FORTH-ICS/TR-339 July 2004

Design of a 32x32 Variable-Packet-Size Buffered Crossbar Switch Chip

Dimitrios G. Simos

Abstract

Switches and routers are the basic building blocks of most modern interconnections and of the Internet, aiming at providing datapath connectivity, while solving output contention, the major problem of distributed, multi-party communication. The latter is accomplished through buffering, access control, flow control, or datagram dropping. Modern high-end switches are called upon to provide aggregate throughputs in the terabit per-second range, which greatly challenges both their architecture and implementation technology.

The aim of this work is to prove the feasibility of a novel buffered crossbar organization, operating directly on variable-size packets. Such operation, combined with distributed scheduling, removes the need for internal speedup, thus fully utilizing the incoming throughput.

We proved the feasibility of this novel architecture by fully designing such a 32x32 buffered crossbar, in the form of an ASIC chip core, providing 300 *Gbit/sec* of aggregate bandwidth in 0.18 μm technology, or higher throughput in more advanced technologies. The design was synthesized, placed, and routed, using a hierarchical ASIC flow, resulting in a 420 mm^2, 6 Watt core in 0.18 μm CMOS technology. In 0.13 μm CMOS, area would be reduced to 200 mm^2, and power consumption to 3.2 W. Power estimation showed that the majority of power is consumed in driving cross-chip wires, while memories and logic are minority consumers.

Hierarchical ASIC flows are difficult to use, but became necessary due to the large size of the design. We present the detailed system design (block diagrams as well as critical circuit details), followed by a detailed description of the design flow, including its numerous intricacies and the lessons that we learnt. In particular, we describe the choice of a hierarchy that is appropriate for effective placement, routing, and timing behavior. The final placement and routing showed that the synthesis tool had underestimated the design area by 30%, due to the dominance of long (end-to-end) wires in this design.

Design of a 32x32 Variable-Packet-Size Buffered Crossbar Switch Chip

Dimitrios G. Simos

Computer Architecture & VLSI Systems (CARV) Laboratory
Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)
Science and Technology Park of Crete
P.O. Box 1385, Heraklion, Crete, GR-711-10 Greece
Tel.: +30-81-391660 Fax: +30-81-391661
email: simos@ics.forth.gr

Technical Report FORTH-ICS/TR-339 – July 2004

Work performed as a M.Sc. Thesis at the Department of Computer Science, University of Crete, under the supervision of prof. Manolis Katevenis

Abstract

Switches and routers are the basic building blocks of most modern interconnections and of the Internet, aiming at providing datapath connectivity, while solving output contention, the major problem of distributed, multi-party communication. The latter is accomplished through buffering, access control, flow control, or datagram dropping. Modern high-end switches are called upon to provide aggregate throughputs in the terabit per-second range, which greatly challenges both their architecture and implementation technology.

The aim of this work is to prove the feasibility of a novel buffered crossbar organization, operating directly on variable-size packets. Such operation, combined with distributed scheduling, removes the need for internal speedup, thus fully utilizing the incoming throughput.

We proved the feasibility of this novel architecture by fully designing such a 32x32 buffered crossbar, in the form of an ASIC chip core, providing 300 $Gbit/sec$ of aggregate bandwidth in 0.18 μm technology, or higher throughput in more advanced technologies. The design was synthesized, placed, and routed, using a hierarchical ASIC flow, resulting in a 420 mm^2, 6 Watt core in 0.18 μm CMOS technology. In 0.13 μm CMOS, area would be reduced to 200 mm^2, and power consumption to 3.2 W. Power estimation showed that the majority of power is consumed in driving cross-chip wires, while memories and logic are minority consumers.

Hierarchical ASIC flows are difficult to use, but became necessary due to the large size of the design. We present the detailed system design (block diagrams as well as critical circuit details), followed by a detailed description of the design flow, including its numerous intricacies and the lessons that we learnt. In particular, we describe the choice of a hierarchy that is appropriate for effective placement, routing, and timing behavior. The final placement and routing showed that the synthesis tool had underestimated the design area by 30%, due to the dominance of long (end-to-end) wires in this design.

Acknowledgments

This work was performed at and financially supported by the Institute of Computer Science (ICS) of the Foundation for Research & Technology - Hellas (FORTH), Heraklion, Crete, Greece, partly through project 002075 "SIVSS" of the European Union FP6 IST Programme. The CAD tools were provided by the University of Crete, through Europractice.

Besides these organizations, I would also like to thank all those people who helped me throughout this work. First of all, I would like to thank my supervisor, prof. Manolis Katevenis for defining the chip architecture after originally observing that a buffered crossbar can operate directly on variable-size packets. Also, more importantly, his priceless suggestions on organizing and carrying out a project, and his simple yet scientific way of thinking and teaching, became valuable guides for me during the last few years. I deeply thank him for that.

I would also like to thank the rest of the Packet Switch Architecture group at ICS-FORTH for their help: Dr. Ioannis Papaefstathiou and prof. Dionysios Pnevmatikatos for their suggestions and observations on all the design stages of the chip –their experience in the development of large projects proved invaluable; Nikolaos Chrysos and Georgios Passas, for our conversations and their help during the last 2 years; Giorgos Kalokairinos for his support.

Also, I give my thanks to Dr. Christos Sotiriou for his suggestions on the back-end flow; Vassilis Papaefstathiou for our valuable hardware-oriented discussions during our breaks; and Michalis Ligerakis for his help.

I would also like to thank Dr. Ioannis Papaefstathiou, prof. Aggelos Bilas, Aggelos Ioannou, Nikolaos Chrysos, Olga Dokianaki, and Evriklis Kounalakis for their corrections on parts of this thesis report.

Thanks are also due to all my friends, especially to: Dimitris Zaharakis, Katerina Iskioupi, Thom Kleitsakis, Haris Kondylakis and Dimitris Christelis. Your support will never be forgotten.

I also thank my brothers, Panos and Vassilis, and, last but not least, my deepest thanks go to my parents, Giorgos and Sevasti, for their support all these years: nothing would have been accomplished without your contribution; I therefore dedicate this work to you.

Contents

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

One important feature of crosspoint queued switches, which had not received much attention until recently, is the ability to route *directly* variable size packets, without the need to segment them at the ingress and reassemble them at the egress; this feature, combined with other crosspoint queueing advantages, allowed us to design a fairly large (32×32) buffered crossbar switch operating at nearly 10 $Gbit/sec$ link speeds, with no internal speedup. The design was synthesized and placed and routed by following a standard hierarchical ASIC flow, which resulted in a 420 mm^2, 6 Watt chip in a $0.18\mu m$ technology. In this chapter we briefly present the most important Output Queueing architectures, along with their corresponding advantages and disadvantages. We introduce Crosspoint Queueing and we present previous work carried out in this field. Finally, we show roadmaps for current and future technologies which prove that large-scale crosspoint queueing will soon overcome its main drawback sofar, memory complexity.

1.1 Queueing Architectures

In order to introduce the reader to buffered crosspoint queueing, we first have to look into the various queueing architectures that have emerged up to now. Single-stage switches can be classified by their switching fabric and buffer architectures. The switching fabric is the physical connection within a switch between the input and output ports; it can be proved that all switches need a crossbar inside their switching fabric [1]. Usually packets need to be queued in buffers when short-term overloading occurs, where the sum of input rates for a single output port exceeds the outgoing link rate; hence, buffering characterizes all kinds of switches. In Figure 1.1 we can see a conceptual derivation and taxonomy of all types of queueing architectures, namely "Output Queueing (OQ)", "Input Queueing (IQ)", "Combined Input Output Queueing (CIOQ - Internal Speedup)", "Shared Buffer", "Block Crosspoint Queueing" and "Crosspoint Queueing (CQ)"; all architectures presented in this figure are being demonstrated on a 8×8 switch - 8 inputs and 8 outputs. In this introductory section, we will analyze OQ, IQ, CIOQ, CQ, and the not-illustrated combination of "Combined Input Crosspoint Queueing (CICQ)".

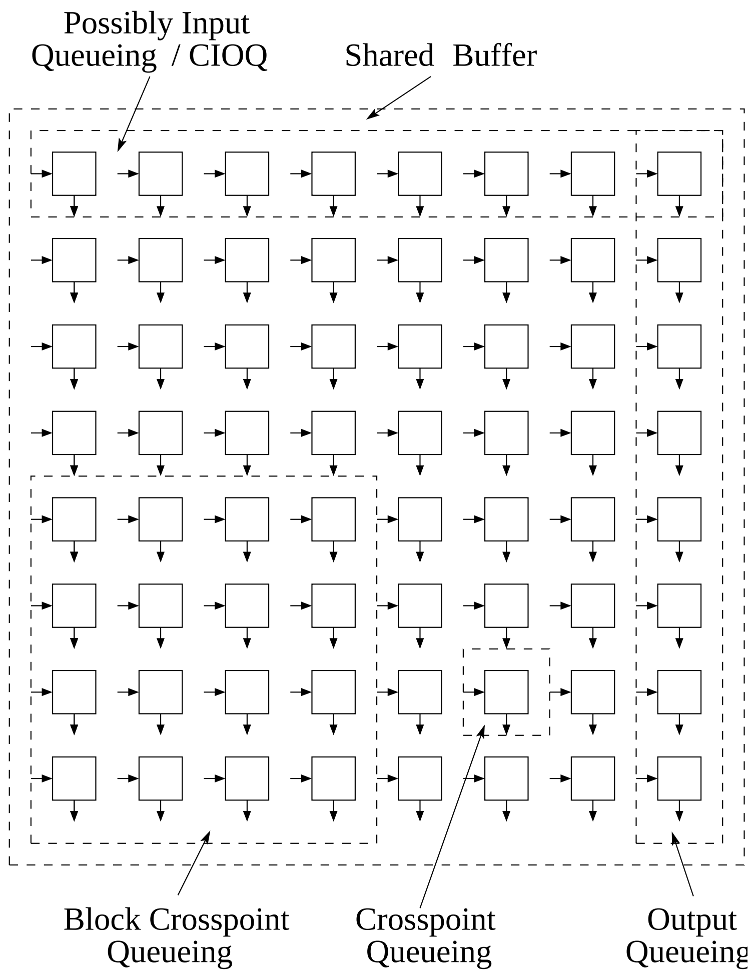

Figure 1.1: Conceptual derivation & taxonomy of Queueing Architectures.

Traditional OQ is depicted in Figure 1.2. It is the reference switch architecture, as it is capable of delivering the best possible performance achievable; this stems from the fact that, in the absense of head-of-line (HOL) blocking at the ingress and internal blocking, minimum delay is guaranteed. HOL blocking occurs when a packet at the head of a queue that is destined to a congested output has to wait and possibly block other packets destined to uncongested outputs. Unfortunately, this "ideal" architecture is infeasible in the case of medium-to-large switch sizes and high link speeds, as it requires large memory throughput, hence it is expensive, or even not implementable: for a N port OQ switch, the buffer memory must operate N times the link speed in order to avoid packet loss; this event can happen when all input ports transfer packets to a single output port [1]. This "hot-spot" case occurs very often in client/server applications, where a popular server is connected to a single switch port and client requests arrive to the other $N-1$ ports [2]. This speedup problem, combined with the fact that link speeds are increasing much faster than memory speeds, makes pure OQ architecture infeasible for gigabit rate networks. Furthermore, another disadvantage of the OQ architecture is the inefficiency of the buffer space partitioning: some memories may remain "almost" empty, whereas others might be "almost" full, although we have "paid" much in memory. This feature characterizes all switch architectures, apart from the Shared Buffer one.

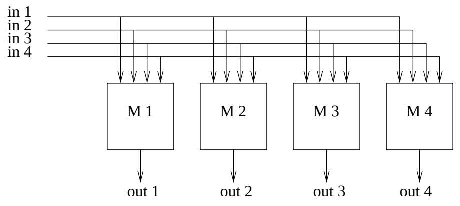

Figure 1.2: Output Queueing (OQ) architecture: a 4×4-switch example is illustrated.

Input Queueing (IQ - see Figure 1.3) uses buffers at the ingress to store incoming data. These buffers must have a constant throughput of 2, and can have a single queue, or multiple queues; in the latter case the architecture is called "Advanced Input Queueing" or "Virtual Output Queueing". IQ switches with a single queue per input were studied and shown by Karol et al. [4] to have a limiting throughput of around 60% of their incoming throughput for Bernoulli packet arrivals with uniformly selected output ports. This limited throughput is due to HOL blocking at the input queues. There are two solutions to this problem: speedup the switching fabric [2], or implement Virtual Output Queues (VOQs) at the ingress line cards; the latter solution has proved to be the most feasible one; as a result, most architectures based on IQ are of the VOQ-IQ style.

Unfortunately, if an unbuffered crossbar is used as the switching fabric of a VOQ-IQ switch: (a) synchronous operation is imposed; and (b) crossbar schedulers are inefficient, as it is hard to implement high throughput global schedulers. Synchronous operation has the result that: (i) variable-size packets have to be segmented into fixed-size cells before entering the crossbar and then reassembled at the egress; (ii) all cells have to be synchronized to a common internal clock, thus expensive synchronization circuits have to be used.

[1]Consider, for example the case when all inputs wish to write to the same output: in the OQ example presented in Figure 1.2, memories must have write throughput of 4 and read throughput of 1; note that if we economize on write throughput, like "Knock-Out" architecture [3] does, then we risk to have to drop some packets on some situations.

[2]Later on we will see that fabric speedup in IQ switches is needed for scheduling reasons, as well.

Furthermore, crossbar schedulers: (*i*) find it difficult to operate in very short cell times, as they cannot offer both high throughput and low latency; and (*ii*) it is hard to offer weighted fair queueing (WFQ) QoS [1]. Scheduling in VOQ-IQ switches is difficult because matching scheduling algorithms require complete "knowledge" of the input-to-output port head-of-line transmission requests [3].

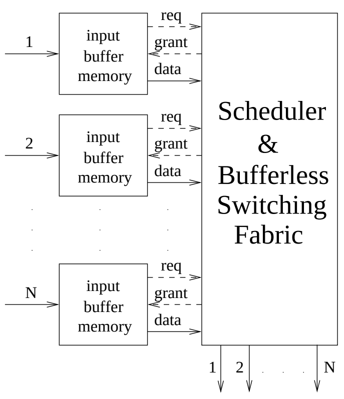

Figure 1.3: Input Queueing (IQ) architecture.

These problems can be overcome in two ways: (a) by using internal speedup and a traditional unbuffered crossbar switching fabric; and (b) by using a buffered crossbar as the switching fabric, but with no (or little) need for speedup. In the case that the switching fabric and schedulers are run at a higher speed, we have the Interal Speedup or Combined Input Output Queueing (CIOQ) switch (see Figure 1.4). Notice that most of the times the buffers at the ingress will be almost empty, while the ones at the egress almost full. Memory throughput still remains constant ($1 + s$, where s is the speedup factor), but this speedup factor, usually two to three in commercial products, actually limits the line rate, compared to the case when another architecture that did not require speedup was used instead.

The other solution is to add buffers to the crosspoints (shown in Figure 1.5); this organization is called Combined Input Crosspoint Queueing (CICQ). By this organization, information entering on different inputs can be destined to any output, as it does not have to be delivered to that output right away; it can be buffered to the corresponding crosspoint buffer instead. As a result, scheduling decisions need not be correlated to each other. Hence, no central scheduler is needed: input transmissions are independent of each other and independent of output transmissions. The N^2 buffers needed in the case of a $N \times N$ switch cannot be too large, due to implementation infeasibility. Instead, small buffers can be used, "backed up" by larger ones at

[3]Two of the most well known such scheduling algorithms are Parallel Iterated Matching (PIM) by Anderson et al. [13] and iterated SLIP (*i*SLIP) by McKeown [12].

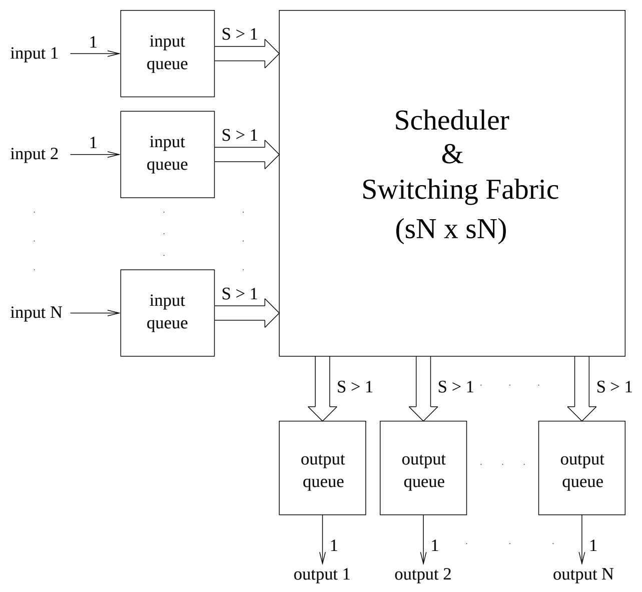

Figure 1.4: Combined Input Output Queueing (CIOQ) or Internal Speedup architecture: speedup factor is s.

the input line cards, in a VOQ organization. Backpressure ensures that no crosspoint buffer will overflow.

Due to the fact that the schedulers operate independent of each other, no synchronized decisions are imposed; hence fixed-size cell operation and synchronization to a common clock are not needed any more. Furthermore, the loosely-coupled input and output schedulers, although not solving the matching problem in a short-term way [4], can find very efficient long-term solutions to the crossbar scheduling problem, and are capable of offering advanced QoS, without the need for speedup. Egress buffering is also not needed, because no packet reassembly is required and no output queue can be built up, as there is no internal speedup.

To summarize, by adopting the CICQ architecture: (a) scheduling is simplified; (b) there is no need to mutually synchronize the line cards and to segment and reassemble packets, as *direct operation on variable-size packets is trivially supported*; (c) no internal speedup is required; and (d) very little (or no) memory is needed at the egress.

The major drawback of the CICQ architecture is the need to partition the switching fabric memory into N^2 (in the $N \times N$ case) buffers that must be located at the crosspoints. This characteristic prevented designers from adopting this organization in the case of medium- or large-scale switch solutions. In a later section we show that sub-micron technology transistor downscaling and new memory technologies have eliminated this drawback.

As a conclusion, provided that an Advanced Input Queued (VOQ) switch

[4]As an unbuffered crossbar would do.

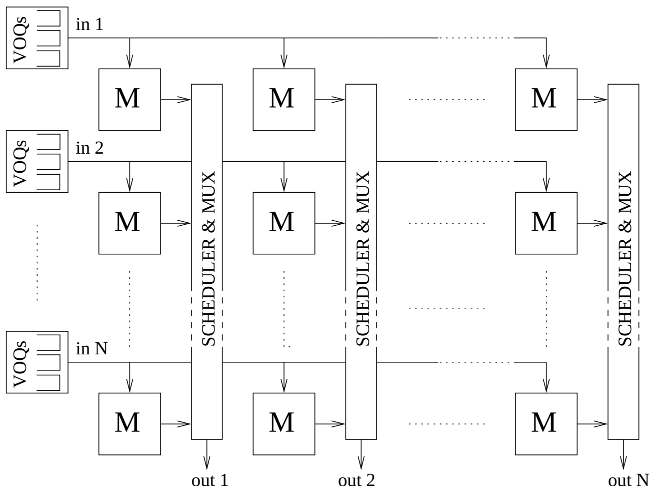

Figure 1.5: Combined Input Crosspoint Queueing (CICQ) architecture: small buffers at each crosspoint are "backed-up" by larger VOQs at the ingress line cards.

had a buffered crossbar as its switching fabric, scheduling would be simpler and packets would not have to be segmented and reassebled; as a result, no speedup would be necessary. In this work, we design, simulate, synthesize and place and route such a (medium-scale) buffered crossbar switch, and demonstrate its feasibility.

1.2 Related Work

In this subsection we present the most important unbuffered and buffered crossbar switch implementations that have emerged during the last few years. The aim is to show the progress that has taken place in this field, as well as highlight the special characteristics of our switch architecture.

1.2.1 Unbuffered Crossbar Implementations

Various researchers have designed and implemented unbuffered crossbar switches in the past.

Heer et al. [16] "Self-routing Crossbar Switch" was a 12 port, 2 $Gbit/sec$ per-port unbuffered switch implementation, running internally at 125 MHz and consuming 2.5 W of power. The overal area of the pad-limited design was 64 mm^2, whereas the core area for the switching matrix, control logic and memories was about 25 mm^2 in a Siemens 0.25 μm tecnology.

McKeown et al. designed a 320 $Gbit/sec$, fixed-size packet, input queued switch, with an unbuffered crossbar switching fabric and a sophisticated scheduler, back in 1996, called "Tiny Tera" [11]. The switch had 32 input ports, operated at almost 10 *Gbit/sec* link speeds, distinguished 4 classes of service and efficiently supported multicast traffic. The switch did not suffer from HOL blocking, due to the use of Virtual Output Queues at the input buffers, whereas the scheduling algorithm was *i*SLIP [12]. The crossbar switch comprised of 1-bit crossbar slices and packets were segmented and sent through the crossbar in 64-bit chunks.

1.2.2 Buffered Crossbar Implementations

Yoshigoe and Christensen simulated the idea of a "Parallel-Polled Virtual Output Queued (PP-VOQ)" buffered crossbar switch [14], with crosspoint memories of 1500 Bytes. The simulations, although of great value, did not take into account: (a) small/large RTT values; and (b) complicated and realistic enough traffic patterns. The same authors also proved the feasibility of a 24-port 10 *Gbit/sec* per-port FPGA implementation of a crossbar switch [15]. The input and output scheduling was plain Round-Robin, but by the use of a new Round-Robin Poller design. Target technology was the Xilinx [9] Virtex II series of FPGAs. The overall design was placed on a bus motherboard, with 6 line cards and one crossbar card attached to it. The crossbar card consisted of four crossbar slices, each of which communicated with the line cards, via the motherboard, through 24 "Rocket I/O" [10] tranceivers (3.125 *Gbit/sec* each). Memory and VOQ datapath was 16 bits wide and crosspoint buffer occupancy was transmitted over a dedicated parallel interface from the crosspoints to the line cards. The auhors also implied that larger switches would be feasible in the future, due to larger amount of fast I/O circuits present in modern FPGAs. The switch did not support direct routing of variable-size packets, which was implied in [14].

Kariniemi et. al. [17] developed a 4×4, 5 *Gbit/sec* ATM buffered crossbar switch on an FPGA, for cable TV backbone networks. Internal SRAMs varied in size, from 32 to one KByte, due to the various ATM rates supported, with a total memory size of 1.12 Mbits.

One of the most important buffered crossbar implementations came from IBM Zurich [18]. It was a 4 *Tbit/sec* incoming throughput switching fabric (but 2.5 *Tbit/sec* incoming throughput switch), single-stage, combined input (with VOQs) and crosspoint queued (CICQ) switch, that supported long Round Trip Times (RTT), with line rates of 10 to 40 *Gbit/sec* (OC-192 to OC-768). Owing to the size of the switching fabric and the number of line cards, the switch was distributed over multiple racks, which could be as far as hundreds of feet apart; totally, 40 switching fabric and 64 interface chips were used. The switch supported 8 classes of service (priorities), but data entering the switch was still segmented into fixed-size packets of 64 or 80 Bytes, resulting in a line rate speedup of 1.6, in order to compensate for the switch packet header and segmentation overhead (hence the 4 *Tbit/sec* and 2.5 *Tbit/sec* difference). The chips were fabricated in a 0.11 μm technology and standard cell design methodologies were used. RTT supported could be as large as 64 packet cycles between the line cards and the switch core, and each crosspoint memory was $2 \times RTT$ large, supporting both unicast and multicast traffic, which translated into at least 8 KBytes each. The same authors also stated that a die size of 250 mm^2, a pin count of 1000 signal I/Os (totally 1500-pin package), and a single-chip power consumption of under 25 W are cost-effective borders for current state-of-the-art CMOS technologies [5].

Today, FPGA vendors provide buffered crossbar switch solutions based on their latest FPGA models and I/O protocols, such as Altera [5] [6] and Xilinx [7] [8]. Such switch proposals however, lack scalability and enable designers to develop switches of sizes up to 16×16; larger switches can be built by slicing the datapath to more than one FPGAs. Needless to say, the bottleneck in this case is the large ($O(N^2)$) memory overhead, which inherently characterizes the buffered crossbar architecture.

Katevenis et al. [19] designed and simulated various speedup-less buffered crossbar sizes operating *directly* on variable-size packets, under complex and realistic Internet backbone traffic patterns and under different Round Trip Time (RTT) values [6]. The results showed that a crosspoint buffer size of $MaximumPacketSize + RTT \times LineRate$ is sufficient in order to achieve full output utilization [7]. Also, the organization proposed outperformed most iSLIP-based unbuffered switches, with speedup factors up to 2, and performed very close to the ideal OQ architecture, both in delay and throughput terms.

1.3 Contributions of this Work

The major contributions of the project within which this work was performed are: (a) to the best of our knowledge, it is the first ASIC core design of a buffered crossbar switch *directly* supporting variable-size packets; (b) the switch supports quite long RTT values efficiently; and (c) our performance is closer to ideal Output Queueing than that of alternative current designs, as proved in [19]. The contribution of this thesis is the design of the ASIC core, from block diagram level, through RTL description, synthesis, placement, routing, timing optimization, verification, and power estimation. The purpose of this thesis was to prove the feasibility of *single-chip* buffered crossbar switch designs in current and future CMOS technologies. In this thesis we describe the internal organization of the switch chip, showing its simplicity. Furthemore, we present the exact synthesis, placement and routing, power

[5]These figures were taken into account in our buffered crossbar switch, too: the 32×32 buffered crossbar switch designed in this work has an extrapolated area of 190 mm^2 and consumes 26.2 W in a 0.13 μm technology.

[6]Traffic generation is analyzed in detail in [29].

[7]For example, for a 1500 Byte maximum packet size (TCP/IP), 10 $Gbit/sec$ link speed and 400 nsec RTT, crosspoint buffer must be 2 KBytes.

estimation, and timing optimization flows based on the available tools, followed by fairly detailed tutorials; lastly, we provide guidelines which should be kept in mind when designing very large chips, such as design partitioning, hierarchy organization, important synthesis advices and place & route common problems and solutions.

1.4 Motivation: Large-Scale CICQ Switch Feasibility

As mentioned earlier, the result of this work was the ASIC-flow implementation of a 32×32 buffered crossbar switch, supporting directly variable-packet-sizes, and resulted in a $420mm^2$, 6 Watt chip in a $0.18\mu m$ technology. This proves the feasibility of the concept: using a non-state-of-the-art technology ($0.18\mu m$), a single-chip, multiport (32×32) switch can be designed and implemented today. There are, however, two points worth mentioning: (a) the area is so large, that fabrication will only be possible with the largest wafer available today and yield will probably be unacceptably low; (b) the switch can handle easily today's traffic protocols (10/100 Mbps Ethernet packet sizes, that is, up to 1500 Bytes), but with “only” 2 KBytes of embedded SRAM per crosspoint, it lacks sufficient support for Gigabit Ethernet. Note that Fast Ethernet (100 Mbps) is currently the most used Ethernet standard, but Gigabit Ethernet is also starting to be used in backbone networks and 10 Gbps Ethernet is also emerging. Both support very large packet sizes, called “Jumboframes”, with length of up to 9 KBytes.

In order to support such long packet sizes, *without* segementation and reassembly, each crosspoint memory would have to be approximately 10 KBytes. Traditional SRAMs of such size have an area density of $12mm^2$ per Mbit[31], resulting in memory requirements of around $1.1mm^2$ for each such memory! To make matters worse, such large on-chip SRAMs have approximately 0.3 mW/MHz power consumption, resulting in $90mW$ for each of them and 2.9 Watts for the average worst-case scenario of a 32×32 switch, where 32 memories are active all the time [8] ; the respective values for 2 KByte SRAMs are $0.28mm^2$, 33 mW per-memory and 1W worst case power consuption.

As a result, feasibility of multiport switches, supporting future bottom-layer protocols and technologies is a very important issue. In this section we will examine various ways to overcome area limitations, by adopting emerging memory technologies. Transistor and chip sizes, as well as chip power consumption are also taken into account in this technology roadmap analysis. The latter seems to be another bottleneck in current and future ASIC designs, as it determines chip performance and, most importantly, its cost.

[8]In section 5.3 we explain why this is the worst-case typical-operation scenario.

1.4.1 Transistor sizes

Transistor sizing has been improving for the past 30 years according to the "Moore's Law", which states that transistor density is doubling every 15-20 months. Although this law is somewhat philosophical and has changed a little bit since it first appeared by Gordon Moore, it still remains the main source for roadmap analyses, one of which appeares in [36].

Transistor down-sizing affects technology size, which is 90 nm today and will fall to 65 nm by 2007 and 45 nm by 2010 [9]. The above roadmap actually implies that logic becomes denser and denser, which means smaller chips. Technology downscaling will allow densities of 220 $MTransistors/cm^2$ at 80 nm (2005) to 450 $MTransistors/cm^2$ at 57nm (2008). A switch chip like the one presented will have an extrapolated area of 200 mm^2 at 130 nm, which becomes 110 mm^2 at 90 nm and 80 mm^2 at 80 nm technology, even reaching 50 mm^2 in the year 2008.

1.4.2 On-Chip Memory

The switch chip designed for this work has a total of 1024 SRAM memories (32×32), one in each crosspoint. This large memory count, along with memory size (2KBytes), has been the main bottleneck in multi-port buffered crossbar switches, as it is an inherent characteristic of the buffered crossbar architecture. One technology that can overcome this limitation is Embedded DRAM (eDRAM). eDRAM offers many advantages, compared to conventional on-chip SRAM: (a) $\underline{\text{it is 1.5x to 4x denser}}$[39], thus enabling smaller die sizes; (b) although active power is comparable for both types of memories, eDRAM draws orders of magnitude less standby current than a respective SRAM; (c) eDRAMs provide very wide I/O, although their random access times still lack those of conventional SRAMs (e.g. 90 nm fast eDRAMs can run at exess of 300 MHz[40, 41], whereas embedded SRAMs have already passed the 500 MHz "barrier"); (d) eDRAMs store larger amount of charge per cell than SRAMs, thus having minimal Soft-Error Rate per bit, whereas SRAMs need specific error checking and correction (ECC) hardware to overcome error limitations, with impacts on area and performance. One potential disadvantage is increased eDRAM cost[42], which is due to: (a) 20% increase to standard 6 Metal Layer process, due to increased mask number required (incremental processing cost); and (b) 5-10% cost increase, due to more complicated testing required. Nevertheless, incremental processing and test costs, like the ones mentioned, are offset by improved silicon yield, as eDRAMs are easily repaired when manufactured.

As a result, a 10 KByte eDRAM-per-crosspoint 32×32 switch will be feasible in the very near future. Furthermore, taking into account that in the

[9]Note that advertisements may state that smaller technologies have been implemented in the laboratory, but it can take 2 to 3 years until the first companies reach actual production.

switch designed for this thesis, memory area accounted for approximately 70% of the total chip core area, the use of eDRAMs of the same size would lower chip size by approximately 50%, thus improving potential yield.

1.4.3 Chip/Wafer Sizes

Despite the tremendous technology improvements that will take place in the next 15 years, companies seem to have agreed on a standard upper wafer diameter limit of $300mm$, which may grow to $450mm$ after nearly a decade. ASIC chip sizes also seem to limit themselves to $570mm^2$. Having the tremendous downsizing caused by transistor technology and a possible adoption of new memory architectures in mind, overall yield will certainly increase, thus making large crossbar switch chips even more feasible and cost-effective.

1.4.4 Power Consumption

Power consumption is possibly becoming one of the most critical elements in hardware design today, as it can be the limiting factor of design performance. Projections show [36] [37] that in a few years: (a) power delivery and dissipation will be prohibitively high; (b) power density (W/cm^2) will grow to enormous figures [10]. This results in large power/ground wiring requirements, sets packaging limits, caused both by technology barriers, as well as cost[11], and impacts on signal noise margins and reliability. As a result, power consumption must always remain as low possible, in order to avoid expensive packaging and cooling solutions.

[10]Today, high-performance processors consume above 100 W (i.e., the 2 CPU, 2 MB L2 cache IBM Power4 consumes 115W in a $3.8cm^2$ die) and have power densities near a nuclear reactor, increasing linearly every year.

[11]$50W/cm^2$ for forced-air cooling and 120 W total power consumption; today's power-dependent pricing is about 1$ per Watt.

Chapter 2

Switch Organization and Operation

2.1 Introduction

In this chapter we present the organization and operation of the 32×32 crossbar switch. We decided to use 2 KBytes of memory per-crosspoint; this agrees with the results presented in [19] and translates into 1500 Bytes for a maximum packet size, plus 500 Byte times for the Round Trip Time; the latter Arbiter corresponds to 400 nsec in 10 *Gbit/sec* link speeds, which is the line rate assumed both in [19] and in this work. The remainder of this chapter is as follows: first we present the switch "block-level" interface, determining the input/output signals needed and outlining the main functional blocks. Then we discuss switch internal block organization; we particularly describe the organization of the Crosspoint, Output Scheduler and Credit Scheduler sub-modules, followed by the enqueue logic, which we developed in order to test the operation of the switch.

2.2 Internal Architecture

A top level block diagram of the switch can be seen in Figure 2.1. Switch interface has thirty-two 32-bit wide input data lines, 32 "Start Of Packet" and 32 "End Of Packet" input lines, thirty-two 32-bit wide data output lines and thirty-two 16-bit wide credit output lines. Apart from the obvious necessity of the data input and output lines, and the credit output lines, we also have to inform the switch that a new packet is about to be transmitted from the line cards to the fabric, as well as inform the fabric about the end of a packet transmission; this is accomplished by asserting the corresponding "Start Of Packet" (`sop`) or "End Of Packet" (`eop`) signal for one clock cycle.

The switch core is divided into three parts: Crosspoint modules (XPs), Credit Schedulers (CSs) and Output Schedulers (OSs). Each crosspoint includes a 512×32 (2 KB) SRAM, along with clock synchronization and mem-

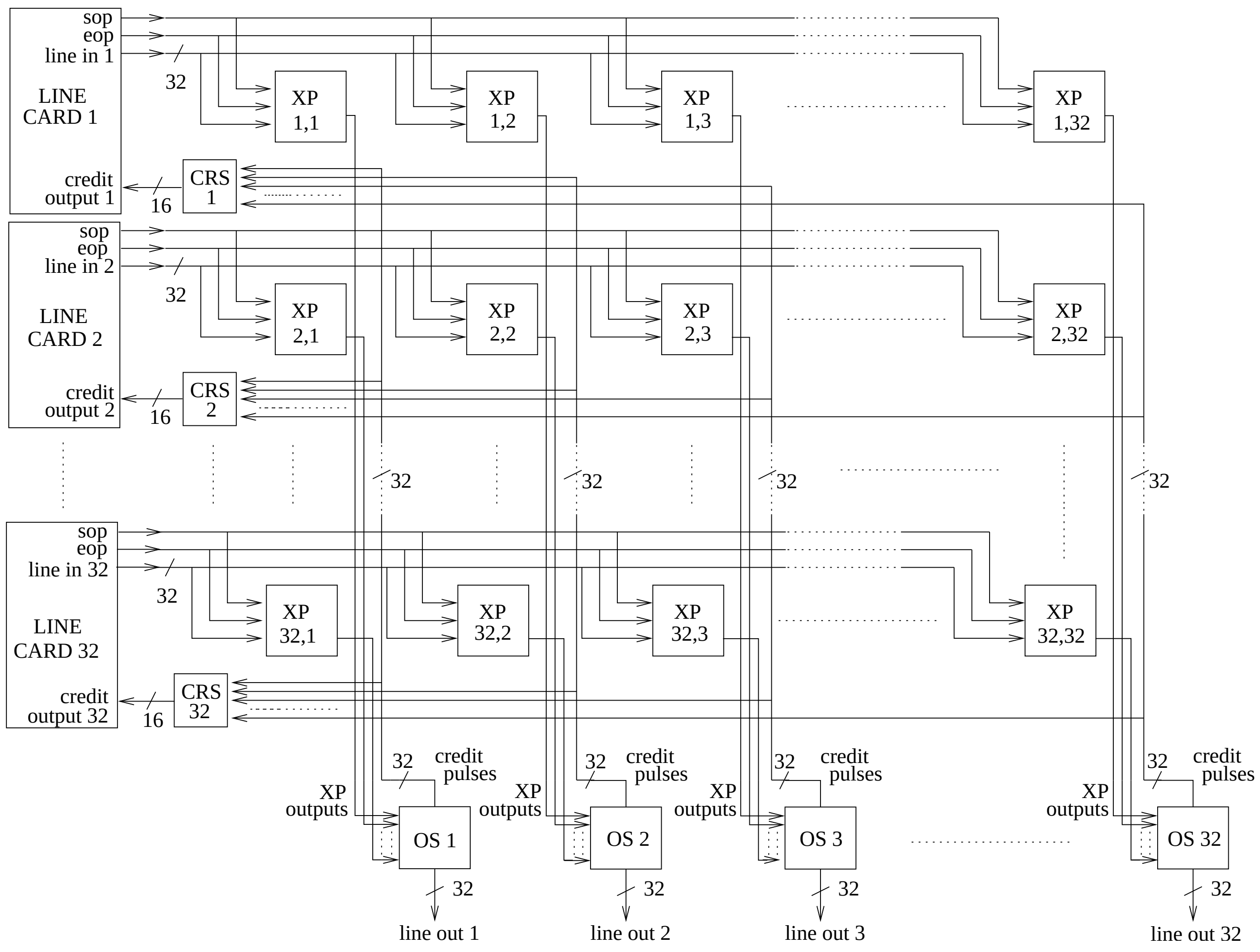

Figure 2.1: 32×32 crossbar switch block diagram: the switch consists of 32 × 32 = 1024 crosspoints, and 32 output and credit schedulers. Line card logic is located outside the switch core.

ory enqueue/dequeue logic. A 32×32 switch needs 32×32=1024 XPs, thus 16 Mbits of memory. Credit Schedulers contain mechanisms for collecting and sending credit information to the input line cards, whereas Output Schedulers are responsible for selecting eligible flows that can send their packets to the chip outputs. Both the Credit and Output Schedulers implement the plain Round Robin discipline. Other scheduling disciplines, being fairer, could also be supported (i.e. [34]).

We place the clock domain boundaries in the crosspoint modules; thus, elastic buffers at the chip inputs are eliminated, which reduces latency and power consumption, as each word of packet payload is only written once into and read once out of a memory during its transition through the chip [19]. Note that the only control information that has to traverse two different clock domains in the switch organization presented, is the “new packet” arrival notification pulse. What is more, the two clock domains must operate in almost the same frequency, as the input and output links have the same speed. These observations significantly simplify crossbar design, as such 1-bit synchronization circuits are very simple and efficient.

Before explaining the three main switch components in more detail, it should be noted that the switch supports directly TCP/IP packets, of sizes 40-1500 Bytes. Packet size must be a multiple of 4 and, if this is not the case, the input line cards have to fill the remaining bytes of the last word and accordingly increase the size of the packet. The only other information needed for the switch to operate is an output destination 32-bit multicast bit-mask, which is sent just before the actual packet; this information is padded by the line card logic and the positions of the "1" indicate the crosspoints in which the packet is destined to or, equivalently, the outputs that the packet must be sent to. Destination bit-mask and packet size are accompanied by a "time of transmission" value and a "packet serial number" in the payload, both of which were added for debugging and statistic purposes (see Figure 2.2).

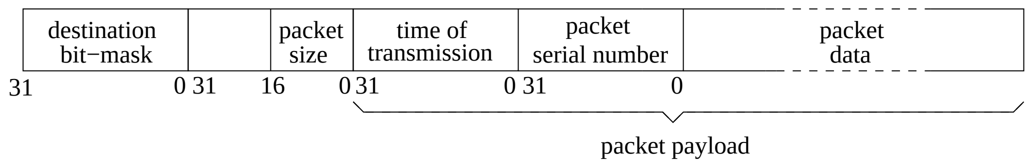

Figure 2.2: Supported packet format: apart from packet size (16 bits), a time of transmission (32 bits) and a packet serial number (32 bits) value was padded for statistic and debugging purposes. Packet payload follows. Notice the 32-bit mask added at the begining of the packet.

2.2.1 Crosspoint Blocks (Crosspoints)

Crosspoints (XPs) are responsible for storing/retrieving packets into/from their corresponding buffer memories. Figure 2.3 shows the top-level block diagram of the crosspoint: a 32-bit wide packet data bus enters each crosspoint, along with the `sop` and `eop` signals. Each crosspoint receives a `deq` signal from the corresponding column output scheduler and sends to the output a 32-bit `data_out` value. Notice the two different clock domains: the clock domain controlled by `clk_wr` contains the writing circuits, while `clk_rd` handles the data reads.

Crosspoint organization is shown in Figure 2.4. Each XP has a 32-bit wide input datapath, along with two control signals: a "Start Of Packet" (`sop`) signal, which must be asserted for one clock cycle during the transmission of the first packet word and an "End Of Packet" (`eop`) signal, which is asserted during the transmission of the last packet words. The `sop` signal informs XP logic that a new packet is being sent. Each crosspoint looks individually at a 32-bit destination bit-mask, described in the previous section,

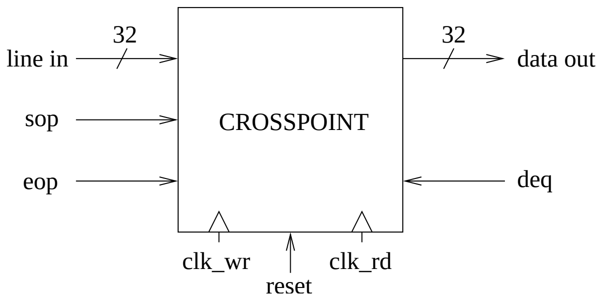

Figure 2.3: Crosspoint block diagram.

and accordingly decides if the certain packet should be handled by it or not. The `eop` signal informs the crosspoint logic that the data enqueueing must stop.

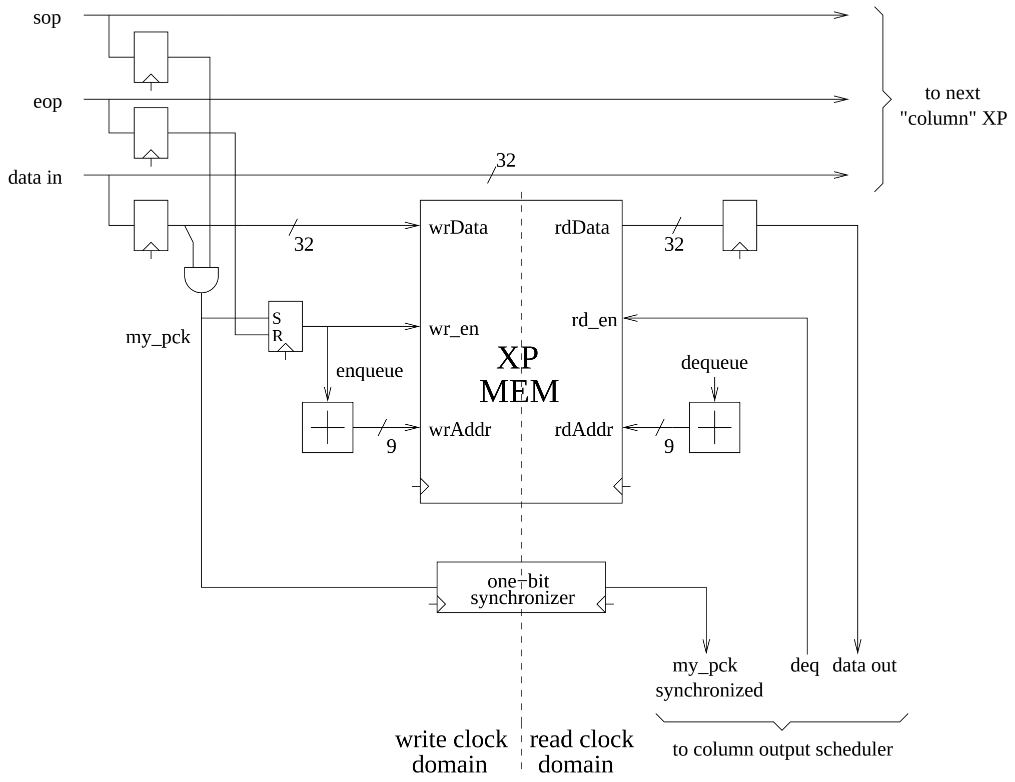

Figure 2.4: Crosspoint organization: notice the two clock domains; they meet in the memory and in the 1-bit synchronizer circuit.

Both `sop` and `eop` signals are needed in order to reduce the complexity of the switch core. In fact, in order to remove those 64 signals, we would have to include some line-card and the packet enqueue control logic *inside* the switch, which would add to its area and complexity. On the other hand, if those logic blocks were placed inside the switch, credits would not have to be sent to the line cards, which would decrease the number of I/O ports needed. Although the last argument is very important in the case of medium- to large-scale buffered crossbars (as chip I/O throughput is limited and has to remain low for pin count, power consumption, and pricing reasons), a clever organization of credit logic can remove this initially predicted overhead [1].

In particular, in order to remove the `eop` signal, extra logic would have to be added inside the switch, which would detect the `sop` arrival, send the enqueue signal to the corresponding crosspoint, capture and decrement the packet's size, and deassert the enqueue signal when the packet is written. During most of the stages of this work, this logic was actually included in the switch, but its necessity proved worthless, as with just a little I/O throughput overhead [2], this logic was easily omitted from the fabric. Note that only 32 such "enqueue logic" blocks were used, as we only needed one per switch row: every crosspoint that actually enqueued a packet would "listen" to the `eop` signal generated by that logic; the rest would just ignore it.

When both `sop` and the corresponding bit of the bit-mask are asserted, the crosspoint logic must perform a packet enqueue to the SRAM. This is carried out by setting a latched signal, called `my_pck`, which in turn asserts the SRAM's write enable and increments the 9-bit enqueue address counter. When `eop` signal arrives, the enqueue FSM deasserts write enable and stops the enqueue pointer from incrementing.

We must also forward the `my_pck` signal to the corresponding output scheduler, informing that a packet is being enqueued to one of the 32 crosspoints connected to the column's scheduler. This event wakes up the output scheduler if the output link has been idle. We thus feed this new-packet-arrival signal to a simple 1-bit synchronizer, which is depicted in Figure 2.5. The signal sets an RS flip-flop and its output Q is sampled by a series of three D flip-flops clocked by the output clock domain (`clk_rd`). When the pulse is received, an acknowledgment signal travels back, which resets the RS flip-flop. The synchronization delay is 5 clock cycles, far less than the minimum packet size (40 Bytes) enqueue time, which, in the case of our 32-bit datapath, corresponds to 10 cycles. Note that the two clock frequencies are always close to each other, which is always the case in switches, since they have the same input and output throughput. Synchronizer design is briefly presented in the Appendix.

Packets are enqueued/dequeued in a 512$\times$32 (2 KByte) Two-Port Register File SRAM from Virtual Silicon Technology [43]. This memory has a 32-bit port for read and an independent 32-bit port for write; read and write addresses are 9 bits wide (see Figure 2.3). The independent read and write cycles are timed with respect to their own clocks, namely `clk_rd` and `clk_wr` respectively. It should be noted that during a read cycle, the output

[1]This organization is thoroughly presented in a following section.

[2]*Not* high chip I/O overhead, though, as "End Of Packet" transmission is carried out at most every "minimum-packet-size" time, which is 10 clock cycles in our 32-bit datapath; hence fast and expensive tranceivers are not needed.

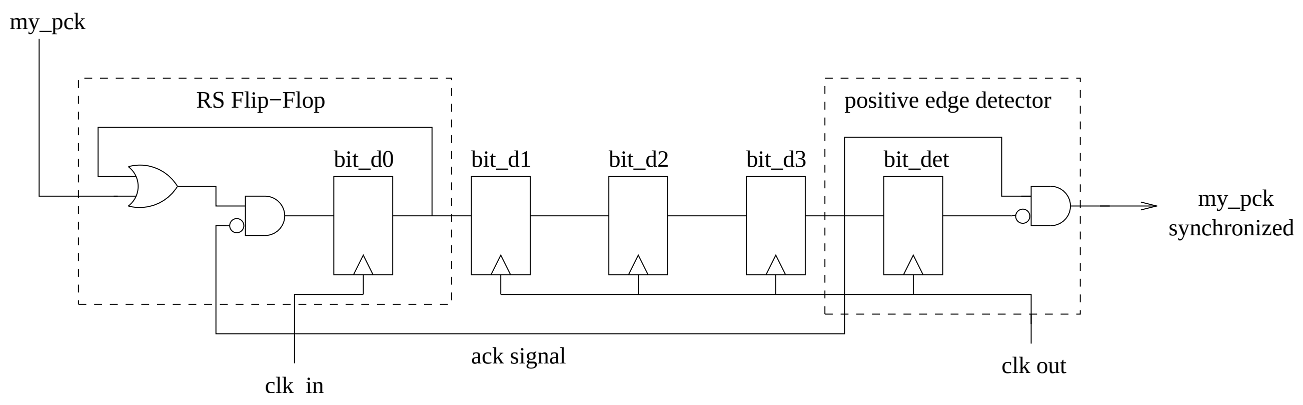

Figure 2.5: 1-bit synchronizer circuit.

bus values are held if read enable is not asserted. This read enable feature is used to save RAM power without the need for external clock gating.

The last part of the crosspoint logic is the dequeue sub-module. Whenever the output scheduler decides to dequeue a packet from the specific crosspoint, it asserts the `deq` signal, which fires the dequeue FSM corresponding to that output; read enable is asserted and dequeue pointer is then incremented. The `deq` signal remains asserted for the time needed to fully dequeue the packet, and is controlled by the output scheduler dequeue logic.

2.2.2 Output Schedulers

The output schedulers are responsible for: (a) selecting the next eligible flow from a certain crosspoint of the same column; (b) initializing the transmission of packets to the specific switch output; and (c) sending a credit back to the line cards. A flow is called *eligible* if the corresponding crosspoint contains packets that are waiting to be sent. If there are more than one eligible flows, the output scheduler has to select one of them; this selection can depend on the flow priority (possibly its *weight* - i.e. in Weighted Round Robin scheduling), or on a plain round robin policy. In this work we did not aim at implementing new and sophisticated output scheduler policies, but rather use a plain round robin scheduler: the output scheduler maintains a list of the eligible flows and after serving one flow, it selects the next eligible flow of the list [3]. The output scheduler knows which flows are eligible or not through the synchronized `my_pck` signals it receives from the crosspoints of the same column. Since the synchronization delay is 5 clock cycles and a packet transmission lasts for 10 cycles, we claim that the scheduler supports *cut-through* operation (with a 5 cycle overhead): when all 32 crosspoints

[3]In the first stages of this thesis, we developed a Weighted Round Robin output scheduler with "smart comparator usage" for a 4×4 crossbar switch, that would be implemented in an FPGA. This scheduler is examined in the Appendix.

of a column are empty and a minimum-size packet arrives to one of them, only 5 clock cycles are needed to initialize its dequeue to the switch output, whereas the time needed for the packet to be written to the crosspoint is 10 clock cycles.

Output scheduler block diagram can be seen in Figure 2.6, while its organization is presented in Figure 2.7. The scheduler has thirty-two 32-bit data

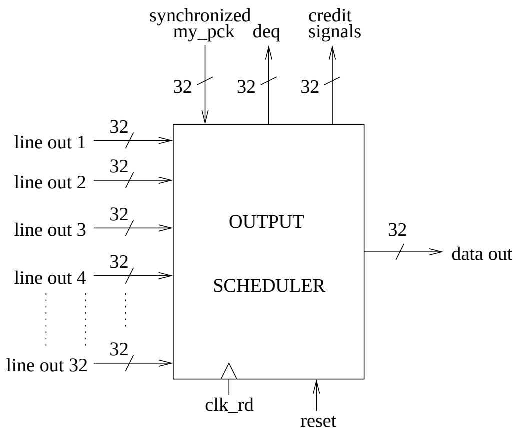

Figure 2.6: RR Output Scheduler block diagram.

inputs, which originate from the column crosspoint memories' data outputs. 32 synchronized `my_pck` signals are also sent from the crosspoint synchronization logic. The output scheduler sends 32 credit signals back to the switch inputs and a 32-bit `deq` bus to the column crosspoints. The dequeue signals act as read enable for the crosspoint memories.

In order to produce the list of eligible flows, the scheduler maintains thirty-two 6-bit packet counters [4], which are incremented each time a synchronized `my_pck` signal arrives and decremented when the selected packet is being dequeued; when these counters are equal to zero, the flow is ineligible. The 32-bit eligibility mask that is created in this way enters a Round-Robin Priority Enforcer, which is responsible for selecting the next eligible flow (its output is a 32-bit value, with one bit equal to "1" and 31 bits equal to zero). This 32-bit crosspoint selection bus is sent to the column crosspoints and at the same time the specific packet counter value is decremented. When the first packet word is dequeued from the crosspoint and arrives at the scheduler, the packet size is stored, in order to allow the scheduler to know when it should stop dequeueing. If, while one packet has been almost dequeued, there exists at least one eligible flow in the same column, the scheduler has to start *pre-scheduling*, in order to choose the next eligible flow and not miss any clock cycles in between. Hence, back-to-back operation is achieved.

[4]Six bits are sufficient because each crosspoint buffer can store at most 50 (minimum-size) packets.

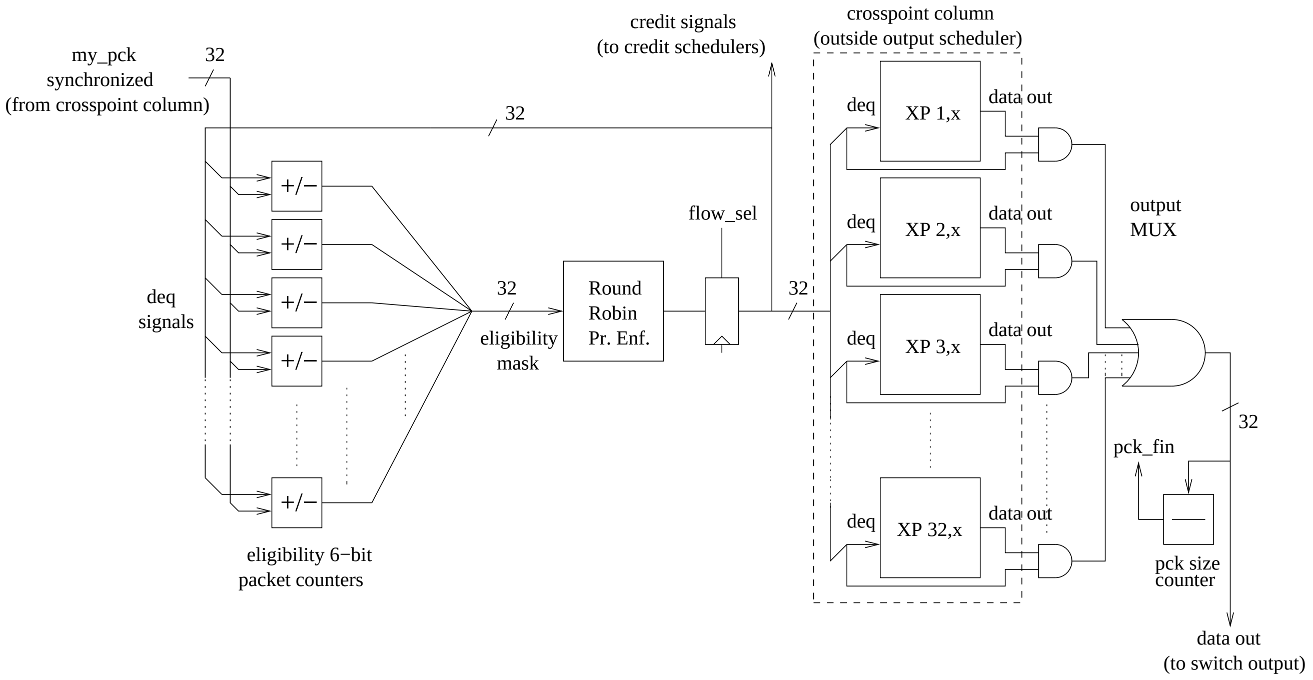

Figure 2.7: RR Output Scheduler organization: crosspoint memories are illustrated for clarity; they do not belong to the output scheduler.

2.2.3 Credit Schedulers

Credit schedulers are responsible for: (a) collecting the credit signals from the output schedulers; and (b) sending them back to the line cards; there is one line card per row. Since it is the line cards' responsibility to recover the receiving clock (as happens with outgoing packet data), credits are sent in respect to the output clock, hence no synchronization circuitry is required.

The credit format should satisfy one constraint: it should contain the absolutely necessary information that the line card should know, in order to save I/O bandwidth (package pins and expensive I/O pads). In order to remove as much logic possible from the switch core, we decided that the line cards should perform all necessary credit computing operations. For error resilience purposes, the credit schedulers adopt a QFC-like [30] approach: instead of sending the information that "the next packet has just left from crosspoint ..." [5], they just send "the total, cumulative number of packets that have left up to now from crosspoint ..., modulo 2^k, is equal to ..." [19]. As a result: (*i*) we do not need to send one credit for every packet; and (*ii*) even if some credits get lost, the next arriving credit carries cumulative information from past ones, too [6]. Line card logic is responsible for concluding

[5]This information could have the form of a 5-bit encoded crosspoint (i.e. output) number, but one such credit loss would have undesirable effects.

[6]We just have to ensure that at least one credit will safely arrive for every 2^k departing

that the credit number just received is different from the previously stored corresponding value, hence a new credit has arrived. Credit format is shown in figure 2.8. It is 16 bits wide, with the most significant 7 bits reserved for future use (possibly 2 bits for error checking purposes, plus 3 bits for supporting up to 8 distinct priorities). The next 4 bits contain the actual credit value (number of packets sent from the corresponding crosspoint), while the 5 least significant bits refer to the crosspoint's ID.

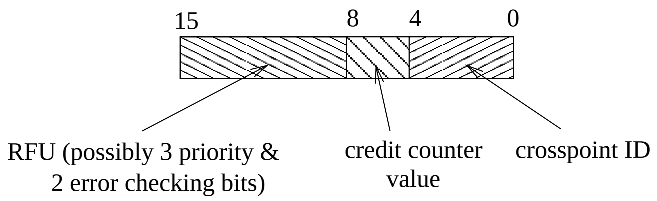

Figure 2.8: Credit format.

Figure 2.9 shows the credit scheduler block diagram. Each credit scheduler module receives a 32-bit credit mask, referring to dequeues from the correspoinding row crosspoints, and sends a 16-bit credit bus to the respective line card logic.

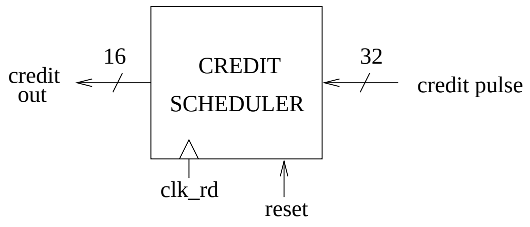

Figure 2.9: RR Credit Scheduler block diagram.

Credit scheduler internal organization is shown in Figure 2.10. The scheduler maintains 32 4-bit "sent-packet" counters, which contain the number of packets that have been dequeued from the respective row crosspoints. Obviously this counter wraps around when 16 packets have been dequeued and the 17-nth credit has just arrived.

Each time a credit arrives, the respective credit counter sets its output `credit_change` bit, informing the scheduler that its value has just changed. The 32-bit `credit_change_mask` value is connected to a Round Robin Priority Enforcer, which selects the next (changed) credit value that will be sent to the line cards. This is indicated by its 32-bit output, `credit_choose_mask`, which, except for selecting the credit value that will be sent to the line cards, is also responsible for *resetting* the `credit_change` flag of the corresponding counter.

packets, i.e. before the counter wraps around.

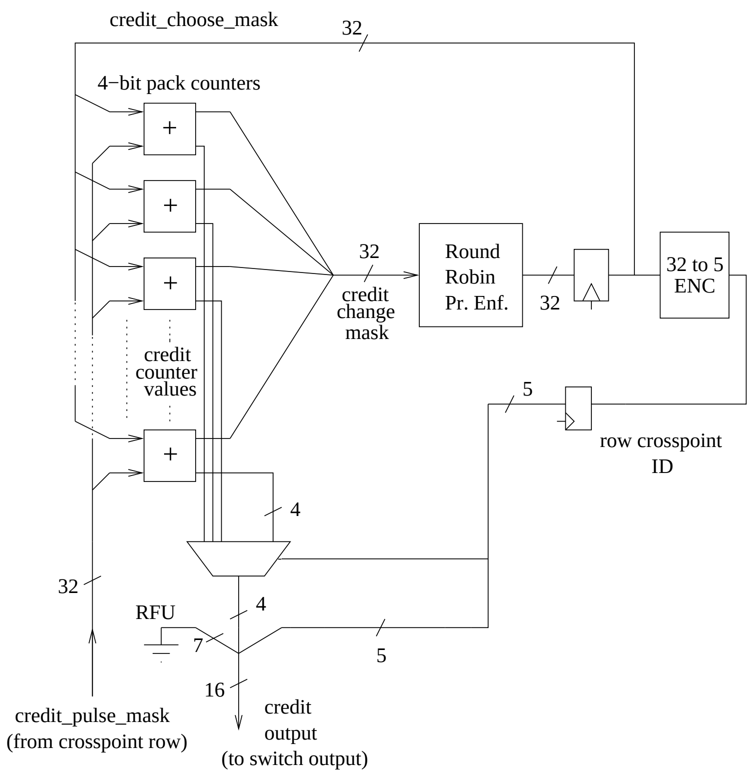

Figure 2.10: RR Credit Scheduler organization.

Assuming that credit switch outputs will use about 1/8 of the bandwidth of every switch packet data input/output [7], each 16-bit credit value will need 4 clock cycles in order to be sent to the line card [8]. As a result, scheduling is carried out in 4-cycle intervals. Since scheduling time is 4 clock cycles, the scheduler needs $4 \times 32 = 128$ clock cycles to send all 32 credit values.

Credit signals can arrive at any time from the output schedulers, with the extreme of all 32 bits of the `credit_pulse_mask` being set in one clock cycle. In this case, `credit_change_mask` is all "1" and one credit counter value is selected to be sent. Due to the fact that the next credit signals will come *at least* one minimum-packet transmission time after this event (10 clock cycles), no counter can overflow sooner than 170 clock cycles have passed; hence, the scheduler has enough time to send all 32 credits before any counter wraps around.

Last of all, it should be noted that if the `credit_change_mask` is all zeroes, meaning that no credit has been received for at least the last 128 clock cycles (nearly 13 packet-times), the credit scheduler sends the stored credit counter values. In this way, the line card logic can check if the credit

[7]This is desirable in order to save package pins and I/O tranceiver pads.

[8]In one cycle, 32 packet data bits are received/sent, hence if credits use 1/8 of this bandwidth (4 bits per cycle), 4 cycles are needed for the transmission of the 16-bit credit value.

value has changed since last time and thus perform error checking.

2.2.4 Line Card Logic

The switch presented sofar could be tested in two ways: (a) by feeding the simulator with trace files containing real or generated traffic; and/or (b) by designing a simple line card and feeding the switch with traffic from the line card's traffic generator. During the initial stages of the switch design, we used traces generated from the simulator which was written in C++ and is thoroughly presented in [29] and [19]. The trace files consisted of packet sizes and the exact time they were arriving at the switch inputs. Later on, the credit logic designed in the simulator changed from the one we used and, as a result, in order to test and verify the switch operation, we designed a Verilog model of the line card.

Line card block diagram is presented in Figure 2.11. Each line card is

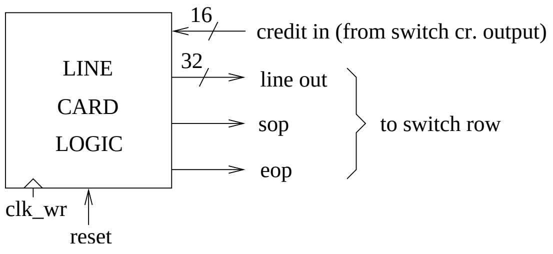

Figure 2.11: Line Card block diagram

responsible for sending packet data (32-bit `line out` bus), and the "Start Of Packet" (`sop`) and "End Of Packet" (`eop`) signals to the respective switch row. Feedback information from the switch is packed in the 16-bit credit bus that is sent through the corresponding switch credit scheduler to the line card block. Line card constists of two parts: (a) packet construction and transmission logic; and (b) credit processing logic. These two parts are completely independent, as a credit can come at any time, regardless of the state of a possible packet transmission. The two parts are discussed in more detail in the next two subsections.

Packet Construction and Transmission Logic

As each line card has multiple (32) Virtual Output Queues (VOQs), 32 FIFO queues are maintained; these contain the size of each packet that has just begun transmission from the line card to the switch. A 32-entry "empty crosspoint buffer size" array is also used, in order to store the empty space of each crosspoint buffer of the corresponding switch row. The packet transmission FSM is shown in Figure 2.12. This FSM is in the `IDLE` state if no packet can/has to be constructed and sent. When the `enq` signal is set, the

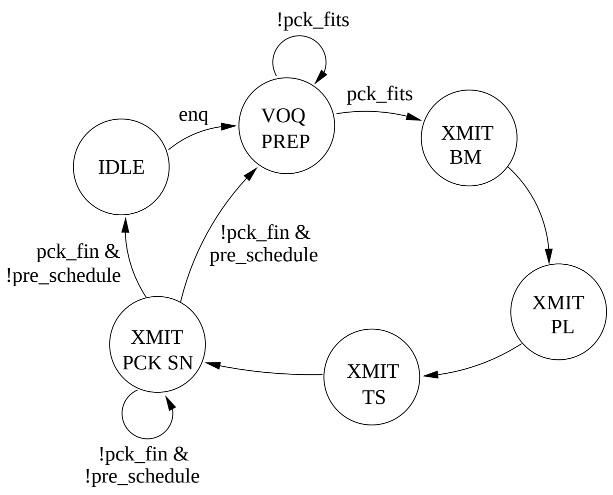

Figure 2.12: Line card packet transmission FSM.

FSM goes to `VOQ_PREP` state. If the new packet's size fits into the corresponding crosspoint buffer, `pck_fits` is asserted, packet size is enqueued in the respective FIFO, the specific crosspoint buffer size is decremented by the `pck_sz` value and the corresponding packet counter is incremented. The next four states are responsible for sending to the switch the packet bit mask (`XMIT_BM`), packet length (`XMIT_PL`), a timestamp (`XMIT_TS`) and the packet's serial number (`XMIT_PCK_SN`); packet format is shown in section 2.2. In order to support back-to-back transmission (`pre_schedule` is asserted), the FSM goes to the `VOQ_PREP` state before the packet's last word is sent to the switch, and the preperations for a new packet transmission start.

Credit Processing Logic

In this behavioral module, a 32-entry credit array is maintained in order to store the previous credit counter values. These values are compared with the newly received ones: if they differ, then we have a new credit for that particular VOQ. As a result, a FIFO dequeue has to be performed, the new credit value must be stored and the empty space of that particular crosspoint has to be incremented by the packet size stored in the head of the FIFO.

Switch Verification

The switch is tested by using the Cadence [20] NCLaunch tool. Three types of traffic are sent: (a) all minimum-size packets (40 Bytes); (b) randomly selected packets; and (c) only maximum-size packets (1500 Bytes). Packets are sent either back-to-back, or with a random interpacket delay. We use two test scenarios: (a) all inputs send packets to all outputs randomly; (b) all inputs send packets to a specific output; obviously, the scenario that puts more stress on the switch core is when smallest size packets are sent back-to-back to a specific output. In that case, the crosspoint buffers of that switch "column" are filled with 40-Byte packets and the output scheduler has to make a decision in 10 clock cycles, which is the transmission time of a minimum-size packet. In this way, we achieve back-to-back operation.

2.3 Conclusions

In this chapter we presented in detail the switch and line card internal organization and operation. Crosspoint logic should be as small as possible, as the $O(N^2)$ crossbar complexity could prove, in the later design stages (synthesis and placement & routing), to be a prohibitive implementation factor. By minimizing the logic at the clock domain boundaries, synchronization becomes easy and simple. Furthermore, no packet segmentation and reassembly is required, by directly supporting variable-size operation and with padding a 32-bit destination bit-mask at the beginning of each packet header. Output and credit schedulers are as simple as possible, whereas their aggregate size is orders of magnitude smaller than total memory area; hence, more complex policies can also be supported, without significantly affecting the total switch core area.

Chapter 3

Logic Synthesis

3.1 Introduction

Logic synthesis is one of the most important phases of the design flow in state-of-the-art circuits. It aims at transforming the HDL (usually Verilog HDL or VHDL) description of the circuit into a technology-dependent, Gate-Level netlist. Through this process, the hardware designer defines the environmental conditions, contraints, compile methodology, design rules and target libraries, in order to achieve certain design goals set by the initial specifications. The Gate-Level representation of the circuit is the input file to the Place & Route tool, which is described in the next chapter.

The tool we use for the logic synthesis of the switch is Synopsys Design Compiler (DC) [46], the most widely used synthesis tool. Design Compiler optimizes logic designs for speed, area and wire routability. From the defined goals, DC synthesizes the circuit and tailors it to a target technology. The rest of this chapter is as follows: first we present the two ways the synthesis procedure can be carried out, flat and hierarchical, and compare them. We then describe the synthesis flow we followed for the 32×32 buffered crossbar switch, accompanied by our synthesis results. Later on, we will compare these results to the placement & routing ones and useful conclusions will be drawn. Finally, we briefly talk about post-synthesis verification.

3.2 Synthesis Flow

The synthesis flow that we follow can be seen in Figure 3.1. We initially *read* the design under synthesis; this enables DC to load all design instances into memory and report possible HDL errors. Next we *set initial design constraints*, such as: (a) maximum circuit area, or zero if ultra "size" minimization has to be performed; (b) circuit clock(s); (c) maximum transition time of specific nets; (d) maximum capacitance of nets, etc. Circuit area and clock cycle specification are usually enough in order to synthesize most circuits. Design must then be *checked*. This enables Synopsys to report mistakes which usually have to do with wrong module instantiations, or interface mismatches. After this "check design" phase is complete, we *optimize the design*, by setting again the appropriate values for the different constraints. Usually, we wish to minimize circuit area and increase operating frequency. Some nets may also have to be constrained in terms of capacitance or timing, but these decisions can usually only be taken into account after actual chip layout is defined. As a result, the latter has to be almost defined *before* synthesis commences. After this final phase, we analyse the reports of the tool and check for unmet constraints. If such exist, the designer usually has to: (a) rewrite HDL code, in order, for example, to meet certain timing constraints; (b) modify the constraints themselves, which unfortunately may result in a slightly different design; (c) change compile attributes, for example, change tool optimization priorities; (d) ungroup (i.e. remove any hierarchy from design blocks), in order to offer the tool the ability to handle larger modules and possibly produce better analysis and optimization results; unfortunately, we cannot completely remove the hierarchy in the case of very large designs. Last of all, we can analyze power consumption and perform power optimizations. Synthesis power estimation is presented in chapter 5, while power optimization techniques can be found in the Appendix.

3.3 Flat vs. Hierarchical Synthesis

Flat designs contain no subdesigns and have only one structural level; they only contain library cells. Design Compiler does not optimize across hierarchical boundaries; therefore, by removing the hierarchy within certain designs, timing results can be improved. Removing of hierarchy is called ungrouping. Through this task, subdesigns of a given level of hierarchy are merged into the parent design.

Although flat synthesis usually provides the best results, this approach cannot be followed in the case of large designs. This is because, at the presence of a large number of multi-instantiated modules, ungrouping will build every instantiated module from scratch, thus needing large computational resources. In such cases, a hierarchical approach is proposed: each module is synthesized, fully optimized, and saved as a separate design. Then, it is loaded back to the tool and linked into the higher hierarchy module. Synopsys will then only have to deal with identicaly-synthesized module instantiations, which will reduce execution time. One disadvantage of the hierarchical approach is that the tool cannot perform low-level optimizations, after the higher-level design is synthesized. Such optimizations are useful when, for example, the designer wishes to optimize the design for power consumption. In that case, cells that are located at non-critical paths are usually replaced by others that are smaller, hence consume less power, but are slower. Such optimizations cannot be performed inside low-hierarchy blocks directly at the top-level of a hierarchical design.

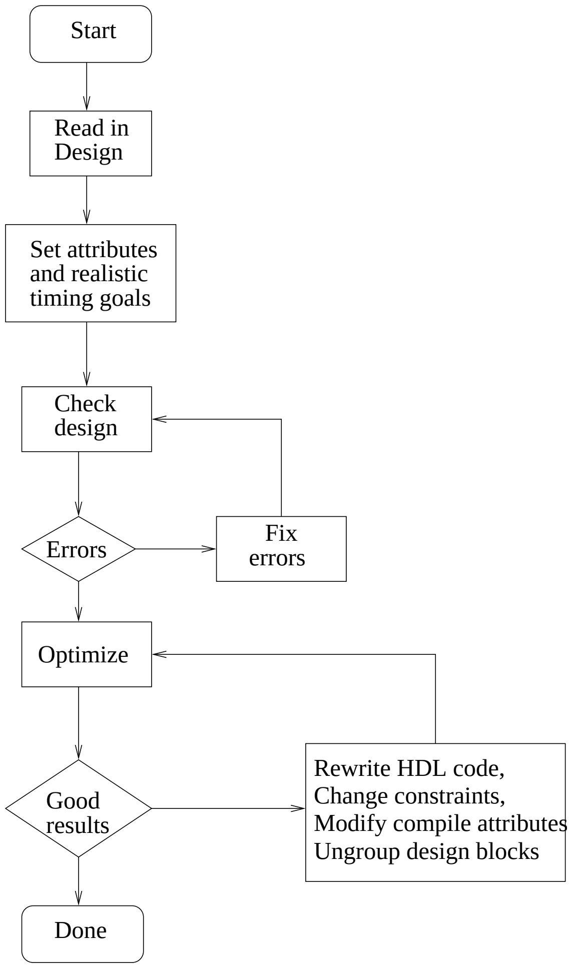
Start
Read in Design
Set attributes and realistic timing goals
Check design
Errors
Fix errors
Optimize
Good results
Rewrite HDL code, Change constraints, Modify compile attributes Ungroup design blocks
Done

Figure 3.1: Generic synthesis flow.

Synthesis hierarchy organization followed can be seen in Figure 3.2. We synthesize low hierarchy blocks, like crosspoints, output and credit schedulers in a flat manner. This is decided in order to achieve maximum optimization on those blocks; the latter is required because, due to the large instantiation number of these blocks (1024 crosspoint and 32 output and credit scheduler modules), lack of optimization can eventually limit the design goals. These low hierarchy blocks are then grouped into higher hierarchy modules: 32 columns include 32 crosspoints and one output scheduler module each, whereas one credit module is used to group the 32 credit schedulers. These groupings are performed in the RTL, but can also be carried out successfully through the Synopsys environment, by using the `group` command. As a result, the top design module consists of merely 32 column and one credit module.

It should be noted that during the Placement and Routing (P&R) design stage, we observed that the tool could not successfully perform P&R of these

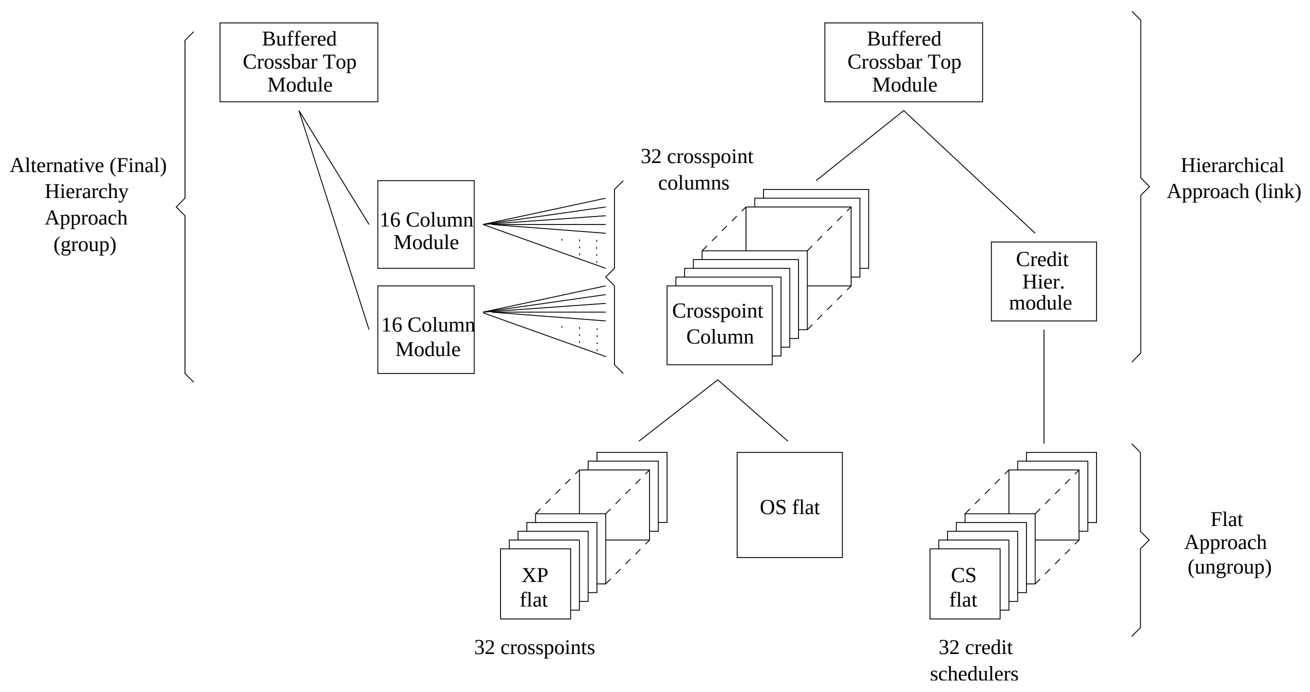

Figure 3.2: Synthesis hierarchy organization: crosspoint modules (XP), output schedulers (OS) and credit schedulers (CS) were synthesized in a flat manner. Each crossbar column was synthesized by linking 32 cosspoints and one output scheduler module. The 32 credit scheduler modules were linked into a higher hierarchy module, as well. The 32 crosspoint columns and the credit module were linked into the top hierarchy module. Later on, it was realized that placement and routing would be performed easier if we added another level of hierarchy, by grouping the 32 columns into two sets of 16; this procedure required circuit resynthesis and is illustrated on the left side of the figure.

33 modules; this was mainly due to the large size of files handled by the tool. As a result, we decided to go back to the synthesis stage and regroup the 32 columns into 2 sets of 16; P&R would therefore be performed hierarchically into these 2 modules, which would connect to the credit block in the top-level. This grouping was carried out through the Synopsys environment, by using the `group` command.

3.4 Synthesis Results

The synthesis process is completed relatively easily and timing contraints are met, while circuit area is kept to a minimum. Timing constraints are of the greatest importance, as we opted for a clock frequency of 300 MHz (3.35 nsec clock cycle); we were constrained to 300 MHz due to the fact that it was the maximum operating frequency of the 2-port SRAMs we had. Hence, we achieve 300 $Gbit/sec$ incoming switch throughput (32 inputs of 9.6 $Gbit/sec$ link throughput). Synthesis results can be seen in Table 3.1.

Module (number of instances)	Gates (K)	Flip-Flops (K)	SRAM 2-port (bits)	Area $0.18\mu m$ (mm^2)	Area $0.13\mu m$ (mm^2)
XPD (1024)	65.0	91.0		9.30	4.2
XPM (1024)			16 M	286.00	130.0
OS (32)	68.0	9.7		3.25	1.5
CS (32)	27.5	6.4		1.45	0.6
Total	162.7	107.1	16 M	300.00	136.3

Table 3.1: Synthesis results: total numbers shown, including all instances. XPD, XPM, OS and CS refer to crosspoint datapath, crosspoint memories, output schedulers and credit schedulers respectively. Memory area is computed from datasheets [43].

Although wiring area is not yet included [1], it is evident that circuit logic area occupies 5% of the total chip core area, leaving the rest 95% for the 1024 memories. $0.13\mu m$ results are computed by extrapolation. The extrapolated area of $140mm^2$ shows that the 32×32 buffered crossbar switch is feasible.

An interesting aspect of any switch architecture is its cost in a per-input, per-output and per-crosspoint basis. Those costs are shown in Table 3.2. In order to produce these results, we design four different switches based on the proposed architecture (4×4, 8×8, 16×16 and 32×32) and average the gate and flip flop sums per input, output and crosspoint, accordingly. Since in each input and output port there is a scheduler whose complexity depends on the numbers of its inputs, the per-port complexity depends on the actual number of the input ports of the switch. The per-input cost corresponds to the costs of the CS modules, the per-output cost consists mainly of the cost of the OS, whereas the crosspoint cost includes the costs of the XPM and XPD. As this table clearly demonstrates, the per-input and per-output costs are not significant. Moreover, although the figures of the crosspoint costs seem relatively high, the table indicates that large switches, based on the proposed architecture, can be easily implemented in today's state-of-the art high density technologies.

3.5 Post-synthesis Verification & Conclusions

Post-synthesis verification is possibly the most important phase of the synthesis flow. It aims at testing whether the initial RTL design has the same behaviour as the Gate-Level netlist produced by the synthesis tool. In most

[1]But is included in the placement & routing results presented in the next chapter.

	Gates	FF
per-input	27 x i	2.5 x i
per-output	70 x i	10 x i
per-crosspoint	63	70

Table 3.2: Per input/output/crosspoint logic costs; i in the number of the input ports of the switch.

of the cases the initial results are not the same, and the designer has to carefully investigate the reason for the erroneous behaviour of the Gate-Level netlist. Usual mistakes happen when the circuit does not reset correctly, a mistake that can pass unseen from the HDL compiler & simulator, but, of course, the actual circuit will not work correctly.

Our synthesized Gate-Level netlist is imported back to the Verilog compiler (Cadence NCLaunch [20]) and is tested within the same environment as the initial RTL design. The final netlist proves to behave correctly. The netlist is now ready to be imported to the Place & Route tool for the final phase of the design process.

Chapter 4

Placement and Routing

In this chapter we explore the Placement and Routing (P&R) process followed when implementing the 32×32 buffered crossbar switch examined in the previous chapters. This process was a major part of this thesis, both in time and effort terms. Although the chip is "regular", as shown in the next section, it proves too large for the available P&R tool to handle in a flat manner; as a result, we have to resort in a hierarchical P&R process, which poses many difficulties, in terms of choosing the optimal hierarchy organization that will reduce interface cost and result in a regular final chip aspect ratio. The rest of this chapter is as follows: at first we compare flat versus hierarchical P&R and present the exact flow followed in our case. Next, we examine the various hierarchy organizations proposed during the P&R process and propose our final organization, followed by a thorough presentation of the P&R process for the major hierarchy components of the switch. Last of all, we present our post-P&R area and power consumption results.

4.1 Flat vs. Hierarchical Placement & Routing

The 32×32 crossbar chip is a highly regular chip, consisting merely of 32 identical "rows" and 32 indentical "columns" (see Figure 2.1). P&R can be therefore be quite straightforward, in the presence of a tool destined for "hand design": the designer places each crosspoint at specific positions and wires are routed through each row and column. Because such a tool does not exist, we have to use automatic P&R tools. It should be noted that automatic P&R tools offer some freedom for user-defined logic cell/block placement, but such a process is difficult when carried out for a large number of logic cells/blocks. Furthermore, the tools we can use have gate limitations in their automatic P&R algorithms, as a result we follow a hierarchical P&R approach. Last, the number and size of instances must also be taken into account, as the presence of a large number of complicated instances (which means timing and layout file complexity) can pose great difficulties in the P&R process.

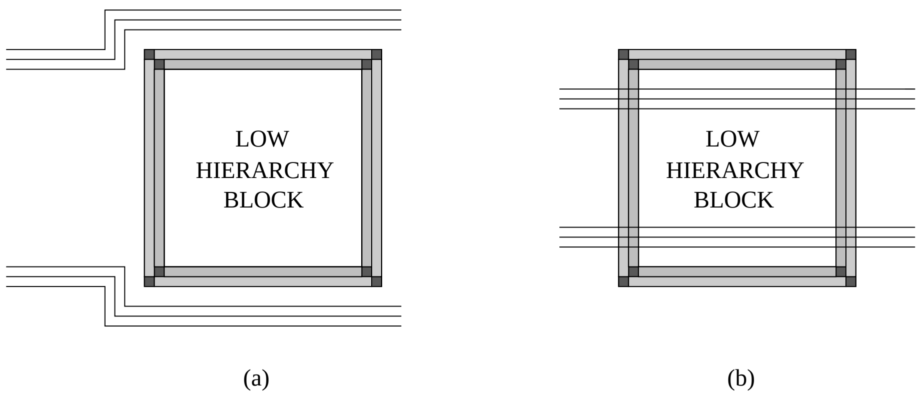

Figure 4.1: Routing global wires in hierarchical designs: hierarchy prevents the P&R tools from routing global wires through lower hierarchy components, unless the designer has explicitly stated so.

We perform hierarchical P&R by placing and routing the lower hierarchy blocks, then importing them to the upper hierarchy level; as a result, the P&R tool treats lower-level blocks as black boxes with specific interfaces, as is the case with memories. Unfortunately, this approach poses some disadvantages, compared to completely flat P&R:

- hierarchy hinders the algorithms that perform global (area, timing) optimizations, which means that hierarchy components must be fully optimized before porting them to the upper hierarchy level
- area is added in the total design, since every block in the hierarchy must have dedicated power rings, in order to terminate the power (VDD) and ground (VSS) lines that traverse the chip horizontally (which provide power and ground connections to the standard cells) [1]

Another hierarchy-related point is that we often wish to allow wires to travel on top of or through the low hierarchy components, which helps the P&R tool to cope with timing violations that might be caused by otherwise being forced to route wires up and down the low hierarchy blocks, as seen in Figure 4.1. The latter can be accomplished in two ways: (a) place and route the low hierarchy blocks at lower metal layers (e.g. up to metal 5), so that higher hierarchy wires can travel *on top* of these blocks; and (b) provide *feedthroughs* in low hierarchy components, thus letting wires travel through each block. Both these solutions are worse than what would be used in a completely flat process, where the tool's advanced algorithms can find the optimal solution. It should be noted that the exact position of the input/outputs (I/O) can also affect our decisions.

[1]Power and ground lines are in metal 1 and 2, as a result these lines cannot be routed through the black boxes present (such as memories at the lowest level of hierarchy).

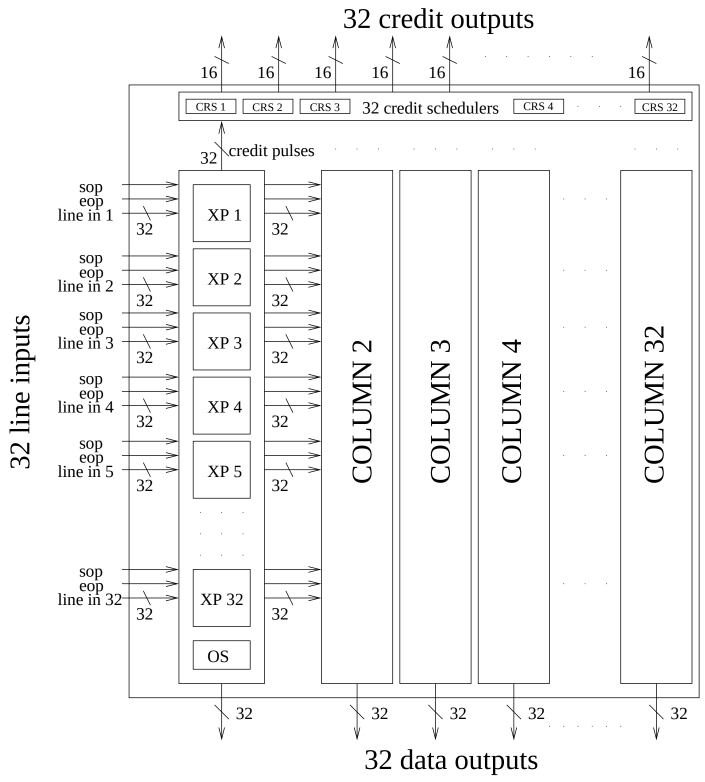

Figure 4.2: Switch internal organization: block interfaces.

4.2 Choosing a Hierarchy Organization

As can be seen in Figure 4.2, each switch column has thirty-two 32-bit data inputs, 64 single bit `sop` and `eop` inputs, one 32-bit data output and thirty-two 16-bit credit outputs; these amount to 1088 inputs and 48 outputs. From the figure it is evident that input lines have to connect to every switch column; as a result, in order to avoid long and unwanted data lines, we come up with the two strategies mentioned in the previous section: (a) place and route each column up to a lower metal layer, so that data lines can travel over every column; and (b) pass all data lines through every column from the RTL level, using dummy wires. We conclude that the latter solution is preferrable, because, as we are aiming at the smalest possible column area, solution (a) will most probably be not as "area-aware" as solution (b). This is because, if we follow solution (a), each column will be placed & routed up to metal layer 5, leaving the (highest) metal layer 6 unrouted by local column wires; hence, column area optimization will not be accomplished.

Another interesting point is memory orientation. As shown in Figure 4.4, memory interface lies on the three of the four sides of the memory, with data lines lying on the south side of its surface. This initially leads us to place every memory with its long side on the "x-axis", which results in a fairly "low" and "wide" column (see Figure 4.3(a)).

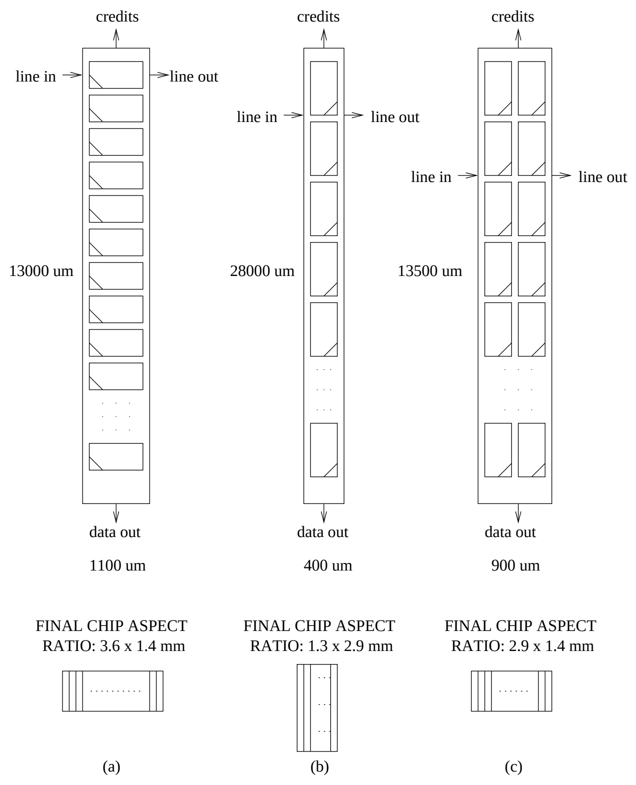

Figure 4.3: Column orientation alternatives: Case (a) has the worst aspect ratio, almost 1:3 and was thus abandoned. Case (b), although having better and acceptable aspect ratio (1:2), was unroutable. Case (c) proved to be easy to route, without making the aspect ratio any worse. Chip core aspect ratios can be seen in the bottom of the figure.

As a result, data lines enter the column from the west sides (and exit from the opposite side), credit signals exit from the north side, leaving the south side for data outputs. Fortunately, the P&R tool we use, Cadence Encounter, offers the ability to manualy place hard blocks (memories) and even soft blocks, like our designed modules, although not so exact, as is the case with hard blocks. Unfortunately, this organization leads to an aspect ratio of 1:3 for the whole chip core, which is obviously not prefferable. By placing the memories with their "small" side on the x-axis (see Figure 4.3(b)), although resulting in an aspect ratio of 1:2, will also result in very long columns, which makes P&R difficult, if not impossible, as long wires have to traverse the whole column height; the tool will thus have to buffer strongly these wires, which adds to the total column area and core power consumption.

On the other hand, the placement of Figure 4.3(c) seems to be the best alternative present. In this organization, the 32 columns are divided in 2 and placed with their "small" sides on the x-axis. As a result, total column width is approximately 900 μm and the corresponding height 14000 μm,

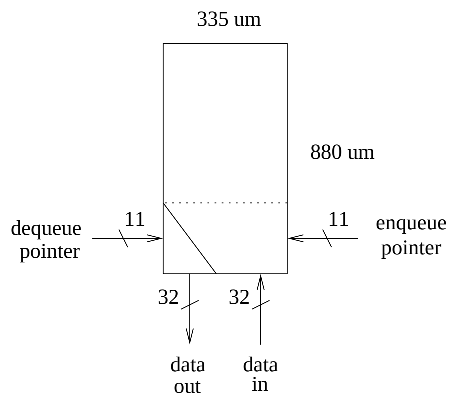

Figure 4.4: Memory layout: all signals connect to the lower 1/3 of the memory (noted with the dotted line). Memory is placed & routed up to metal layer 4.

which results in a $15000\mu m \times 29000\mu m$ switch core, with total area 420 mm^2 and aspect ratio of 1:2.

4.3 Placement and Routing Flow

We place & route the switch chip core with Cadence Encounter [20]. P&R flow followed can be seen in Figure 4.5. It comprises of various stages, some of them being optional, although important for new design cases; we present a comprehensive P&R tutorial in the Appendix. At first the design is *imported* to the tool. The files needed are technology dependent and are either given by the technology vendor (in our case Virtual Silicon UMC 0.18μm [21]) or are produced by the synthesis tool (in our case Synopsys Design Compiler). The vendor provides the P&R tool with (a) *technology library - .lib* files, which contain the exact electrical characteristics of the library standard cells, (b) *layout - .lef* files, which include standard cell actual layout, pin placement and metal layer usage and (c) *verilog - .v* files, containing the interface of every standard cell. The same vendor also provides the memory models, so the memory *timing library - .tlf* file was also included. From the synthesis tool point of view, the only information needed to be passed on to the P&R tool is the Synopsys *technology mapped - .v* Gate-Level representation of the design.

After importing the design, the designer has to *Floorplan* the various verilog modules and/or black boxes in the actual chip. Modules can be either "softly" placed, that is the tool can alter their shape, or "hard" placed, in which case their shape cannot change significantly or at all during the placement phase. Power rings are created and block rings are added for power/ground termination purposes. Input/Output pins are also placed in this stage: from the switch layout provided in Figure 4.2, it seems obvious to

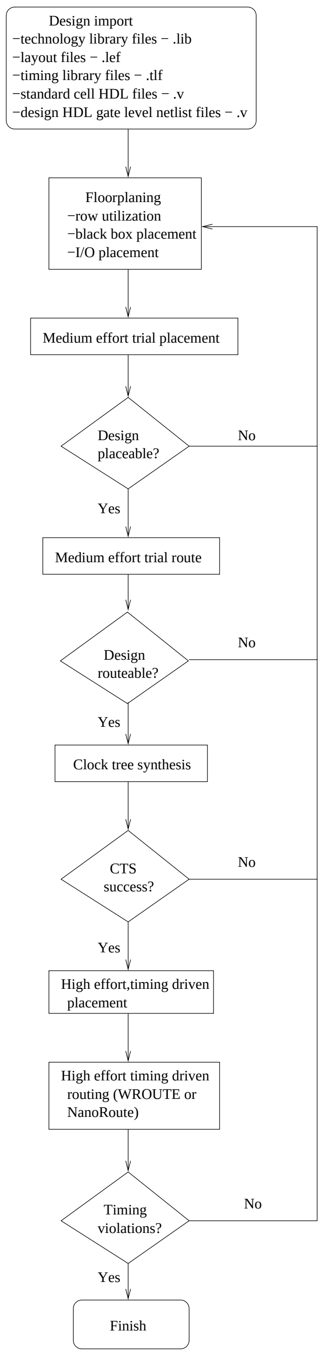
Design import
−technology library files − .lib
−layout files − .lef
−timing library files − .tlf
−standard cell HDL files − .v
−design HDL gate level netlist files − .v
Floorplaning
−row utilization
−black box placement
−I/O placement
Medium effort trial placement
Design placeable?
No
Yes
Medium effort trial route
Design routeable?
No
Yes
Clock tree synthesis
CTS success?
No
Yes
High effort,timing driven placement
High effort timing driven routing (WROUTE or NanoRoute)
Timing violations?
No
Yes
Finish

Figure 4.5: Placement and Routing Flow followed. Some stages (pad placement, crosstalk checks, DFT) were omitted, as we opted in demonstrating the feasibility of the idea, *not* manufacture an actual chip.

place all input pins (data and `sop` / `eop` signals) on the west side of the chip, data output lines on the south side and credit outputs on the north side. Note that decisions carried out in this stage are of the greatest importance, and need thorough “paper-design”, as they affect the whole design process and the final result, both in area and in timing terms. The most important decisions are, of course, chip exact layout, utilization percentage (i.e. amount of “empty” row space among standard cell rows used for wiring) and I/O placement. Utilization value was not given explicitly, as we are aiming at a predefined size of the module under P&R; so we state utilization implicitly by specifying the exact sizes of all the modules.

Low effort *Trial Placement* comes next. During this stage, the tool tries to place the standard cells and/or black boxes of the design and produces the initial placement results, that will provide us with information regarding design size errors and wrong utilization decisions, as well as an estimation of net length. The tool uses heuristic algorithms and tries to improve placement results (net length) for about 15 “trial rounds”.

Next comes a medium effort *Trial route* phase, in which the tool tries to route the previously placed design. This phase will let us know whether the design is routable and in which locations on the actual chip wire overflows occur. The latter can be due to lack of space for wire routing among standard cells and/or black boxes and is the result of optimistic utilization decisions or wrong placement of black boxes or I/O pins in the floorplan stage. If the tool does not provide us with good and promising results, we have to go back to the floorplaning stage and “give some space”, or rearrange the blocks/pins. During this phase, an initial timing check of the design can be carried out, although the clock signal is not yet placed and routed. This enables us to check wrong placement desicions, which have resulted in extra and needless wiring.

When trial routing is complete, we move to the *Clock Tree Placement* phase. During this stage the P&R tool reads the *clock specification* file and tries to place and route the clock tree (or trees, in our case) in the design. This is carried out with buffer insertions and thorough computations of the tree node weights, so as to minimize clock skew (uncertainty) produced by unbalanced clock tree placement.

After the clock tree placement and routing, we perform a highly detailed *Timing Driven Placement*, which possibly moves some standard cells to new positions, according to the timing contraints we have imported at the initial stage of the process and taking into account the clock tree and the register positions.

This phase is followed by the most important *Final Route* phase, which is the most time- and resource-consuming part of the whole P&R process. Setup and hold times of registers are taken into account and, having in mind the operating frequency constraint, the tools tries to buffer wires and resize standard cells in order to minimize clock skew. If the clock frequency requirements are not met after this phase is complete, we have to perform in-place optimizations; should they fail, we have to go back to the initial stages and re-floorplan the module, which means changing row utilization and/or moving black boxes to new positions.

4.4 Column Placement and Routing

As mentioned earlier, after thorough investigation of various column placement and routing organizations, we came up to a solution that seemed to be nearly optimal; this was accomplished by organizing all 32 memories of each crossbar column into 2 columns of 16 each and placing the memories with their "small" sides on the x-axis (see 4.3(c)).

By this organization, the input lines enter the column from the west side, the data outputs are located on the south side, and the credit outputs come out from the north side of the column. The input line feedthroughs (data input lines and `sop` and `eop` signals) exit the column's east side, in order to enter the next column's respective west inputs. Note that all inputs and the east outputs are the internal interface of the crosspoint modules, whereas data and credit outputs come from the output scheduler module.

After placing the memories "by hand", we also had to restrict the output scheduler module placement to the lower half of the column; this was carried out because it was noted that if there were no restrictions imposed on the module placement, the tool could not place the output scheduler's standard cells cleverly enough, in order to avoid timing violations. In such cases the designer must "assist" the tool, either by restricting the area (up to a reasonable size, of course), changing I/O pin placement or even changing the timing requirements themselves.

Finally, we should note that the credit signals, which originate from the output scheduler module, have to be routed across a long way, until they output the column module. As a result, if we do not locate "by hand" the 32 output credit pulse registers of each column at the column's northest position (right "below" the credit outputs), the tool is unable to meet the timing goals. Actually, this problem is faced in conjunction with the respective credit scheduler inputs, which are located in another component on top of the crossbar columns (see Figure 4.2). A final placed and routed view of the column can be seen in Figure 4.6.

4.5 Credit Scheduler Placement and Routing

As mentioned earlier, there exist 32 credit schedulers in the crossbar chip and are responsible for sending in a round-robin manner the credit values to the line card schedulers. There are also two alternatives in this P&R process: flat and hierarchical. Flat P&R of the credit schedulers proves to be a very time consuming task, as it has to be carried out during the final stage of the P&R, in which case the tool has to handle all 32 column instances and the flat credit schedulers as well. As a result, large layout, netlist and timing files have to be processed by the tool, and at the same time, the 32 credit scheduler blocks have to be placed & routed in a flat manner. As a result, we decide to P&R the credit schedulers in a very narrow "line" and then

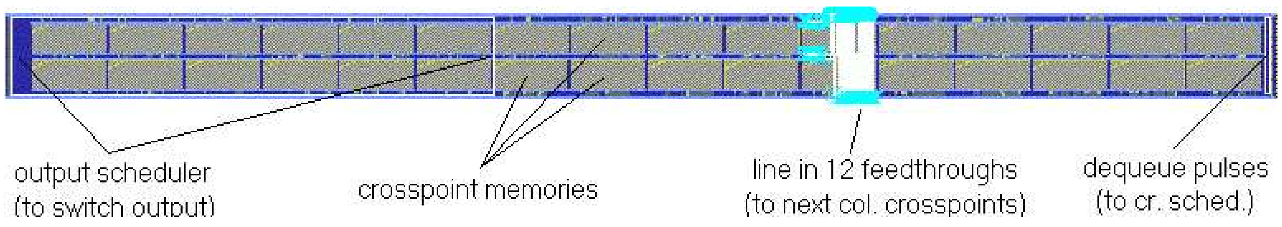

Figure 4.6: Final crosspoint column layout: The picture is rotated 90^o clockwise for space reasons. The output scheduler is located on the left (actual bottom) half of the column, with the data output bus coming from the west (actual south) side. Notice the line in/out metal 6 feedthroughs, that connect to the corresponding line in data inputs of the adjacent column. Dequeue pulse registers were manually located on the east (actual north) column side, in order to avoid inter-block timing violations (their outputs, credit pulses, connect to the credit schedulers, which will be placed on the right (actual north) side of the columns.

"hand-place" that block just above the column blocks (see Figure 4.2).

By following this organization, the credit scheduler input signals (1024 credit signals) enter the block from the south side, while the credit outputs (32 16-bit buses) come out from the north side. We also have to place the credit scheduler modules "by hand" at specific positions inside the "line" block and restrict those modules, in order to prevent the tool from moving standard cells of the 32 modules at far located positions.

Credit scheduler P&R is thus straighforward and is completed easily, also meeting the timing requirements. The credit input pulse registers must be placed as close to the credit scheduler block inputs possible and the tool manages to comply with this requirement, without human intervention. As a result, the credit scheduler input registers and the column credit pulse output registers (destination and source respectively) are located very close to each other, thus meeting the clock requirements of these adjacent blocks. Note that those requirements are not met with the initial column placement decisions and are only solved by placing "by hand" the credit pulse output registers, as mentioned in the previous subsection.

4.6 Adding Another Level of Hierarchy: Top Level Placement and Routing

After organizing the switch core in 32 column modules and one module consisting of credit schedulers, we realize that the tool cannot handle the placement and routing of so many and so large instances within a respectable time. This has to do primarily with the machine that was used to P&R the chip [2] and the large size of timing and layout files that have to be analysed by the tool in the final stage. Note that these files are crucial in determining whether timing will be met in the final chip.

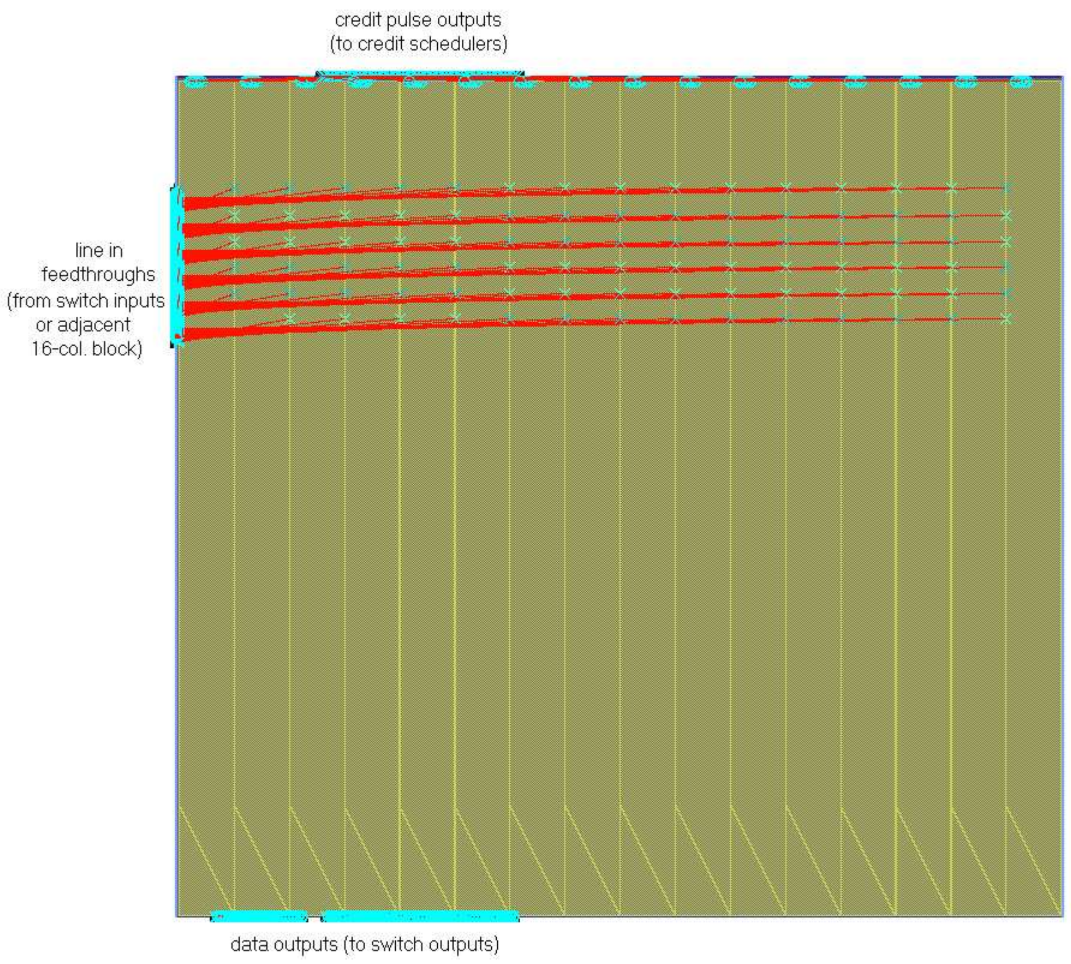

Figure 4.7: 16-column higher level P&R: Data inputs can be seen connecting to all 16 columns via feedthroughs.

As a result, we conclude that if we add another level of hierarchy in the column organization, the problem will be solved. Organizing the columns in higher levels of hierarchy is an easy task and is carried out by grouping them during the synthesis procedure by using the Synopsys `group` command. A reasonable hierarchy choice is to group the 32 columns in 2 pairs of 16 each, placing and routing these two blocks separately and then combining only three highest hierarchy blocks (two groups of 16 columns and one block of credit schedulers). This task is carried out successfully; a 16-column resulting

[2]The only machine we can use has 4 SPARC processors, 400 MHz each, and 3 GB RAM. Use of a faster machine can possibly produce faster results, but hierarchical approach will most probably be anavoidable.

block can be seen in Figure 4.7, while a final chip core layout is shown in Figure 4.8.

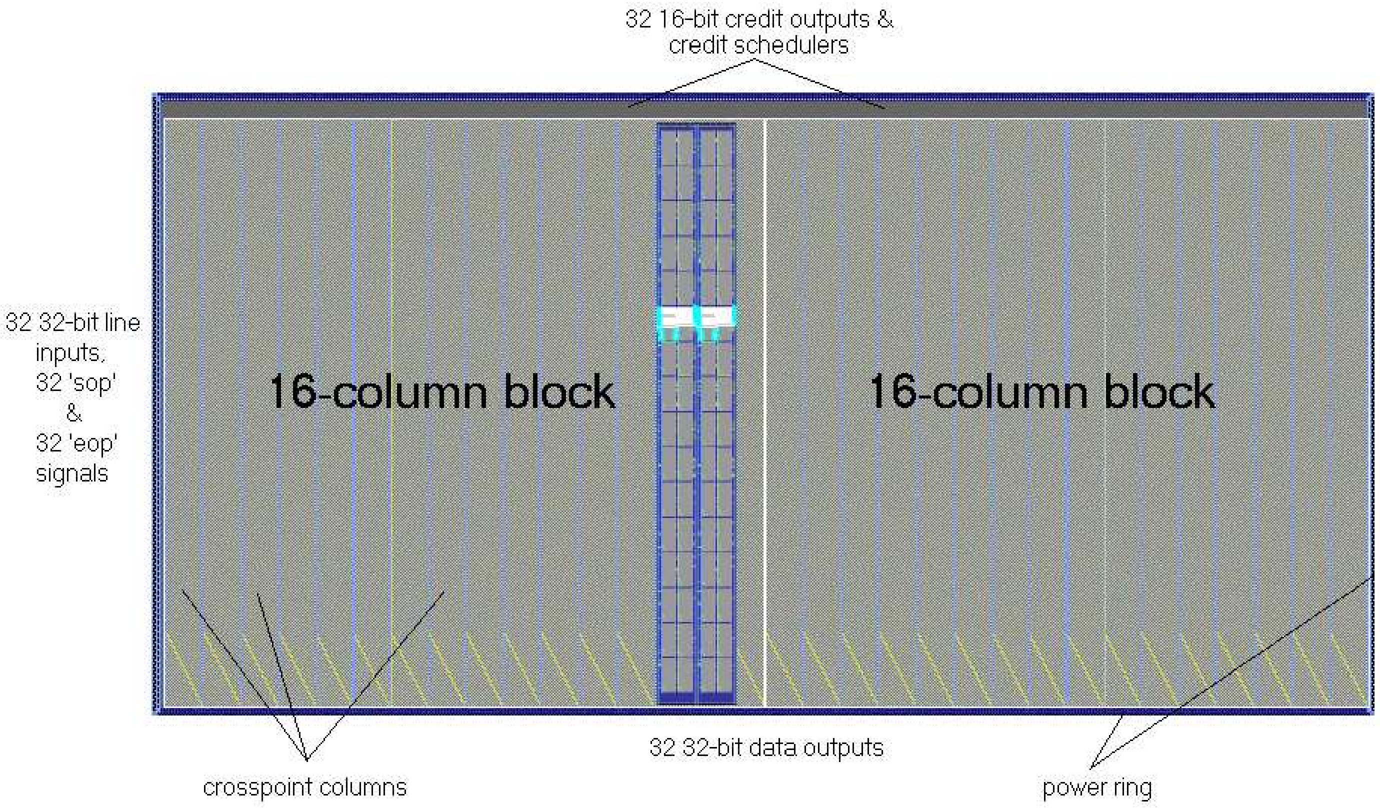
32 16-bit credit outputs & credit schedulers
32 32-bit line inputs, 32 'sop' & 32 'eop' signals
16-column block
16-column block
crosspoint columns
32 32-bit data outputs
power ring

Figure 4.8: Final switch core layout: Core aspect ratio is 1:2. Data inputs are fed from the west side of the chip, while credit and data outputs come from the north and south side respectively. The two higher-level 16-column blocks are also visible. Two adjacent columns are also visible, with data lines of input 12 passing through them. Core dimensions are 14×30 mm ($420mm^2$).

4.7 Chip Packaging and Placement and Routing Results

4.7.1 Chip Packaging

The most important result of the P&R process is, of course, the total chip core area, which usually has to be as small as possible, in order to decrease manufacturing cost and improve yield. Having final chip core area and package type, die area can be computed. There are times, however, when chip core area cannot be brought to a minimum, as the chip I/O pins/pads are so many or so large, that the core is *pad-limited*; this is because: (a) the bonding pads have to be placed at some minimum distance from each other, but the periphery of the core cannot fit this requirement; (b) the tranceiver circuits (SERDES) that have to be used occupy periphery larger than that of the core. The opposite situation is called *core-limited* and means that the core is large enough to eventually determine die area.

Note that depending on the number of I/O signals and chip periphery length, some packaging technologies may result in a pad- and others in a core-limited chip. For example, one of the most popular package types used today for ASICs is Ball Grid Array (BGA) [38]. BGA can be either *Wire Bond* or *Flip Chip*. In Wire Bond BGA, I/O pins have to be placed at the chip core periphery, as shown in Figure 4.9(a), and then connected to bonding pads through special I/O cells. Bonding pads are then connected via bond wires to the BGA package balls. Pads cannot be placed too close to each other: today's values are $70\mu m$ pitch for single-row and 40-$50\mu m$ staggered pitch if two rows are used. This places restrictions on the maximum number of I/Os that the chip can fit, as they are proportional to its periphery; bonding pad number is also proportional to die area square root. As a result, although this packaging method is cheap, it cannot be used when we have large number of I/Os. Furthermore, bonding wires are relatively long (3-5 mm) and therefore the connections have high inductance. Flip Chip BGA 4.9(b) can be used instead: Although more expensive, this method allows much more I/O pins to be connected to the (flipped) chip core, which saves die area in pad-limited designs. Signal inductance is also limited, as interconnect length is an order of magnitude less than that of Wire Bond BGA.

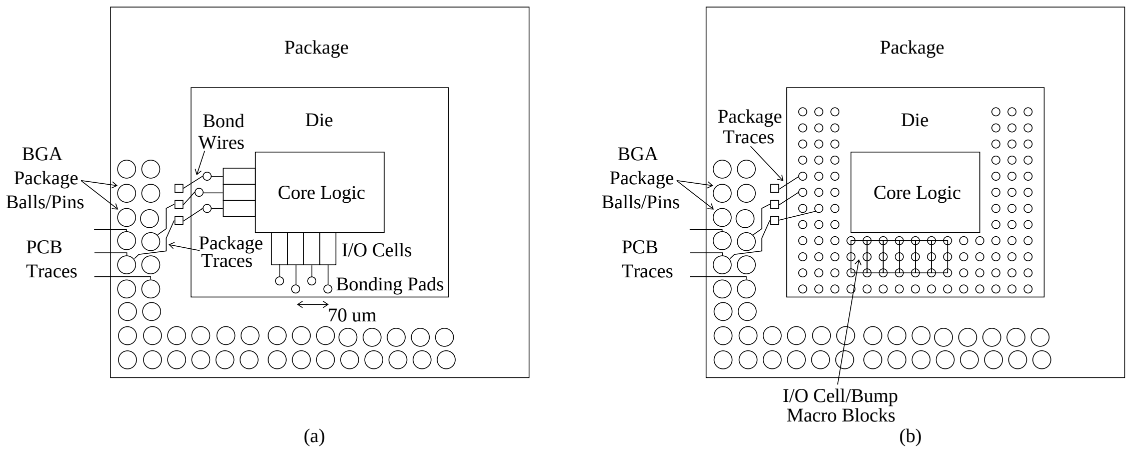

Figure 4.9: Wire Bond (a) and Flip Chip (b) BGA packaging solutions: if the chip proves to be pad-limited in a Wire Bond approach, turning into Flip Chip may prevent the core area increase that would have to be adopted if (a) was used.

In the case of the 32×32 buffered crossbar switch, fast and relatively large I/O tranceivers (SERDES) have to be used, as input and output data link speed is nearly 10 $Gbit/sec$, whereas credit output link speed is 1/4 of the former, namely 2.5 *Gbit/sec*. If 3.125 *Gbit/sec* tranceivers (2.5 *Gbit/sec* useful rate) are used, then 4 and one uni-directional tranceivers are needed for every data line and credit line respectively. Thus, we need $(32+32)\times4+32 = 288$ uni-directional tranceivers for the whole chip, in addition to 64 traditional wires for the `sop` and `eop` signals. If the 32×32 buffered crossbar switch chip is to be placed in a Wire Bond package, then all tranceiver modules will have to placed on the core periphery. Taking into account that a uni-directional 3.125 *Gbit/sec* tranceiver has an approximated area of $500 \times 500\mu m$ in a 0.18 μm technology [3], then periphery length must be 144 *mm*. From the results presented shortly, chip core dimensions are $1.4 \times 3cm$, thus its periphery is 88 *mm*. As a result, our design is pad-limited. Use of the alternative Flip Chip packaging method might eliminate this phenomenon.

4.7.2 P&R Results

The 32×32 buffered crossbar chip we place & route has a total area of 420 mm^2, with an aspect ratio of around 1:2 (1.4×3 cm). Area results, as well as gate and flip-flop count per module can be seen in Table 4.1.

Module (number of instances)	Gates (K)	Flip-Flops (K)	SRAM 2-port (bits)	Area 0.18μm (mm^2)	Area 0.13μm (mm^2)
XPD (1024)	68	91.0		9.5	4.3
XPM (1024)			16 M	286.0	130.0
OS (32)	100	9.7		4.0	1.8
CS (32)	50	6.4		2.0	0.4
Clock Tree Buffering	7			0.7	0.3
Long Wire Optimization Buffering	38			1.5	0.7
Wiring/Hier. Overhead				116.3	53.0
Total	263	107.1	16 M	420.0	190.5

Table 4.1: Placement and routing results: total numbers shown, including all instances. XPD, XPM, OS and CS refer to crosspoint datapath, crosspoint memories, output schedulers and credit schedulers respectively. Memory area is computed from its datasheet [43]. Optimization buffering includes buffers added after the post-route optimization phase. 0.13μm area results are computed by extrapolation.

Note the area differences compared to the corresponding synthesis results (shown in Table 3.1): Area is 40% larger, while gate count has increased by 60%. These discrepancies are due to: (a) wiring; (b) hierarchy overhead; and (c) clock tree and optimization buffer insertion. It is reminded that each

[3]This result is an approximation taken from [27]: there, a single-chip *bi-directional* 3.125 *Gbit/sec* tranceiver circuit is $1 \times 1mm$ in 0.18 μm.

column had to be fully optimized before importing it to the top-level core. As a result, in most cases the tool added buffers in all hierarchy levels in order to meet timing constraints; these buffers, although usually small, are numerous: if we take into account only each column interface, then buffering each column outputs adds almost 35000 buffers in the switch core, thus increasing its area and power consumption [4].

4.8 Conclusions: Dealing With Large and Regular Designs

The P&R process presented in the previous subsections poses many differences compared to flat P&R, which is usually followed in the case of small designs. First of all, this process was imposed because of the large number of gates, but, mainly, by the large number of black boxes (memories) which lie in the switch. The latter results from the O(N^2) complexity of the crossbar switch, which is an inherent characteristic of the crossbar architecture. Hierarchical P&R has itself many challenges, the most important being the correct decision of the hierarchy that must be chosen in order to help, not the designer, but the tool, in performing quick and resource-aware P&R.

Wiring seems to be the most crucial factor of the hierarchical P&R process. Wires that interconnect low-level modules must be taken into account, in order to prevent the tool from using unneedlessly long wiring while connecting the modules. In our case, from the solutions investigated, manual creation of feedthroughs within the modules seemed to be the most appropriate, as it led to straightforward layout with small area overhead. Another important factor that must be taken into account is the final pin layout, which inevitably influences the P&R process speed and success. It should be noted that hierarchical decisions taken during P&R phase may not be the same as the ones taken during synthesis. These hierarchical changes can be performed both in the RTL and the Gate Level, the first having the disadvantage that a slight change in hierarchy directly affects RTL, needing careful and error-prone changes. In the Gate Level, hierarchy changes can be easily carried out by using the Synopsys `group` command.

Of course, hierarchical P&R most often leeds to larger designs, as there is area overhead that has to be added in order to P&R lower hierarchy blocks. Unfortunately, in our case, we had to make a tradeoff between following hierarchical P&R, because of the tool's limitations, and producing the smallest chip possible, as the design area (during synthesis) was already closing 300 mm^2. Thus, the tremendous difference between the Gate Level area result and the final placed & routed chip area result (300mm^2 and 420mm^2 respectively), should not surprise us: a growth of 40-50% should be a rule of thumb in large designs, especially when a hierarchical P&R approach is followed.

[4]Buffer insertion effect on power consumption is discussed in the next chapter.

Place & route flow is, undebatably, the most time consuming part of the chip design, as: (a) it demands careful "hand" design and analysis of the alternative organization ways; and (b) each stage is time consuming and usually the designer has to restart the flow from the very beginning if something goes wrong. This "back-and-forth" situation can be carried out tens of times before the first promising results are reported; therefore, for industrial designs, careful management of the project has to be performed in order to meet time to market demands; the faster the design is designed, verified and synthesised, the better.

Finally, it should be noted that the regularity of the design actually assisted us, both by making hierarchy decisions easy and by helping the tool with the P&R process, as the number of different modules in each level was limited: For example, each "column" comprised of 32 identical crosspoint modules and one output scheduler module, whereas the switch was made up of 32 "columns" and one credit scheduler "line". If there was no regularity, P&R of such a large design would probably have been much more time consuming.

Chapter 5

Power Consumption

Total power consumption is the sum of two different types of power: *dynamic* and *static* [47]. Dynamic power is dissipated when the circuit is active, that is, when the voltage on a net changes due to stimulus applied to the circuit. Notice that because voltage on a net can change without necessarily resulting in a logic transition of the output, dynamic power can be dissipated even when an output net doesn't change its logic value (see Figure 5.1). Static power is the power dissipated by a gate when it is not switching, that is, when it is inactive or static. Static power can be dissipated in several ways; the largest percentage results from the source-to-drain transistor subthreshold leakage caused by reduced threshold voltages preventing the gate from completely turning off. Static power is also dissipated when current leaks between the diffusion layers and the substrate of a transistor; For this reason, static power is often called *leakage* power.

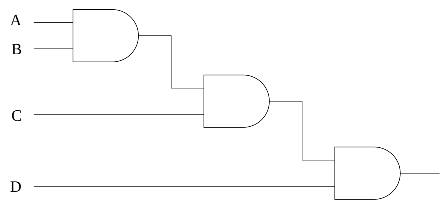

Figure 5.1: Dynamic power is dissipated even if the output does not change: Logic cell inputs A, B and C can switch from logic 0 to 1, and net D can switch from 1 to 0, but the output will remain unaffected, althought dynamic power was consumpted.

The dynamic power dissipation of a circuit is composed of two factors: *switching* power and *internal* power. Switching power results from charging and discharging the load capacitance of an output cell [1] and equals to $P_{SW} = p_{sw} f_{ck} C V_{DD}^2$, where p_{sw} is the probability that a node will switch up *and* down, f_{ck} is the clock frequency, C is the load capacitance and V_{DD} is the

[1]Cells are complex or simple gates in which logic is transformed by synthesis.

supply voltage. Internal power is any power dissipated within the boundary of a cell. During switching, a circuit dissipates internal power by the charging and discharging of any existing capacitances internal to the cell. Internal power includes power dissipated by a momentary short circuit between the P and N transistors of a gate, called *short-circuit* power.

Switching power is the dominant (90% and relatively decreasing) component in current CMOS technologies. Internal (mainly short-circuit) power constitutes about 8% of the total power consumption, with an absolute decreasing trend; the rest 2% (although relatively increasing) comes from leakage (static) power. Although for circuits with fast transistion times (i.e. fast clocks), short-circuit power consumption can be small, for slow-transition ones the short-circuit power dissipation can account for as high as 30% of the total power dissipated by the gate. Short-circuit power is affected by the dimensions of the transistors and the load capacitance of the gate's output. If we ignore short-circuit and leakage power consumption, we observe that for the same technology and same circuit working characteristics (i.e. the same clock frequency, supply voltage and switching activity), switching capacitance is the only factor that can be controlled and reduced (buffering of long nets is a possible solution) [2]. Short-circuit power dissipation is illustrated in Figure 5.2: if we apply positive voltage at the input then, as the signal transitions from low to high, the N-type transistor turns on and the P-type transistor turns off. However, for a short time during the input transistion, both P- and N-type transistors are switched on simultaneously. During this time, current I_{SC} flows from V_{DD} to GND, causing the dissipation of short-circuit power (P_{SC}). Figure 5.2 also illustrates where exactly

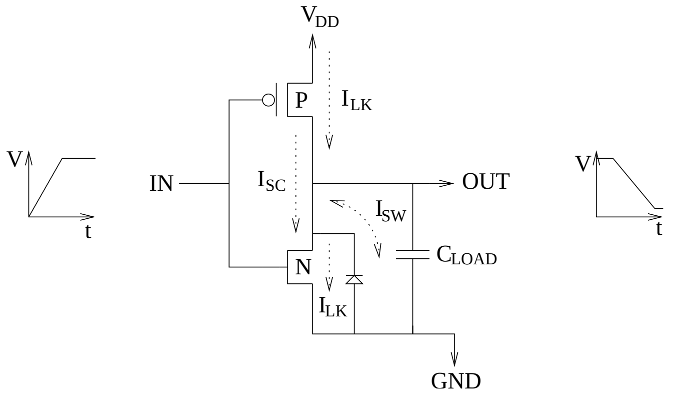

Figure 5.2: Components of power dissipation illustrated in the case of an inverter: I_{LK}, I_{SC}, and I_{SW} are leakage, short-circuit and switching currents respectively.

dynamic (switching and short-circuit) power is dissipated.

[2]Capacitance limitation is also the key factor that has to be minimized in order to avoid timing violations.

In the following sections we will talk about the work carried out in measuring and reducing switch (especially crossbar) power consumption, briefly introduce the Gate Level and Placement & Routing power estimation flows, and present the power estimation results of our 32×32 buffered crossbar design. In the appendix we talk about the most important power minimization techniques that can be applied in the RTL and Gate Level design stages.

5.1 Crossbar Switch Power Consumption Estimation: Work Done Sofar

Various researchers have tried to come up with crossbar power consumption conclusions, as well as discover the most power consuming components of such devices. Ye et al. [44] performed Verilog RTL simulations on the four most important switch fabric topologies (crossbar, fully connected, Banyan and Batcher-Banyan) and introduced a framework to estimate their power consumption [3]. They divided a switch into four main parts (ingress and egress packet processing units, arbiter and switch fabric) and analysed only the latter component. "Bit Energy" is also introduced; it comprises of the energy consumed by a single bit travelling on switching nodes, in buffer memories and on interconnection wires. They concluded that (a) interconnect contention increases significantly switch power consumption [4]; (b) for switches with small number of ports, internal node switching dominates, while in multi-port switches (beyond 32×32), interconnect power is the dominant factor. Their simulation results show that a 32×32 bufferless crossbar switching fabric consumes approximately 300 mW, under uniform traffic conditions and load almost 60%.

Wang et al. [45] developed an architectural-level power-performance simulator for switches and routers, called Orion. They introduced router power models of the three most power consuming router building blocks, namely memories, crossbars and arbiters. The authors applied their theoretical models to two commercial routers, the Alpha 21364 and the IBM Infiniband 8-port 12X. Their most interesting (for the purposes of this work) result was the power estimation of the Alpha 21364's switching fabric, which comprises of two 8×5 crossbars: the estimated average power consumption was about 700 mW, for 100% input load, whereas worst case power estimation is almost double the figures of the average case. Their results were close to the estimated values published by the companies that design these products. They also came to the conclusion that: (a) crossbar and buffer memory power consumption grow approximately linearly to the data arrival rate, as they are both "datapath" components; (b) the larger the memory, the higher the

[3]Switch I/O consumption is not taken into account in their study.

[4]Nevertheless, a crossbar switching fabric does not suffer from internal blocking, hence interconnect contention is negligible in our case.

power it will comparably consume; (c) router links [5] consume a large amount (almost double) of the switch core total power consumption. Based on the above assumptions, we can conclude that our switch's crosspoint datapath power consumption will be at least 500 mW [6].

Various researchers have provided approximate I/O and power consumption figures and predictions, in order to assist designers in performing efficient power budgeting. According to [35], single-chip I/O power consumption factor is expected to grow, from 16% in the case of a 6 *Gbit/sec* 16×16 switch, to 50% for a 64 *Gbit/sec* 32×32 switch. [18] states that any state-of-the-art switch is severely limited by I/O pins and their power consumption: high-speed and high-density SERDES (tranceivers) available in ASIC libraries today give approximately 125 mW per 2.5 *Gbit/sec* useful bandwidth full-duplex (3.125 *Gbit/sec* in 8B/10B encoding); as a result, a 4 Tbit/sec switch would require about 10^4 pins and would dissipate roughly 256 W of I/O power! This is obviously prohibitive for single-chip implementations: in order to comply with Network Equipment Building Standards (NEBS) rules [33], a single board must not dissipate more than 150-250 W, while single-chip power dissipation must remain under 25 W to avoid hot spots and expensive cooling solutions.

Last of all, [31] states that on-chip buffer memories ($0.18\mu m$) consume 5-10 *mW* in order to support sustainable Gbit bandwidth, while the same figure for chip I/O power consumption grows to 40 *mW/Gbps* in $0.18\mu m$ and 25 *mW/Gbps* in $0.13\mu m$. Thus, I/O throughput is 5 to 10 times higher than on-chip buffer memory throughput in terms of power consumption, which eventually limits the total achievable single-chip switch throughput: a single switch chip with 64 10 *Gbit/sec* incoming links will have an approximate I/O power consumption of 32 W in a $0.13\mu m$ technology. To make matters worse, in order to limit package pin count, the designers would favor high-speed links (e.g. 3.125 *Gbit/sec* SERDES/ tranceivers), which are large enough (almost $1mm^2$ per 3.125 *Gbit/sec* tranceiver in $0.18\mu m$ [27]) to result in a pad-limited chip core. Hence, I/O power consumption is possibly one of the most important factors when designing single-chip switches; moreover, in a buffered crossbar, special care should be taken for memory consumption, as well.

5.2 Power Analysis Flow: RTL vs. Gate Level vs. Post-P&R

Power analysis can be carried out in many design stages, the lowest levels being RTL, Gate Level and post-P&R. The more "abstract" the circuit's

[5] Links include expensive Serializer-Deserializer (SERDES) circuits; the term "tranceiver" is usually used instead.

[6] The actual figure will grow, due to the extra logic included in each crosspoint; such logic is not needed in the unbuffered crossbar implementations analyzed above.

description is, the more imprecise the power analysis results are; hence, RTL power estimation is considered erroneous, whereas the Gate Level and post-P&R ones are more accurate. In this thesis we estimated the power consumption on both the Gate Level and post-P&R level (on some design components). In the remainder of this section we will briefly present these power analysis flows, both in the Gate and post-P&R level.

5.2.1 Gate Level Power Estimation

The Gate Level power estimation is carried out by the Synopsys Power Compiler, which is included in the framework of the Synopsys Design Compiler [47]. Synopsys Power Compiler uses various formulas and the information modeled in the logic cell technology library, in order to evaluate the static power of a design. Apart from using equations, Power Compiler uses two- or three-dimentional lookup tables that contain power consumption values for various output load capacitances and input transition times. Note that cells often consume different amounts of internal power depending on which input pin transitions take place or depending on the state of the cell; hence, internal power is state- and path- dependent. For example, in a cell with many levels of logic (see Figure 5.1), a transition on an input signal that affects many levels consumes more power than that of an input signal that affects less levels; Power Compiler takes such differences into account. An example of a cell with varying state-dependent internal power consumption is a RAM, where different amounts of power is dissipated depending on whether it is in read or write mode. Dynamic power is computed by the formula mentioned in the introductory section; capacitance information is obtained by the wireload model [7] specified by the user or the tool, as well as by the technology library information for the gates connected to the net [8].

Cell internal power and net toggling (i.e. the frequency of the input transitions) have a direct effect on the dynamic power of a design. Power Compiler needs such information in order to perform power reporting or optimizations; net toggling is also called *switching activity*. Switching activity can be given to the tool in two ways: (a) by specifying the toggle rate in terms of static probability, in which case the results are probably inaccurate; and (b) by measuring the switching activity under a certain simulation scenario and back-annotating it to the power estimation tool; measurements can be carried out on some or all design objects. Switching activity annotation is performed by compiling and simulating the design within an HDL simulator that can capture switching activity; in our case, Cadence Verilog XL [20] `verilog_toggle` command was used. We can also include interconnect and cell delay information into the annotation tool; this is carried out by loading

[7]A large wireload model, e.g. 80K, should thus be preffered upon a smaller one in cases when the module under synthesis includes large amounts of wiring. This decision, however, cannot be usually made unless physical layout decisions have been made.

[8]Capacitance information can be also back-annotated after physical design.

the “Standard Delay Format” file (`.sdf`) written out by Design Compiler during the synthesis phase. Furthermore, a library forward-annotation file, that contains information from the technology library regarding the cells’ state- and path-dependent power models, can be used; this file, as well as the simulator switching activity annotation result file, is written in the Switching Activity Interchange Format, hence its extension is `.saif`. After switching activity is annotated, it is loaded into the Power Compiler, read and analyzed. The power estimation flow is shown in Figure 5.3. In the Appendix, we describe this procedure in detail.

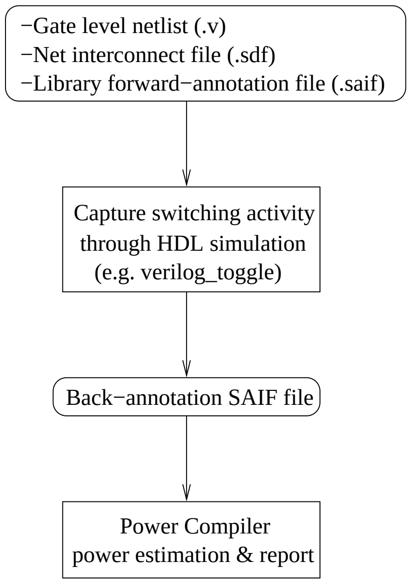
−Gate level netlist (.v)
−Net interconnect file (.sdf)
−Library forward−annotation file (.saif)
Capture switching activity through HDL simulation (e.g. verilog_toggle)
Back−annotation SAIF file
Power Compiler power estimation & report

Figure 5.3: General Gate level power estimation flow.

5.2.2 Post-P&R Power Estimation

The post-P&R power estimation is carried out after the circuit is placed and routed. This kind of power estimation requires the same information as the Gate Level one; that is, the net and cell delay file (`.sdf`), the library annotation file (`.saif`) and, of course, the post-P&R netlist of the design (`.v`). The latter file, as well as the Standard Delay Format (`.sdf`) file are produced by the P&R tool; the `.saif` file is the same one that was used for synthesis. The power estimation procedure is carried out in the Synopsys environment by the Power Compiler tool, and the steps described in the last section are followed.

5.3 Switch Power Consumption Breakdown

We measure both Gate Level and post-P&R steady-state [9] power consumption of the 32×32 buffered crossbar switch; in this section we present the most accurate post-P&R results [10]. Power estimation is carried out by sending minimum-size packets (40 Bytes) back-to-back (100% load), from every input to uniformly selected outputs. Packet payload is selected so as to maximize net switching.

The power consumption breakdown can be seen in figures 5.4 and 5.5.

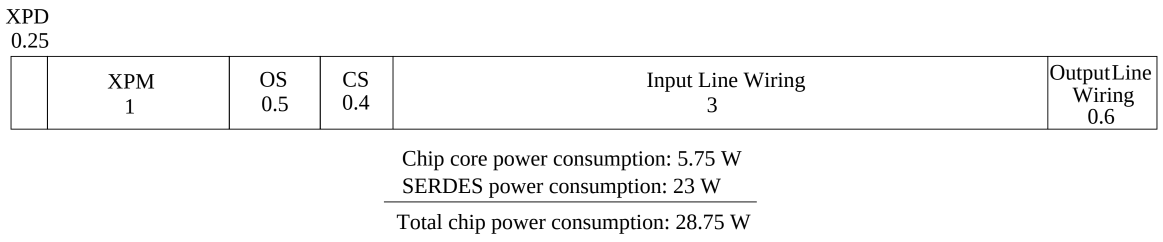

Figure 5.4: Power consumption breakdown: XPD = crosspoint datapath, XPM = crosspoint memories, OS = output schedulers, CS = credit schedulers. All numbers are in Watt.

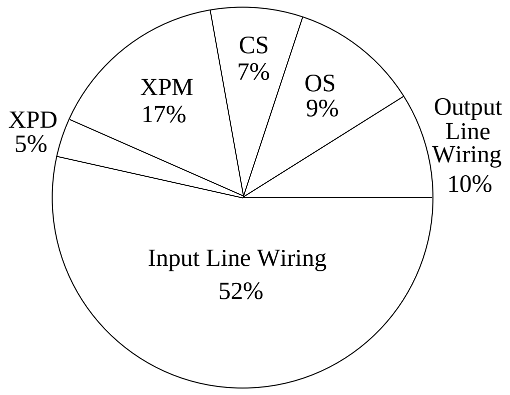

Figure 5.5: Power consumption percentage breakdown of the 32×32 buffered crossbar switch.

Crosspoint datapath consumes $250mW$ of power, while output and credit schedulers dissipate 500 and $400mW$ respectively. Crosspoint memories con-

[9] "Steady-state" power consumption differs from the absolute worst-case one; this is analyzed shortly.

[10] Post-P&R power estimation is followed in order to measure the consumption of each switch component. Power estimation of the total switch core, however, is carried out at the Gate Level.

sume $1W$ of power; this is measured from the memory datasheet [43]. Wiring accounts for $3.6W$ of a total of $5.75W$ chip core power consumption; this breaks down into $3W$ for the long input data lines and `sop` and `eop` signals that traverse the chip's longest dimension, and $600mW$ for the output data wires that traverse the switch columns. It is reminded that there are thirty-two 32-bit input wires and $64(32 + 32)$ `sop` and `eop` signals. These signals are fed into each switch column through high-metal layers. Because a hierarchical P&R approach was imposed, each column should be optimized before importing it into the top-level core; as a result, small buffers were added at each column output, in order to avoid possible future top-level timing violations (see Figure 5.6). This inevitably increases input wire power

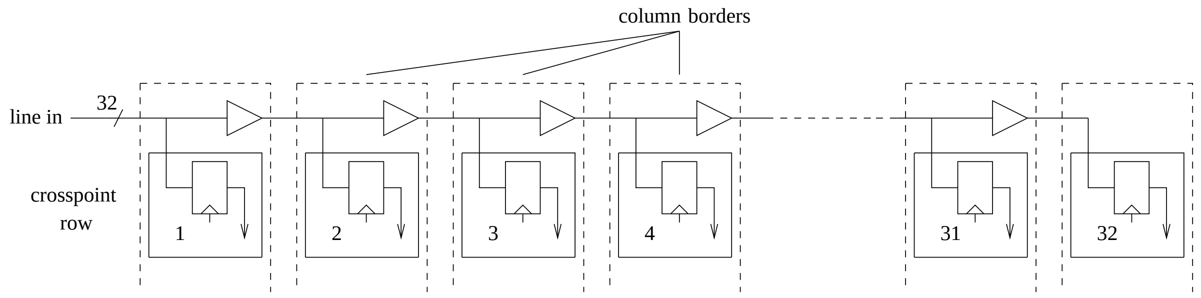

Figure 5.6: Buffering of long input lines: each 32-bit long input line is buffered at the outputs of each column. For 32 such lines and their 64(32+32) accompanying signals, approximately $32 \times 32 \times 32 = 32000$ buffers are needed. These contribute by 85% to total input wire power consumption.

consumption. From the $3W$ consumed for wiring, $2.5W$ are dissipated from the buffers themselves, whereas the rest $500mW$ account for "net switching" power consumption (power dissipated during charging/discharging of wires). If we could P&R the core in a flat manner, then less buffers would be needed. In fact, after synthesizing and placing & routing a single 32-bit input line of length $3cm$ (like the ones needed in this work), its power consumption dropped from $3mW$ to $2.2mW$, and a different buffering approach was followed by the P&R tool (half the buffers were used, but with stronger drive strengths); buffer to wiring consumption ratio remained the same. One solution to decrease wiring power consumption would be to switch off the wire from the last receiver crosspoint on; this can be accomplished by using pipeline registers and disabling the path that does not have to be switched. Adoption of this idea could decrease wiring power consumption of the long horizontal wires considerably, but would increase switch latency, too. It should be noted that this typical switch core *steady*-state consumption of $5.75W$ drops to $3.2W$ in a $0.13\mu m$ technology [11].

[11]This result is estimated by extrapolation based on the technology libraries.

Note that, as also described in [19], worst-case instantaneous power consumption occurs when all inputs receive multicast packets destined to all outputs, in which case all 1024 crosspoint buffers perform write operations. This situation can last up to 1.6 *μseconds*, which is the time needed to fill up a 2 KByte buffer at 10 *Gbit/sec* link speed. Following that, backpressure stops the incoming traffic, and buffer memories perform read accesses only, one crosspoint memory per column (i.e. output) at a time. The *read* power cosumption of crosspoint buffers corresponds to their read throughput, which never exceeds 32 buffers operating in parallel. These buffers are located on the diagonal of the switch. *Write* power consumption corresponds to write throughput; the long-term average write throughput cannot exceed read thoughput, since every byte that is ever written once, must also be read once. Last of all, all 32 output and credit schedulers are functioning at all times during the simulation. Worst-case steady-state scenario is illustrated in Figure 5.7.

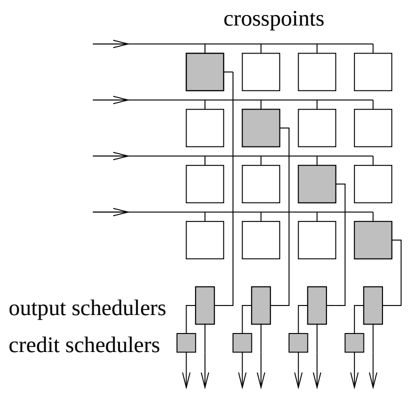

Figure 5.7: Module power activity in worst-case steady-state operation: in the scenario presented (all inputs send randomly minimum-size packets to all outputs), all output and credit schedulers are active. On the other hand, only one crosspoint placed on the diagonal is active per-row and per-column; all others are "filled" with packets. Active modules are in grey.

5.4 Conclusions

It is evident that average power consumption is dominated by pad drivers and SERDES; core consumption accounts for just 20% of the total chip consumption. In turn, core consumption is dominated by long-wire drivers: driving 1024 input data wires across the chip width and 1024 output data wires to the chip outputs (32 links x 32 bits/link = 1024 wires). Buffer memories end up consuming, on the average, only 15% of the core power and 3-4% of the chip power. The almost 30 W total switch power consumption shows the feasibility of our design, as with a medium-priced packaging technology, its power dissipation will not cause any problems.

Chapter 6

Conclusions and Future Work

In this work we presented a novel buffered crossbar organization, and proved its feasibility by designing a 32×32 speedup-less such switch core, with aggregate incoming throughput of 300 $Gbit/sec$; final chip core area was 420 mm^2 in a 0.18 μm and 200 mm^2 in a 0.13 μm CMOS technology, while its power consumption dropped just below 6 Watts (3.2 W in 0.13 μm). Taking into account that a similar switch would approximate 110 mm^2 in the emerging 0.09 μm technologies, we can claim that the adopted organization can become the switching building block of the future.

The fact that output scheduling was distributed over every output, removed the need for a very fast, complicated scheduler, that would control all connection requests; as a result, speedup due to scheduling inefficiencies was eliminated. Switch novelty stems from the fact that no packet segmentation and reassembly (SAR) had to be carried out; hence no speedup due to SAR had to be adopted, either.

We divided the switch into 2 clock domains: the use of 2-port SRAMs at each crosspoint removed the need for elastic buffers at the switch inputs, in order to synchronize all incoming data to a single local clock; the only other synchronization needed concerned a single-bit control signal at each crosspoint, and was dealt with an effective 1-bit synchronizer. The simple and effective Round Robin policy was followed at both the output and credit schedulers. The latter sends credits to the input line cards in a QFC-like manner, thus compensating for possible credit losses.

We synthesized and placed and routed the switch by following hierarchical flows, which was due to the large number of memories (1024) and the resulting chip size. Such flows are usually carried during the development of large or complicated designs, and were completely new approaches to us. Hierarchical flows posed some difficulties though, in particular choosing the optimal hierarchy organization that would assist the tools in producing fast results. P&R, being more complicated, error-prone, and time consuming, covered almost half of this work's duration.

Multiple priority support is a usual feature of commercial switches. To support multiple priorities under the proposed architecture and maintain full efficiency (no HOL or buffer hogging) we need $L*2KB$ of buffer space, where L is the number of priorities. For example, if 8 priorities were used, then our implementation would need $8 \times 2 \times 1024 = 16MBytes$ of total memory, which will be only feasible in future CMOS and/or memory technologies. Fortunately, as Chrysos et al. [24] have shown, *only two* 2KByte buffers are needed per crosspoint: through simulation, and assuming 8 priority levels, the $2 \times 2KByte$ per-crosspoint proposed system performs almost identically to the $8 \times 2KByte$ per-crosspoint ideal system under realistic traffic, whereas, even under highly irregular traffic, average delay of any priority level is not increased by more than 75%, compared to the ideal system. If this organization is adopted, then 8 priorities will only need 32 Mbits of total SRAM memory, divided into 2048 2Kbyte SRAMs, which is feasible in current state-of-the-art CMOS technologies ($0.13/0.09\mu m$).

Also, another interesting point is the fact that, in fast networks, maximum packet size has been increased from 1500 Bytes, to almost 9 KBytes. This poses significant difficulties in case the proposed organization is followed, due to the fact that buffer size equals $RTT + MaximumPacketSize$; hence, under medium RTT values, each buffer would have to be almost 10 KBytes, as a result, a 32×32 single-priority, buffered crossbar would need 80 Mbits of distributed, on-chip SRAM memory, in order to operate efficiently under heavy load. This memory area is too large, even if we take into account the technology downscaling fashion. Embedded DRAM (eDRAM) is a possible solution as, being up to 4 times denser than conventional SRAM, could lower chip area to normal sizes. eDRAM adoption would, of course, impact on the crossbar organization, as well.

Appendix A

Organization of a 4-input WRR Scheduler

As noted in chapter 2, during the initial stages of this thesis we investigated the design and implementation of a 4×4 buffered crossbar switch that would be downloaded in an Altera FPGA. The number of input/output ports (switch size) was bounded by the memory requirements, as 32 KBytes are needed for a 16-crosspoint switch. Operating frequency was specified to 50 MHz, hence the switch would have link speed of 1.5 Gbit/sec link. Although this attempt was later abandoned (and replaced by the more "ambitious" 32×32 buffered crossbar ASIC chip design & implementation), a 4-input Weighted Round Robin output scheduler was designed, in order to select the appropriate flow among the 4 crosspoints of each column of the 4×4 switch. In the next sections we will briefly present various scheduling policies that are being used in network traffic management circuits, followed by a short introduction to Weighted Round Robin (WRR) discipline. Finally, we will present the internal organization and operation of our 4-input WRR output scheduler.

A.1 Introduction to Scheduling Policies

In this subsection we will assume N flows and P priorities; each flow has one of these P priorities [31]. The simplest scheduling discipline is the *strict (static) priority scheduling*, where each flow [1] belongs to a certain static priority level. The scheduler's goal is to serve the highest-priority eligible flow. This discipline is very easy to implement by using a priority enforcer/encoder. Unfortunately, the strict priority scheduling scheme suffers from a starvation issue: if flow i 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; that is why we have to ensure that all levels but the last (lowest priority) one will be be policed or regulated.

[1]Each flow may be an aggregate of many sub-flows

The other extreme of priority scheduling is *plain round robin (equality) scheduling*, which was actually the only scheduling discipline used in the 32×32 buffered crossbar switch presented in this thesis. In plain round robin, the scheduler visits each eligible flow, regardless of its priority or persistency; hence, all flows have the same priority. Round robin scheduling can be implemented in two ways: (a) by using a circular priority encoder/enforcer and (b) by using a linked list of the eligible flow IDs. Method (a) needs simple circuitry (round robin priority enforcers), while efficient organizations exist for small, medium, or large number of flows. Implementation (b) has the difficulty of where to reinsert a flow that was ineligible and has just become eligible, with most of the possible solutions leading to unfairness.

A.2 Weighted Round Robin Scheduling

Last of all, there is a way to serve each flow according to its priority, but with a round robin manner as well. If each flow had a *weight*, that would correspond to its priority, then the round robin policy could be changed by the following way: assume that we maintain a (varying) set of eligible flows and we associate a value, called "Next Service Time" (NST) with each of them. Then, in summary, the scheduler has to (a) find and serve the (eligible) flow that has the minimum (earliest) NST, and (b) re-schedule for a future time the flow just served.

When scheduling starts, each flow is possibly served randomly [2]. When a flow is served, the (virtual) time it will be served again is computed by multiplying its weight [3] by the packet size that is being dequeued by the flow and adding the result to its current "service time". Thus, intuitivelly, the flow will be served some (virtual) time in the future, relative to the time it was being served and its priority (weight). An WRR scheduling example is shown in Figure A.1, where 4 flows, `A` to `D` , with various weights and eligibility statuses are being scheduled. Flow initial service order is random. Last of all, it should be noted that according to the reinsertion time of a flow that was ineligible and suddenly becomes eligible, as well as the Next Service Time computation, many variants of WRR scheduling have been examined in the past, with the most important being "Weighted Fair Queueing (WFQ)", "Self-Clocked Fair Queueing (SCFQ)", "Worst-Case Fair Weighted Fair Queueing (WF^{2}Q)", "Start-time Fair Queueing (SFQ)" and "Virtual Clock".

[2]In the case of multiple low-rate flows and less high-rate flows, unfavorable initilization may cause large jitter to the latter; aggregation of low-rate flows and round robin service inside them solves the problem.

[3]Sometimes the inverse value is used, called "Service Interval" (SI).

Flow	Eligibility status	Weight Factor	Service Interval (SI)
A	YES	50	20
B	YES	30	33
C	YES	10	100
D	NO	10	100

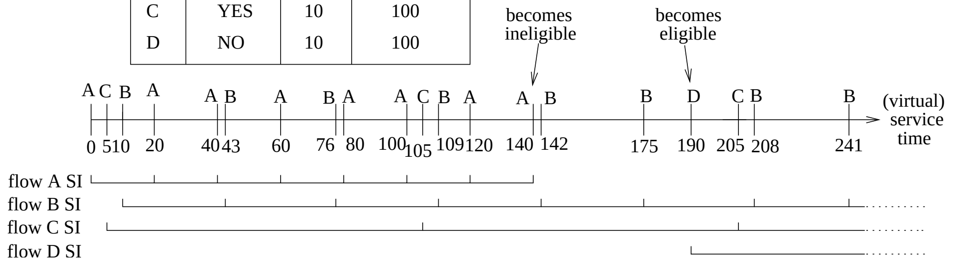

Figure A.1: Priority queue WRR scheduling example: Flows A to D have weights 50%, 30%, 10% and 10% respectively; hence their Service Intervals (SIs) are 20, 33, 100 and 100. Initial service is random. Notice that flow A, which was served at (virtual) time 0, will be served again in time $0+20 = 20$, whereas flow B was served at time 10 and will be re-served at $10 + 33 = 43$.

A.3 4×4 Output Scheduler Organization

Output scheduler block diagra can be seen in Figure A.2. The scheduler has 4 32-bit input data buses, accompanied by 4 synchronized “Start Of Packet” (`sop`) signals. The block outputs a 32-bit `data_out` bus to the switch outputs, as well as a 4-bit wide `deq` bus, which connects to the corresponding `rd_en` signals of the column crosspoints’ memories.

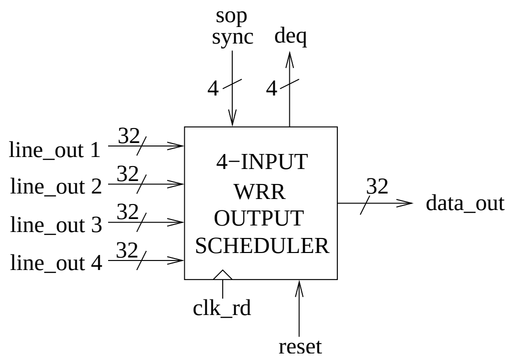

Figure A.2: 4-input WRR output scheduler block diagram.

Figure A.3 presents the WRR scheduler internal organization. Flow eligibility flags are computed via 4 packet adders: if an adder equals zero, its flow is ineligible. Packet counters are incremented by the synchronized `sop` signal that the column crosspoint that enqueues the packet sends to the output scheduler. This eligibility flag fires the FSM that controlls packet dequeues and WRR operation.

The most important operation of the specific scheduler is the computation of the minimum NST, among 4 values. This can be accomplished in two ways: (a) by using 3 comparators, organized in a tree fashion, along with three 32-bit multiplexors (top left of Figure A.3); (b) by using 6 comparators, which compare all 4 NST values with each other (center of Figure A.3). Although the first method needs only half the comparators compared to solution (b), it also requires 3 2-input, 32-bit multiplexors, which are considered expensive in this context; hence, we eventually used method (b). By comparing all NST values (from now on called `NST_1` to `NST_4`) with each other, 6 "less than" signals are generated: `NST_4` less than `NST_3`, `NST_4` less than `NST_2`, `NST_4` less than `NST_1`, `NST_3` less than `NST_2`, `NST_3` less than `NST_1` and `NST_2` less than `NST_1`. These signals, along with the 4 eligibility flags, are fed into a combinational logic 3-level circuit, which chooses the next eligible, minimum NST value. For example, the logic used for `NST_4` is as follows:

```
if (NST_4 eligible)
   if (NST_3 eligible AND NST_4 < NST_3) OR NST_3 ineligible, OR
   if (NST_2 eligible AND NST_4 < NST_2) OR NST_2 ineligible, OR
   if (NST_1 eligible AND NST_4 < NST_1) OR NST_1 ineligible, OR
   then min_NST = NST_4
...
```

where `min_NST` is the minimum Next Service Time among the 4 flows. Three similar `if` clauses are used for the rest of the flows. The `flow_sel` result of the combinational logic curcuit is used to select (a) the `line_out` bus that will be sent to the scheduler output, (b) the SI value that will be used for multiplication with the packet's size, (c) the NST register that will be updated with the *new* NST value.

Four 16-bit registers are used to store the SIs; these values are multiplied by packet size, in order to compute the NST of the flow currently being dequeued. The multiplier has a two clock cycle latency and its exit is the $16x16 = 32-bit$ value that must be added to the previous NST of the specific flow. After the NST is updated, all ineligible NSTs have to be brought to the current NST, too (Weighted Fair Queueing policy).

Dequeueing stops either when the packet is fully dequeued (`eop_deq`) and there is no other eligible flow, or when there still exist eligible flows, thus next-flow minimum NST computation has to start before the currently dequeueing finishes (`pre_schedule`).

Last of all, it should be noted that the 32-bit adder, the 16-bit multiplier and the 32-bit comparators all have a 2 clock cycle latency, thus they all need 40 nsec to produce a result. This constraint was imposed by FPGA limitations.

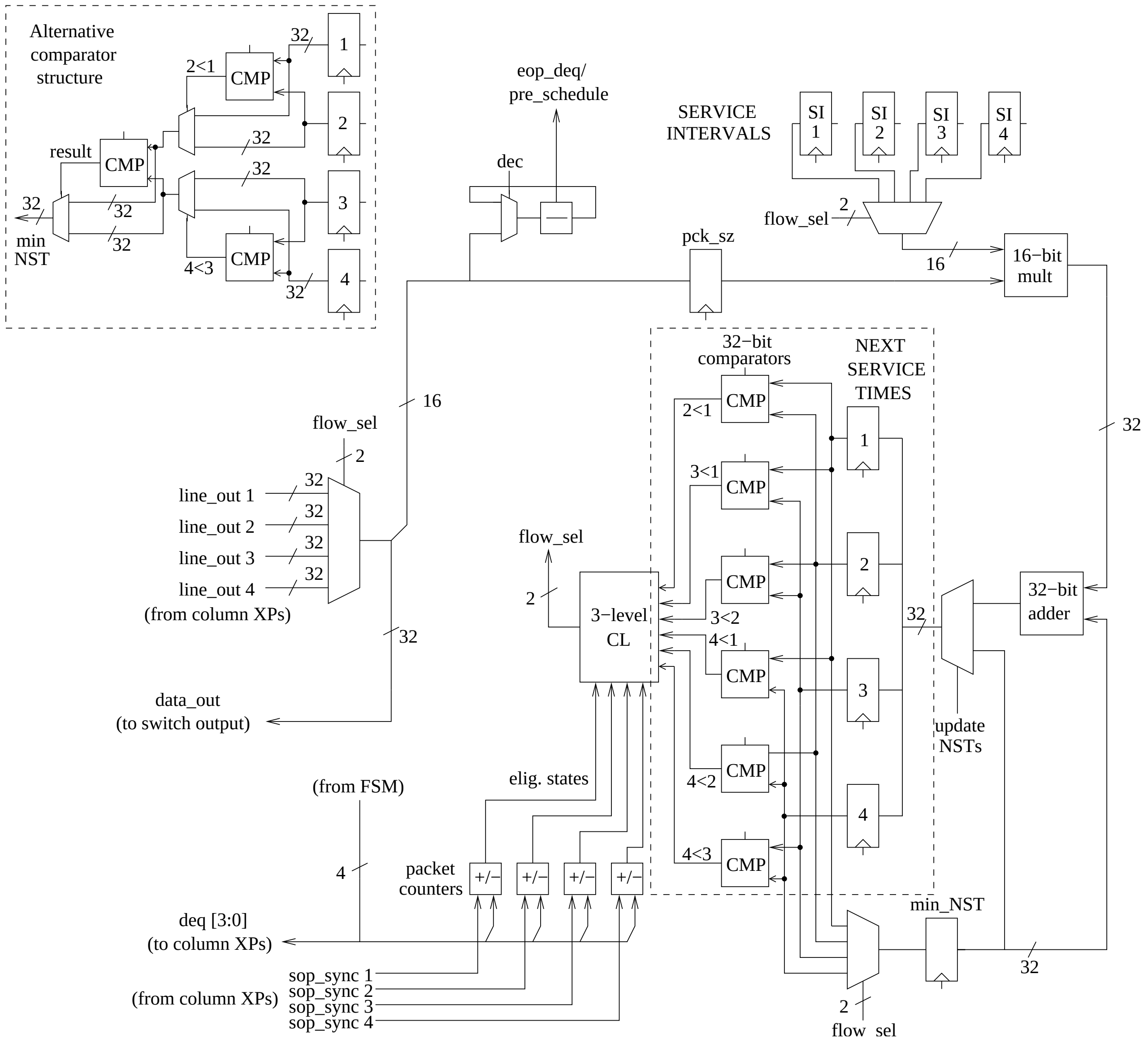

Figure A.3: Weighted Round Robin (WRR) output scheduler internal organization. 6 comparators are used in order to compare all 4 Next Service Times (NST) with each other. This comparison stage can be otherwise implemented by using 3 comparators, in a tree order, and 2 32-bit multiplexors. This organization is shown on the top left of the figure.

A.4 WRR Output Scheduler Implementation

We intended to download the WRR output scheduler (along with the rest of the switch) in an Altera EP20K1500E FPGA (1, 500, 000 ASIC-equivalent gates, 51, 840 Logic Elements, 442, 368 RAM bits), located on the Altera SOPC Development Board [26]. Target operating frequency was 50 MHz; this would result in a 4×4 crossbar switch with link rates of 1.5 Gbit/sec.

This attempt was later abandonded, since it was decided to move on the design & development of the 32×32 crossbar switch, which was thoroughly presented in the first chapters of this thesis. Nevertheless, we measured output scheduler area and speed, in the case of an ASIC construction. Thus, we followed the chip design flow presented in the previous chapters to design an single output scheduler chip [4]. Scheduler clock achieved was 100 MHz [5] , while chip area was 360×360 μm. The chip, along with the scheduler's major part physical placement, can be seen in Figure A.4.

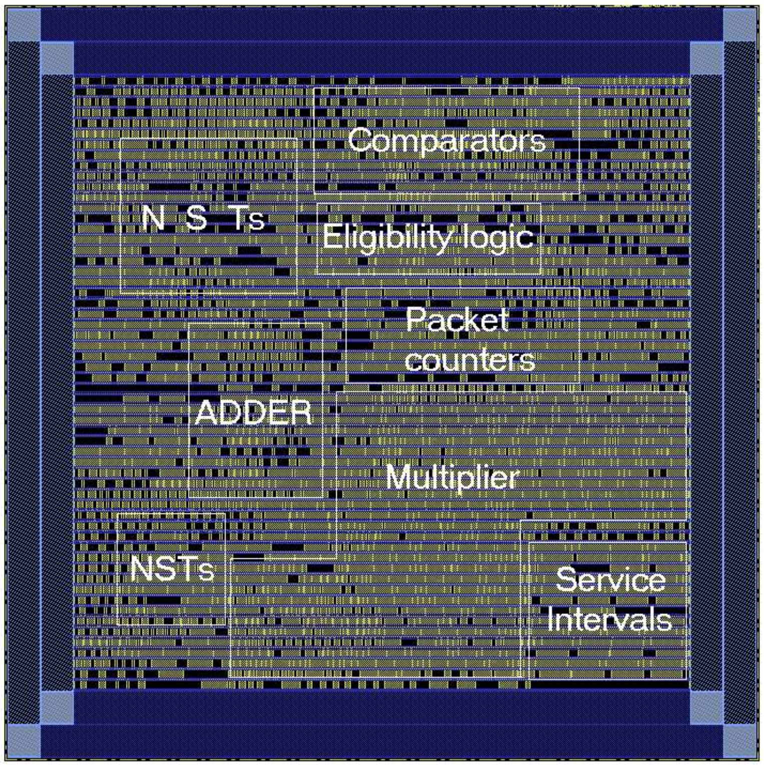

Figure A.4: WRR output scheduler chip layout: total chip area is 360×360 μm (0.13 mm^2).

[4]Multiplier, adder and comparator latency contraints were insignificant in the case of the ASIC implementation, since almost double operating frequencies could be accomplished.

[5]This could result in a 4×4 3.2 Gbit/sec link speed crossbar switch.

Appendix B

Power Estimation Tutorial

Circuit power estimation can be carried out in all "detail" levels: system, algorithm, RTL, Gate Level and post-P&R levels are possible candidates. Although the largest power savings (nearly 75%) can be achieved at the System Level, power can be saved during the rest design stages as well [22]. On the other hand, power estimation becomes more exact as the design proceeds to its latest stages (see Figure B.1). In this chapter we will analyze

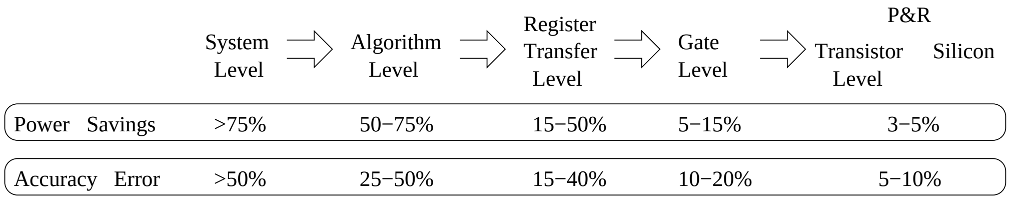

Figure B.1: Power savings vs. power estimation per design level: in the early design stages (system/algorithm/RTL) power reduction decisions are easily made and power saving decisions can be significant. On the other hand, later design stages (Gate Level/P&R) provide better power estimation results.

in detail the power estimation procedure followed in the two final stages of the design process, namely Gate Level and P&R.

B.1 Gate Level Power Estimation

In order to perform Gate Level power estimation, we must first decide on the accuracy of the desired estimation. If we want to analyze, not only cell power dissipation, but interconnect consumption as well, we must synthesize the circuit and write the interconnect and cell delay file (`.sdf`); this information is read with the command `$sdf_annotate`. The Gate Level netlist (.v) is also needed. Notice that, if we aim at estimating the consumption of a specific design component, we can use the RTL description for the rest of the design, and the Gate Level netlist for the specific component. This accounts for the

`.sdf` case as well. Last of all, the library forward annotation SAIF file must be included, too (its format is `.saif`); this file is written from the Synopsys environment with the command

```
lib2saif -output XX.saif
```

were `XX.saif` is the library output annotation file. Toggle region must be also specified; in this way, the designer explicitely defines the component under power estimation. Switching activity is captured with the command `$toggle_start` and this process is finished with the command `$toggle_stop`; switching activity report is written with the command `$toggle_report`.

The files mentioned above must be read and analyzed by an HDL compiler and simulator that supports switching activity computation, such as `verilog_toggle`. These files must be included in the design's testbench; for example, in order to simulate a crosspoint column module called `xp_col`, we need the files `xp_col.sdf`, `tech_library.saif`, as well as the component's Gate Level netlist `xp_col_mapped.v`. The `.sdf` and `.saif` files are read wherever, in the testbench, the designer wants to compute the circuit's switching activity, with the following commands (this process is called *SAIF annotation*):

```
$sdf_annotate("xp_col.sdf", bufXbar32x32_tb.bx.xp_col_inst_32, , "sdf_annot.log");
$read_lib_saif("xp_col.saif");
$set_toggle_region(bufXbar32x32_tb.bx.xp_col_inst_32);
$toggle_start;
...
... ( t e s t b e n c h )
...
$toggle_stop;
$toggle_report("xp_col.rpt.saif", 1e-12, "bufXbar32x32_tb.bx.xp_col_inst_32");
$finish;
```

This procedure outputs the *SAIF back-annotation* file, which is then imported to the Synopsys Design Compiler (Power Compiler). The design must be annotated with the information written in the SAIF back-annotation file before and after each compile of the design; for example, in the case of the crosspoint column presented above, the Design Compiler script will look like the following:

```
...
read_saif -input xp_col.rpt.saif -instance bufXbar32x32_tb/bx/xp_col_inst_32
compile -incremental
change_names -rules verilog -hierarchy
read_saif -input xp_col.rpt.saif -instance bufXbar32x32_tb/bx/xp_col_inst_32
report_power
...
```

Note the `change_names` command; it is invoked in order to prevent Design Compiler from changing the design naming rules, from the "traditional"

Verilog format to Synopsys internal naming conventions. For example, Synopsys internal naming conventions might change a bus name, from `XX[]` to `XX_1`, `XX_2` etc. Failing to do so has the result that Synopsys will not recognise those nets' toggling behavior [1]. Incremental compile ensures that Design Compiler will only make changes that will not affect the previously achieved contraints. The process described above is presened in Figure B.2.

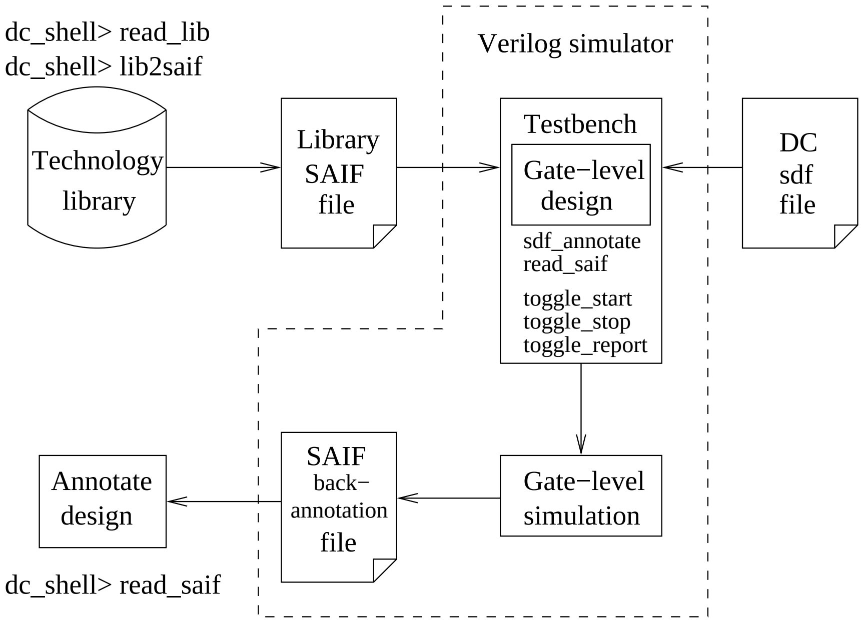

Figure B.2: Detailed Gate-Level power estimation flow.

B.2 Post-P&R Level Power Estimation

In order to perform post-P&R power estimation, we need the post-P&R netlist of the module under estimation, as well as the net interconnect and cell delay file (`.sdf`) from the same phase; the library forward annotation file (`.saif`) remains the same. The procedure followed is exactly the same as the one followed in the Gate Level power estimation.

[1]Depending on the Synopsys environment setup, name changes may not be needed.

Appendix C

Cadence Encounter Tutorial and Scripts

As mentioned in chapter 4, P&R was carried out in the Cadence Encounter environment. The tool is very memory-consuming, especially in the case of large designs [1]. In the following sections, we provide a quick tutorial on the usage of the tool, accompanied by conclusions drawn during the P&R phase of the switch design.

C.1 Setting the Environment

Cadence Encounter is run with the command `encounter`, after an include script is executed. The tool is operated in two modes, command line and graphics. The first is used to output the tool's results after each operation; those results are also written in log files (extension `.logXX` , were X is a file serial number), whereas the commands executed are appended in command files (extension `.cmdXX`). The graphics mode is used to offer the freedom to the designer to observe the placed and routed design and physicaly alter it.

Before running the tool, it is crucial to have a concrete directory organization. This is due to the large number of different types of files needed to import and use during the P&R phase. These directories are:

- `/lef`, with the layout header and technology files. Layout files contain the real physical layout of all standard cells of the technology library.
- `/lib`, with the technology library file. This directory contains the electrical characteristics (capacitance, timing) of all the standard cells.
- `/tlf`, with the timing library files of black boxes. When black boxes, like memories, are used in the design, it is usual to describe their timing characteristics by using such `tlf` files; `lib` files can be used instead.

[1]For example, in order to fully P&R in tight space constraints a 32-crosspoint switch column, along with its output scheduler, time consumed was almost 8 hours.

- `/verilog`, with the verilog descriptions. This directory contains: (a) the interfaces of all standard cells and/or black boxes; (b) the HDL Gate-Level netlist of the design, which is written by the synthesis tool. This HDL file must be *uniquified* by running the Encounter command `uniquifyNetlist`. Unification simply changes multi-instantiated module names into discreet ones.

- `/save`, where Encounter will save the design after each phase of its execution. A rule of thumb is: (a) to save the whole design after each phase is complete; (b) the name of the saved file and directory should be `design_name_AFTER_phase_name`, where `phase_name` is a discreet name for the phase that has just completed, like `TRIAL_ROUTE`.

The last two other files needed are related to the circuit clock(s): (a) the `.sdc` file contains the clock net names and their frequencies; (b) the `.clockspec` file refers to the first and also informs the tool about the electrical characteristics of the clock net and of the buffers that the tool may use when creating the clock tree. The `.clockspec` file is the only file imported at a later stage. All other files must be imported during the initial stage of the P&R process (Design Import phase).

C.2 Placement & Routing Stages

In this section we will present the exact flow followed during the P&R process and also provide the reader with useful advices. In Figure C.1 we can see a chip core, along with the most often used terms in this short tutorial.

C.2.1 Design Import

As mentioned earlier, in this stage all files needed by the tool are loaded into memory. The tool loads the standard cell library and the design in memory, as well as performs initial checks on the design. Obvious mistakes that have not been corrected during the synthesis stage are reported during this phase.

C.2.2 Floorplaning & Pin Assignment

During this phase the design is floorplaned; this means specifying the exact size of the chip core and the width of the power ring, putting certain black boxes into certain locations etc. Initially, the designer has to decide upon the size of the core, its utilization and the power ring width. The core dimensions can be either explicitly stated, that is to give the exact dimensions of the rectangle, or given by a row utilization number. In the latter case, the core remains square and the utilization percentage defines its exact size. Row utilization is entered as a percentage and determines the amount of unused row space that will be put between adjacent rows. This space will be later

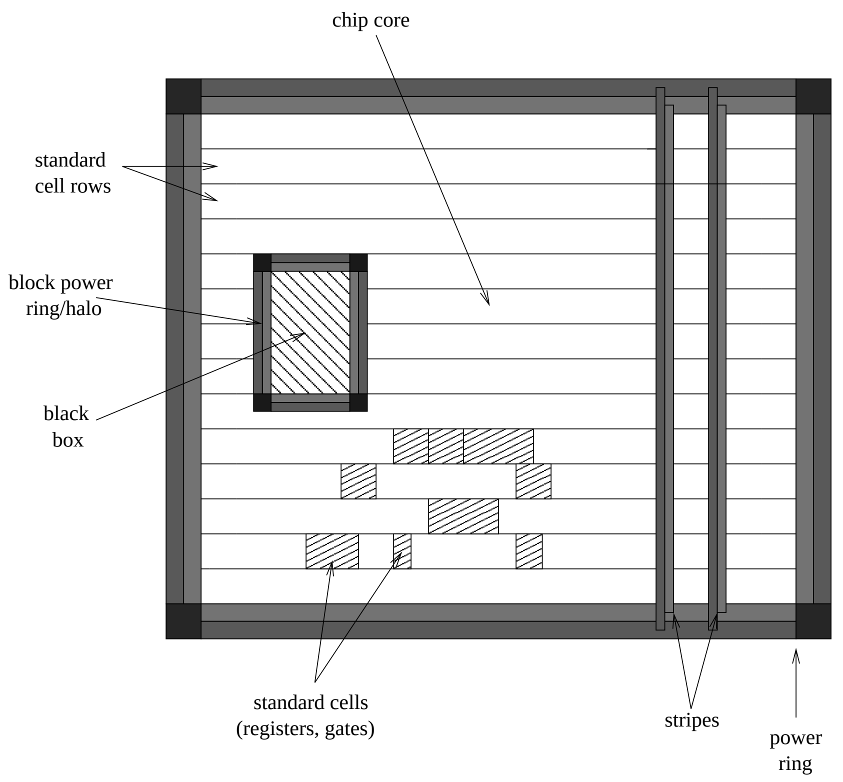

Figure C.1: Chip Overview: Chip core, rows, power ring, power stripes, black boxes and standard cells are visible; routing is omitted.

used for wiring. Should the designer decide to use the second approach in order to define the core size, care should be taken in case the design has a lot of “wiring”, as is the case with switches. In that case, a row utilization of 50% seems enough; in other cases, usual utilization figures are in the area of 70%.

Floorplaning also involves setting the width of the power ring. The power ring is made of two parallel metals, one of which carries the power, while the other acts as the ground. The power rings should be wide enough to be able to feed the whole circuit with power. In cases of large chips, power/ground stripes should also be included. These traverse the chip in horizontal or vertical parallel lines at certain distances from each other.

When black boxes are included, care must be taken. The designer usually wishes to place them at certain positions inside the core, so the tool provides this freedom. Those black boxes have to be connected to the chip’s power/ground lines and must be surrounded by a very narrow power ring themselves. This is because the power lines which will be at the top and bottom of each cell row need to be terminated and the black boxes would obscur them from reaching the other end of the core. Last of all, black boxes must be surrounded by a “block halo”, on top of their power rings, which will prevent standard cells from being placed too close to them.

Pin assignment should also be carried out in this phase. This is accomplished through an Encounter embedded application, called "Pin Editor". Although the Graphical User Interface (GUI) of the application is staightforward, care should be taken in the assignment: before it starts, "group bus" should be enabled, in order to make the assignment easier. Also, it was noted that, although we chose to place some pins on the north, or east, or west sides of the chip, the application put them all on the south side. This bug is overcome by re-placing the wrongfully placed pins. Note that the "preplace pin" opion must be always selected, as in the opposite case the pins are not fixed and it was noticed that sometimes the tool moved the pins to other sides of the chip during the later stages. After pin assignment is complete, it must be saved with the command

```
saveIoFile XX.io
```

where `XX.io` is the pin assignment file and then immediately loaded with the command

```
loadIoFile XX.io
```

in order to prevent the tool from "forgetting" the assignment. After the pin assignment is saved, it can be loaded from the beginning of the design process, along with the other design files ("setting the environment phase"); as a result, concurrent P&R attempts will do not require new pin assignments.

C.2.3 Medium Effort Placement

After the design is flooplaned, an initial placement of cells and/or black boxes has to be performed. This task is carried out automatically by Encounter and provides the designer with the initial results of the P&R process, which must always be taken into account. The tool reports initial net length, which includes the length of all wires connecting the placed standard cells and tries to improve this result, by running the medium effort placement algorithm for about 15 times. If the design fits into the area provided by the floorplaning phase, medium effort placement is successful; otherwise the tool will inform the designer about the violations. Another form of violation is the "black box" one, and is created when a black box is wrongfully placed by the designer outside the chip borders, or when the block halo does not exist and standard cells are located too close to the black box. Medium effort floorplaning is run with the Encounter shell command

```
amoebaPlace
```

while placement can be checked with the command

```
checkplace
```

A possible medium effort report might look like the following text:

```
<CMD> amoebaPlace
Extracting standard cell pins and blockage ......
Pin and blockage extraction finished
Extracting macro/IO cell pins and blockage ......
Pin and blockage extraction finished
*** Starting "Amoeba(TM) placement v0.254.2.15 (mem=244.1M)" ...
Options: gp=mq-medium dp=medium
Options: congestDrivenI75 congestDrivenQ congestDrivenR1
*info: there are 2 cells with cell padding / block halo
#std cell=47808 #block=2 (0 floating + 2 preplaced) #ioInst=0 #net=52033 #term=156067 #term/net=3.00, #fixedIo=2624, #floatIo=0, #fixedPin=2368,
 #floatPin=1
Total std cell len = 237.4733 (mm), area = 1.4628 (mm^2)
*info: identify 1 tech site  => 1 tech site pattern
*info: estimated cell pwr/gnd rail width = 0.770 um
*info: average module density = 0.024
*info: AmoebaPlace has switched to wireLength driven mode for this low density design.
*info: density for module 'crs_pes_col_inst' = 0.024
       = stdcell_area 359808 (1462835 mu^2) / alloc_area 14715214 (59826174 mu^2).
*info: density for the rest of the design = 0.000
       = stdcell_area 0 (0 mu^2) / alloc_area 12975183 (52751904 mu^2).
*info: identify 96 spare/floating instances which are grouped into 96 clusters.
*info: fixed term percentage = 5.906%
*info: fixed term center of gravity = (9751152 6169775) in coreBox (100320 100240, 35100320 19100240)
*info: partition deviation = 0.770
 0|: Est.  net length = 1.806e+08 (7.48e+07 1.06e+08) cpu=0:02:24 mem=890.3M
 1-: Est.  net length = 1.132e+08 (7.41e+07 3.91e+07) cpu=0:00:14.9 mem=890.3M
 2|: Est.  net length = 1.029e+08 (6.39e+07 3.91e+07) cpu=0:01:15 mem=890.3M
 3-: Est.  net length = 8.440e+07 (6.12e+07 2.32e+07) cpu=0:00:33.4 mem=890.3M
 4|: Est.  net length = 7.671e+07 (5.35e+07 2.32e+07) cpu=0:01:17 mem=890.3M
 5-: Est.  net length = 7.283e+07 (5.33e+07 1.95e+07) cpu=0:00:33.1 mem=890.3M
 6|: Est.  net length = 7.052e+07 (5.10e+07 1.95e+07) cpu=0:01:03 mem=890.3M
 7-: Est.  net length = 7.104e+07 (5.06e+07 2.04e+07) cpu=0:01:01 mem=890.3M
 8|: Est.  net length = 6.956e+07 (4.92e+07 2.04e+07) cpu=0:01:01 mem=890.3M
 9-: Est.  net length = 6.720e+07 (4.76e+07 1.96e+07) cpu=0:00:54.4 mem=890.3M
10|: Est.  net length = 6.656e+07 (4.70e+07 1.96e+07) cpu=0:00:36.4 mem=890.3M
11-: Est.  net length = 6.512e+07 (4.53e+07 1.98e+07) cpu=0:01:43 mem=906.6M
12|: Est.  net length = 6.515e+07 (4.49e+07 2.02e+07) cpu=0:03:11 mem=916.1M
13-: Est.  net length = 6.537e+07 (4.51e+07 2.02e+07) cpu=0:03:41 mem=931.2M
14|: Est.  net length = 6.555e+07 (4.50e+07 2.05e+07) cpu=0:03:06 mem=952.0M
..+: Est.  net length = 6.761e+07 (4.76e+07 2.00e+07) cpu=0:05:01 mem=1054.9M
*** cost = 6.761e+07 (4.76e+07 2.00e+07) (cpu for global=0:27:35) ***
*info: there are 2 cells with cell padding / block halo
Options: gp=mq-medium wireLengthDriven dp=medium
Starting fine tune place ...

  CPU Time for Phases I and II = 0:00:21.8, Real Time = 0:00:22.0
Statistics of distance of Instance movement in detailed placement
  maximum (X+Y) =       137.50 um
  inst (crs_pes_col_inst/crs_32x32_RR_4/RR_pe_cr/PEnfAll/U52) with max move: (35087.6, 18075.1) -> (34956.2, 18069)
  mean    (X+Y) =        12.23 um
*** cpu=0:00:34.6   mem=1054.9M ***
Total net length = 5.992e+07 (4.238e+07 1.754e+07) (ext = 3.254e+07)
*** End placement (cpu=0:37:49, real=0:38:01, mem=1054.9M) ***
<CMD> checkPlace
*info: there are 2 cells with cell padding / block halo
Options: gp=mq-medium dp=medium
Check Place(new) starts ...
** Info: Total stdCell area is 5.535816e+08.
** Info: Total core area is 6.650000e+08.
** Info: Density is 8.324535e-01.
** Info: Placed = 47810
** Info: Unplaced = 0
```

We can see that the placement algorithm tries to fit all standard cells and/or black boxes into the specified core area, while keeping net length to a minimum, in order to minimize possible timing violations in later stages.

C.2.4 Trial Routing

Trial routing is carried out next. During trial routing, Encounter routes the nets of the design and tries to minimize their length, or minimize the chip via usage (vias add a lot to signal delays). From the trial route report, we can see whether the the design is routable or not. If the nets are routed too close to each other, crosstalk interference is introduced, and the tool reports this violation in a “congestion distribution” table (the “-” lines refer to violated routing). Too many and too strong violations of that kind make the design unroutable and in that case the designer has to go back to the flooplaning phase and change the area of the core, or the positions of the hand-placed black boxes. Small violations (usually up to "-4"), on the other hand, might be met during the optimization phases of the P&R process, so they should not be taken into account. Trial route is executed by the shell command

```
 trialRoute -maxRouteLayer X
```

where `maxRouteLayer` is the maximum routing layer we wish to used by the tool [2]. A possible output of the trial route phase follows.

```
<CMD> trialRoute -maxRouteLayer 6
*** Starting xroute (mem=312.5M) ***

options: glbDetour reduceNode obstruct2 obstruct4 spreadOut multiOuterRow moveTermZ fixAirConnect spacingTable pinAndObsSpacing multiVoidRow wid
eBlockageSpacing MSLayer
Number of redundant fterm=0
routingBox: (0 0) (37000560 22000160)
coreBox:    (1000560 1000160) (36000360 20995520)
nrIoRowLo/Hi = 173/174 nrIoColLo/Hi = 123/123
Mem for trkArr = 446.2M, nrETrk used = 0.
Init all gpins ...
Number of multi-gpin terms=0, multi-gpins=0, moved blk term=1086/1086
Number of initial reassigned term = 0

Phase 1a route (0:00:14.8 1654.4M):
Est net length = 5.795e+07um = 3.872e+07H + 1.923e+07V
Suboptimal net = 2 out of 4738 (0.0% nets) (0.0% len over box)
Usage: (4.6%H 2.1%V) = (3.872e+07um 1.925e+07um) = (11531036 3141827)
Obstruct: 54606508 = 27303254 (70.0%H) + 27303254 (70.0%V)
OvInObst: 2228162 = 2220362/27303254 (8.13% H) + 7800/27303254 (0.03% V)
Overflow: 790340 = 635415 (5.44% H) + 154925 (1.33% V)
Number obstruct path=480 reroute=0
...
...
...
Phase 1f route (0:01:04 1730.4M):
Usage: (4.6%H 2.4%V) = (3.879e+07um 2.193e+07um) = (11547280 3576809)
OvInObst: 0 = 0/27303254 (0.00% H) + 0/27303254 (0.00% V)
Overflow: 1101 = 856 (0.01% H) + 245 (0.00% V)

Congestion distribution:

Remain  cntH            cntV
--------------------------------------
 -4:    1        0.00%  0        0.00%
 -3:    31       0.00%  0        0.00%
 -2:    142      0.00%  1        0.00%
 -1:    599      0.01%  244      0.00%
--------------------------------------
  0:    172990   1.48%  61079    0.52%
  1:    150670   1.29%  86077    0.74%
  2:    44762    0.38%  32148    0.28%
  3:    21519    0.18%  19627    0.17%
  4:    22164    0.19%  13860    0.12%
  5:    16793    0.14%  11306    0.10%
  6:    19731    0.17%  8547     0.07%
  7:    20861    0.18%  10747    0.09%
  8:    18863    0.16%  7694     0.07%
  9:    17268    0.15%  30224    0.26%
 10:    16931    0.14%  346064   2.96%
 11:    17222    0.15%  277659   2.38%
 12:    15600    0.13%  5036916 43.09%
 13:    14148    0.12%  4530174 38.76%
 14:    14351    0.12%  27628    0.24%
 15:    19822    0.17%  323453   2.77%
 16:    16375    0.14%  1841     0.02%
 17:    18400    0.16%  4238     0.04%
 18:    23292    0.20%  426549   3.65%
 19:    29309    0.25%  396033   3.39%
 20:    10996138        94.08%  35873    0.31%


4000/4739 (84.41%) net (0:01:01 est=0:00:11.3 1961.4M)
Nr short=228 83%, med=13 5% medR=34 long=0, huge=0
Phase 1l route (0:03:05 1961.4M):
```

[2]Sometimes it is useful to route a design up to a lower routing layer, eg. up to metal 5, so as to allow long wires of the upper hierarchy level to be routed over it.

```
Number of deextended terms = 1086
Prep phase2 (0:00:00.0 1961.4M):
...
...
...
*** Completed Phase 1 route (0:35:35 1961.4M) ***

Phase 2a route (0:01:49 1961.4M)
Phase 2b route (0:01:45 1961.4M)

Total length: 6.070e+07um, number of vias: 23989
M1(H) length: 7.300e+00um, number of vias: 7
M2(V) length: 6.764e+06um, number of vias: 2925
M3(H) length: 1.833e+07um, number of vias: 5829
M4(V) length: 1.099e+07um, number of vias: 9178
M5(H) length: 2.046e+07um, number of vias: 6050
M6(V) length: 4.159e+06um
*** Completed Phase 2 route (0:03:34 1961.4M) ***

*** xroute (cpu=0:39:09 mem=1961.4M) ***
<CMD> saveDesign /proj/carv/users/simos/unix/MSc/bufXbar32x32/SOC_ENCOUNTER/bx_top/save/bx_top_AFTER_TRIAL_ROUTE.enc
*** Completed saveRoute (cpu=0:00:07.6 real=0:00:09.0 mem=1961.4M) ***
```

We can see the congestion distribution table, with minor horizontal and vertical violations; experience showed that such small violations can be minimized in later phases. Trial routing involves two phases, and each phase consists of many sub-phases; each phase tries to minimize congestion violations, obstructions and overflows. Especially phase 1.f shows the biggest changes, compared to the improvements achieved between other concurrent phases. If the design still includes large distribution violations after that phase, it is certainly unroutable [3] .

C.2.5 Clock Tree Synthesis

The clock nets are by far the most important ones in a synchronous design; hence, they are routed separately in the clock tree synthesis phase. Of course, clock nets must be buffered inside the chip as evenly as possible, in order to avoid clock skew (uncertainty - see Figure C.2).

The `.clockspec` file is included during this phase, so the tool knows which types of buffers to use when producing the clock tree. Obviously, space must be left inside the core, in order to place those buffers, so floorplaning should not "stress" the core area. The tool produces full reports of the clock tree depth and of the final clock skew. Clock tree synthesis is performed by the following commands: first we have to specify the clock specification file,

```
specifyClockTree -clkFile XX.clockspec
```

Next we create the clock save directory, with the command

```
createSaveDir dir_name
```

Clock checks are then performed by running

```
checkUnique
```

[3]Note that routing violations are visible in the Encounter graphics mode, but the designer must have selected that option.

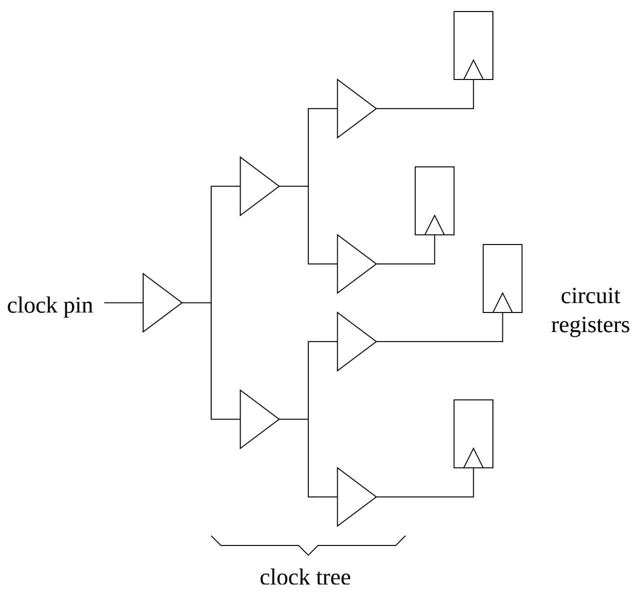

Figure C.2: Clock Tree Example

Last of all, the actual clock synthesis procedure is executed, with the command

```
ckSynthesis -rguide dir_name/XX.guide -report dir_name/XX.ctsrpt
```

where `XX.guide` and `XX.ctsrpt` are internal Encounter files [4]

A sample output of the clock synthesis, though inevitably "cut" due to space limitations, can be seen in the following text.

```
<CMD> specifyClockTree -clkfile xp_col_rd.clockspec
specifyClockTree Option :  -clkfile xp_col_rd.clockspec
RouteType               : FE_CTS_DEFAULT
PreferredExtraSpace     : 1
Shield                  : NONE
PreferLayer             : M2 M3
Avg. Cap                : 0.157351 (ff/um) [0.000157351]
Avg. Res                : 0.221429(ohm/um) [0.000221429]
M1 w=0.24(um) s=0.24(um) p=0.56(um) es=0.88(um) cap=0.146(ff/um) res=0.321(ohm/um)
M2 w=0.28(um) s=0.28(um) p=0.66(um) es=1.04(um) cap=0.153(ff/um) res=0.221(ohm/um)
M3 w=0.28(um) s=0.28(um) p=0.56(um) es=0.84(um) cap=0.162(ff/um) res=0.221(ohm/um)
M4 w=0.28(um) s=0.28(um) p=0.66(um) es=1.04(um) cap=0.153(ff/um) res=0.221(ohm/um)
M5 w=0.28(um) s=0.28(um) p=0.56(um) es=0.84(um) cap=0.162(ff/um) res=0.221(ohm/um)
M6 w=0.44(um) s=0.44(um) p=1.32(um) es=2.2(um) cap=0.129(ff/um) res=0.0932(ohm/um)
...
...
...
********** Clock clk_wr Pre-Route Timing Analysis **********
Nr. of Subtrees                : 1
Nr. of Sinks                   : 2
Nr. of Buffer                  : 44
Nr. of Level (including gates) : 39
Max trig. edge delay at sink(R): xp_hier_col_2_inst/clk_wr 4992.4(ps)
Min trig. edge delay at sink(R): xp_hier_col_1_inst/clk_wr 4989.4(ps)

                                 (Actual)               (Required)
Rise Phase Delay               : 4989.4~4992.4(ps)      0~5000(ps)
Fall Phase Delay               : 5600.1~5679.3(ps)      0~5000(ps)
Trig. Edge Skew                : 3(ps)                  500(ps)
Rise Skew                      : 3(ps)
Fall Skew                      : 79.2(ps)
Max. Rise Buffer Tran.         : 482.6(ps)              500(ps)
```

[4]Due to the large number of commands that must be executed, clock tree synthesis is more prefferably run from the Enounter graphics mode.

```
Max. Fall Buffer Tran.         : 471.7(ps)             500(ps)
Max. Rise Sink Tran.           : 900.2(ps)             500(ps)
Max. Fall Sink Tran.           : 628.5(ps)             500(ps)


Generating Clock Analysis Report bx_top_cts/bx_top_cts.ctsrpt ....
Generating Clock Routing Guide bx_top_cts/bx_top_cts.guide ....
Clock Analysis (CPU Time 0:00:00.2)


<CMD> saveClockNets -output bx_top_cts/bx_top_cts.ctsntf
<CMD> saveNetlist bx_top_cts/bx_top_cts.v
Writing Netlist "bx_top_cts/bx_top_cts.v" ...
<CMD> savePlace bx_top_cts/bx_top_cts.place
<CMD> saveDesign /proj/carv/users/simos/unix/MSc/bufXbar32x32/SOC_ENCOUNTER/bx_top/save/bx_top_AFTER_CLK_SYN.enc
```

C.2.6 Timing Driven Placement

After the clock tree is placed and routed, the placement stage is re-run, but in a *high-effort, timing driven* mode. This allows the tool to possibly move some standard cells to new positions, in order to achieve timing goals. An example output of this process is shown below. We can see that, in this case, the tool has actually moved some cells.

```
<CMD> amoebaPlace
*** Starting "Amoeba(TM) placement v0.254.2.15 (mem=241.3M)" ...
Options: gp=mq-medium dp=medium
Options: congestDrivenI75 congestDrivenQ congestDrivenR1
...
...
...
Starting fine tune place ...

  CPU Time for Phases I and II = 0:00:22.6, Real Time = 0:00:23.0
Statistics of distance of Instance movement in detailed placement
  maximum (X+Y) =       162.36 um
  inst (crs_pes_col_inst/crs_32x32_RR_24/cr_choose_reg_1) with max move: (761.64, 17791.8) -> (617.76, 17773.3)
  mean    (X+Y) =        18.84 um
*** cpu=0:00:35.9   mem=1052.0M ***
Total net length = 4.134e+07 (3.025e+07 1.109e+07) (ext = 1.223e+07)
*** End placement (cpu=0:38:31, real=0:39:33, mem=1052.0M) ***
...
```

C.2.7 Timing Driven Final and Global Route

This is the last main phase of the P&R process. In this phase, the tool routes once more the design and tries to use the minimum net length and number of vias, choose the appropriate metal layer for each net, and possibly add buffers or resize cells, in order to achieve unmet timing constraints. [5] The designer can choose which final router the tool is going to use: WRoute or NanoRoute. The latter is a new router and is actually suggested by Cadence, as it is considered to be faster and better, although the author has observed that WRoute has provided better results in some cases; hence, possibly both must be tried. A limited output of the NanoRoute final router, along with the commands executed, can be seen below [6]:

[5] It is reminded that the time t needed to load a wire equals $t = C\Delta V/I$, where C is the wire's capacitance, ΔV the voltage drop/increase and I the current flowing through the wire. Hence, for the same ΔV, time is reduced if (a) we allow more current to flow through the wire, as a result the wire must be wider, in order to have small resistance; (b) load capacitance is small; this can be achieved by buffering the net.

[6] Due to the large number of commands that must be executed, we suggest to run NanoRoute in the Enounter graphics mode, than in the command line interface.

```
<CMD> getNanoRouteMode -quiet routeSiEffort
<CMD> getNanoRouteMode -quiet
<CMD> setNanoRouteMode -quiet drouteFixAntenna true
<CMD> setNanoRouteMode -quiet routeInsertAntennaDiode false
<CMD> setNanoRouteMode -quiet routeAntennaCellName default
<CMD> setNanoRouteMode -quiet routeWithTimingDriven true
<CMD> setNanoRouteMode -quiet routeWithTimingOpt true
<CMD> setNanoRouteMode -quiet optimizeBi true
<CMD> setNanoRouteMode -quiet optimizeGs true
<CMD> setNanoRouteMode -quiet optimizeFixSetupTime true
<CMD> setNanoRouteMode -quiet optimizeTargetSetupSlack 0.000000
<CMD> setNanoRouteMode -quiet optimizeFixMaxCap false
<CMD> setNanoRouteMode -quiet optimizeFixHoldTime false
<CMD> setNanoRouteMode -quiet optimizeTargetHoldSlack 0.000000
<CMD> setNanoRouteMode -quiet optimizeFixMaxTran false
<CMD> setNanoRouteMode -quiet optimizeDontUseCellFile default
<CMD> setNanoRouteMode -quiet routeWithSiDriven false
<CMD> setNanoRouteMode -quiet routeSiEffort min
<CMD> setNanoRouteMode -quiet siNoiseCTotalThreshold 0.050000
<CMD> setNanoRouteMode -quiet siNoiseCouplingCapThreshold 0.005000
<CMD> setNanoRouteMode -quiet routeWithSiPostRouteFix false
<CMD> setNanoRouteMode -quiet drouteAutoStop true
<CMD> setNanoRouteMode -quiet routeSelectedNetOnly false
<CMD> setNanoRouteMode -quiet routeFixPrewire false
<CMD> setNanoRouteMode -quiet drouteStartIteration default
<CMD> setNanoRouteMode -quiet envNumberProcessor 1
<CMD> setNanoRouteMode -quiet routeTopRoutingLayer default
<CMD> setNanoRouteMode -quiet drouteEndIteration default
<CMD> deleteSwitchingWindows
<CMD> cleanupDetailRC
<CMD> globalDetailRoute

globalDetailRoute
...
...
...
```

WRoute is run with the command

```
wroute -timingDriven
```

WRoute and NanoRoute perform *Physically Knowledgable Synthesis - PKS*, that is, they both try to optimize design timing by actually changing some standard cells (cell resizing), or adding buffers to nets, while having in mind the actual physical layout of the design. Usually, after many optimization attempts, the algorithm finishes and reports the "worst" path in therms of timing. A possible report of that kind is shown below:

```
+----------------------------------------+
| Report           | report_timing       |
|------------------+---------------------|
| Options          |                     |
+------------------+---------------------+
| Date             | 20040417.162148     |
| Tool             | pks_shell           |
| Release          | v5.9-s043           |
| Version          | Apr 18 2003 19:47:51 |
+------------------+---------------------+
| Module           | crs_pes_col         |
| Timing           | LATE                |
| Slew Propagation | WORST               |
| PVT Mode         | max                 |
| Tree Type        | worst_case          |
| Process          | 1.00                |
| Voltage          | 1.98                |
| Temperature      | 0.00                |
| time unit        | 1.00 ns             |
| capacitance unit | 1.00 pF             |
| resistance unit  | 1.00 kOhm           |
+----------------------------------------+
Path 1: MET Setup Check with Pin crs_32x32_RR_32/cr_choose_reg_4/CK
Endpoint:   crs_32x32_RR_32/cr_choose_reg_4/SE (^) checked with  leading edge of 'clk_rd'
Beginpoint: crs_32x32_RR_32/cr_change_32_reg/Q (^) triggered by  leading edge of 'clk_rd'
Other End Arrival Time      9.24
- Setup                     0.11
+ Phase Shift               3.35
= Required Time            12.48
- Arrival Time             12.03
= Slack Time                0.45
     Clock Rise Edge                 0.00
     + Drive Adjustment              4.11
     = Beginpoint Arrival Time         4.11
     +------------------------------------------------------------------------------------------------------------+
     |              Instance                |      Arc       |           Cell            | Delay | Arrival | Required |
     |                                      |                |                           |       |  Time   |   Time   |
     |--------------------------------------+----------------+---------------------------+-------+---------+----------|
     |                                      | clk_rd ^       |                           |       |   4.11  |    4.56  |
     | clk_rd__L1_I0                        | A ^ -> Z ^     | BUFD16                    |  2.49 |   6.60  |    7.06  |
     | clk_rd__L2_I0                        | A ^ -> Z ^     | BUFD16                    |  1.01 |   7.61  |    8.07  |
     | clk_rd__L3_I0                        | A ^ -> Z ^     | BUFD16                    |  0.15 |   7.76  |    8.21  |
     | clk_rd__L4_I0                        | A ^ -> Z ^     | BUFD16                    |  0.10 |   7.86  |    8.31  |
     | clk_rd__L5_I0                        | A ^ -> Z ^     | BUFD16                    |  0.21 |   8.07  |    8.52  |
     | clk_rd__L6_I0                        | A ^ -> Z ^     | BUFD16                    |  0.39 |   8.46  |    8.91  |
     | clk_rd__L7_I13                       | A ^ -> Z ^     | BUFD16                    |  0.63 |   9.09  |    9.54  |
     | crs_32x32_RR_32                      | clk_rd ^       | cr_sched_32x32_RR_SPC_31  |       |   9.09  |    9.54  |
     | crs_32x32_RR_32/cr_change_32_reg     | CK ^ -> Q ^    | DFEPQ2                    |  0.46 |   9.55  |   10.01  |
     | crs_32x32_RR_32/RR_pe_cr             | In[0] ^        | RR_prior_enf_cs_SPC_31    |       |   9.55  |   10.01  |
     | crs_32x32_RR_32/RR_pe_cr/U3          | A1 ^ -> Z ^    | NOR2M1D2                  |  0.16 |   9.71  |   10.16  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar     | In[0] ^        | CRS_0_SPC_31              |       |   9.71  |   10.16  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U95 | A4 ^ -> Z v    | NOR4M1D2                  |  0.04 |   9.75  |   10.20  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U90 | A2 v -> Z ^    | NOR4D4                    |  0.19 |   9.94  |   10.39  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U14 | A4 ^ -> Z v    | NAN4M2D1                  |  0.15 |  10.09  |   10.54  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U47 | A v -> Z ^     | INVD2                     |  0.10 |  10.18  |   10.64  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U83 | A2 ^ -> Z v    | NAN2D1                    |  0.07 |  10.26  |   10.71  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U81 | A3 v -> Z ^    | NAN4M2D2                  |  0.08 |  10.33  |   10.79  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U85 | A2 ^ -> Z v    | NOR2D2                    |  0.08 |  10.41  |   10.87  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U92 | A4 v -> Z ^    | NAN4M2D2                  |  0.06 |  10.47  |   10.93  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U21 | A4 ^ -> Z v    | NAN4M1D2                  |  0.08 |  10.55  |   11.01  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U87 | A1 v -> Z ^    | NOR4D1                    |  0.12 |  10.68  |   11.13  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U69 | A2 ^ -> Z ^    | AND2D2                    |  0.15 |  10.83  |   11.28  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U38 | A ^ -> Z v     | INVD1                     |  0.08 |  10.91  |   11.36  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U74 | A3 v -> Z ^    | NOR4D2                    |  0.13 |  11.04  |   11.49  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U73 | A2 ^ -> Z v    | NAN2D2                    |  0.08 |  11.12  |   11.57  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U15 | A3 v -> Z ^    | NOR3M1D2                  |  0.12 |  11.24  |   11.70  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U116| A3 ^ -> Z v    | NAN3M1D2                  |  0.10 |  11.34  |   11.79  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar/U11 | A3 v -> Z v    | OR4D1                     |  0.24 |  11.58  |   12.03  |
     | crs_32x32_RR_32/RR_pe_cr/PEnfPar     | OneDetected v  | CRS_0_SPC_31              |       |  11.58  |   12.03  |
     | crs_32x32_RR_32/RR_pe_cr/U65         | A v -> Z v     | BUFD20                    |  0.13 |  11.71  |   12.16  |
     | crs_32x32_RR_32/RR_pe_cr/U24         | SL v -> Z ^    | MUX2DL                    |  0.32 |  12.03  |   12.48  |
     | crs_32x32_RR_32/RR_pe_cr             | Out[4] ^       | RR_prior_enf_cs_SPC_31    |       |  12.03  |   12.48  |
     | crs_32x32_RR_32/cr_choose_reg_4      | SE ^           | SDFPQ1                    |  0.00 |  12.03  |   12.48  |
     +------------------------------------------------------------------------------------------------------------+

            Timing driven total CPU time = 620.930000 seconds. <PLC-530>.
    Info:   Writing "/dev/null" ... <PLC-601>.
    Info:   Written out 582720 gcells for 6 layers <PLC-601>.
    Info:   End routability analysis: cpu: 0:12:30, real: 0:12:38, peak: 387.36 megs. <PLC-601>.
...
...
...
Begin antenna checking ...
Layer          H-Length   V-Length   Down-Via  Violation (   Antenna)
--------------------------------------------------------------------
 1st routing     189230        987          0          0 (         0)
 2nd routing       7714     918427     235465          0 (         0)
 3rd routing    1906908        366     128602          0 (         0)
 4th routing        150     195163      46695          0 (         0)
 5th routing     410392         51       4363          0 (         0)
 6th routing        333     138479       1752          0 (         0)
--------------------------------------------------------------------
                2514730    1253475     416877          0 (         0)
End antenna checking: cpu: 0:00:19, real: 0:00:20, peak: 94.87 megs.
End final routing: cpu: 0:17:20, real: 0:17:29, peak: 94.87 megs.

Begin DB out ...
Writing "crs_pes_col.wdb" ...
Written out 11 layers, 6 routing layers, 0 overlap layer
Written out 510 macros, 82 used
Written out 47975 components
  47975 core components: 0 unplaced, 47975 placed, 0 fixed
Written out 1538 physical pins
  1538 physical pins: 0 unplaced, 1538 placed, 0 fixed
Written out 49129 nets, 49033 routed
Written out 2 special nets, 2 routed
Written out 263166 terminals
Written out 232684 real and virtual terminals
End DB out: cpu: 0:00:00, real: 0:00:04, peak: 94.87 megs.
...
```

C.2.8 Timing Reports and Design Optimizations

Timing violations are obviously the most important ones in the whole design process. Therefore, Encounter is able to report design timing violations, both in pre- and post-routed clock design instances. In the first case, clock is considered to be ideal. This report is generated in the very early design stages, that is right after the trial route phase. In this way, the designer can have a very early picture of the design's timing violations, and thus conclude on whether it is worth continuing the P&R process, or go back to earlier stages. The same timing report can be generated after the global and final routing phase and provides the designer with the actual timing violations that possibly exist. In order to produce a timing violation report, the design must first be RC extracted: at first, the designer has to define the RC extraction model detail, with the command

```
setExtractRCMode -detail/-default -reduce 5 -noise
```

and then RC extraction is performed with the shell command

```
extractRC -outfile XX.cap
```

where XX.cap is the capacitance information output file. A possible output of these two commands is shown below:

```
RC Extraction for instance crs_pes_col
RC Database Name : crs_pes_col.rcdb.gz
Detail Parasite RC Extraction for crs_pes_col
wire [660 220 20019570 169940] IOBox[0 0 20020560 170080]
bounrary [0 0 20020560 170080]
max_track=10
**WARN: Since cap. table file was not provided, it will be created internally with the following process info:
* Layer Id            : 1 - M1
      Thicknesss       : 0.48
      Min Width        : 0.24
      Layer Dielectric : 4.1
* Layer Id            : 2 - M2
      Thicknesss       : 0.58
      Min Width        : 0.28
      Layer Dielectric : 4.1
* Layer Id            : 3 - M3
      Thicknesss       : 0.58
      Min Width        : 0.28
      Layer Dielectric : 4.1
* Layer Id            : 4 - M4
      Thicknesss       : 0.58
      Min Width        : 0.28
      Layer Dielectric : 4.1
* Layer Id            : 5 - M5
      Thicknesss       : 0.58
      Min Width        : 0.28
      Layer Dielectric : 4.1
* Layer Id            : 6 - M6
      Thicknesss       : 0.86
      Min Width        : 0.44
      Layer Dielectric : 4.1
...
...
...
Nr. Extracted Resistors     : 258800
Nr. Extracted Ground Cap.   : 211830
Nr. Extracted Coupling Cap. : 2001384
Detail RC Extraction Completed (CPU Time= 0:01:25  MEM= 260.6M)
RC Extraction Completed (CPU Time= 0:01:25  MEM= 260.6M)
```

Last of all, the actual timing model is extracted by the command

```
genTlfModel
```

whereas, after these commands are executed, timing report can be generated with the command

```
reportViolation -outfile XX.tarpt -num 50
```

where `XX.tarpt` is the output file and `-num 50` is the maximum number of reported violations. In the case when violations still exist, the designer can run “Post-route Optimization”, with the following command

```
runPostRouteOpt -allEndpoints -dfUseOpt 0 -dfAllPt 1 -ffUseOpt 0
                -ffAllPt 1 -inAllPt 1 -outAllPt 1 -fdAllPt 1
                -macUseOpt 0 -macAllPt 1
```

It is suggested to run the last command from the graphics mode, in order to handle all possible optimization options in a more convenient way. It is also noted that Encounter includes other optimization algorithms, as well; depending on the design and its violations, the designer can try them, too. During post-route optimization, the tool finds all violating paths and tries to minimize each by cell resizing (thus area increases). By minimizing local slacks, the tool also minimizes the up-to-now worst slack of the circuit. There are times, however, when, after many attempts and running time, the tool “gives up” for specific violating paths. A sample snapshot of a post-route optimization process is shown below:

```
...
   +--------------------------------------------------------------------+
   |                              xp_col                                |
   |--------------------------------------------------------------------|
   |  Cell area  | Utilization | Worst slack | Local slack | CPU(s) Mem(M) |
   |-------------+-------------+-------------+-------------+--------------|
   |   99053.00 |      13.07% |     -2.6842 |     -0.0699 |    904    181 |
   +--------------------------------------------------------------------+
          Critical Begin Point(s): os_32x32_RR_inst/deq_mask_reg_18/Q
          <TOPT-515>.
          Critical End Point(s): xpoint_14_x/TP512X32_inst/REN {TO_MACROS}
          <TOPT-516>.
...
```

Note that if post-route optimization does not succeed in meeting all timing requirements, P&R process has failed. Also, it is interesting to note that the tool’s global and final algorithms (WRoute and NanoRoute), as well as the post-synthesis optimizer, can cope with very strict timing requirements, even in cases when the initial timing analyses report violated slacks of even 15 nsec!

Appendix D

Switch Synchronization Details

D.1 Introduction

Switches are multi-clock ASICs. This is because, if the input links are fast (i.e. more than 1 Gbit/sec), tranceiver circuits used also carry the other-end transmitter clock; this clock is recovered by the tranceiver circuits located at the switch inputs (see Figure D.1). If the links are long, then the phase

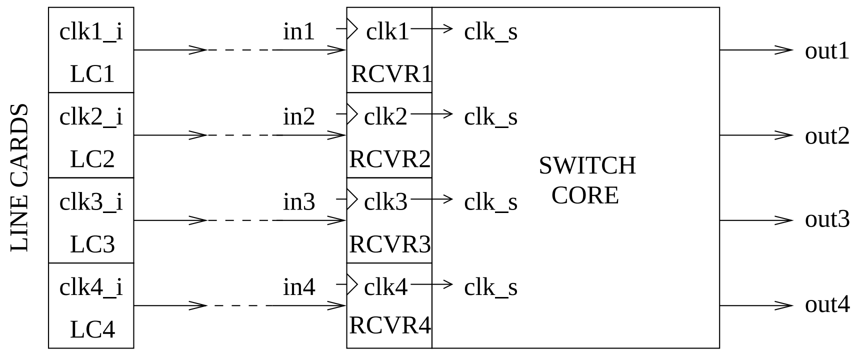

Figure D.1: The problem of communicating across clock domains: clocks clk1 to clk4 are the recovered versions of the initial line card (LC) clocks clk1_i to clk4_i. Clock recovery is carried out by the switch tranceiver circuits. These clocks differ from each other, due to phase variations, and they all have to be synchronized to the switch core's clock.

of the recovered clock, at the receiver (i.e. switch), can vary widely relative to the phase of the transmitted clock (i.e. line card), at the other end. This poses *metastability* issues.

D.2 The Metastability Problem

Figure D.2 shows a typical positive edge flip-flop timing specification. Input data must be stable setup time (t_{SU}) before the edge arrives and must remain the unchanged hold time (t_H) after it is latched. If these requirements are met, the flip-flop will output the latched value after its propagation delay (t_{PROP}). If the new value arrives inside the setup & hold time window, then

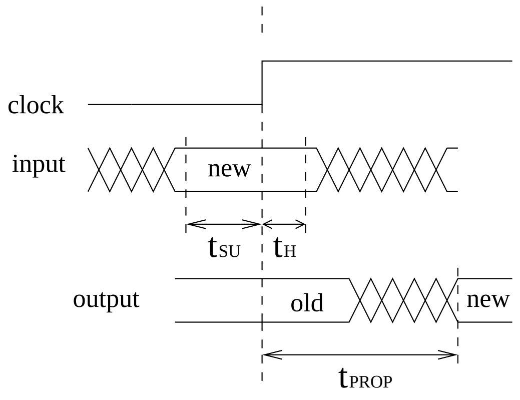

Figure D.2: Flip-flop timing specification.

conflicts occur (see Figure D.3): the new output value may be delayed if it arrives within the "long delay danger window" (W), whereas arrival within the "metastability danger window" (w) may lead to unknown output; this behavior is called metastability. Usually, the long delay window is 10 times larger than the metastability one ($W \approx 10 \times w$). Because both long delay

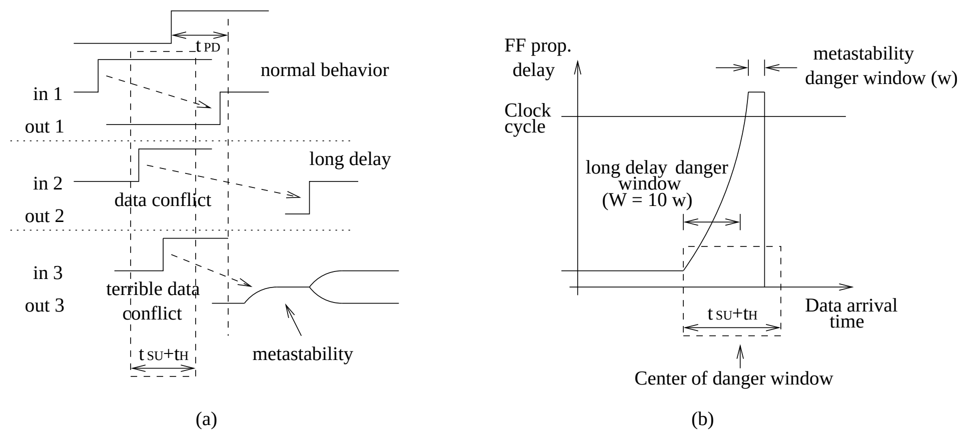

Figure D.3: Long delay and metastability due to data conflicts: (a) illustrates the results of data arriving inside the setup and hold window. (b) shows the relationship between the "long delay" and "metastability" danger windows.

and metastability are undesirable effects, the designer should be aware of the combined window; from now on, we will use W for the whole window.

The probability that the arriving data enters metastability is a function of the clock cycle T (or its inverse clock frequency F_C), the width of the danger zone, W, and the data arrival rate F_D. If data arrives during every cycle, then the probability p equals $p(\text{enter metastability}) = W/T = W \times F_C$. When data arrival rate is F_D, the rate R of entering metastability equals $R(\text{entering metastability}) = W \times F_C \times F_D$. As an example, consider the "new packet" notification pulse that traverses the two clock domains of the switch presented in this work. In a $0.18\mu m$ technology, gate delay is approximately $60psec$ and W is estimated to be approximately $90psec$; clock frequency is $300MHz$ (clock cycle $3.35nsec$). If data must be synchronized every 10 clock cycles (as is the worst case with the "new packet" signal), then the metastability rate equals $R = W \times F_C \times F_D = 800KHz$, that is, once every 1.25 nsec, which is unacceptably large.

After a metastability conflict occurs: (a) the output may send the correct value after a long delay; or (b) the output may go to 1 or 0 in random. In practice, the decision time of a synchronizer is exponentially distributed [32]. Thus, if a system is designed to wait a time, τ, before sampling the synchronizer, the probability of synchronization failure decreases exponentially as τ is increased. Given metastability at $t = 0$, the metastability probability at $t > 0$ is $e^{-t/\tau}$. A failure exists if metastability remains during the next clock cycle, too. Failure will be thus $p(\text{enter m.s.}) \times p(\text{stil m.s. after T})$, and the corresponding failure rate will equal

$$Rate(failure) = Rate(\text{enter m.s.}) \times p(\text{still m.s. after T}) = W \times F_C \times F_D \times e^{-T/\tau}$$

Mean Time Between Failure (MTBF) will then equal

$$MTBF = 1/Rate(\text{failure}) = e^{T/\tau}/W \times F_C \times F_D$$

Depending on the parameter τ, MTBF can be almost eliminated.

D.3 Synchronizer Requirements

From the above, it is evident that all recovered clocks have to be synchronized to a common clock, `clk_s`. Switch outputs, on the other hand, can be synchronized to the switch core clock `clk_s`; this reduces outgoing latency.

There is large number of possible synchronization solutions. The decision on which to use depends mostly on the latency requirements and the synchronization protocol used. Switches usually use elastic buffers at the inputs, in order to synchronize the received multi-bit information (packet data) [31]. Elastic buffers are merely 2-port SRAMs with two asynchronous ports; a read and a write pointer are used to select the read and write position in the memory array; these two pointers are synchronized to their respective clocks. Elastic buffers are large and complicated circuits, and require the use of an SRAM in every synchronization boundary. The 32×32 buffered crossbar switch would need one such elastic buffer at each input.

But since SRAMs were inevitably used at the switch crosspoints, choosing to use 2-port versions would immediately solve the *data* synchronization problem, without the need for elastic buffers. The only other information that has to be synchronized is a *control* signal: the 1-bit "new packet" notification pulse. This signal is produced at most every minimum-size packet time, 10 clock cycles. Since 1024 (32×32) such signals have to be synchronized, the synchronization circuit has to be the simplest possible. Also note that the various incoming clocks must have approximately the same frequency as the switch core's clock, as the switch incoming throughput is the same as the outgoing one, and no speedup is required, due to the adopted architecture; hence the synchronizer used must be able to efficiently deal with small frequency and phase variations, while keeping its complexity to a minimum.

D.4 1-bit Synchronizer Design

The two flip-flop synchronizer is possibly the most efficient solution in such pleisiochronous situations (see Figure D.4). Two flip-flops are used in order to

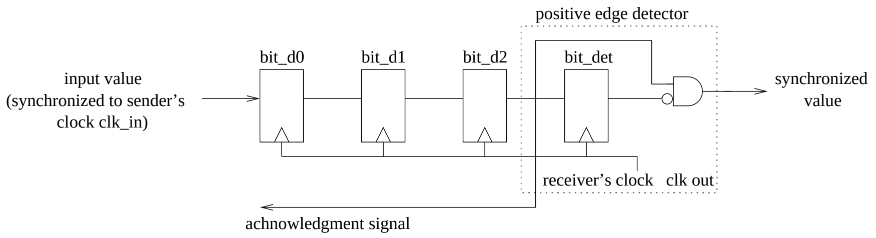

Figure D.4: The 2 flip-flop synchronizer.

allow the single-bit data a complete cycle for settling; hence, settling time is 2 clock cycles, whereas synchronization latency is 3 clock cycles. We decided to use 3 flip-flops for settling, in order to be more redundant. The synchronized signal is driven into a positive edge detector, which eventually lathes the signal and outputs a complete pulse. This protocol is called "4-phase", since a complete cycle of the incoming pulse triggers one synchronization event.

Appendix E

Power Optimization Techniques

Low power consumption has become a crucial issue in IC design. In portable devices, low power consumption has long been considered one of the main design contraints. In high-performance processors, on the other hand, power consumption was traditionally the secondary constraining factor; however, this has changed due to the concerns on the cooling and packaging cost. What is more, the increase in power consumption has resulted in higher temperature, which in turn reduces reliability [1]. Therefore, low power techniques are important for current and future VLSI design. In this section we will examine theoretically various methods which, if used correctly, can decrease a circuits's power consumption. Some of these methods are adopted by the Synopsys Power Compiler power optimization algorithms. The methods presented can also be found in [25].

E.1 RTL Power Optimization

E.1.1 Glitch Minimization

A design's structure in the RTL level can have a significant affect on inter-block glitches. This seems logical when we face a circuit's blocks as large gates that connect to each other. Glitch propagation through these RTL blocks can affect power.

Path Balancing and Depth Reduction

Glitch can be reduced if we balance signal paths, as well as by increasing circuit depth [2]. For example, in Figure E.1 we can see two implementations of a 4-number adder. If all inputs arrive at the same time, in the case of

[1]Realiability is a crucial demand in switching devices: 5 minutes/year downtime is considered to be a reasonable value.

[2]Of course, increasing the depth , increases the capacitance; on the other hand, reducing the depth imposes the use of more registers, hence power is increased. Such tradeoffs will be largely encountered in the following techniques as well.

the chain adder of Figure E.1(a), adder 2 will compute twice every clock cycle and adder 3 will compute 3 times per cycle. This results in useless transitions that consume power. On the other hand, in the case of the tree adder of Figure E.1(b), all paths are balanced, hence glitching is minimal.

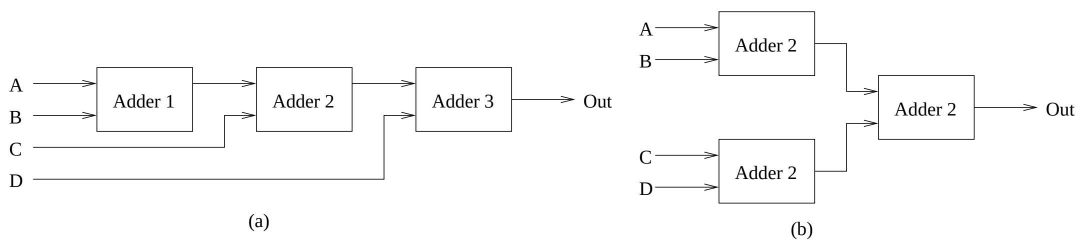

Figure E.1: Path balancing and depth reduction power optimization: case (a) is an unbalanced chain adder, while (b) is a completely balanced, almost glitch free tree adder.

E.1.2 Exploitation of Resource Sharing

Resource sharing is a tradeoff between area and performance, but it can affect power consumption as well. Usually, when we share a unit, we need multiplexors at its inputs, which results in increased power consumption. For example, we can choose to use a parallel architecture (by duplicating the resources) in case we need a high-throughput circuit, or a time-multiplexed architecture for low-throughput circuits. As an example, in Figure E.2, the parallel implementation (E.2(a)) approach consumes less power than the time-multiplexed one (E.2(b)), because the latter involves using multiplexors and due to the fact that the time-multiplexed approach destroys data correlation.

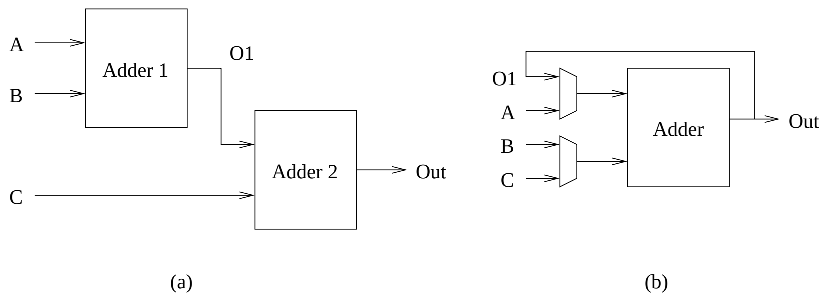

Figure E.2: Exploitation of resource sharing power optimization: case (a) is a parallel implentation, while (b) time-multiplexes input data, in order to consume less area, but is not as power-aware as approach (a).

E.1.3 Dynamic Power Management

The idea of dynamic power management is to shut down some blocks (ALUs, registers) that are not performing valuable jobs all the time. There are 4 approaches to this idea.

Pre-computation

With pre-computation we compute some of the circuit outputs one or more cycles earlier, in order to decrease the switching of the combinational circuit. This can be achieved by implementing predictor functions, which must cover as many circuit states possible. This method can achieve as much as 60% power savings, but with small area overhead in pipelined circuits, and is difficult to implement in sequential circuits. For example, the predictor functions in the case of the comparator shown in Figure E.3 are very simple: If we compare the multi-bit numbers C & D, functions g_1 and g_2 are $g_1 = C[n-1]D[n-1]'$ for $C > D$, and $g_2 = C[n-1]'D[n-1]$ for $C < D$; as a result, $g_1 + g_2 = (C[n-1]XORD[n-1])'$. For equiprobable inputs, XNOR power consumption is $p(XNOR) = 0.5$, so we achieve an average of 50% power reduction.

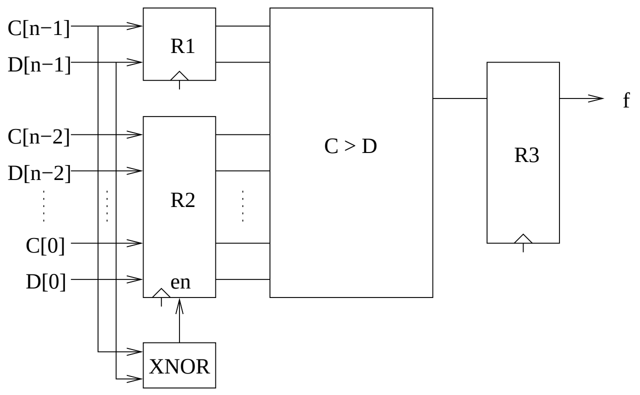

Figure E.3: Pre-computation power optimization: the result of the $C > D$ comparison is evaluated by first comparing the most significant bits of C and D and then by comparing the rest. Power consumption savings can be as much as 60%.

Gated Clocks

Clock gating can be directly implemented on our code. The designer finds the cases when a specific subcircuit must remain idle and builds an activation function. Gating is performed by AND-ing the real clock with the latched output (in order to avoid glitches) of the activation function. Such techniques can reduce the power consumption of FSMs by almost 30%. For testability purposes, we wish to have a way to bypass clock gating. It is also better to locate the clock gating latches at the high hierarchy levels. Synthesis tools, such as Synopsys, can automatically implement clock gating in circuits as long as such an option is explicitely activated in the synthesis scripts and the code contains `always` clauses of the form

```
always @(posedge clk)
if (enable)...
```

As an example, consider the FSM block diagram of Figure E.4. When activation function f is 0, the gated clock stops.

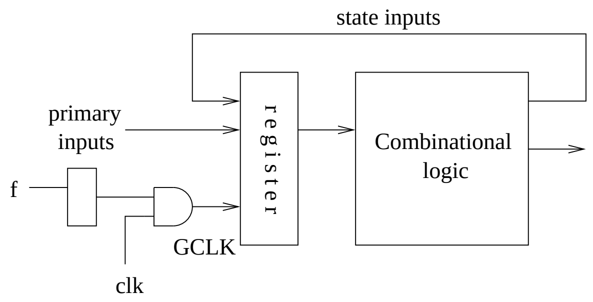

Figure E.4: Clock gating power optimization: when activation function f is 0, gated clock is stopped; through clock gating, power can be reduced even by 30%.

Extraction of Computational Kernels

Combinational circuits often "pass" through a relatively small set of states, hence they have a "typical" and a "non-typical" behavior. Typical behavior can be built by using small, fast and low power logic, called "computational kernel". Thus, the designer can use a parallel implementation, where the "kernel" logic of the FSM will be executed often, while the rest of the FSM logic will be seldom executed. This method achieves 20-50% power savings, but adds less than 15% delay, while its area overhead, due to parallelism, is around 20-70%.

Operand Isolation

This method is similar to clock gating, but is used in computational elements: components that will perform redundant operations are isolated by the use of control logic. This method achieves power savings of 15-30%, but its area overhead is in the range of 5-25%, while the slack is 30-50% larger.

E.2 Gate Level Power Optimization

E.2.1 Technology-independent Techniques

Combinational Circuits

Don't Care Minimization Through this method the designer tries to minimize the cost function of each node, that is, it is favourable to have a small number of cubes that implement the function in the Karnaugh-map. Also, by trying out different implementations of the function and by hypothetizing on the node switching activities, an implementation can be found that will minimize power, perhaps with an area overhead. Care must be taken though, as this method does not always guarrantee power savings, so it must first be ckecked thoroughly on paper or by special tools.

Sequential Circuits

State Assignment With state assignment, the designer attempts to minimize the state-to-state transitions. The extreme case is to use one-hot encoding, but with area and possibly power (as capacitance increases) overhead. As a result, states must be encoded in conjunction with how often the specific transitions occur; this means assigning state codes with small transitions to states that are often active. For example, in Figure E.5 we can see a 4-state FSM. We can derive a cost function $C = \sum_{i,j} w_{ij} H(i,j)$ which is indicational of the switching activity and area penalty of each case. By correctly selecting state assignments, we can decrease the FSM cost C, from 1.2 to 0.8.

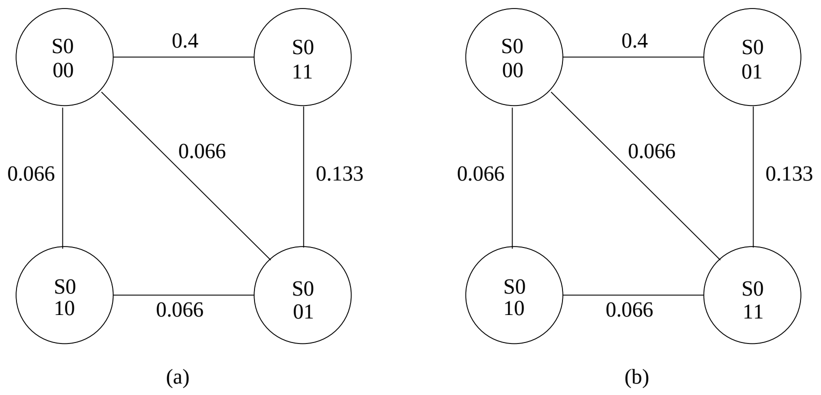

Figure E.5: State assignment power optimization: by carefully selecting state encoding, the designer can minimize the cost function $C = \sum_{i,j} w_{ij} H(i,j)$, which indicates switching activity. Case (a) is a power-consuming state encoding selection giving $C = 1.2$, while (b) gives $C = 0.8$.

FSM Decomposition With this technique, the FSM is divided into one ore more sub-FSMs; this is done because not all the sub-FSMs will be active at any time, hence some power management can be achieved. FSM partitioning into sub-FSMs is carefully carried out in order to minimize inter-FSM communication. The overhead of this method, an example of which can be seen in Figure E.6, is the area overhead for the addition of the sub-FSM activation/inactivation logic.

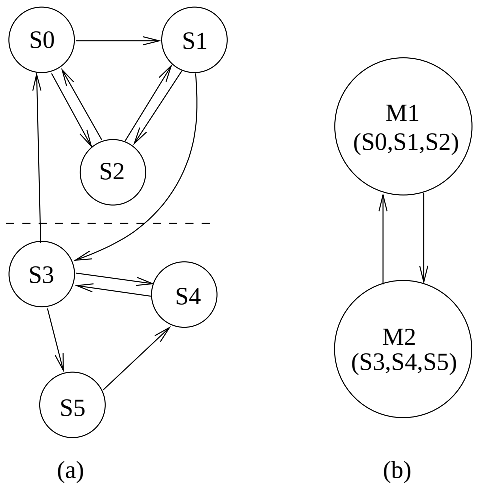

Figure E.6: FSM decomposition power optimization: in (a) we can see a traditional FSM, which can be partinioned into two sub-FSMs, M1 & M2, (b), only one of which will be active at any time.

Retiming & Guarded Evaluation Retiming is a performance oriented method. Latches are added to the glitchy nodes and 5-10% power savings can be achieved in pipelined circuits. Area overhead is not negligible, though. This optimization is not supported by Synopsys Power Compiler, thus the designer has to perform it on his own (see Figure E.7). With guarded evaluation, dynamic power management is carried out, but in the Gate-Level: guard logic (latches with enable) are added to block inputs.

E.2.2 Technology-dependent Techniques

Technology Mapping

The idea of technology mapping is to hide high-switching nodes inside the gates. If, for example, the tool synthesizes a function f, at first it performs "decomposition of f", thus building the "subject graph", which is the circuit that implements f, built from primitive elements. Next the tool performs "subject graph covering", that is, it chooses the appropriate gates from the technology library, in order to "hide" some high switching activity inside the

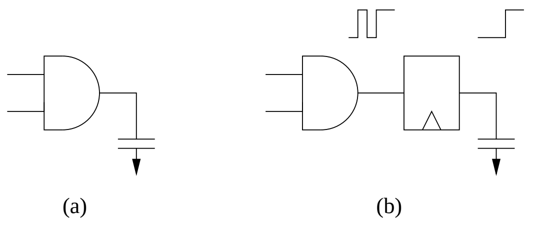

Figure E.7: Retiming power optimization: lathes are added (b) to glitchy nodes; power reduction is in the range of 5-10%, but area overhead can be significant.

gate; these gates will be directly built on silicon, so their switching activity will as low possible. An example of this mapping can be seen in Figure E.8.

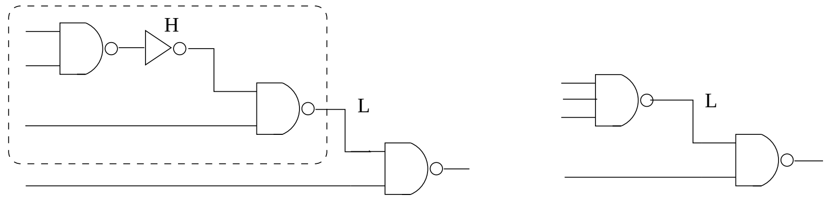

Figure E.8: Technology mapping power optimization: the covered circuit on the left is the "subject graph", which is then "covered" into a three input NAND gate, shown on the right; thus, the high switching internal node of the subject graph is "hidden" inside the gate implementation.

Gate Resizing

All technology libraries contain a number of basic primitives (gates, multiplexors, registers), but for every primitive, multiple editions exist, with different characteristics (size, speed, power consumption), as shown in Figure E.9. The synthesis tool chooses which instance to use depending on the contraints the designer has imposed. Hence, the tool chooses to use fast, but power consuming primitives in the critical path, and small, slow and low power primitives for other, non-critical, paths. Power savings can be up to 30%.

Buffer Insertion

Having a fully mapped gate level netlist, with exact timing data, we can add buffers and thus decrease the capacitance and the transition time of a node. For example, in Figure E.10 we can see how a buffer insertion between a driving NAND gate and two receiver flip flops reduces capacitance (hence power consumption) and node transition time.

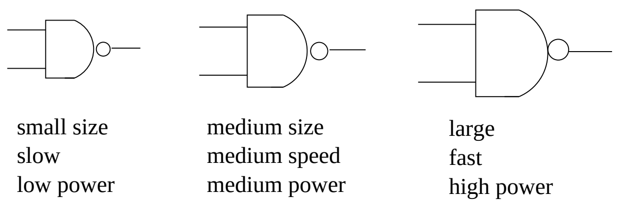

Figure E.9: Technology library primitive characteristics: the library contains different editions of the same primitives, which differ in area, speed and power consumption. Hence, the synthesis tool can choose to use a specific primitive, depending on the contraints set by the designer.

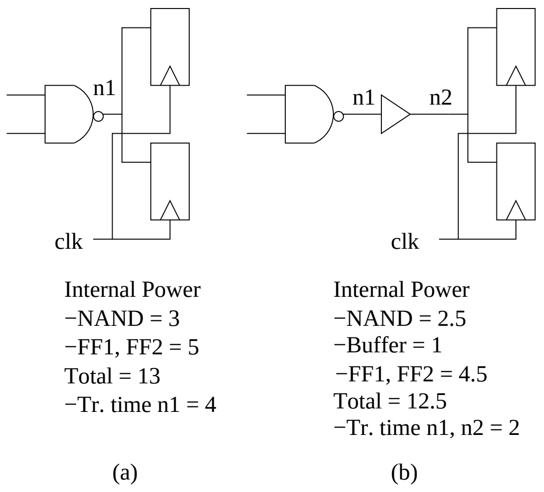

Figure E.10: Buffer insertion power optimization: case (a) shows a NAND gate driving two flip flops in parallel. By adding a buffer in the node (case (b)), NAND gate and flip flop power consumption decrease. Of course, buffer insertion should be carefully performed: mistakes may increase total power.

Dual-voltage Gates

Gates with low supply voltage consume small amounts of power, but are slow, too. The synthesis tool might be able to replace gates that do not affect the timing contraints with others of a lower supply voltage (see Figure E.11). Generally speaking, no more than two different supply voltages should be used in a circuit; this method is also preferrably followed locally in a design, as the special circuits needed at the low voltage outputs (level shifters), have large area and consume power [3].

[3]Special algorithms do exist that minimize the use of level shifters.

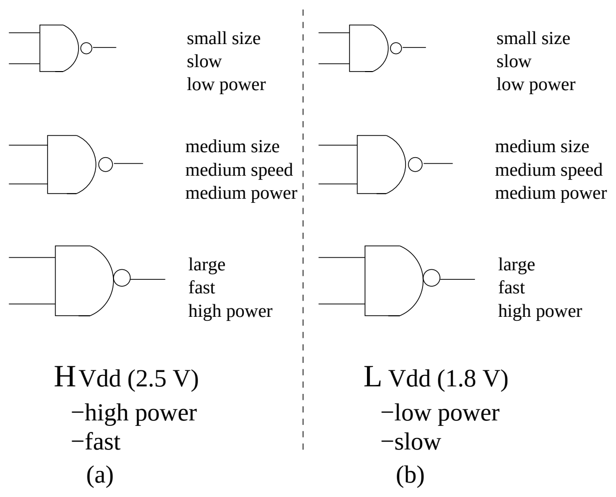

Figure E.11: Dual-voltage gate power optimization: case (a) is a high voltage (2.5V) technology library of three NAND gates, which are fast and non-power aware. Case (b) is a low voltage (1.8V) version of the same technology; it is slower, but consumes less power.

Leakage Power Minimization

By synthesizing the design with gates of high threshold voltage $V_{Threshold}$, we can achieve a minimum leakage power implementation. Timing constraints are then met by replacing the gates of the critical path with gates of smaller threshold voltage. Of course, switching power decreases as threshold voltage drops, and at the same time leakage power increases; thus, care must be taken.

Appendix F

LFSR Random Number Generators

Switch verification is a difficult task, with the most challenging part being the generation of realistic input traffic that must resemble, as close possible, real internet traffic. Because of the high link bandwidth (almost 10 Gbit/sec) supported by our 32×32 buffered crossbar switch, emulating internet *backbone* traffic was desirable. After traffic analysis data is acquired, traffic must be generated and sent to the switch inputs. Although this task is usually performed in software [1], generating traffic that follows specific mathematical distributions is quite difficult in hardware. In this section we will present the most important internet traffic analysis results, and propose a way to produce number sequences that follow a specific distribution in hardware, especially in FPGAs. Random number generation is mostly analysed, since every other distribution can be implemented by transformating a random number sequence.

F.1 Traffic Analysis

Traffic analyses were found in [23], while a cumulative distribution of packet sizes, and of bytes by the size of packets carrying them is presented in Figure F.1. The figure is a good example of the difference between per-packet and per-byte analyses. As we can see, there is a predominance of small packets, with peaks at the common sizes of 44, 552, 576, and 1500 bytes. The small packets, 40-44 bytes in length, include TCP acknowledgement segments, TCP control segments such as `SYN`, `FIN`, and `RST` packets, and `telnet` packets carrying single characters (keystrokes of a `telnet` session). Many TCP implementations that do not implement Path MTU Discovery use either 512 or 536 bytes as the default Maximum Segment Size (MSS) for nonlocal IP destinations, yielding a 552-byte or 576-byte packet size. A

[1]Such software traffic generation is briefly presented in [19], whereas a more technical and detailed presentation can be found in [29].

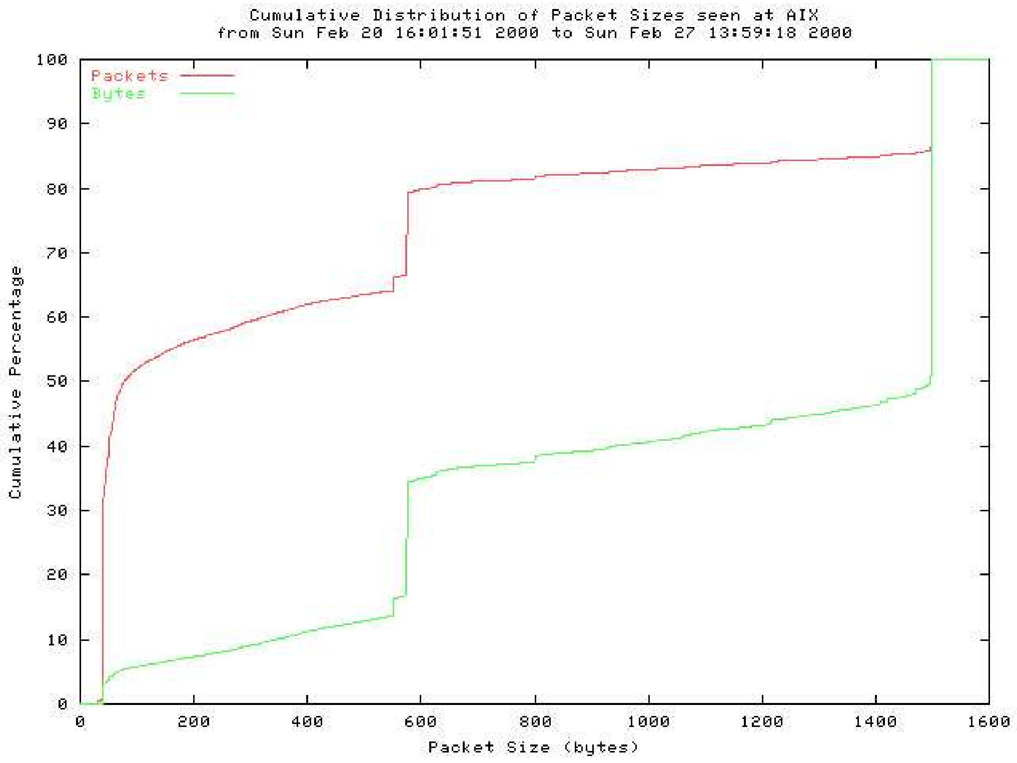

Figure F.1: IP packet length distribution: Data taken from a *www.caida.org* year 2000 simulation.

Maximum Transmission Unit (MTU) size of 1500 bytes is characteristic of Ethernet-attached hosts. Almost 75% of the packets are smaller than the typical TCP MSS of 552 bytes. Nearly half of the packets are 40 to 44 bytes in length. Note, however, that in terms of bytes, the picture is much different. While almost 60% of packets are 44 bytes or less, constituting a total of 7% of the byte volume, over half of the bytes are carried in packets of size 1500 bytes or larger.

F.2 Hardware Random Number Generators

Generally speaking, there are several requirements that a random number generator (RNG) must satisfy, the most important being: (a) uniformity, (b) independence, (c) long period, (d) repeatability, (e) portability, and, of course, (f) efficiency [28]. The latter requirement is the most difficult to meet in hardware and the most important, too, as hardware recourses are critical, both for ASICs and FPGAs.

RNGs can be either “truly random” or “pseudo random”, with the former to excibit true randomness (so the value of the next number is unpredictable), and the latter to only *appear* random. Truly RNGs can only be built in hardware by using thermal noise (possibly from a resistor), which is then amplified, latched and sampled. This obviously analog circuit is not preferrable in most cases.

The most efficient 1-bit RNG implementation uses the Linear Feedback Shift Register (LFSR). It is based on the fact that the next random value is a function of the older values, with 1 to 3 XOR operations between them. The LFSR is a very efficient pseudo-RNG, as for an m-bit random number, only an m-bit shift register and 1-3 XOR gates are needed; it also shows very nice statistical properties, while its period can become very large, too.

In order to design a multiple-bit RNG, various methods have been proposed. The most obvious is to use multiple copies of single-bit LFSRs. This method has the same operation speed as the signle-bit RNG, but has two major disadvantages: (a) it requires a different seed for each single-bit LFSR, in order to avoid correlation; and (b) it is inefficient, as it needs large hardware resources.

The answer to the above problems is the "Multiple-bit Leap-forward LFSR", which utilizes only one LFSR and shifts out several bits, with all shifts being performed in one clock cycle. This method is based on the observation that an LFSR is a linear system and the register state can be written in vector format $q(i+1) = A*q(i)$, where $q(i+1)$ and $q(i)$ are the contents of the shift register at $(i+1)^{th}$ and i^{th} steps, and A is a transition matrix. After the LFSR advances k steps, the equation becomes $q(i+1) = A^k * q(i)$. If we compute A^k then we determine the XOR structure and the circuit can leap k steps in one clock cycle [2]. An example of a 4-bit Leap-forward LFSR can be seen in Figure F.2. Leap-forward LFSR method utilizes extra combinational

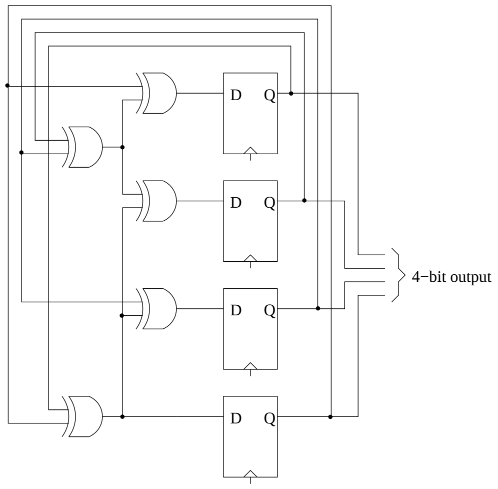

Figure F.2: 4-bit Leap-forward Linear Feedback Shift Register (LFSR) organization.

circuit, instead of duplicated LFSRs. For small k, this combinational circuit

[2]Transition matrix computation is presented in [28].

is not very complex and is thus ideal for FPGAs, since it balances register and combinational circuitry. However, for large k, the XOR structure grows very large and becomes the dominant factor; in that case, a "Lagged Fibonnaci" RNG can be used instead.

F.3 Non-uniform Number Hardware Sequence Generation

In order to produce general, non-uniform random numbers that follow a specific distribution, a similar procedure can be followed: first we generate a uniformly-distributed random number, and then we perform an extra processing step, in order to produce the desired value. There are two methods to implement that extra step, but the easiest and most straightforward is the "inverse tranform method": assuming a random variable X, with a probability distribution function (pdf) $P(X = x_j) = p_j$, where $j = 1, \ldots, m$ and $\sum_j p_j = 1$. We generate an n-bit random number U and set X as

$$X = \begin{cases} x_1 & \text{if } 0 <= U < p_0 2^n \\ x_2 & \text{if } p_0 2^n <= U < p_1 s^n \\ \cdots & \end{cases}$$

Hence, in the worst case, we need a 2^n by $log_2 m$ ROM look-up table. By computing the above values off-line, storing them in a ROM and addressing it with a uniformly distributed random number (possibly computed by a LFSR-RNG), the desired values are generated.

Appendix G

Thesis Overview

This thesis was carried out from February 2003 until June 2004 and was composed of 5 stages: (a) initial work, (b) switch architecture, HDL coding and verification, (c) synthesis and verification, (d) placement and routing and (e) thesis report. Figure G.1 shows the time spent in each of those stages; it should be noted that most stages overlapped.

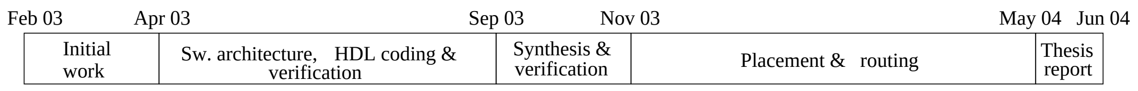

Figure G.1: Project timeline.

Entire code size of the project can be seen in table G.1. "Core Verilog Code" code infers to the 32×32 switch code written from scratch; "Automatically Generated Code" had to be produced in order to instantiate the 1024 crosspoints and the 32 credit and output schedulers, as well as organize the switch in columns. This was carried out by a C file. "Miscellaneous Verilog Code" was mostly written for the development of a 4×4 buffered crossbar switch with WRR schedulers, as well as for tests on LFSR random number generators.

Core Verilog Code	5500
Autom. Generated Verilog Code	10500
Miscellaneous Verilog Code	2000
Synopsys & SoC Encounter Scripts	800
C Code	600

Table G.1: Thesis code size.

Bibliography

[1] "Buffered Crossbar (CICQ) Switch Architecture", *Computer Architecture and VLSI Laboratory, Institute of Computer Science, Foundation for Research and Technology, Hellas (FORTH)*; http://archvlsi.ics.forth.gr/bufXbar/

[2] Kenji Yoshigoe and Kenneth J. Christensen: "An Evolution to Crossbar Switches with Virtual Output Queueing and Buffered Cross Points", *IEEE Network, September/October 2003*

[3] Yeh, Hluchyj, Acampora: "The Knockout Switch: A Simple, Modular Architecture for High-Performance Packet Switching", *IEEE Journal of Selected Areas in Communications*, pp. 1274-1283, Oct 1987.

[4] M. Karol, M. Hluchyj, and S. Morgan: "Input versus Output Queueing on a Space Division Packet Switch", *IEEE Trans. Commun.*, vol. 35, no. 12, Dec. 1987, pp. 1347-56

[5] Altera Corporation: "Crosspoint Switch Matrices in MAX II & MAX 3000A Devices", *Altera Application Note 294*; http://www.altera.com/literature/an/an294.pdf

[6] Altera Corporation: "Using Stratix GX in Switch Fabric Systems", *Altera White Paper*; http://www.altera.com/literature/wp/wp_switch_fabric.pdf

[7] Vinita Singhal, Robert Le: "High-Speed Buffered Crossbar Switch Design Using Virtex-EM Devices", *Xilinx Application Note 240*; http://direct.xilinx.com/bvdocs/appnotes/xapp240.pdf

[8] Xilinx Corporation: "Building Crosspoint Switches with CoolRunner-II CPLDs", *Xilinx Application Note 380*; http://www.xilinx.com/ publications/products/cool2/apps_pdf/xapp380.pdf

[9] Xilinx Corporation; http://www.xilinx.com

[10] Xilinx Corporation: "Rocket IO Multi-Gigabit Tranceiver"; http://www.xilinx.com/products/virtex2pro/rocketio.htm

[11] Nick McKeown, Martin Izzard, Adisak Mekkittikul, William Ellersick, Mark Horowitz: "The Tiny Tera: A Packet Switch Core", *HOTI V*, Stanford University, August 1996; http://tiny-tera.stanford.edu/tiny-tera/papers/papers/HOTI_96.ps

[12] Nick McKeown: "The iSLIP Scheduling Algorithm for Input-Queued Switches", *IEEE/ACM Transactions on Networking*, vol. 7, no. 2, April 1999, pp. 188-201; http://tiny-tera.stanford.edu/ nickm/papers/ToN_April_99.pdf

[13] T. Anderson et al.: "High-Speed Switch Scheduling for Local-Area Networks", *ACM Trans. Computer Systems*, vol. 11, no. 4, Nov. 1993, pp. 319-52

[14] K. Yoshigoe and K. Christensen: "A Parallel-Polled Virtual Output Queued Switch with a Buffered Crossbar", *2001 IEEE Workshop on High Performance Switching and Routing*, pp. 271-275, May 2001; http://www.csee.usf.edu/ kyoshigo/hpsr01.pdf

[15] K. Yoshigoe, K. Christensen, and A. Jacob: "The RR/RR CICQ Switch: Hardware Design for 10-Gbps Link Speed", *Proceedings of the IEEE 2003 International Performance, Computing, and Communications Conference*, pp. 481-485, April 2003; http://www.csee.usf.edu/ kyoshigo/ipccc03.pdf

[16] Christopher Heer, Andreas Kirstadter, Christian Sauer: "Self-Routing Crossbar Switch with Internal Contention Resolution", *International IEEE Conference on Electronics, Circuits, and Systems, ICECS 2001*, Sept. 2-5, Malta; http://www.lkn.ei.tum.de/lkn/ mitarbeiter/akirstaedter/papers/2001_ICECS_Chipset.pdf

[17] Heikki Kariniemi, Jari Nurmi: "A Crossbar-based ATM Switch on FPGA for 2.488 Gbits/s CATV Network with Scalable Header Remapping Function", *Institute of Digital and Computer Systems, Tampere University of Technology*; http://www.scit.wlv.ac.uk/ in8189/CSNDSP2002/Papers/C1/C1.4.pdf

[18] F. Abel, C. Minkenberg, R. Luijten, M. Gusat, I. Illiadis: "A Four-Terabit Packet Switch Supporting Long Round-Trip Times", *IEEE Micro Magazine*, vol. 23, no. 1, Jan/Feb 2003, pp. 10-24

[19] M. Katevenis, G. Passas, D. Simos, I. Papaefstathiou, N. Chrysos: "Variable Packet Size Buffered Crossbar (CICQ) Switches", *Proc. IEEE International Conference on Communications (ICC 2004)*, Paris, France, June 2004, vol. 2, pp. 1090-1096; http://archvlsi.ics.forth.gr/bufxbar/

[20] Cadence Corporation; http://www.cadence.com

[21] UMC 0.18 micron Libraries; http://www.umc.com/english/design/b3.asp

[22] Xiang Li: "Saving Power with Power Compiler"; http://vlsi1.engr.utk.edu/ xiang/652/POWER-COMPILER.ppt

[23] Cooperative Association for Internet Data Analysis: "Packet Length Distributions"; http://www.caida.org

[24] N. Chrysos, M. Katevenis: "Mutliple Priorities in a Two-Lane Buffered Crossbar", *to be published in Globecom 2004*; http://archvlsi.ics.forth.gr/bufxbar/mprio_bxb_chrys_04-03.pdf

[25] S. Devadas and S. Malik: "A Survey of Optimization Techniques Targeting Low Power VLSI Circuits", *Proc. IEEE Design Automation Conf.*, pp. 242-247, June 1995; http://portal.acm.org/ ft_gateway.cfm?id=217536&type=pdf&coll=portal&dl=ACM&CFID=24235476&CFTOKEN=52253054

[26] Altera Corporation: "System-on-a-Programmable-Chip Development Board User Guide"; http://www.altera.com/literature/ug/sopcug.pdf

[27] Ahmed Younis et al.: "A Low Jitter, Low Power, CMOS 1.25-3.125 Gbps Tranceiver", *Proc. of 2001 ESSCIRC*, pp. 148-151; http://www.imec.be/esscirc/esscirc2001/Proceedings/data/79.pdf

[28] Pong P. Chu and Robert E. Jones: "Design Techniques of FPGA Based Random Number Generator", *Military and Aerospace Applications of Programmable Devices and Technologies Conference, 1999*; http://academic.csuohio.edu :8080/chup/chu_web_stuff/research_stuff/Random.PDF

[29] Georgios Passas: "Performance Evaluation of Variable Packet Size Buffered Crossbar Switches", *Institute of Computer Science, FORTH*, Technical Report 328, Heraklion, Crete, Greece, November 2003; ftp://ftp.ics.forth.gr/tech-reports/2003/2003.TR328_Evaluation_Packet-Size_Buffered_Crossbar_Switches.pdf

[30] Quantum Flow Control (QFC) Alliance: "Quantum Flow Control: A cell-relay protocol supporting an Available Bit Rate Service", version 2.0, July 1995; originally at http://www.qfc.org but no longer there – copy at http://archvlsi.ics.forth.gr/ kateveni/534/qfc/

[31] "CS-534, Packet Switch Architecture", *Computer Science Department, University of Crete*, lecture notes; http://www.archvlsi.ics.forth.gr/ kateveni/534/04a/s22_chip.html

[32] William J. Dally, John Poulton: "Digital Systems Engineering", Cambridge University Press, ISBN 0521592925

[33] Network Equipment Building Standards (NEBS); http://www.nebs-faq.com/, http://www.telcordia.com/

[34] K. Harteros: "Fast Parallel Comparison Circuits for Scheduling", *Institute of Computer Science, FORTH*, Technical Report FORTH-ICS/TR-304, 78 pages, March 2002; http://archvlsi.ics.forth.gr/muqpro/cmpTree.html

[35] Cyriel Minkenberg, Ronald P. Luijten, François Abel, Wolfgang Denzel, Mitchell Gusat: "Current Issues in Packet Switch Design", *HOTNETS '02*, 10/02 Princeton, NJ, USA; http://www.acm.org/sigs/sigcomm/HotNets-I/papers/minkenberg.pdf

[36] "International Technology Roadmap for Semiconductors", 2003 Edition, Executive Summary; http://public.itrs.net/Files/2003ITRS/Home2003.htm

[37] "ECE260B - CSE241A", *University of California, San Diego*, Winter 2004, lecture notes; http://vlsicad.ucsd.edu/courses/ece260b-w04/

[38] "ECE260B - CSE241A: Packaging", *University of California, San Diego*, Winter 2004, lecture notes; http://vlsicad.ucsd.edu/courses/ece260b-w04/ece260b-w04-packaging.pdf

[39] IBM Microelectronics Presentation: "Embedded DRAM Comparison Charts", *IBM Microelectronics*; http://www-306.ibm.com/chips/techlib/techlib.nsf/products/Embedded_DRAM

[40] John Barth, Darren Anand, Jeff Dreibelbis, Erik Nelson: "A 300MHz Multi-Banked eDRAM Macro Featuring GND Sense, Bit-Line Twisting and Direct Reference Cell Write", *ISSCC 2002*, Session 9, DRAM and Ferroelectric Memories, 9.3; http://www-306.ibm.com/chips/techlib/ techlib.nsf/techdocs/ 48D7E84C4F67E55287256DD700021AA1/$file/isscc02_D09_3.pdf

[41] Chorng-Lii Hwang et al.: "A 2.9ns Random Access Cycle Embedded DRAM with a Destructive-Read Architecture", *IBM Microelectronics*; http://www-306.ibm.com/ chips/techlib/techlib.nsf/ techdocs/E95CC111124D5A5587256DD6007DE26D

[42] Steve Tomashot: "An Embedded DRAM Approach", *IBM Corporation*; http://www-306.ibm.com/ chips/techlib/techlib.nsf/ techdocs/C4DA54943C004A1987256DC8004EA766/$file/emdramprez.pdf

[43] "*e*Si-RAM/2P^{TM} Two-Port Register File SRAM", 512 Words 32 Bits per word, Data Sheet

[44] Terry Tao Ye, Luca Benini, Giovanni De Micheli: "Analysis of Power Consumption on Switch Fabrics in Network Routers", *Proceedings of Design Automation Conference, DAC 2002, IEEE*, pp. 524 -529, June, 2002; http://www.gigascale.org/pubs/408/34-02-ye.pdf

[45] Hang-Sheng Wang, Li-Shiuan Peh, Sharad Malik: "A Power Model for Routers: Modelling the Alpha 21364 and Infiniband Routers", *IEEE Micro*, Jan-Feb 2003, pp. 26-35; http://www.princeton.edu/ peh/publications/powermodel.pdf

[46] Synopsys Corporation; http://www.synopsys.com

[47] "Power Compiler User Guide", *Synopsys Corporation*, June 2003