comparison. In contrary, the path from the element bits to the result is 2-gate delay. This is not a severe disadvantage in a binary structure configuration because usually the longest path is the one along the 2-element comparator and not along the

levels of the tree (from the leaves to the root).

3.3.1 The Binary Three with Ripple Comparators.

A part of a binary tree with ripple configuration of the 2-element comparator is shown in Figure 3.16 The elements compared by such a tree are N= 8, having each k = 4 bits. Seven comparators must be used for the comparison of 8 elements. The tree has 3 levels and each 2-element comparator uses 4 2-bit comparator cells. The first cell is different from the others because the carries have taken their initial values. The last cell is also different from the others because the final value of $found_0$ is not needed. *Choose* indicates the winner. The critical path of the calculation is highlighted. It is observed that at least 5 gates delay per level is introduced. This is the case until the signals reach the top level. There the delay added is 1 gate per bit.

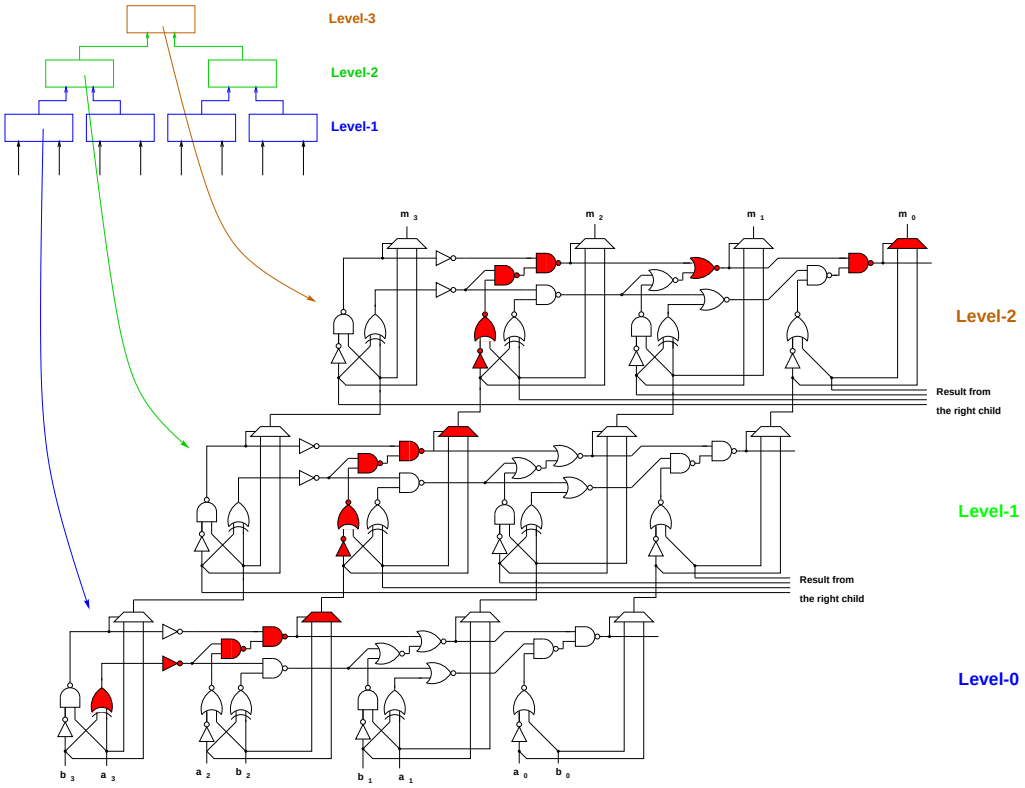


Figure 3.16: Propagation delay for the three 2-element comparator structures in the binary tree.

Assuming that all gates have the same delay equal to 1 (this is not always true but it stands for our case due to the small fan-in and fan-out of every gate), an equation for the delay of the binary tree can be derived.

$$t_{\text{ripple tree}} = k - 2 + 5 \cdot \log_2 N$$

In contrary to the delay the area of the circuit is growing linearly with the number of elements. This is also the case for the power.

3.3.2 The Binary Three with Carry Select Comparators.

Another version of the binary tree comparator is designed with carry select topology for the 2-element comparator. Equalizing the delays between 2-element comparators lying at sequential levels of the tree is needed. Figure 3.17 shows the problem which occurs when all the 2-elements comparator blocks are designed as shown in Figure 3.12 for all the levels of the tree. The delay of the ripple block can be derived by Figure 3.16 .

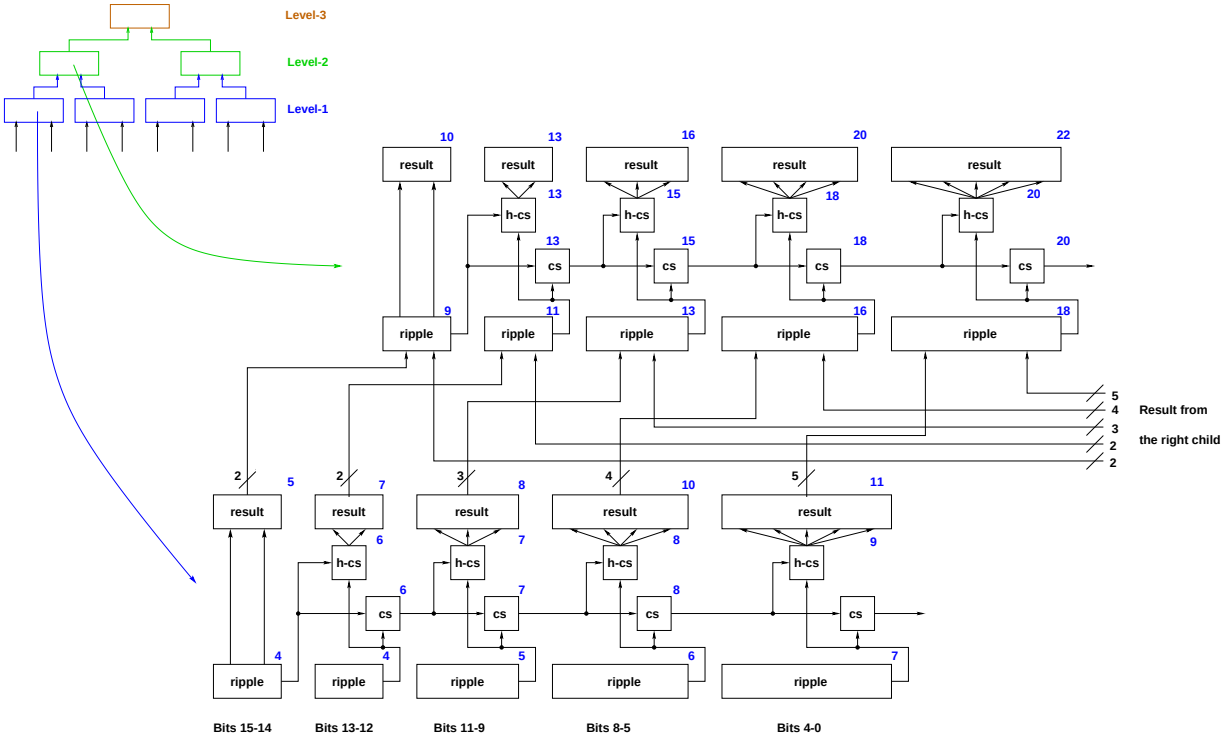


Figure 3.17: Propagation delays of the binary tree structure with carry select comparators.

This block is the same with the ripple comparator's without the multiplexor part. Thus, it is 4 gates for the first two bits and one gate delay for the following cs-cell chain. The mux part is moved to the result block. At Level-0, the first cs-cell in the chain introduces 2 gates delay in contrary to the others having only one. This is also the case for the hs-cells. Finally, the delay of the muxes must be added. The effect of large fan-out is significant only in the last parts of the comparator and for elements with many bits. At Level-1, a different situation exists. The output carries of the first ripple part of the 2-element comparator is ready before the carries of the second ripple part. Thus, the

evaluation of the first cs-cell block is delayed. The evaluation of the second cs-cell block in the chain has to wait for the third ripple part of the carry select to finish the calculation. Thus, the problem is the delay of propagating *the choice* along the 2-element comparator through the chain of cs-cells, because the inputs to the ripple blocks are delayed.

The solution is to feed the ripple parts earlier, shown in Figure 3.18 . A 5-level binary tree is designed with different 2-element comparators at each level of the tree. Only one comparator per level is shown for simplicity. Level-0 is constructed with a 2-element comparator as shown in Figure 3.12 . This is not the case with the rest of the levels, where comparators are hybrid ripple and carry select. The percentage of ripple is defined by a simple rule: *Moving one level up means moving the last part of 2-element comparator from carry select to ripple section of the comparator.* This rule equalizes the propagation delays of the different parts of the 2-element comparator at each level of the tree. The equalization of the delays is necessary for the parallelism property to be valid and the proper propagation of *the choice* along the 2-element comparator through the cs-cells. Thus, careful partitioning of the bits of the resulting element to feed the next level 2-element comparator is needed.

All the comparators are calculating in parallel in all the levels. The critical path of the circuit has to be defined. The complexity will be derived by the known properties of the binary tree and the carry select comparator. Beginning from the bottom level comparator, the critical path is the one shown at Figure 3.12 . Thus, *the choice's* signals have to propagate along the 2-element comparator and the complexity is $t_{comparator} = O(\sqrt{k})$ for k-bit elements. The rest of the delay is propagating the last part of the carry select bits from the leaves to the root of the tree. The number of those bits is decreasing linearly passing through the levels of the tree. The complexity of this topology is calculated as follows: Assume l is the number of bits calculated by the last part of the 2-element comparator that lies at the leaves. Then

$$l = K + P - 1$$

For all the comparators at the tree comparing N elements the sum S of those parts is

$$S = l - 1 + l - 2 + \cdots l - \log_2 N \Rightarrow$$
$$S = \log_2 N \cdot l - \frac{\log_2 N \cdot (\log_2 N + 1)}{2} \Rightarrow$$
$$S = \log_2 N \cdot [l - \frac{\log_2 N + 1}{2}]$$

Substituting l

$$S = \log_2 N \cdot [K + P - 1 - \frac{\log_2 N + 1}{2}]$$

usually K=2

$$S = \log_2 N \cdot [P - \frac{\log_2 N - 1}{2}]$$

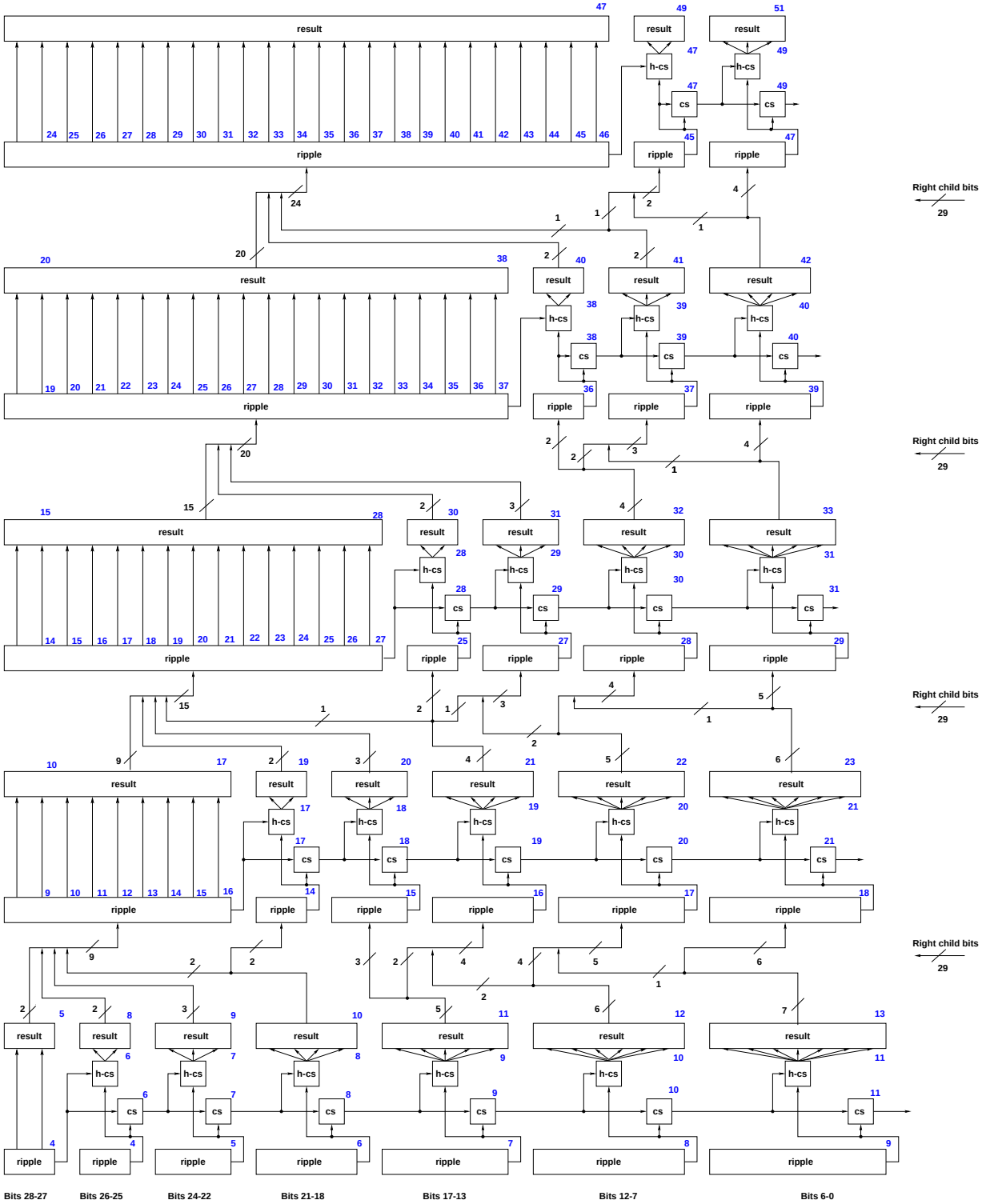


Figure 3.18: A 5-level binary tree with carry select comparators. Only one comparator per level is shown for simplicity. The hybrid comparators are shown. The delays(with blue) of the signals are matched.

As shown in Section 3.2.2

$$\mathrm{P} = \sqrt{2 \cdot \mathrm{k}} \Rightarrow$$

and substituting at S

$$\mathrm{S} = \log_2\mathrm{N} \cdot [\sqrt{2 \cdot \mathrm{k}} - \tfrac{\log_2\mathrm{N}-1}{2}]$$

Thus we conclude to an equation of S having only the number of elements N and the number of bits per element k. This equation gives us the total complexity of the binary tree

$$\mathrm{O}(\sqrt{\mathrm{k}}) + \mathrm{O}(\log_2\mathrm{N} \cdot [\sqrt{\mathrm{k}} - \log_2\mathrm{N}])$$

This relation is valid for

$$\sqrt{\mathrm{k}} > \log_2\mathrm{N})$$

The complexity of the hs-cells and the large fan-out is not taken in mind. This complexity depends logarithmically on l and is discarded in this approximation.

The area of this scheme is increased compared with the one of binary tree with ripple 2-element comparators. This is also the case for the power.

3.3.3 The Binary Three with Carry Look Ahead Comparators.

The last version of the binary tree comparator is the one that uses carry look ahead topology for the 2-element comparator block. Figure 3.19 is shows the mentioned topology. The critical path of the tree is highlighted. This path includes the gates with the largest fan-in and fan-out in the circuit. The property of parallelism is conserved in this topology. This is the result of the linearly increasing fan-out of the gates. The solution to large fan-in of the gates is to replace each one with a binary tree of 2 fan-in gates. Tis results to $\mathrm{O}(\log_2\mathrm{N})$ delay. To deal with the large fan-in of the gates, a sequence of buffers is used. The drive strength of those buffers is grsdually increased.

Figure 3.19 shows a circuit, where the 2-element comparators cannot be programmed. A initial condition to the comparators cannon be applied. This is due to the fact that the initial value of *the choice*$_N$ is replaced with "A equals B". Thus, the a 2-element comparator could not start comparing with initial condition different than "A equals B". In different case, a significant amount of large fan-in gates and wires should be added, increasimg the dealy of the circuit. This is not the case with the 2 previously analyzed topologies: 3-4 gates suffices, increasing slightly the overal delay. Programmability of the 2-element comparators is necessary for the proper function of the WRR scheduler, as will be proved at next sections.

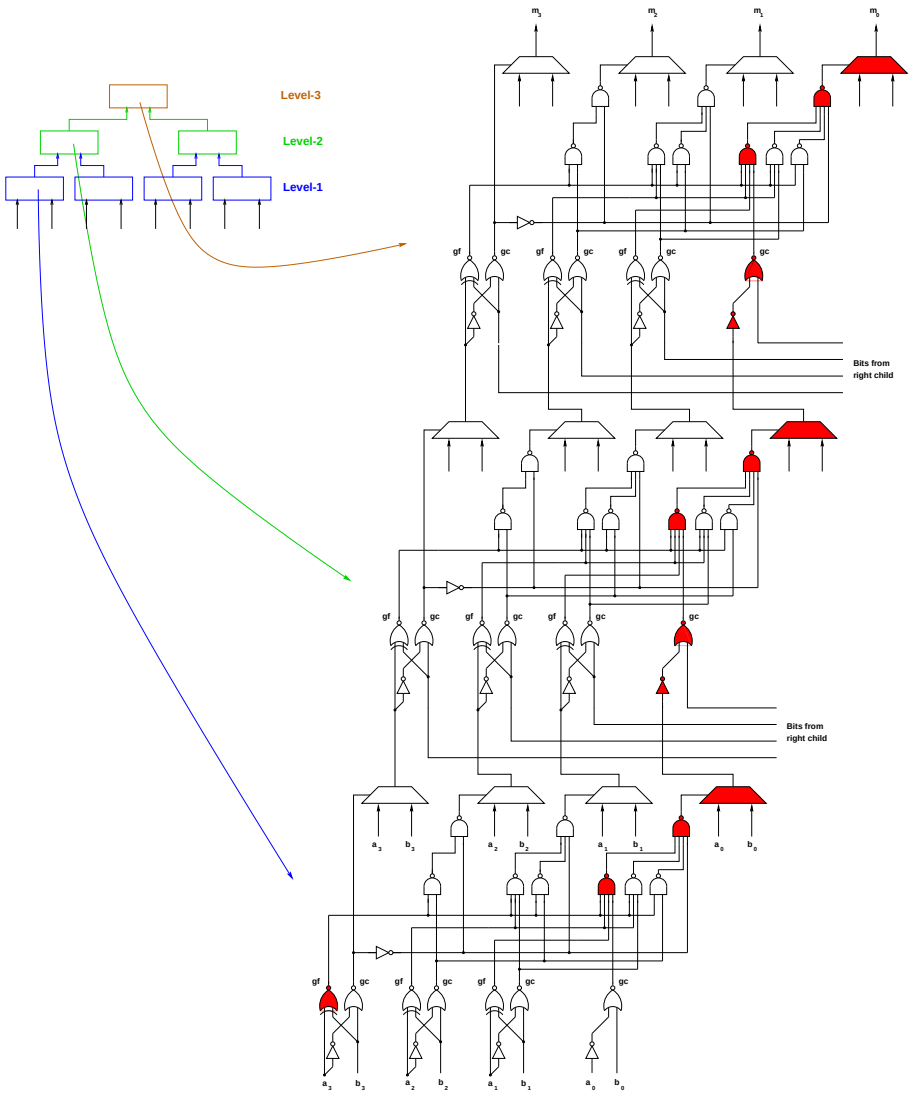


Figure 3.19: A 3-level binary tree with carry look ahead comparators.Only one comparator per level is shown for simplicity.

Chapter 4

Design of the Scheduler

In previous sections, the WRR disciplines were introduced. Those raised the necessity of a priority queue data structure. The latter is characterised by specific operations, which must be supported by the scheduler. The operations are defined and analyzed at the following sections. Afterwards, the datapath and the control will be designed step by step.

4.1 Operations

The discussion of this section involves the operations to be performed by the scheduler The basic operations for the dynamically change priority queue are *insert delete* and *replace*.

Delete operation is performed to ineligible flows that are scheduled to be served in the future and make the transition eligible $\Rightarrow$ ineligible. This situation is common in the scheduler and appears for several reasons (last packet of the flow is served, backpressure signals, bank interleaving). The NST of this flow must not participate to the competition for the minimum.

Insert operation is performed, when the flow becomes from inactive state to active one. This is the result of a new packet arrival in an empty queue and all the reasons mentioned for *delete*. In order to be consistent with the WRR disciplines, the new flow must be inserted to the current "time" (NST¡current "time"), or the NST defined (NST¿ current "time"). If multiple flows become active simultaneously, all of them will be served at current "time", using RR.

Replace operation is performed when a flow packet is served, and it is not the only one from the same flow waiting for service. The updated NST must participate to the

competition for the winner, so it has to replace the previous one in schedule.

4.2 Tasks and Interface

In addition to the operations performed to manage the priority queue, there are some features, the scheduler must have for the proper operation. Those tasks are independent from the basic operations mentioned earlier. They include information for the activity of the flows, stopping the scheduler, adding the FSI to the NST of the flow and others analyzed in the next paragraphs.

In order to define the status of the flows, the usage of *ready bits* is necessary. Every ready bit corresponds to a flow showing its eligibility. Those are *inputs* to the scheduler and at every iteration. The queue manager, who inserts and deletes packets from the queue of each flow and receives flow control signals, provides the *ready bits*.

WRR disciplines define the current "time". This definition separates the NSTs in two divisions: those greater and those smaller than current time. In every iteration those divisions have to be updated. This will help the scheduler to reinsert ineligible flows, which become eligible. The flows of the first division (NST_i<current "time") NSTs are reinserted in the schedule at current "time". In contrary, the others remain to the (NST_i> current "time") NST, at which they are scheduled to be served.

Every flow has its P-bit FSI, which is inverse proportional to the weight of that flow. This number must be provided to the scheduler, every time a new request for connection is granted. It is stored to an SRAM, which is accessed at every cycle. Flows are indexed with i, $i \in [0,N-1]$. This index is used as address internally to the scheduler and as address input to the SRAM block. The FSI for a new connection and the index i are *inputs* to the scheduler.

The decision of the scheduler for the flow to be served is not always granted by the queue manager. This is usually due to limited link capacitance or latency problems introduced in memory interface etc. The scheduler must know if the flow is really served in order to provide the service according to the weight of the flow. If this information is not provided, the NST of the flow will be updated and the latter will receive less bandwidth and more latency. Thus, a signal which carrying the information about the action performed on the flow from the queue manager is *input* to the scheduler.

Some times scheduler has to stop working, for power saving. Thus, a signal indicating this situation is provided to the scheduler, and the latter stops all the internal tasks. This

signal is *input* to the scheduler.

In every iteration scheduler should *output* a number, which is the index of the chosen flow. Also, it must indicate if this signal is valid or not. The latter is *output* from the scheduler.

4.3 Element Representation and Wrap-around

In Section 2.4 , an assumption was made about the representation of the numbers in the scheduler. The choice taken in favour of unsigned integers against floating point is based on two parameters: the hardware simplicity and the precision wanted. The first parameter deals with the amount of hardware wanted for the addition of floating point numbers, which includes adder, incrementor and barrel shifters, for aligning and normalizing the mantissa. In addition, the result occurs in more than one cycle. In contrary to those, unsigned addition needs only one fast adder, which is translated in less delay, area and power consumption. Furthermore, the implementation of WRR disciplines does not need the precision of floating point numbers. It works well enough for unsigned integers, with not so many bits (15 - 25 in this work).

There exist a problem emerging, which is associated with the numeric values of the elements participating at the contest. A scheduler may need to increase the NST of the flow in order to reinsert in different position in the priority queue. This may be repeated arbitrarily large number of times before the element is requested to leave the queue. The values of the elements increase monotonically, eventually leading to a saturation of the number field of the element, no matter how many bits wide it is. Sooner or later wrap-around will occur. Assume a k-bit number, reaching its maximum value 2^k-1. A consecutive increment will bring it back to the beginning, starting from "00". However, such a number should continue to be considered greater than another close to the maximum value 2^k-1, according to the service schedule. A normal comparison will make a mistake in this case. The solution is shown in Figure 4.1 .

The space of allowed element values is segmented in four subplanes, according to the encoding to the two MSBs of the elements. Each color defines a rage of numbers, at which the corresponding condition is valid. In order to compare without mistakes, all numbers should have the same color. To achieve this, for k-bit elements, the FSI added should not be larger than k-2 bits. Increment is always performed on the minimum value element, thus the range of the values of the elements is bounded at most in two subplanes. Assume, all the numbers begin from "00". Thus their color is green. In certain time some numbers

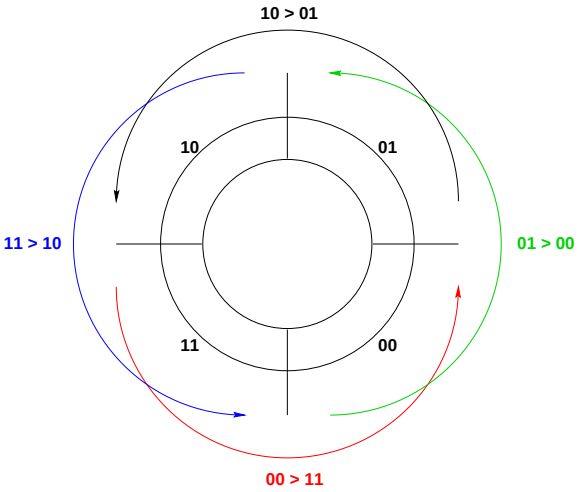


Figure 4.1: Managing wrap around. The elements values can be only one color.

will be at "01" subplain, while others will be at "00". As soon as the last element value passes from "00" to "01", the color of the set becomes black. As a result, the two first bits of a 2-element comparator works different than the rest.

At the previous paragraphs, the interface of the scheduler is described and the tasks that it must perform are defined. Scheduler's block is shown if Figure 4.2 . In the next section, pipelining is introduced before building the datapath and the control block.

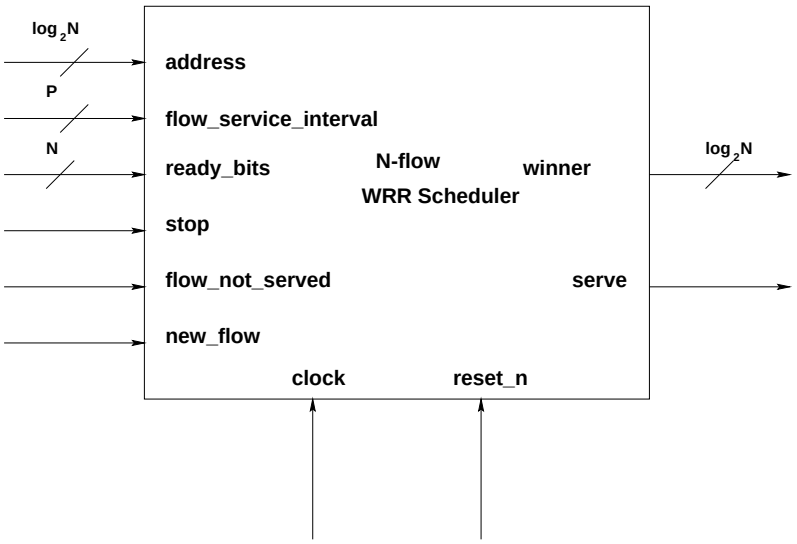


Figure 4.2: The top block of the N-flow WRR scheduler.

4.4 Pipelining

After the previous tasks were defined, a chance of pipelining the scheduler rises. To achieve a high task throughput for the scheduler, new iterations should issued and start executing before a previous one has completed. To achieve such parallelism, each iteration already issued, should overlap with the already existing one. The best way to do this is to separate the "updating" of already found NST and the finding of a new NST. Thus, a 2-stage pipelining can be introduced. As long as the binary tree of comparators tries to locate the minimum among N elements, access to SRAM can be issued and the addition of the FSI of the flow to the current "time", can be performed. This optimization introduces a couple of issues to deal with.

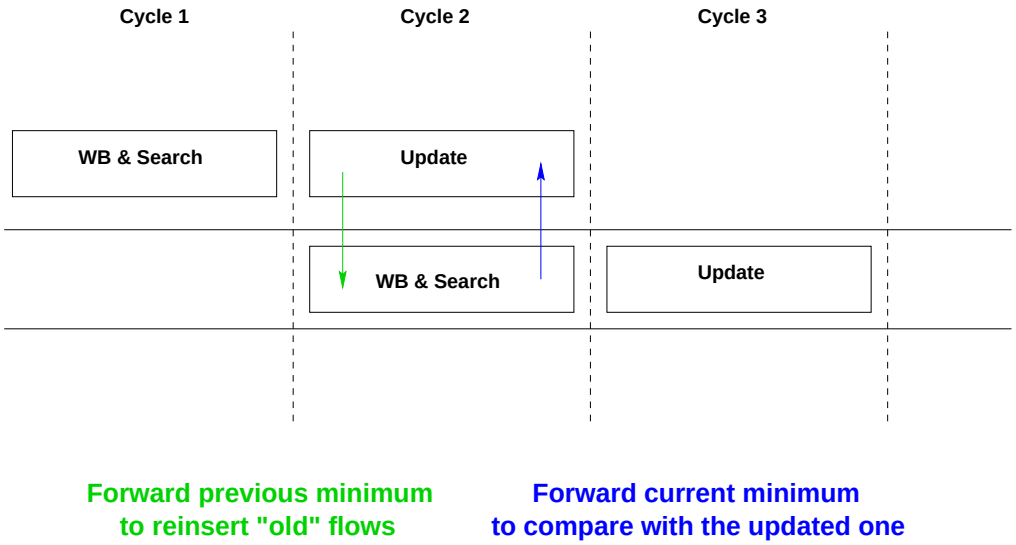


Figure 4.3: Forwards in the pipeline.

The first one concerns the value of the updated winners element, which must participate to the new competition. Referring to Figure 4.3 this value is generated at cycle-1. At cycle-2, this value is not yet updated, while simultaneously, a new iteration is issued. The winner, generated at cycle-1 and not yet updated, is removed from the contest. As soon as the competition at cycle-2 finishes the current winner is forwarded to the "update" section, in order to be compared to the updated previous winner. With this trick, the updated value of the previous winner is reinserted in the contest. The forwarding of the winner element to the update section solves another problem. This concerns the situation that the *flow_not_served* signal is raised. This means that the previous winner is not served. Thus, it must reinserted into the schedule. This is done by forwarding the current winner and compare with the *non-updated* version of the previous winner.

The second concerns the reinsertion to the schedule of *old flows*. The *Old flows* are ineligibles, with NSTs smaller than the current "time". The reinsertion is performed at always at current "time". Thus, the latter becomes the NST of those flows, and participates to the competition. So, forwarding the winner from the previous competition to that just started, is necessary.

Another, point for discussion is updating of SRAM with a new FSI. Keep in mind that SRAM is busy at every cycle, providing the FSI for the winner element. A single-port SRAM is used because the writes are significantly less than reads. A write occurs every time a new connection is established, for the new FSI. The number of packets served by a switch is larger than the number of connections. Because of that reads are more than writes. A *structural hazard* occurs on updating the contents of the SRAM. Thus, updating the winners FSI has to be done again and the pipeline is stalled for one cycle.

In this section pipelining is discussed and the problems with data and structural hazards solved. The next section deals with the building of the data path.

4.5 Building the Datapath

Previous discussion has prepared the introduction of the datapath. Figure 4.4 shows the basic blocks and it will be the next to discuss. The presentation of the datapath begins with blocks performing simple operations, in order to understand clearly the functionality of the circuit. As a guideline to the design of the datapath, the following statements are followed. Stage-1 of the pipeline is the most time-consuming, because of the binary tree of comparators block. Thus the main concern is to remove tasks from stage-1 and put them in stage-2.

The description starts with the first pipelining stage. The most significant block is the *N-1 2-element comparators Binary Tree*. This block accepts N elements at the inputs and calculates the minimum of that element as shown in the previous chapter. Those elements are stored in N registers.

Another important block is the *Locate the address of the minimum*. The name of the block explains its functionality. As inputs accepts (N-1)-bit *the_choice_bits*, which are the final choice of every 2-element comparator in the tree. The output of this block is the $\log_2$N-bit *winner_address*. A detailed diagram is shown in Figure 4.5 . A simple observation of the diagram shows the idea behind the implementation of this block. Every father node, knows which child won the contest, by just reading the last value of the carry

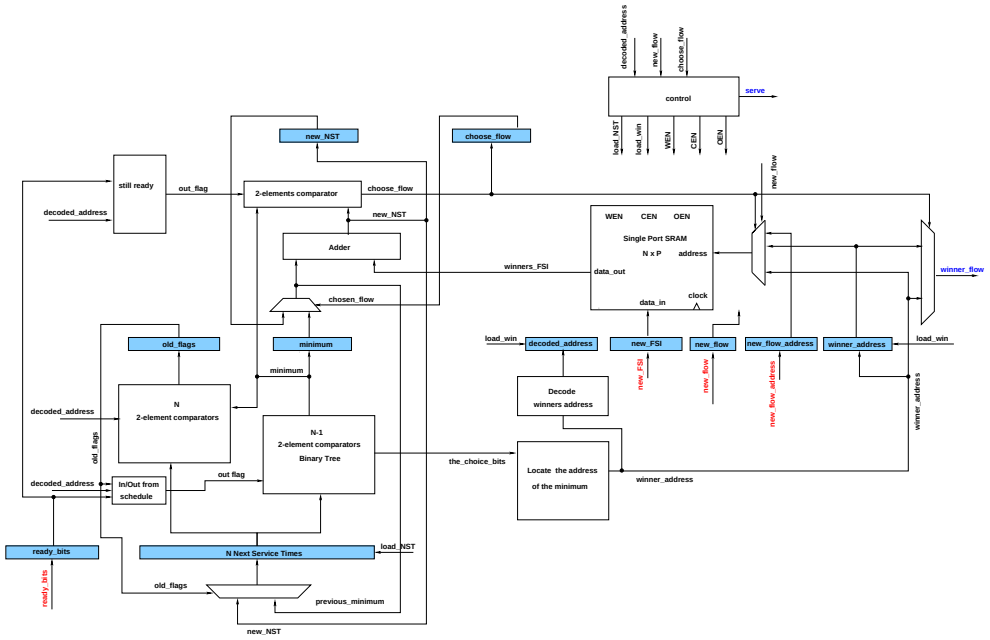


Figure 4.4: The datapath of the WWR scheduler: a first approach.

$choose_0$.

In Figure 4.5 , this value is written in the boxes. Every box is a node and represents a 2-element comparator. If the indexing of the flows is done the way Figure 4.5 shows, then by reading the values in the boxes, the address of the winner element can be located. Reading begins from the root and ends to the leaves. The path followed, is the same with that of the winner element. The implementation of this idea can be done by using *trees of multiplexers*. One tree is dedicated to one address bit . For N elements the number of trees is $\log_2$N - 2. In order to calculate the i_{th} bit (0 bit is the LSB bit), a binary tree of multiplexers with depth $\log_2$(N - 1 - i) is needed. Thus, for N elements the number of multoplexors needed is

$$\begin{aligned}
\mathrm{S} &= (2^1 - 1) + (2^2 - 1) + \cdots 2^{\log_2 \mathrm{N} - 1} \Rightarrow \\
\mathrm{S} &= 2^1 + 2^2 + \cdots 2^{\log_2 \mathrm{N} - 1} - (\log_2 \mathrm{N} - 1) \Rightarrow \\
\mathrm{S} &= 2^0 + 2^1 + 2^2 + \cdots 2^{\log_2 \mathrm{N} - 1} - \log_2 \mathrm{N} \Rightarrow \\
\mathrm{S} &= 2^{\log_2 \mathrm{N}} - 1 - \log_2 \mathrm{N} \Rightarrow \\
\mathrm{S} &= 2^{\log_2 \mathrm{N}} - (\log_2 \mathrm{N} + 1)
\end{aligned}$$

The propagation of signals to all the tree of muxltiplexers is following that of comparators. The colors of the *select* signals at each multiplexer in Figure 4.5 pinpoint the arrival of signals. The fan-out of $choose_0$ increases linearly from the leaves to the root for the comparators tree. This is a disadvantage of the circuit, but it will be solved later.

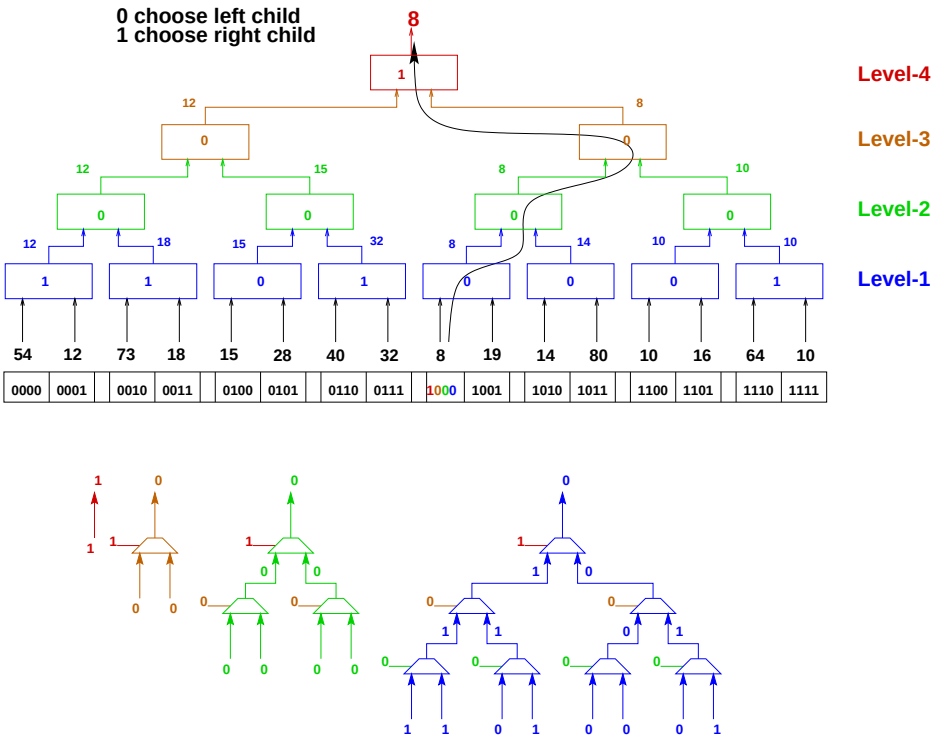


Figure 4.5: An example of locating the address of the winner element.

The output from *Locate the addressof the minimum* block is stored in a register, in order to be used for addressing the SRAM, at the next pipeline stage. In addition, it feeds the *Decode winners address*. The output from this block is stored in a register and it will be used at the next clock cycle.

The next block of Figure 4.4 discussed is the *N 2-element comparators*. This block contains comparators used for the detection of flows with NSTs smaller than current time. Every comparator accepts as inputs one of the N elements and the minimum from the root of the tree. In addition, the *decoded_address* register feeds this block. The purpose of this is simple. The flow served at the previous cycle updates its NST at the stage 2 of the pipeline. But the non-updated NST is still at the register. A comparison performed with the winner, probably will find this flow left behind in time and probably eligible. The next step is to put this flow at current "time" and serve it again, which is unacceptable. The output of the block is *old_flags* (one bit per flow), indicating which flow is left behind in time. The comparators of this block are the same with the one shown in Figure 3.7 without the "result" part.

The last block at stage 1 of the pipeline is the *In/Out from schedule*. This is responsible for the insertions and extractions of flows in schedule. It performs simple but significant tasks. Inputs to that block are the decoded_address and the old_flags calculated in the

previous cycle, among with the *ready_bits*. The latter shows which flows are active. The outputs of this block are the *out_flags*. All inputs and outputs are N-bits wide. The calculations performed in this block are the same for every bit. The out_flag for a flow is asserted if it is "not ready" or if it was served in the previous cycle. Thus, 3 states occur for one flow: i) do not participate at all to the contest, ii) participate with its NST (NST ¿ current "time" and iii) participate with the current "time" as NST (NST ¡ current "time"). Table 4.1 shows the states that may occur for a flow i in the scheduler.

$state_i$	$decode_address_i$	old_flag_i	$ready_bits_i$	out_flag_i	$choose_min_i$
Ineligible NST>current_time	0	0	0	1	0
Eligible NST>current_time	0	0	1	0	0
Ineligible NST<current_time	0	1	0	1	0
Eligible Reinsert at current time	0	1	1	0	1
Last in queue packet served	1	0	0	1	0
Served and reinserted	1	0	1	1	0
X	1	1	0	x	x
X	1	1	1	x	x

Table 4.1: The possible states for a flow in the scheduler.

The equations for the out_flag is

$$\mathrm{out_flag_i} = \overline{\mathrm{ready_i}} + \mathrm{decode_address_i}$$

The use of out_flag needs a little explanation. A pair of those signals is input to the 2-element comparators *at the leaves*. Keep in mind that the 2-element comparators have two carries with input condition "A equals B"(Table 3.3). Assume two flows i, j at the inputs of the same k-bit 2-element comparator. Thus, the out_flag signals change this condition according their values, as shown in Table 4.2 .

	out_flag_i	out_flag_j	$choose_{k-1}$	$found_{k-1}$
i-in, j-in	0	0	0	0
i-in, j-out	0	1	0	1
i-out, j-in	1	0	1	0
i-out, j-out	1	1	0	0

Table 4.2: Apply the out_flag to the initial condidtion of the 2-element comparator.

The equations for the two signals are

$$\mathrm{choose_{k-1}} = \mathrm{out_flag_i} + \overline{\mathrm{out_flag_j}}$$

$$found_{k-1} = \overline{out_flag_i} + out_flag_j$$

The out_flag signal is propagated from the leaves to the root, accompanying the element value. Any parent 2-element comparator accepts those flags from its childs. Eventually, the path towards the root, for elements with asserted out_flags, is short. In the common case they stop not far from the leaves.

The last block discussed from stage-1 is the *still ready*. This block guards the eligibility of the updated NST, which is compared with the new winner element. Assume a flow with small FSI and one the last packet in the queue and next to be served. After updating the NST at stage-2 it will be compared with the new winner element in order to reinsert in schedule. But according to the ready_bits arrived at stage-1 this flow must not participate to the contest, since its last packet was served. This information is not available to the comparator at stage-2, which choose wrong the flow with no packets (and small FSI) to be served again. The *inputs* of the block are the ready_bits and the decode_address bits. Those are used for bit mask to the ready_bits to isolate the corresponding bit of the flow. After that a tree of OR gates is used, in order to confirm or not the existence of one 1. The *output* of this block is the *still_ready* flag. This operates to the initial condition of the 2-element comparator analogous to the out_flags.

The main block of stage-2 is *SRAM*. It is accessed at every cycle and gives results to the *Adder* block. This single-port SRAM has a single address port and separate data input and data output ports. Read and write cycles are timed with respect to a single edge of the clock. During both read and write cycles, the write enable (WEN) and cell enable (CEN) inputs are sampled by the rising edge of the clock. The data out bus has an asynchronous 3-state output enable control (OEN). A multiplexer at the input of the memory selects the address according to the operation the scheduler performs. The writes in the SRAM, which occurs every time a new connection is established, is a two-cycle operation for the interface. Initially the *new_flow*, *new_flow_address* and *new_FSI* signals are latched by corresponding registers. Thus the output of the multiplexer (at the memory input) is prepared and latched at the positive edge of the next cycle. Thus, consecutive writes to the memory are permitted. During write operation, although the previous winner's NST is not updated, the multiplexer which calculates *winner*, outputs a result. The latter is not valid, so the signal *serve* has to be deasserted. In addition, at the next cycle there is nothing to load to the NST registers. After write completes, scheduler's output *winner* is valid at the next cycle.

Adder block is used to update the NST of the served flow. The output of the adder is used to perform write back to the NST registers for the served flow. In addition, it

feeds a *2-element comparator* block. The other input of the latter is the current minimum element. Thus, reinsertion of the updated NST is performed by this block which compares the current minimum with the previous one updated. The output of the 2-element comparator is a select signal to a multiplexer, which outputs the index of the winner flow. What happens if the updated NST is smaller than the new minimum of the remained flows? There are 2 solutions: i) put a multiplexer at the input of the *minimum* register to select between the new minimum and the previous updated minimum, with select signal the *choose_flow* and ii) store the new NST and the choose_flow in registers in order to be used in the next cycle, and put a multiplexer at the output of the *minimum* register to with select signal the stored *choose_flow*. The solution chosen is the second. Although the first solution takes advantage over the second in area and power, loses in delay. Instead, the solution followed removes delay from the stage-1 to the stage-2.

At this section, the main operations of the datapath analyzed. A first approach to the WWR disciplines implementation is introduced. In the next section more features are added to the design and solutions to a couple of arrised problems are proposed.

4.6 Adding more Features

The datapath presented in the previous section, does not complete the design of the scheduler. The *flow_not_served* and *stop* signals must be added to the design. The updated datapath is shown in Figure 4.6

The chosen flow to be served from the scheduler is not always served by the switch. Scheduler uses *flow_not_served* signal as input to make appropriate operations. The first one is not to load the updated NST of the "not served" flow. The second is to reinsert this flow into the schedule with the previous NST (not the updated) by comparing with the new minimum. Thus, a new 2-element comparator block is added to datapath of Figure 4.4 . A multiplexer selects the between the two output signals of the 2-element comparators. This signal arrives near the end of the cycle period, because the queue manager wants time to perform some operations before desiding to evaluate.

The *stop* signal is used to stop the scheduler from performing any operations. This is used to minimize the power consumption of the circuit. This is mainly achieved by stopping the operation of the tree. It prevents from writing back the NST and loading new ready_bits.

Figure 4.6: The datapath of the WWR scheduler

4.7 Optimizations for Fan-out

After designing the datapath some observations should be done in terms of delay, area and power. The critical path of the circuit must be detetermined and optimized.

As mentioned earlier the cycle period is determined by stage-1. The critical path passes through the binary tree of comparators and througn the comparators at stage-2 and finishing at output multiplexer. The most time-consuming block is the binary tree. However, significant delay is introduced by the N multiplexers which are drived by the *previous_minimum* signal, due to large fan-out(1 to N). In addition, large fan-out (1 tp N) is introduced at the root of the tree. *Minimum* is driving the N comparators in order to find which flows are left back in time. The proposed solution deals with the binary tree. The idea is to increase the number of root 2-element comparators by *inversing the tree*. However, this must be done carefully, in order to get significant improvement in delay. This is a trade-off between area, power and delay. An **invariant condition** must be satisfied for any block in the structure:

The fan-out of any block in the tree must be less or equal to 2.

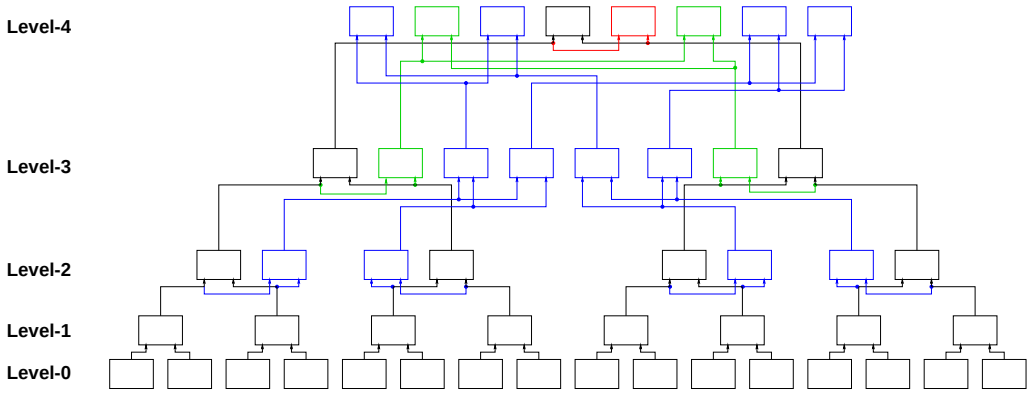


Figure 4.7: An example of reducing the fanout of the root. 2-element comparators from Level-1 to Level-3 have fan-out 2.

The implementation is shown in Figure 4.7 . The black blocks are the boxes of the binary tree while the colored are those added. The construction of this architecture is done, following a sequence of steps indicated by the color of the blocks. Initially, the red box is added nearby the root. Thus, the fan-out of the root 2-element comparators is reduced to N/2 and the fan-out of their children is increased to 2. Thereafter, the 2 green blocks are added, initially to Level-3. In order to benefit from the invariant condition, 4 green blocks are added to Level-4. Thus, the blocks at Level-2 and Level-3 have fan-out 2. The root blocks have fan-out N/4. In the same way the blue blocks are inserted in

the architecture. The fan-out of the root blocks is N/8. The extra area added and the fan-out of the root block are given by the formulas

$$\text{area} = \sum_{i=1}^{\log_2 N} i \cdot 2^{i-1}$$

$$\text{fanout} = K/2^i$$

where $\log_2$N is the height of the tree and K the fan-out of the root element. Index i is "running" from the root to the leaves(e.g. i=1 is the root-level, i=2 is next level below the root).

Inserting such a tree in the scheduler yelds some interesting opportunities for decreasing the fan-outs of the multiplexers. By increasing the number of registers storing the minimum and the multiplexers in front of them, the number of identical *previous_minimum* is increased. Thus, the fan-out if this signal is decreased. Further more, by increasing the number of previous_minimum signals, the number of adders can be increased. Thus, the number of identical new_NST signals is increased. Thus, the fan-out of this signal is decreased. However, increasing the number adders costs an increase to the fan-out of the memory. This fan-out does not affect the cycle period, because it is not in the critical path. On the other hand, the designer should be very careful not to overload any signal and finally actually decrease the performance.

4.8 Economizing on Power

Another observation that uncovers one advantage of the circuit, deals with the power consumption. The latter was the main disadvantage of the binary tree of comparators because of the recalculation of the minimum from the begining at every iteration. Despite this fact, keep in mind that the cycle period is few ns. This deals with the reinserions of new elements to the contest. Assume a 4-level binary tree. A possible input configuration is shown in Figure 4.8 .

The winner's value stored in the register at the leaves is updated with the new NST = 16. This will activate only the 2-element comparators, which are on the path followed by the previous value 8. Any other comparator will remain inactive! Thus, for every reinsertion only $\log_2$N comparators will calculate. This is the common case for the scheduler. However, multiple insertions may occur by inserting "old flows" in the schedule at the same cycle. This will activate more than $\log_2$N comparators for one cycle. Anyway, the mean value of active comparators is near $\log_2$N. This is not the case for the 2-element comparators calculating the old_flags. All of them are active every time the minimum changes. Thus this is the most power-consuming block of the datapath.

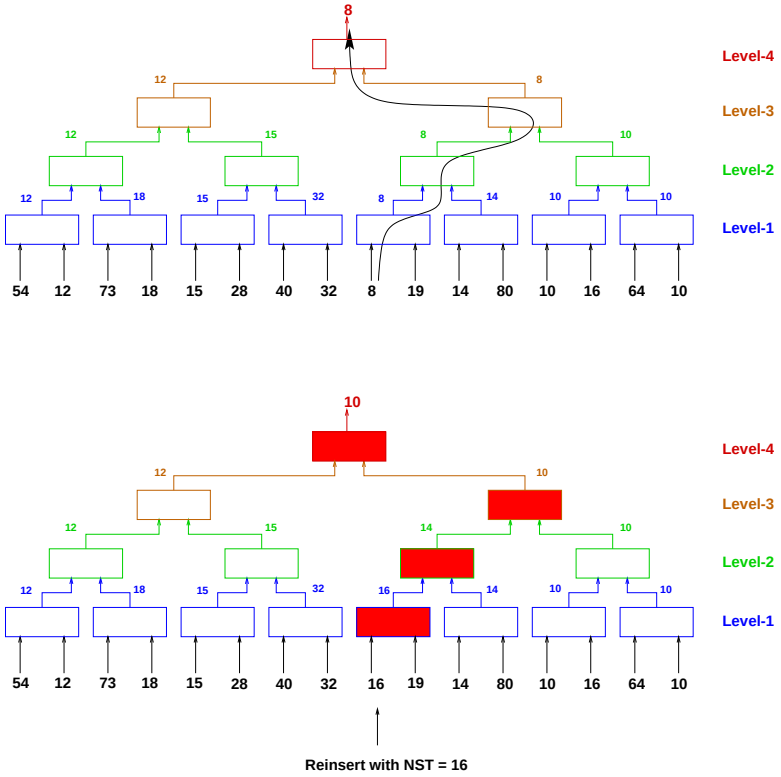


Figure 4.8: The power consumption after updating one element. The active 2-element comparators at this cycle are highlighted.

4.9 Conclusions on Datapath Design

The WWR disciplines define certain requirements for a scheduler. Priority queue data structure together with per-flow queueing can implement those requirements. The designed datapath performs the proper operations to the priority queue elements and scheduler produce one valid output per cycle. Problems ocurred with large fan-out were faced satisfyingly decreasing the cycle period. Furthermore, the power consumption of the binary tree is proofed to be less than estimated initially. At the next section the design flow is analysed and synthesis results are presented.

Chapter 5

Design Flow

The design flow of a circuit plays significant role to the final performance. Many design flows are proposed for verification and synthesis. The one used here contains several steps for power and delay optimization. Figure 5.1 shows the design flow used for this circuit. The design of the circuit is segmented in two parts. Initially the choices of binary tree of comparators are investigated. Afterwards the scheduler is synthesised with fixed parameters N-number of flows and k-number of bits per element. This design flow is used for both parts.

The pattern generator concerns only the scheduler. The C code for the binary tree comparator is very simple. In addition, it creates the number of elements for input to the verilog code. Those elements are written to file that the verilog code uses as input. In contrast to that, the test vectors for the scheduler are more sophisticated, in order to examine the most of the cases that may occur. A describes the pattern generator and the interface to the verilog and C code.

The synthesis block in the previous figure hides enough information. This is shown in Figure 5.1 . The synthesis flow followed is using gate level simulation for capturing the switching activity of the nodes. This helps the synthesis tool to perform more accurate power consumption calculations.

In the bottom-up strategy used, individual subdesigns are constrained and compiled separately. After successful compilation, the designs are assigned the *dont_touch* attribute to prevent further changes to them during subsequent compile phases. Then the compiled subdesigns are assembled to compose the designs of the next higher level of the hierarchy, and these designs are compiled. This compilation process is continued up through the hierarchy until the top-level design is synthesized. This method permits to compile large

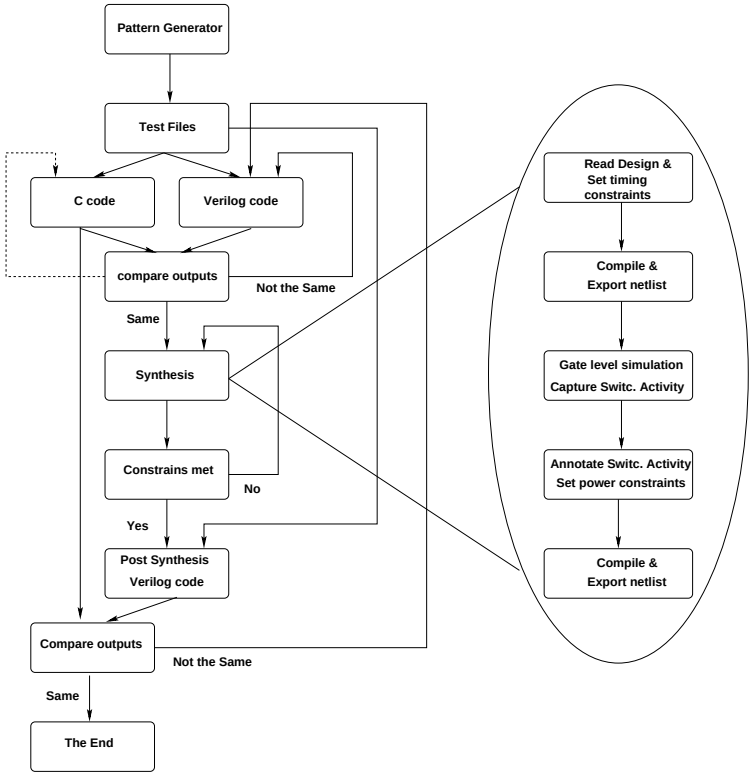


Figure 5.1: The general design flow followed

designs because Design Compiler does not need to load all the uncompiled subdesigns into memory at the same time.

5.1 Technology used for Synthesis

The CAD tools used for verification and synthesis were Cadence Verilog-XL and the Synopsys tools. The technology and library used for implementing the design were provided via Europractice by Virtual Silicon Technology, Inc. (http://www.virtual-silicon.com): umcl18u250t2 library, for 0.18um CMOS technology.

5.2 Binary Tree of Comparators

The binary tree of comparators, as shown in Section 3 , can be designed in three ways, according to the implementation of the 2-elements comparator: ripple, carry select and carry look ahead. The parameters for these designs were the number of elements (N: N$\in\{2, 4, 16, 64, 256\}$) and the number of bits per element (k: k$\in\{8, 16, 24\}$). The delay constraints initially set for the tree was 10ns.

Initially, the 2-bit comparator cell was synthesised. Afterwards, the 2-element comparator, which uses the synthesised 2-bit comparator cell was mapped to gates. At this level is necessary to start using the Power Compiler for the power optimization of the 2-element comparator. This interference at early stages of synthesis produces better results than late ones.

5.2.1 Delay Results

The results for the three implementations of the 2-element comparator is shown in Figure 5.2 .

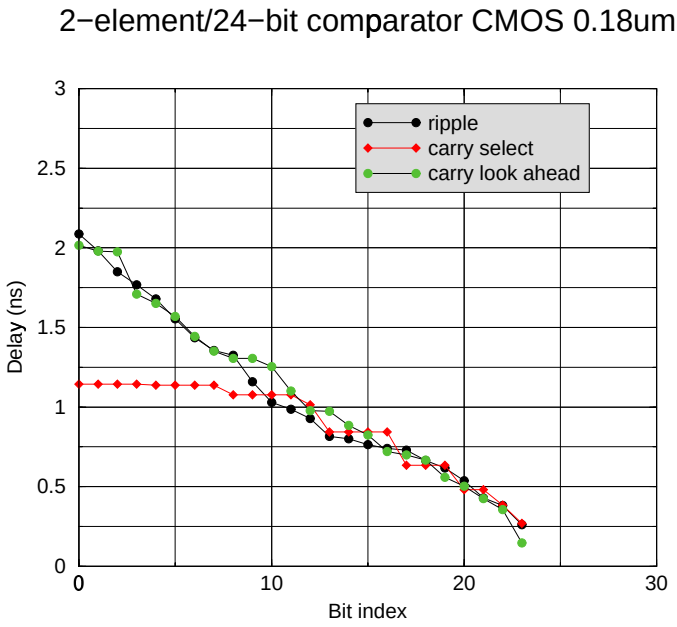


Figure 5.2: Delay comparison of 2-element comparator circuit.

The carry select implementation appears to have better performance than the others. In Figure 5.2 , 24-bit comparators are used. As proved earlier, the carry select implementation is faster than ripple and carry select. The performance of the carry look ahead topology is poor, because of the large fanout. The latter increases linearly from the MSB to the LSB, thus the delay results approximately linear. The next thing to do is to comparare for the binary tree. Section 5.3 - Figure 5.5 shows delays for the values of the width and the number of elements. Initially 8-bit elements will be used. Results are shown in Figure 5.3 . The number of bits is small, so the carry select comparator will not have superior performance related to the other implementations. For small number of elements performance is the same for all topologies. As number grows, the carry

look ahead is winning. This happens because the number of bits of the elements is small, resulting to small fan-in and fan-out for carry look ahead.

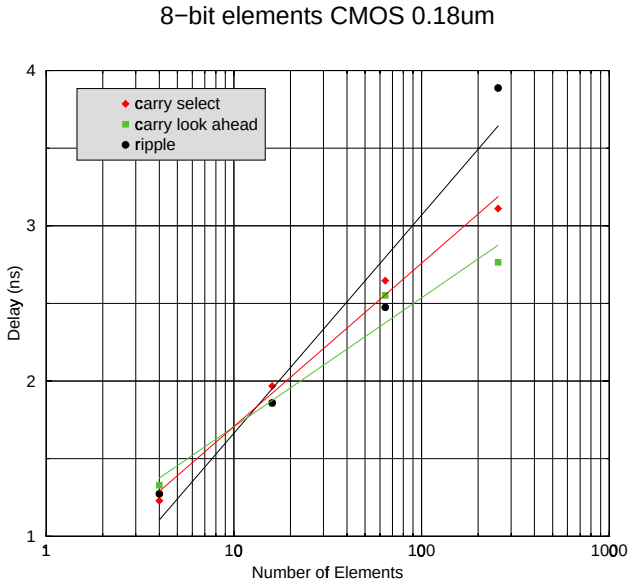


Figure 5.3: Delay comparison for binary tree with 8-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

Afterwards, 16-bit elements is used for the binary tree. Results are shown in Figure 5.4 . Carry select has better performance than other two. The element width is quite enough for the nice features of the comparator to be shown. Ripple and carry look ahead compete each other. The former is a little more fast than the latter in many numbers. Finally, the delay to calculate the minimum among many 24-bit elements is measured. The results are shown in Figure 5.5 . The carry select topology is faster than the other two. The delays of ripple and carry select are similar.

Concluding for the delays of the binary tree, the carry select comparator has better performance than the ripple and the carry look ahead in terms of delay. The delays increase logarithmicaly with the number of elements. This behaviour is inherited from the binary tree.

5.2.2 Area Results

In Figure 5.6 - Figure 5.8 area results are shown for the binary tree. Area increaes linearly with the number of elements. Area is measured in number of cells. This is more accurate than mm^2, because things change after place and root. There is a compact topology for the place and root of large binary trees, proposed in [YoSi89]. Thus, the impact of wires on the area of the circuit is limited. However, the synthsis tool does not include

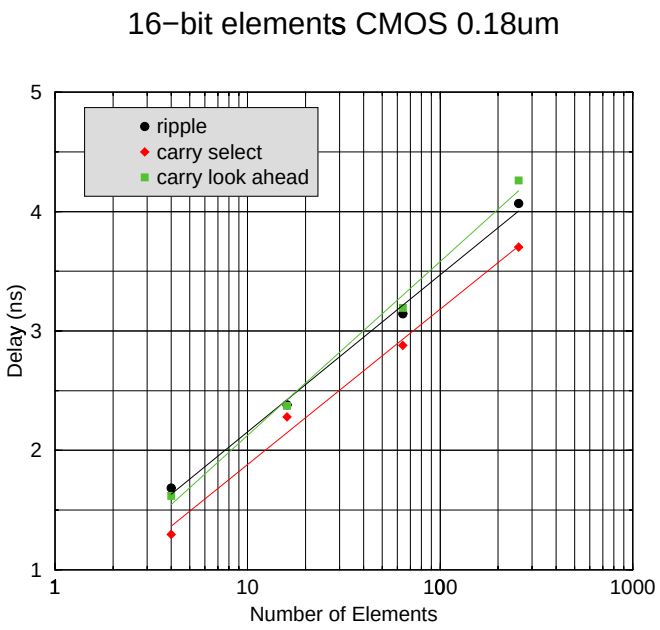


Figure 5.4: Delay comparison for binary tree with 16-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

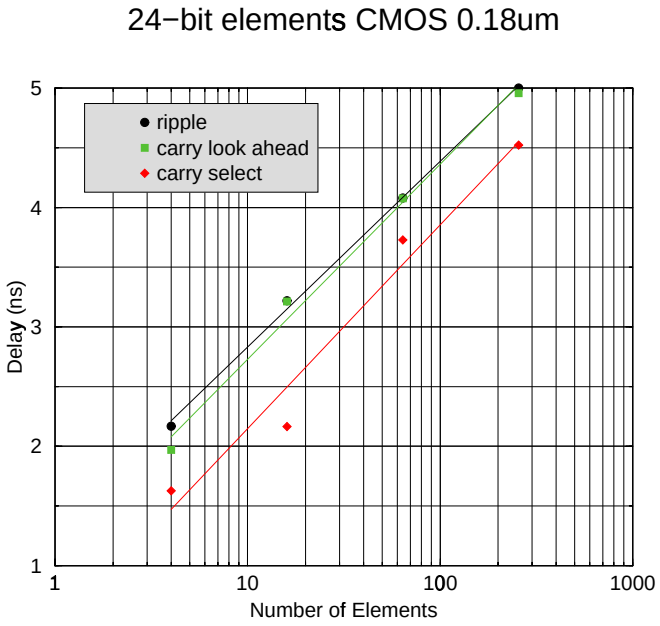


Figure 5.5: Delay comparison for binary tree with 24-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

the area of wires in the calculations. The area for 24-bit 256-elements ripple topology is approximately 0.75mm^2.

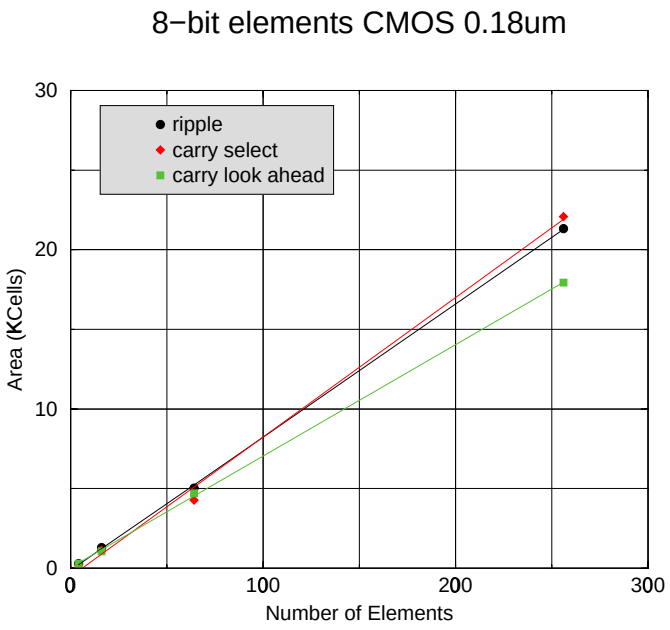


Figure 5.6: Area comparison for binary tree with 8-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

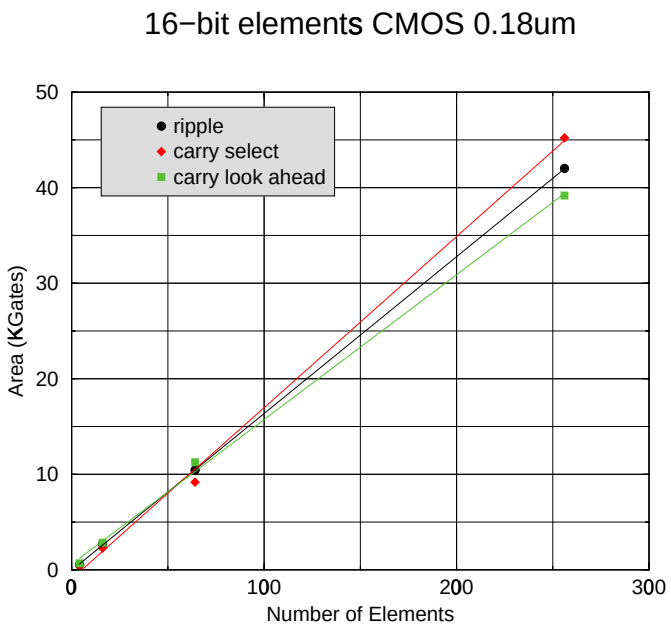


Figure 5.7: Area comparison for binary tree with 16-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

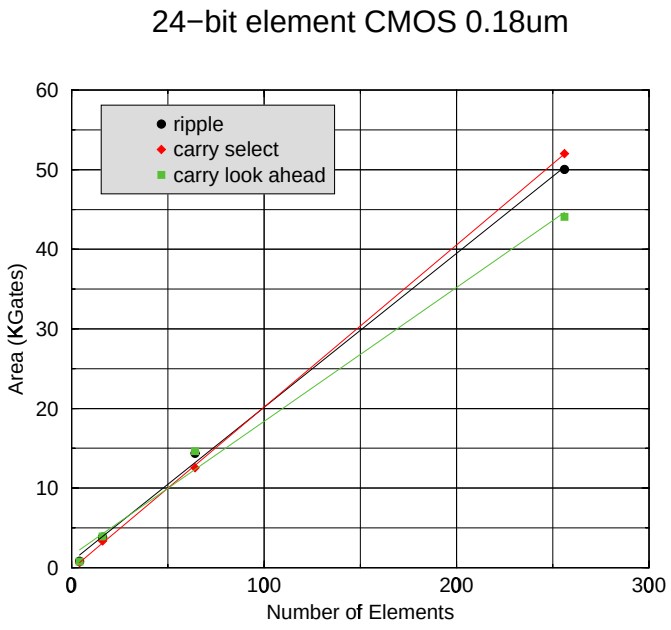


Figure 5.8: Area comparison for binary tree with 24-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

5.2.3 Power Results

This section presents synthesis power results. The power analysis sequence is explained in details in B. The values shown in the figures are close to the worst case. Initially, zeros introduced at the inputs of the binary tree comparators. At the next cycle, a non-zero values set of elements is introduced at the inputs. Those values decrease gradually from the maximum-value element. For k-bit numbers the maximum value is 2^{k-1}. Thus, for N elements the range of values is [2^{k-1},2^{k-1}-N]. The purpose is to activate as much nodes of the tree as possible.

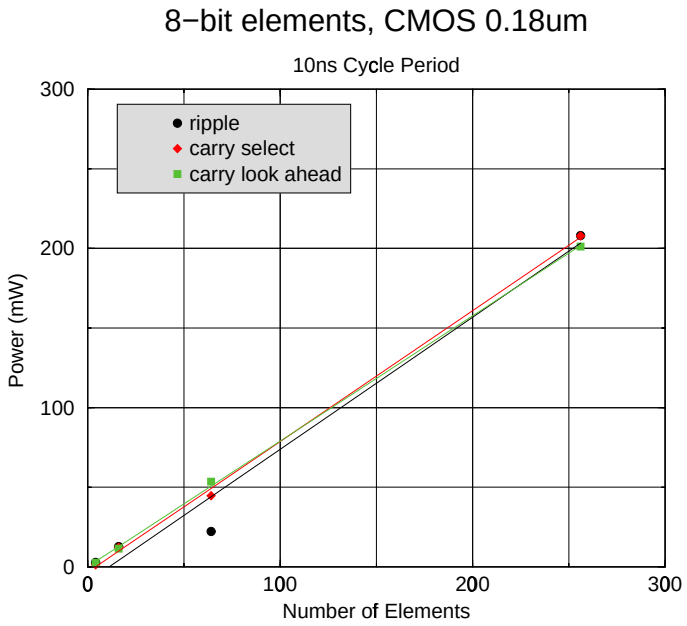


Figure 5.9: Power comparison for binary tree with 8-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

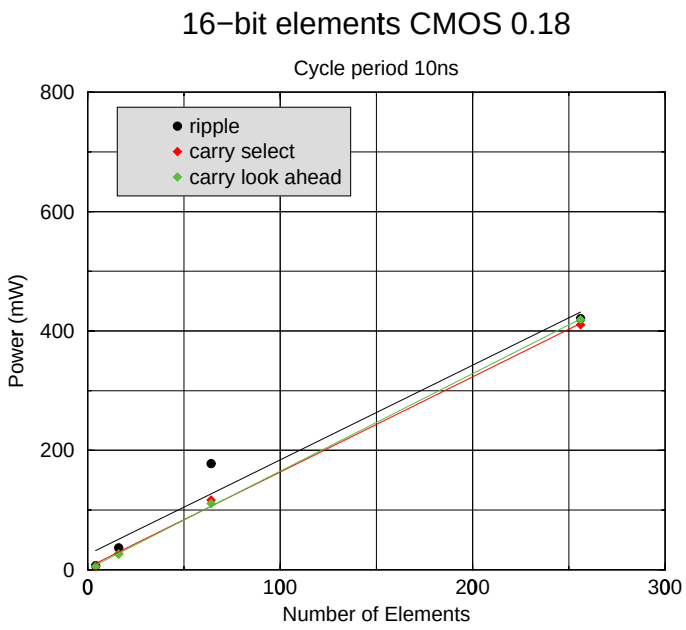


Figure 5.10: Power comparison for binary tree with 16-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

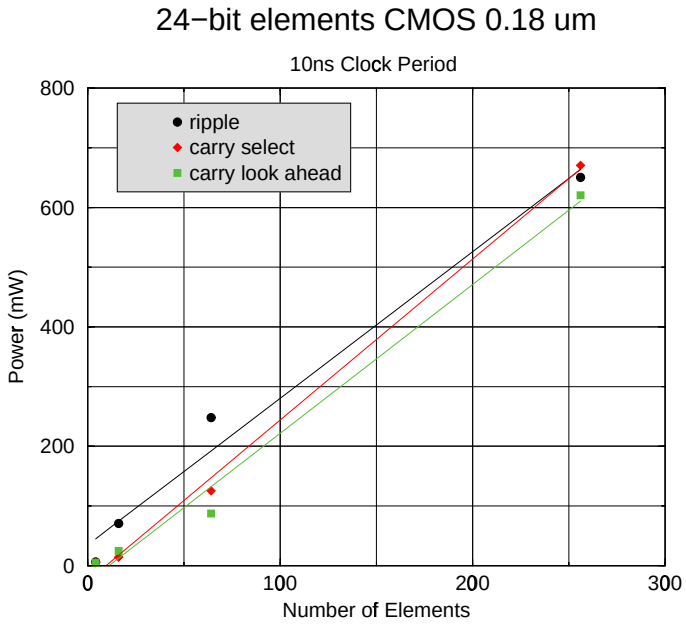


Figure 5.11: Power comparison for binary tree with 24-bit elements. Three topologies are used for the 2-element comparator: ripple, carry look ahead and carry select.

Chapter 6

Conclusions

The motivation for this work was to design a new hardware structure to support high-speed operation in a priority queue, where the set of eligible flows changes arbitrarily fast. The approach followed was to find the minimum (or maximum) of an arbitrary (non-sorted) set of elements in a non-sorted set. In order to do this, a fast algorithm, where the minimum (or maximum) is found by a binary tree of comparators is proposed. We developed an innovative organization for the tree, where signals are propagated across each 2-element comparator as well as the tree levels, at the same time; in this way, the delays of the individual comparators and the delays of the tree levels are placed in parallel, rather than in series. A carry select 2-element comparator is designed and optimized for each level of the tree. The complexity of such a structure is sublinear inherited from the carry select. The number of elements, which participate at every comparison, is easily changeable at every iteration of the algorithm.

Synthesis results show that the delay of the binary tree, where carry select comparators are used, is less than 5 ns for 256 elements for a 0.18 CMOS process. The same results show that carry select topology for the binary tree has superior performance than the ripple and the carry look ahead. The property of parallelism is proved by synthesis results. In addition, synthesis results show nice performance in terms of array and power consumption.

The binary tree of comparators is the heart of a weighted-round-robin scheduler that we designed. The scheduler produces a valid output at every cycle. Synthesis results for the tree shows that the clock cycle of the scheduler is 6 ns for a 256-element scheduler.

Appendices

Appendix A

Pattern Generator

Pattern generator is written in C (1800 lines). Initially the name of the test must be provided by the user. Afterwards, the user chooses among three general paths.

The first is named *Explicit Definition of Sequence* and it can be used for the generation of tests for extreme and special cases. Furthermore, it can be used at the initial stages of the design, where simple test vectors are used. The instructions are generated in specific order, set by the user. The second path is named *Explicit and Random Sequence*. There, the user can select only the cases it is interested for and randomise the appearence of others (e.g. all flows "ready"). This is used to test the correlation between cases, namely if the scheduler executes properly one specific operation, in the presence of different environment. The sequence of instructions is random. The third path is called *Random sequence*. Here the generator outputs a random sequence of instructions. This is used for the generation of long tests.

After choosing a path, the user is called to select the instructions to be produced. Those are 5: reset, add one flow, do not serve scheduled flow, stop and continue. The total number instructions to be generated, is requested. Furthermore, the percentage of each instruction is requested, because every one has a certain field set and the user can randomise the others or fill them with the wanted bits or both. The set of different versions of the same operation produced is fractioned by the user(e.g. 50% add one flow with random ready bits and 50% add one flow with all ready bits asserted).

Finally, five files are produced, one per instruction field. The verilog top module reads those files and store them in "memories". In every cycle an index selects the fields from the "memories" to construct one instruction, which is presented to the inputs of the scheduler. The outputs of the scheduler are written in two files, in order to be compared with those of the C code, which also accepts as input the five test files.

Appendix B

Synthesis Scripts

The Verilog code is read and first compilation is done with the hierarchy preserved. The switching activity is captured using the following sequence.

** Grouped design**

- Read in the design.
- Export the SAIF file with dc_shell command rtl2saif before the compilation.
- Compile
- Use the saif2trace unix command to produce a trace file as input to the Simulator
- Simulate the design for the most common case of sequence of inputs. The Simulator exports a .vcd file.
- Use the vcd2saif unix command to produce a post simulation SAIF file.
- Import the post SAIF file to dc_shell using the command read_saif.
- Set power constraints
- Compile incremental based on the previous netlist.
- Write design .v , .db.
- Read again the post SAIF file and export reports for delay, power and area.

Faster design is produced when the ungroup command is used. Ungroup removes the hierarchy of the design and produces a flat circuit. The optimization is enhanced due to

the fact that the space of solutions is expanded. The synthesis tool has more choices to make. In addition the interface among modules is defined exactly.

Ungroup design

- Read the previous .db file.
- Ungroup the design.
- Compile.
- Export the SAIF file with dc_shell command rtl2saif.
- Use the saif2trace unix command to produce a trace file as input to the Simulator

A critical observation has to be mentioned here for the SAIF file that is produced at this stage of synthesis. The rtl2saif is said to be working only in pre-mapped design. Using a trick the output SAIF file can be used for gate-level power optimization. The SAIF files include information about the ports of a cell. They doesn't include any information about the structure of a cell. Thus, a change in the power optimization sequence permits the more accurate gate-level power optimization. The SAIF file now contains the instances of the cells of the technology library is used. The synthesis sequence is continued by writing a post synthesis .v file. This file is used by the simulation tool together with the SAIF forward annotation file.

- Write .v post syntheis file
- Simulate the design for the most common case of sequence of inputs. The Simulator exports a .vcd file.
- Use the vcd2saif unix command to produce a post simulation SAIF file.
- Import the post SAIF file to dc_shell using the command read_saif.
- Set power constraints
- Compile incremental based on the previous netlist.

At this point the forward annotation SAIF file has to be rewritten due to the fact that the cells has changed after the compilation.

- Export the SAIF file with dc_shell command rtl2saif.

- Use the saif2trace unix command to produce a trace file as input to the Simulator
- Simulate the design for the most common case of sequence of inputs. The Simulator exports a .vcd file.
- Use the vcd2saif unix command to produce a post simulation SAIF file.
- Read again the post SAIF file and export reports for delay, power and area.
- Write design .v, .db.

The accuracy of this synthesis flow is dependent only on the simulation and the capturing of the activity of the nodes in the design. This is a useful flow if the interface verilog_toggle for gate-level optimization is not available.

Bibliography

[CK02] N. Chryssos, M. Katevenis, *"Weighted Max-Min Fair Scheduling for a Crosspoint-Buffered Crossbar"*, M.Sc. Thesis, Computer Science Department, University of Crete, April 2002 (under preparation).

[IK01] A. Ioannou, M. Katevenis, *"Pipelined Heap (Priority Queue) Management for Advanced Scheduling in High-Speed Networks"*, Proc. IEEE International Conference on Communications (ICC' 2001), Helsinki, Finland, June 2001.

[GM99] P. Gupta, N. McKeown: *"Designing and Implementing a Fast Crossbar Scheduler"*, IEEE Micro, Jan.-Feb. 1999, pp. 20-28.

[KaSM97] M. Katevenis, D. Serpanos, E. Markatos, *"Multi-Queue Management and Scheduling for Improved QoS in Communication Networks"*, Proceedings of EMMSEC'97 (European Multimedia, Microprocessor Systems, and Electronic Commerce Conference), Florence, Italy, Nov. 1997, pp. 906-913; http://archvlsi.ics.forth.gr/muqpro/classSch.html

[Kat01] M. Katevenis *"CS-534: Packet Switch Architecture"*, lectures, Fall 2001 http://archvlsi.ics.forth.gr/~kateveni/534/

[Kes97] S.Keshav, *"An Engineering Approach to Computer Networking"*, Addison Wesley, 1997, ISBN 0-201-63442-2.

[KKVK97] G. Kornaros, C. Kozyrakis, P. Vatsolaki, M. Katevenis, *"Pipelined Multi-Queue Management in a VLSI ATM Switch Chip with Credit-Based Flow Control"*, Proc. 17th Conf. on Advanced Research in VLSI (ARVLSI'97), Univ. of Michigan at Ann Arbor, MI USA, Sep. 1997, pp. 127-144; http://archvlsi.ics.forth.gr/atlasI/atlasI_arvlsi97.ps.gz

[MRS00] S. Moon, J. Rexford, K. G. Shin, *"Scalable Hardware Priority Queue Architectures for High-Speed Packet Switches"*, Trans. on Computers, vol. 49, No 11, November 2000

[Rab96] Jan M. Rabaey, *"Digital Integrated MRS Circuits, A Design Perspective"*, Prentice Hall, Inc., 1996.

[Tho83] C. D. Thompson, *"The VLSI complexity of sorting"*, IEEE Trans. Computers, vol. C-32, pp. 1171-1184, 1983.

[VM93] M. Vai and M.M. Moy, *"Real-time maximum value determination on easily testable VLSI architecture"*, IEEE Trans. Circuits System I, vol. 40, pp. 283-285, Apr. 1993.

[YoSi89] H. Y. Young, A. D. Singh, *"On Implementing Large Binary Tree Architectures in VLSI and WSI"*, IEEE Trans. on Computers, vol. 38(4), pp. 526-537, 1989.

[Zha95] H. Zhang, *"Service Disciplines for Guaranteed Performance in Packet Switching Networks"*, Proceedings of the IEEE, vol. 83, no. 10, Oct. 1995, pp. 1374-1396.