Architecture for a Reliable Router Control Plane
Arkitektur for et pålideligt router control plane

Rune M. Barnkob (s973431)

April 29, 2005

Supervisor: Michael S. Berger
Research center COM
Technical university of Denmark

Abstract

In this report we will design and implement a cluster of cooperating LiABs. The motivation for this is an attempt to increase the reliability of the routing plane in a core router. This will be done through the development of a printed circuit board suitable for connecting 4 LiABs to a Xilinx FPGA. The cluster of 3 LiABs is connected through a queue to the fourth LiAB emulating the actual router forwarding plane. The firmware developed for this connection is described. Finally, programming and usage of the design is elaborated.

Contents

List of Figures

List of Tables

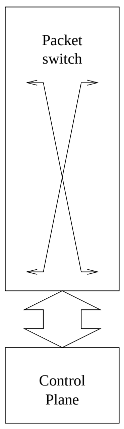


Figure 1: Division of routing plane and forwarding plane in a router.

1 Introduction

This project aims at designing a platform for the development of a reliable router control plane. It thus attempts to increase the availability and reliability of a node in the network.

Though packet switched nets (like the internet) was designed to be very resilliant towards failures of a single node, telcos do not like backbone routers dropping out as this may have very unfortunate side-effects such as loss of traffic. This is emphazised in the survey performed by BTExact, a part of British Telecom responsible for research. Their survey ([1] and [2]) indicated that the main consideration for telcos were reliability, not cost. Serveral companies have provided the telcos with solutions for high availability routing systems, such as Avici's NSR. In their whitepaper [3] Avici lists serveral different approaches to increasing availability of routers. Of the suggested methods only one is described as feasible and the design suggested is persued in this report.

1.1 Problem

As shown in figure 1, a router is considered two distinct layers - a routing layer and a packet switch/forwarder. This division is supported by contemporary router architectures as described in [6] and [11], where high-speed interconnects between the line cards and forwarding engines are used. This design has the added advantage that even though the routing layer should fail (deliberately or accidentially), the packet switch can continue operation (shown in [20]). As all complex software, software for the routing plane is likely to be bug-ridden and must therefore sometimes be updated or patched and might even crash. This is a cause for much of the downtime in the current routers. While the routing layer is unavailable due to updates or crashes, the router is not able to respond to routing messages in the network and update the routing tables accordingly. Not transmitting the heartbeat messages used in most internal routing protocols [7] [10] or keeping the TCP connection used in BGP [8] might cause eg. route-flapping, where the lowest-cost route moves to another route temporarely, only to be swapped immediately back to the original route when the routing plane of the affected router becomes available again. Resolving this problem in the protocol has been suggested, such as "I'll-be-back" messages in OSPF [4], but unfortunally, this would only solve expected outages in case of eg. software updates - not a random crash. A graceful restart mechanism such as the one described in [11], would also allow continue usage of the forwarding plane of the router and facilitate a rapid restart of the routing plane.

The IBB extension suggested in [4] solves many problems related to expected outages by continuing to use the router that has announced an outage of the routing plane while still able to continue forwarding packets. The extension considers what messages related to routing decisions it might have been lost in the routing plane. This way, the IBB extension tries to avoid loops and black holes by considering what effects the lack of updates to the link state database used in OSPF, has. One major problem remains in relation to this approach - the routing layer must announce its expected outage to the surrounding routers. Hardware failures can thus not be handled gracefully by this approach. It also requires all vendors to support these protocol extensions [3], a situation not likely to be resolved soon. The advantage of this solution is that it can be solved entirely in software and needs no extra hardware in order to support it. This, in my opinion, has the unfortunate side-effect that it requires changes to the protocol, which is unlikely to happen soon in an autonomous network such as the Internet.

In the other trench we have full hardware redundancy. This solution works by using serveral routing processors, most likely solving the same task so we have serveral answers we can compare in order to ensure the correct answer is returned. This solution, though, is costly in terms of hardware, that must be specially developed to support this majority voting principle. This is not as much an issue in relation to telcos as they are willing to pay the needed extra price to obtain reliability and scalability [1]. If these routing engines run in lock-step (same algorithm, same instruction stream, at almost same point), we do not solve problems relating to software errors as the routing engines fail at the exact same time. This was most noteably show in the case of Ariane 5, that had fully redundant hardware but lacked software redundancy.

This problem could be relieved by using n-version programming. If we ask n independant department to develop n independant solutions to the requirements put forward, they are likely to come up with n somewhat different solution (by ensuring that they to not communicate extensively) and it is though unlikely that the same errors exists in all n different versions. This protects us against software failures by also having a redundant software running. The disadvantage is of course price. It is widely known that the most expensive part of developing a hardware-software solution is the software, so n 'plifying this cost is certain to be very expensive.

A more cost-consious solution to alliveate the possibility of crashes in the software is checkpointing. By checking results before committing them to the softwares states and roll-back to the last known good state of the software, some protection against software errors are possible. This also allows us to perform very rapid restarts of the system as we already have the state of the program ready. The UNIX `fork()` actually resembles this somewhat though it does not allow us to inspect the old state of the process before the fork. This technique is implementet in [27] and a performance-increasing implementation of it has been proposed in [28]. The problem with this approach is that it does not really have the redundancy of the previously mentioned method - it only allows very rapid restarts in case of crashes. Errors in hardware is not handled by this approach.

Unlike OSPF and RIP, that uses UDP messages to transfer routing information, BGP uses a TCP connection. During the routing engine, the connection may timeout causing the other router to consider this router failed. This poses a problem as this BGP router will refrain from using the failed gateway and thus might be forced to use suboptimal routes. As BGP is often used in telcos back-bone routers, this suggest to us we need a way to seamlessly ensure that the other end of the BGP connection never discovers a temporary outage of our control plane.

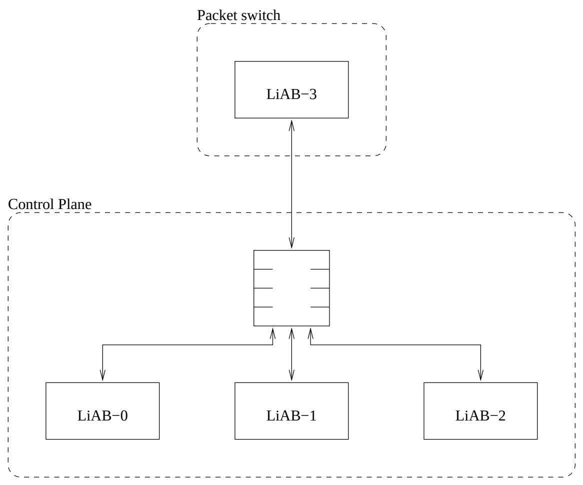


Figure 2: The concept of a cluster with a quarentine queue towards a final LiAB acting as the packet switch

2 Concept

The basic concept of the design is shown in figure 2, where a cluster of 3 LiABs will function as the routing plane connected to the packet switch through a quarentine queue. In the quarentine queue, the messages directed at the packet switch must either be accepted by at least two LiABs or pass a time-out period without being flagged as defect by two LiABs. This is somewhat similar to the NSR solution by Avici [3], though even more closely knitted together. One of the LiABs is the elected master of the cluster and is hence the one responsible for answering all incoming requests. The two slaves in the system works both as hot stand-by's ready to take over the role as master, and as monitors for the correct functioning of elected master. This solution is not fully redundant and assumes the correct function of a cluster coordinator. As this most likely is implemented in hardware, failing hardware will be the most probable cause of complete system failure. By designing the hardware as simple and robust as possible, it is easier to debug and verify. The default behaviour of letting a message pass the quarentine queue after a given period is chosen to assure the cluster will keep functioning, but slowly, with only one LiAB running. If two LiABs are running, the elected master is assumed to be working correctly and, if the other LiAB agrees, the functioning will be fast. With three LiABs running, it is able to do full triple-voted functioning where the output from the cluster is assured by at least two LiABs to be correct.

This design should allow the outside of the cluster to communicate to a very reliable virtual system and, as the clustered machines are so closely knitted together, the cluster should be able to hand-over a TCP-connection between a former master and a slave without terminating it. This so-called TCP hand-over is one of the major goal of the cluster and is especially important for BGP-based routing solutions such as those most commonly found in back-bone routers. The state of each LiAB must be somehow communicated between the machines in the cluster without being forwarded to the packet switch.

As I do not have access to a packet switching network, the fourth LiAB depicted in figure 2 must work as an emulator for the actual hardware packet switch and be able to inject messages into our triple-redundant routing engine.

In order to support this cluster, I will attempt to design a simple PCB, described in section 3, suitable for connecting a cluster of LiABs (Linux-in-A-Box [22]) together using a smart bus implemented inside a Xilinx FPGA. The choice of a bus as the interconnect will be explaned in section 4. To interface towards the FPGA, a driver closely knitted to the firmware uploaded to the FPGA, needs to be developed - the requirements for this driver is described in section 5.

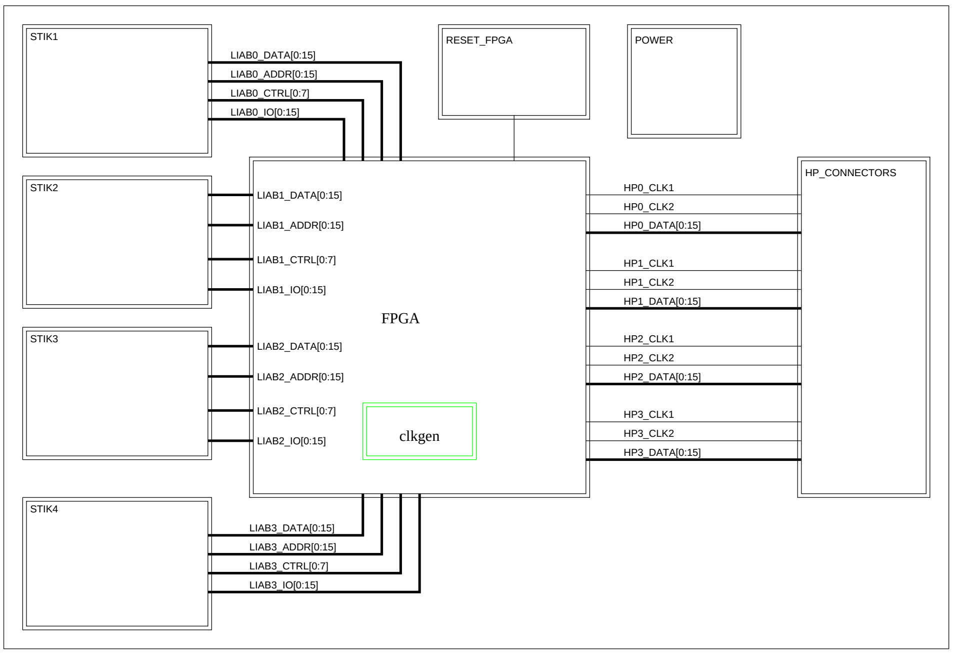


Figure 3: Division of schematic in FUBs - the 8 fubs comprising the main level of the schematic.

3 PCB

The purpose of the PCB is to connect 4 LiABs to an FPGA containing a core capable of sorting messages. Three LiABs comprise the cluster being the reliable control plane and the last LiAB emulates the packet switch. Therefore, the PCB consists of quite few components needed to achieve this simple interconnect purpose and a couple of blinkenlichten [29].

The schematic has on purpose been kept simple as to reduce the likelihood of a failure on this part - if a fully redundant system where to be implemented, the complexity of the PCB would increase substantially as the number of separate FPGAs must be increased to 4 to allow for majority-votes among the FPGAs. Instead, I have chosen to focus on making the contents of the FPGA sufficiently robust to ensure a reliable system.

The tool from Mentor Graphics, DxDesigner, operates with FUBs (FUnction Blocks), and as seen on the overview in fig. 3, the schematic consist of 8 major blocks, 4 LiAB connectors, an FPGA, a clock-generator, a group of HP connectors for a logic analyzer, a reset pulse generator and a power FUB. To ease the PCB routing of signals, I grouped the signals in busses where appropriate, but in order to fill the fixed size busses (8-bits or 16-bits), I took signals that should be only somewhat related to each other and grouped them in a bus, eg. CS0-CS2 and A0-A9.

3.1 STIK1, STIK2, STIK3 and STIK4

These four FUBs contain the LiAB connectors. Address bus consists of A0-A9 and CS0, CS1 and CS2 plus IORD, IOWR and ALE. These signals are all related to the addressing of the output. The 16 data pins and 16 I/O pins was placed in two busses, and finally the remaining signals was grouped in the CTRL bus. A

68Ω resistor was placed on each input pin in order to protect the FPGA against undesirable spikes.

3.2 HP_CONNECTORS

The design contains connectors for 4 HP connectors. Each HP connector has 16 bits of data and 2 clocks, one of which is connected to the on-board clock generator and the other one is connected to the FPGA to allow the FPGA generate a clock signal for the HP connector.

The HP connector contains a pin called Vcc; this pin is not connected as it is a power source from the HP analyzer itself to possible on-board electronics. In our case, connecting it would have caused fault current damaging the PCB.

3.3 RESET

The reset circuit is a simple pulse generator making a pulse of a given length to the output. This was a design taken from another project called "Switch Emulator Board" by JH and YY.

3.4 POWER

Quite boring as it just contain the connectors coupled to a large number of capacitors for decoupling the power and assuring a constant supply to the quite power-hungry FPGA.

The board uses three voltage levels, 5V, 3.3V and 1.8V, each has an input terminal.

I must admit, though, that I probably overdid the decoupling according to our engineer, as I basicly had 1 capacitor for each Vcc pin on the FPGA.

3.5 CLKGEN

Hidden within the FPGA is a FUB containing the clock generator and a clock distribution chip CDC-VF-2405. This low skew driver chip is used to generate 9 amplified copies, of which 5 are used, of the on-board XTAL 100MHz clock. It also contains a jumper to allow an external clock to be fed into the board instead. The design includes a SMA connector, but the connector was never fitted on the board.

When the board was produced, it turned out I had overlooked an enable pin on the CDC chip, but luckily, a Vcc pin was located right next to it so a small solder fixed this little error.

3.6 FPGA

3.6.1 Configuration

The configuration of the FPGA will be done by two XC18V04 chips [13] placed in cascade, together able to hold 8Mb configuration data; more than the 6.3M needed for the XCV812E [12]. I originally intended to use the parallel configuration of the FPGA as this method is the fastest for configuring the FPGA, but this idea was rejected by Brian M. S. as it would require an entire bus to be routed from the XCV18 chips to the FPGA. I was recommended the Master Serial programming mode instead, where the FPGA generates a clock-signal to the EEPROMs to use for sending data. This programming technique has previously been used on COM with positive results and requires far less connections. The programming of the FPGA can be monitored on an attached LED, which will be lighted when the programming has been completed. Two switches has also been added to restart programming and to force the FPGA to restart its initialization process. Additionally, a JTAG interface has been added to the schematic to allow reprogramming of the EEPROMs embedded in the XC18V chips. The JTAG chain also covers the FPGA and can be used to configure only the FPGA with a test configuration before the final configuration is ready.

3.6.2 Switches and LEDs

A number of LEDs has been placed on the design in order to be able to output information from the FPGA. LEDs can usually be placed either so the device drains power through the LEDs or sources power through them. I choose the former method with a 330Ω resistor as it is usually easier for a chip to sink a current than to source it. This has the odd effect, though, that the LEDs will light when a low level is output and remain dark if a high level is put on the output - obviously contrary to the intuitive use. Unfortunally, this turned out to have the unfortunate effect that the diodes was turned the opposite way on the final board.

The board also contains 12 DIP switches, 3 of which are used to set the programming mode of the FPGA. These three switches must be placed according to the table on pg. 12-13 in [12], ”000” in master serial programming mode if EEPROMs are used, ”001” if the JTAG interface is used for configuration. All switches have weak pull-ups using 4K7 resistors.

3.6.3 Clock inputs

The FPGA features 4 clock-domains, 3 of which has been attached to LiAB 0-2, the fourth has been attached to the fixed 100MHz clock source on the board. Originally, the intention was to split clock domains between the different LiABs over the embedded BlockRAM and clock the internal bus at 100MHz, but this turned out to be impossible as the Place&Route tool did not allow us to use this many clocks and the 100MHz was thus chosen for the entire FPGA design.

3.6.4 Pin mappings

I had originally routed the signals logically close to each other without regard to the actual package pins - this turned out to yield some problems when the PCB was layouted - therefore, Brian chose to move some of the busses to allow a smaller layout on the PCB. Seen from the inside of the FPGA, this does not pose any problems as it is only a change to the pin mappings.

The pin mappings was written in a CSV (Comma-Separated-Values) in a simple text file containing pin name and location on the package. I never made DxDesigner export the schematics pin mappings (couldn’t make BoardLink work), so it was manually entered. Each logical function was placed in its own file (see appendix A) and was verified as wrong pin mappings could cause difficult-to-find bugs and maybe even damage to the board. This actually did turn up a couple of typos. It later turned out that Xilinx ISE actually preferred to receive input as an UCF file with lines in the format:

```
NET "xx" LOC = "xx" ;
```

A small Perl-script shown in figure 4 was made to convert the CSV files to UCF format - and assuming the Perl script did not touch the values, the UCF files need not be verified again.

Note that the script actually throws pin-directions aways - this is because the pin names must match the entity ports in the VHDL code, and the entity ports already has directions.

```
while ($l = <STDIN>) {
  if ($l =~ m/^([A-Za-z]+[0-9]+), ?(.+), ?(InOut|Input|Output)/) {
    ($loc,$net,$dir) = (uc $1,$2,$3);
    $net =~ s/</\(/g;
    $net =~ s/>/\)/g;
    print "NET \"$net\"  LOC = \"$loc\"  ;\n";
  }
}
```

Figure 4: Perl script to translate CSV files to something.

An important other thing to note is that the UCF files may not contain nets that are not used. Usually, this would not be a problem if your entities declare the ports, unless the synthesis-tool removes the unused pins. For the fore-mentioned reason it is important to note that # denotes a comment in the UCF files.

3.7 Component placements

In figure 5 we see an overview of the produced PCB. Power is supplied in J3 (5.0v), J4 (3.3v) and J5 (1.8v). U2-U5 is the connectors to the HP Aglient logic analyzer, whose 16 bits can also be used for other purposes. U1,U7,U8 and U9 is the four LiAB connectors. J8 is the Multilinx JTAG connector used to program the board. U10 and U11 is the two EEPROMs to store configuration data in. There is no need to use these two EEPROMs when trying out FPGA configurations - configurations of the FPGA can be programmed directly into the FPGA (U12). The mode bits controlled by S1's bit 1-3 must be set according to [12].

To recapitulate the important components on the board, they are listed in table 1. The original PCB print can be found in appendix B. All component placements, layout/routing and assembly of the board was performed by Brian M. Sørensen.

3.8 Summary

The PCB has been produced, is ready and has passed initial test - the FPGA can be programmed using the JTAG chain as was intended. The PCB has the layout depicted in a reduced version in figure 5. The main change between the actual pcb and the reduced overview is the removal of resistors and capacitors to improve readability (the original can be found in appendix B. It should, though, still be able to navigate around the board - note the position of the power supplies.

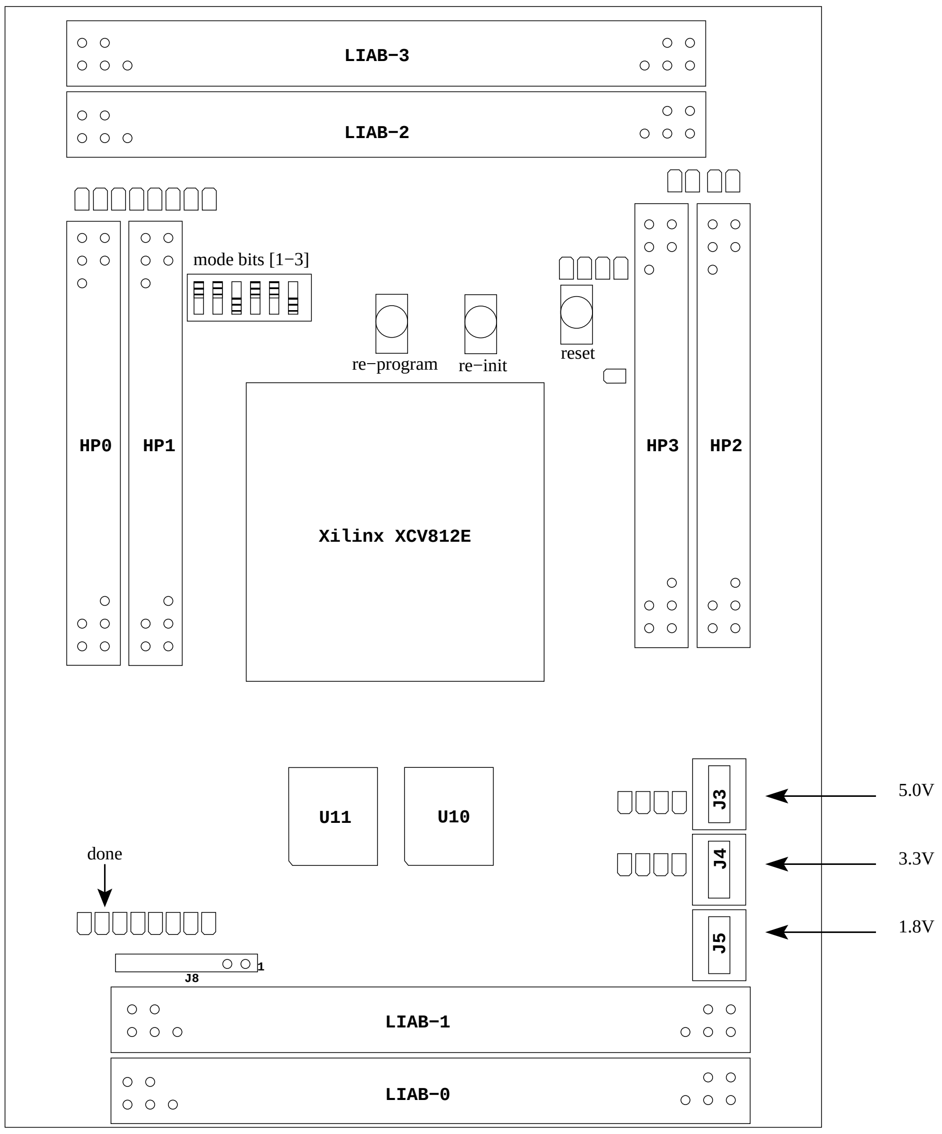


Figure 5: Overview of the final PCB.

Ident	Purpose
J1	reduced JTAG input
J8	multilinx JTAG input (connect to PC)
D49	program done LED
J3	5.0v Vcc supply
J4	3.3v Vcc supply
J5	1.8v Vcc supply
J7	clock source select (SMA-connector or on-board clock)
U3	HP0 connector
U5	HP1 connector
U2	HP2 connector
U4	HP3 connector
U1	LiAB-0
U7	LiAB-1
U8	LiAB-2
U9	LiAB-3
SW1	init
SW2	reprogram
SW3	reset signal to FPGA
S1(1-3)	mode bits

Table 1: The most important components on the PCB.

4 Firmware

The FPGAs purpose is to act as an intelligent interconnect between the LiABs. It could be configured either as a switch, in which case it should not do anything but move data around, or, as we decided early in the design process, it should have larger responsibilities.

Several different designs of the interconnect was considered. The requirements to the interconnect is described in the first sections, followed by designs of the interconnect. The bus-based design was selected and optimized. The implementation proved to be very time-consuming even though the bus-based design was chosen for its simplicity. The problem was an unexpected delay in the on-chip routing of signals consuming up to 80% of the worst-delay path. The time-consuming implementation therefore took away time required for the development of the driver and the modified TCP stack along with the test-case. These are hence only described as would-be designs.

Conventions used in the diagrams:

- blocks with round corners denotes combinatorial functions
- large squares are blocks containg storage, eg. latches or registers
- signal "clk" is implicitly used whenever squares with unconnected triangles are used[1].
- All state diagrams in this section are clock driven - if no conditions are attached to a transition, it will be taken on next clock, otherwise, it will be taken on a clock where conditions are met.

Though very advanced fault-tolerant designs are possible, such as those described by [18] and [17], the techniques will not be pursued as the design is supposed to be placed in shielded environments and the techniques described are most suited for no-access environments such as space applications.

4.1 Messages

As mentioned in section 2, the design uses a quarantine queue, which allows the two non-master LiABs to vote against a message sent from the third LiAB. This is accomplished by allowing a LiAB to either sent a pass or a stop message regarding a message directed to the packet switching matrix. The pass and stop messages will be very small as they only need to tell us which message they refer to. As the PS must also be allowed to send data to the LiABs (otherwise, the design would be pointless), we need a total of 4 messages for quarantine handling of messages - to-PS, from-PS, pass and stop.

In order to allow our cluster to maintain a consistent state between the 3 LiABs composing the cluster, we must be allowed to transmit messages between the three LiABs that does not propagate to the PS. When a LiAB has been IPLed, it will not have a consistent state, and instead of leaving it unable to join the cluster until it has collected enough information to be in sync with the cluster, it must be allowed to request a state dump from the current master. We therefore need three messages regarding state updates, requests and dumps.

To allow us to consider the current state of the cluster, we must know which LiABs are currently available. There are at least two ways of assuring a LiAB is alive - one is to query it to see if it responds, the other is to require it to tell us it is

[1]As I realized LeonardoSpectrum insist we use global clock-buffers for all clocks and only 4 GCB are available, it was irrelevant to do a multi-clock-domain design.

alive at periodic intervals. If you choose the latter approach, these messages should of course be generated by the software processing messages on the LiABs, not by some timer interrupt on the LiAB! I have chosen to use a heartbeat approach to the system as this is easiest to implement. If the heartbeat is absent for an appropriate period of time, the LiAB is considered crashed, and if it is the master, another LiAB is elected as the master of the cluster.

These considerations lead us to summarize the following messages that must be directed at either the cluster, the PS or the FPGA itself:

- FROM-PS
  message from the packet switch to the control plane.
- TO-PS
  message from the control plane to the packet switch.
- PASS
  message regarding another message, the message should be passed.
- STOP
  message regarding a faulty message - the message should be dropped from the quarantine queue.
- HEARTBEAT
  "I'm alive" message from a LiAB in the cluster.
- STATE-REQ
  "I need to know your state", an IPLed LiAB need the cluster state.
- STATE-DUMP
  Message containing state.
- STATE-UPD
  "change this in your states" message to the cluster.

To ease the quarantine queues work with sorting data, I decided that the first few octets of data in the message must be directed at the FPGA itself. Also, it gives us the advantage that we know what data a message contains before we complete the transmission of data and we can thus start processing data even before it is completely transferred.

4.2 Cells

Early in the process, it was decided that moving fixed-size cells around between the LiABs would be easiest to implement. The size of these fixed cells was dictated by the available hardware - namely the Xilinx XCV812Es on-board 4KiBit BlockRAM. Originally, the intention was to make the cells 1024 octets as this would be somewhat above a common MTU on the Internet of 576 bytes [5] (X.25). Obviously, a cell size of 1024 octets is below the Ethernets 1536 octets MTU [6], but 1536 was considered an annoying number to implement as it was not a power of 2. To include Ethernets MTU, we could have chosen 2048 octets, but these large cells would have caused an unnecessary large overhead when transmitting very small messages as an entire cell must be filled each time. I also considered very small cells like ATM in the range of 64 octets, but decided that this would cause IP to be forced to fragment unless the interconnect was connection-based like ATM. It was therefore discarded and a cell-size of 512 octets matching the BlockRAM was chosen.

Cells also gives the advantage that we know the size of the message transmitted and can thus schedule a time-slice which will always contain an entire message, no more, no less.

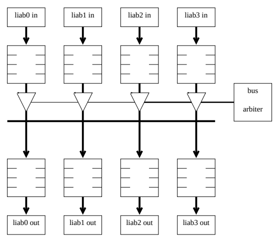


Figure 6: Bus overview.

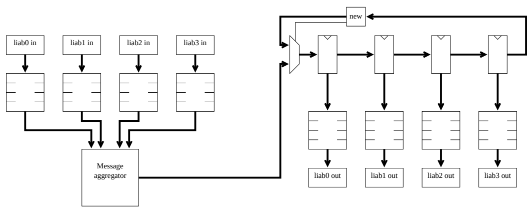


Figure 7: Ring example.

4.3 Interconnects

The interconnect in the FPGA must be quite robust and preferably simple to ease debugging and reduce the likelihood of failures in the hardware. It must also be capable of dealing with malfunctioning LiABs intending to force the FPGA to congest by outputting immense amounts of faulty cells and the interconnect should therefore have a static bandwidth allocation and/or early cell sorting.

It is also important to note that many transmissions in the fabric will be of a broadcast nature - namely the masters transmission to the packet switch, state updates and the packet switch to the master. Hence, a fabric suitable for broadcasting should be chosen as it is easier to sort away information than to obtain it.

4.3.1 Bus

The bus is a well-known interconnect with certain bandwidth-limiting properties as all connected units must content for the common medium and we therefore need some sort of MAC[2] algorithm to ensure the messages get transferred. The Ethernet originally used this topology and a CDMA/CD[3] [5], which unfortunally in our design has the lack of certainty of transmission [5] - especially if a LiAB should malfunction and output defect cells. This could be easily handled by instead using a TDM[4] MAC algorithm. This MAC algorithm is also very easy to implement.

4.3.2 Ring

The ring topology was inspired by the large amount of broadcast transmissions. By using a ring to interconnect all LiABs, they would inherently have access to all data transmitted through the interconnect. The advantage of this interconnect is the ease at which the design supports retries. It will continue to attempt to deliver the cell to a given destination until the cell is removed from the ring. This is also

[2]Medium Access Control
[3]Carrier-Detect-Multiple-Access / Collision-Detect
[4]Time-Division Multiplexing

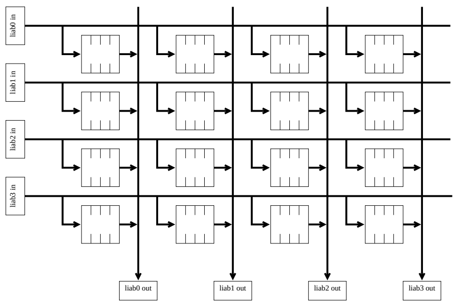


Figure 8: Crossbar overview.

the rings disadvantage - it does not preserve order of cells and a cell could be stuck in the ring unless a time-to-live mechanism is implemented.

4.3.3 Cross-bar switch

The crossbar switch is highly optimized for point-to-point transfers with a lot of distributed buffers. It has a very high throughput as described in [19], but is unsuited for our design as we need to be able to sort cells in a quarantine queue. This means that we have to have an additional set of buffers below the matrix. [20] describes an algorithm to improve the reliability of the cross-bar switch, but it is still not particularly suited for the broadcast transmissions we need.

Like the crossbar, the Kaleidoscope is optimized for point-to-point, even more than the crossbar. Where the crossbar could facilitate our need to broadcast most of the traffic, the kaleidoscope approach does not facilitate this feature easily. In our design, the kaleidoscope does not achieve some of the desirable advantages and is thus abandoned. When working with only 4 LiABs to interconnect, the kaleidoscope actually more resembles a ring. Fault-tolerant designs based on toruses, such as described in [21], are best suited for larger interconnects than our target of just 4 units and is considered too complex to gain anything from.

4.3.4 Interconnect decision

From these different approaches, we consider the bus most appropriate since it is easy to implement, easy to extend and reliable with TDM. The problem is the major SPOF - the unit responsible for sharing the bus. If kept sufficiently simple the unit should have a high MTBF.

The bus width has been selected to be 16 bits, as the dual-port Xilinx RAMB4 is available only in the finite set of sizes mentioned in table 2 taken from [13] pp. 25:

Of the mentioned available Xilinx primitives, RAMB4_S16_S16 was selected as that would provide the widest internal bus and match the writes from the LiAB. The choice of a dual-port entity was motivated by the possibility of doing clock desync in the RAM block, but as mentioned later, this turned out to be impractical (see 4.5).

4.4 Internal protocol

As we now have decided to use a TDM bus and transmit cells of 512 octets on a 16-bit bus, its time to determine the exact layout of the cells. The cell layout is partly dictated by the LiAB - we will use the term word to describe 16 bits as the LiAB will perform transfers in words. The first word (the cell header) of the

Primitive	Port A width	Port B width
RAMB4_S1	1	-
RAMB4_S1_S1	1	1
RAMB4_S1_S2	1	2
RAMB4_S1_S4	1	4
RAMB4_S1_S8	1	8
RAMB4_S1_S16	1	16
RAMB4_S2	2	-
RAMB4_S2_S2	2	1
RAMB4_S2_S4	2	4
RAMB4_S2_S8	2	8
RAMB4_S2_S16	2	16
RAMB4_S4	4	-
RAMB4_S4_S4	4	4
RAMB4_S4_S8	4	8
RAMB4_S4_S16	4	16
RAMB4_S8	8	-
RAMB4_S8_S8	8	8
RAMB4_S8_S16	8	16
RAMB4_S16	16	-
RAMB4_S16_S16	16	16

Table 2: The sizes block RAM is available in the Xilinx Virtex-E series.

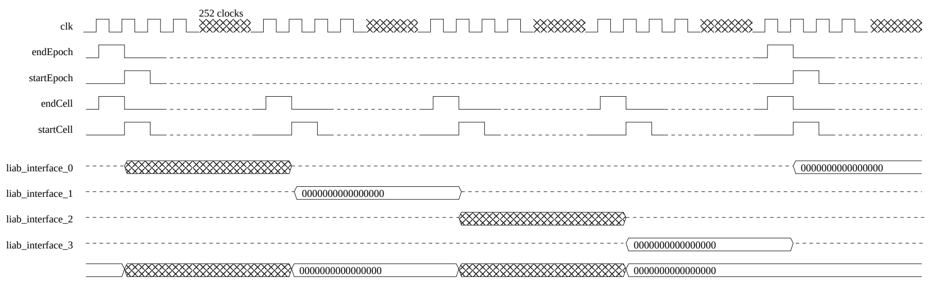


Figure 9: Internal bus timings.

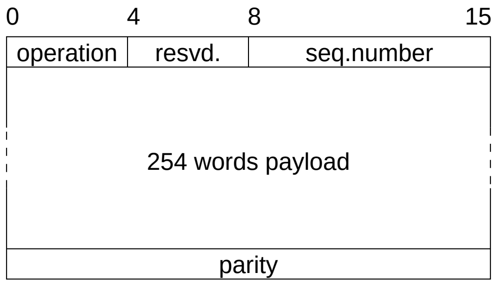


Figure 10: Layout of a transmission cell.

transfer describes the contents of the cell following and the last word is the parity of the entire cell calculated as XOR of all words initialized with "00...00". The cell header selects what to do with the cell and is split into 3 fields - command (4 bits), reserved (4 bits) and sequence number (8 bits). The layout of the field is shown in figure 10 and a sketch of the timing on the bus is show in figure 9.

Bits	Mnemonic	Sender	Description
0000	IDLE	A	To ease implementation, we reserve a cell header of pure zeroes to represent an empty cell.
0001	HEARTBEAT	A	To signal a LiAB is still alive, it should use its bus assignment to transmit a HEARTBEAT message - this is a message, that contains no contents, but signals that the LiAB is still alive and ready to act as master.

As seen, a cell header with all zeroes constitutes an empty cell. This was important to know as this is how the internals distinguish an idle bus from a working bus - if the first word transferred from any unit is a zero, the unit has nothing to say in this bus epoch. Originally, the `CellQueue` would start to transfer whenever its `do_get` signal was asserted and the queue contained an entire cell, even if it was out-of-sync with the bus - it would just get in-sync in the next epoch, where it would restart transmission. Due to a new architecture internally in the `CellQueue`, this was behavior was eliminated.

Bits	Mnemonic	Sender	Description
0010	FROM_PS	P	The packet switch directed this message at the control plane.
0011	TO_PS	M	This cell must be forwarded in the hierarchy after its quarantine time is successfully completed.

These two messages relates to the communication with the outside of the cluster. If a FROM_PS is transmitted on the internal bus, it is passed to all LiAB, so they may figure out what the proper response to this message is. TO_PS is put in quarantine until a time-out period is elapsed. Only the LiAB emulating the packet-switch is allowed to send FROM_PS, likewise for TO_PS - only the currently elected master may send such a message. These messages should be sorted in the incoming queue.

Bits	Mnemonic	Sender	Description
0100	DROP	S	Sent by a non-master to indicate that the cell with the sequence number of the current cell should be discarded from the bridge. If all non-masters agree on the cell should be dropped, the bridge may not forward the cell. When 4 cells have been dropped, a new master must be elected.
0101	ACCEPT	S	Sent by a non-master to indicate that the cell with the sequence number of the current cell is accepted.

These two messages are used by LiABs in order to stop or accept a TO_PS message. If two units agree to drop message, the message is removed from the quarantine queue before being passed on to the Packet Switch. If one (other than the sender) sends an ACCEPT with an appropriate serial number, it is passed on. These messages may be send from any non-PS LiAB, hence a master may actually DROP its own cell, thus indicating to the FPGA that just one other LiAB needs to drop this cell.

Bits	Mnemonic	Sender	Description
1000	GET_STATE	S	A new/rebooted LiAB has joined the cluster and needs a copy of the masters current state. The master must intercept this message and schedule a flow of STATE_DUMP messages.
1001	STATE_DUMP	M	This cell contains data representing the current state of the master in the cluster. Higher-layer protocols are needed to ensure the state is correctly transferred.
1010	STATE_UPD	M	Whenever the master changes something in its state, it should send a cell with this content type. If a non-master sends this message, it should be reset as it is not entitled to make state updates. The slaves have to decide if this update makes sense.

The messages in this table relates to maintaining the clusters state.

Bits	Mnemonic	Sender	Description
1111	SOFT_HOVER	M	It is an request to the FPGA to elect a new master in the cluster and flag this LiAB as failed.

And finally, we need a method to do an intentional hand-over if the LiAB knows it will time-out we might as well signal this and force an election of a new master.

The clever reader might have noticed that no sender IDs are ever used. As the bus is assigned to a LiAB, the units needing a sender can look at the current bus-assignment and thereby determine the sender. Likewise, there is no explicit destination as the message is either directed at the PS, in which case everybody needs a copy, from the PS (everybody needs a copy) or something only relevant for designated units in the FPGA. In any case, the destination can be derived from the cell-header, so there was no need to waste 4 bits (yes, I know, we do that anyways to reserved storage). If it was included in the cell-header, a LiAB might be able to fake the source address and lead the FPGA to fail - eg. an evil (faulty) LiAB might decide to send accept cells with fake sources and thus make its cells pass immediately. This will not happen when the source of a cell is dictated by a bus owner-signal. The LiAB would also need some method to determine their source-address, which would further increase complexity without gain.

4.5 The bus-based VHDL design

VHDL [16] and Verilog are the two languages supported by ModelSim, the available hardware simulator. VHDL is preferred as VHDL has certain desirable features for good design practices, eg. types. Verilog does not have the same type-mechanism and thus makes it easier to make errors regarding signal meanings. The Verilog syntax is more like C, short and effective, unlike VHDLs approach, which contains

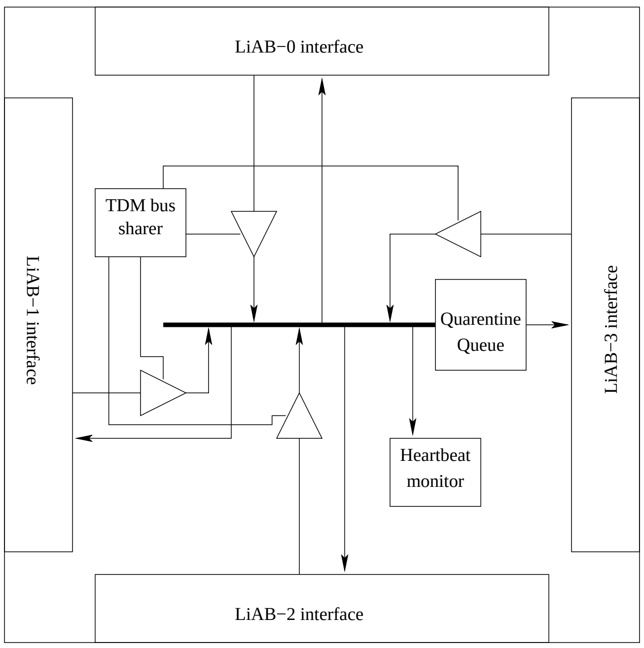


Figure 11: Top entity in the VHDL design - the Reliplan.vhdl file.

a large number of keywords. Both languages are fully supported by the available synthesis-tools.

As seen on figure 11, a bus was chosen for the final design. In this section, we will dvelve into the details of the different modules, namely `QuarentineQueue`, `LiAB_Interface`, `HeartbeatMonitor` and `BusSharer` to look at how they finally was implemented and details regarding progress towards the final implementation.

Regarding Mentor's two products for synthesis I had available; Mentor LeonardoSpectrum and Mentor Precision RTL. Precision RTL has problems using the tri-state drivers used extensively throughout the design to ensure an almost constant cost with a larger number of cells in the queues. As far as I could deduce, Precision RTL tried to implement my busses using muxes which becomes very expensive in latency when using more than 4 cells in a queue. This reduced the possible speed of the design implemented in Precision RTL to significantly less than the desired 100MHz (closer to 45MHz with 8 cells in the queues). LeonardoSpectrum did not have these quirks and was therefore chosen for implementing the design. Unfortunally, LeonardoSpectrum does not run correctly under Linux/WINE.

I originally desired to do a multi clock-domain design with clock boundaries through the `LiAB_Interface` using the possibility of feeding two independent clocks to each BlockRAM. This idea was discarded as I realized that LeonardoSpectrum would always assign one of the global clock signals to any signal used as a clock or latch enable. As I intended to use 5 independent clocks and 4 latch enable (GPALE) signals, I soon realized that this multi clock-domain design was unfeasible and the idea was therefore discarded (the original design for my LiAB-to-FPGA interface can be seen in figure 18).

4.5.1 BusSharer

As shown in figure 11 we use a TDM bus. This bus needs a manager, and the easiest way to make one of these is to use a one-hot ring. The disadvantage of using a naive

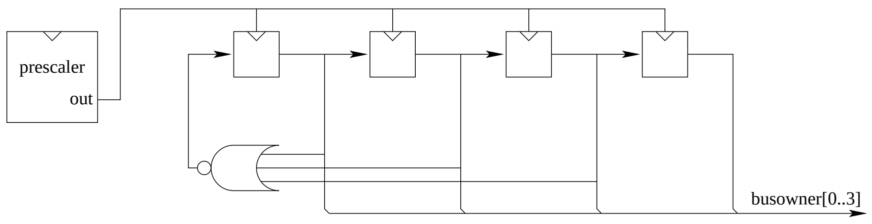


Figure 12: The DFF-chain design for the BusSharer.

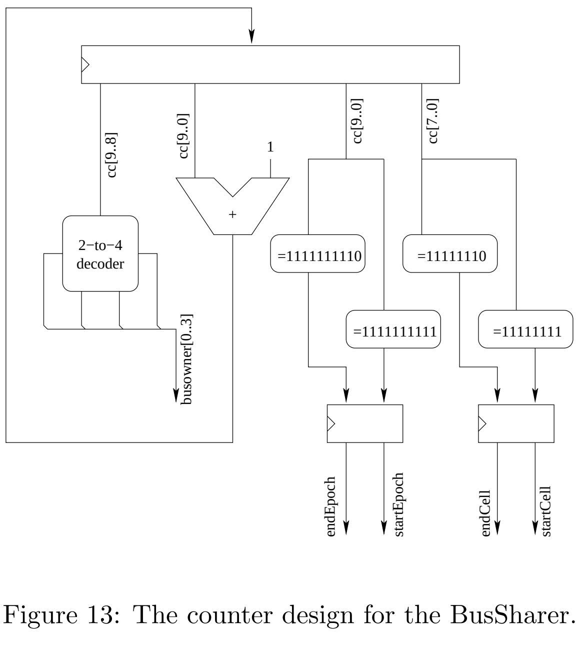


Figure 13: The counter design for the BusSharer.

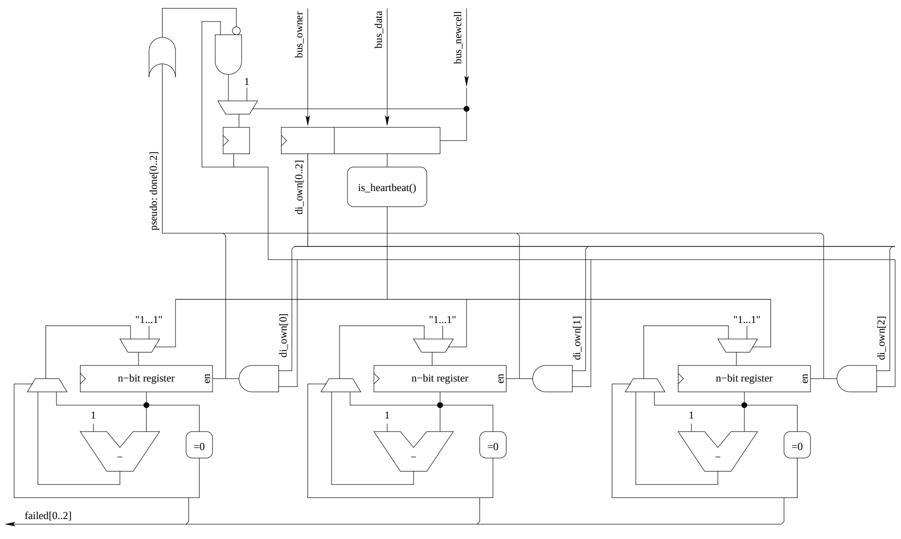


Figure 14: The HeartbeatMonitors count-to-zero.

ring is in case the hot bit disappears, the ring would no longer work. Therefore, a chain of flip-flops with a clock prescaler in front, as shown in figure 12, was chosen for the initial design. This design would ensure that two-hot would not happen unless storage in the DFFs was contaminated. This could additionally be assured not to happen by adding AND-gates on the outputs keeping output low if it detects more than its own input high. This design is easier to extend to triple-redundant operation as described in [17].

The ring design had problems being implemented in LeonardoSpectrum due to the clock being output from the prescaler (and I somehow never thought of enable-signals) and the begin-epoch and end-epoch signals needed to synchronize the bus. I therefore ended up using a far simpler design (in terms of code, not area) shown in 13. The major advantage of this design is that LeonardoSpectrum uses a primitive to implement the adder-register part of the design.

4.5.2 HeartbeatMonitor

The HeartbeatMonitor consists of three count-to-zero registers shown in figure 14. If we receive a Heartbeat message from a LiAB, the appropriate register is set to ones. If the counter reaches 0, the LiAB related to this register is assumed crashed and if it was master, a new master is elected amongst the LiABs having reported positively in. This is done as shown in the state diagram in figure 15. The timeout value is measured in epochs and is given in a number of bits during synthesis as a generic supplied to the HeartbeatMonitor. This is to simplify the implementation.

The width of the timeout registers is set to 16 bits equalling 65536 epochs, which is $256 \times 4 \times 65536/100 \times 10^6 s^{-1} = 671ms$. This value was selected to ensure a rapid timeout but not to force the LiAB to spend too much time sending Heartbeat messages to the FPGA.

Important is it to consider the time to transfer a single cell from the LiAB to the FPGA - this is assumed to be a high-priority task for the LiAB and thus not interrupted. In that case, the transfer should take 256 cycles of a word-transfer to the FPGA over the GPB (General Purpose Bus) and an additional 256 reads from memory for the SC520. A single word transfer over the GPB takes n cycles to complete as seen on figure 16. We assume that the delay, which is customizable in

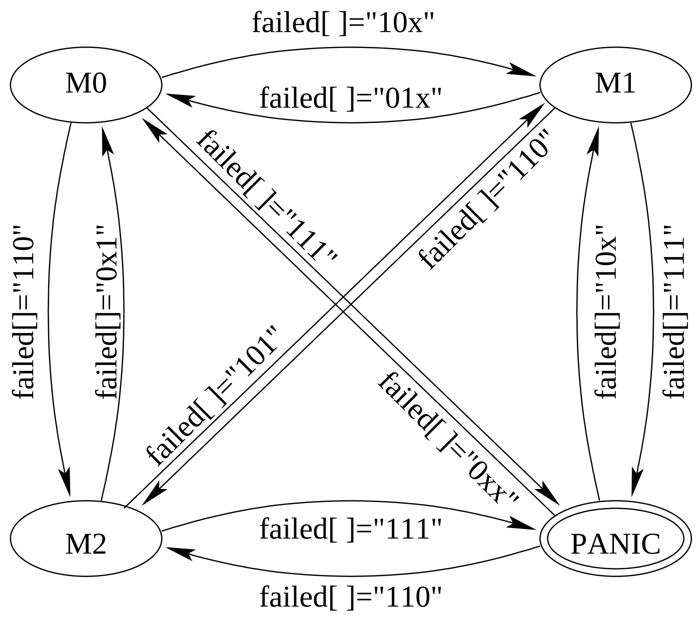


Figure 15: Election of new master

the SC520, is set to 0 for these burst moves. As mentioned previously, we assume that the GPB in the transfer period is only used by our transfer (could be ensured by a `rep outsw`). To sum up the election process shown in figure 15, the heartbeat monitor priorities LiABs in the order 0,1 and 2 when leaving PANIC. The heartbeat monitor starts out by assuming all attached LiABs have failed and only selects a new master whenever the current master fails.

4.5.3 DataFeed and LiAB

I find it important to point out these two files as they are the data-sources used in testing the design. `DataFeed` is a simple VHDL entity to read data from a file into the design. `DataFeed` supports only a single test vector and two different commands. The format of the input is simple - the first character in each line describes what the line does - `p` for pause a number of clock-ticks, `d` for data. Data must have same number of bits as the `std_logic_vector` used for the port.

Unlike `DataFeed`, `LiAB.vhdl` tries to emulate the signals from the LiAB with proper timings. This was used to test the function of `LiAB_Interface`. In order to accurately emulate a LiAB's signals, the `LiAB` entity has additional commands compared to `DataFeed`, but the concept is the same - first character of line determines the command. `LiAB` supports 'p','i','r' and 'w' commands.

In case of 'r' and 'w' (read and write), two addressing vectors must be supplied, one containing the 3 `CS0-CS2` chip select bits and one for the 10 bit `A9-A0` address. Depending on the command, an additional data vector of 16 bits may be appended. In figure 16, a reduced version of the diagram shown pp. 70 in [23] is shown. The diagram shows what I consider important in relation to interfacing with the LiAB and what the interfaces has been geared towards.

'p' functions like `DataFeed`, it waits for a number of clock periods. 'i' is important in simulating a real LiAB - it waits until the interrupt number supplied after 'i' occurs, eg. 'i 2' is expanded to "wait for interrupt 2".

Of course it should be noted that neither of these entities are synthesiable and

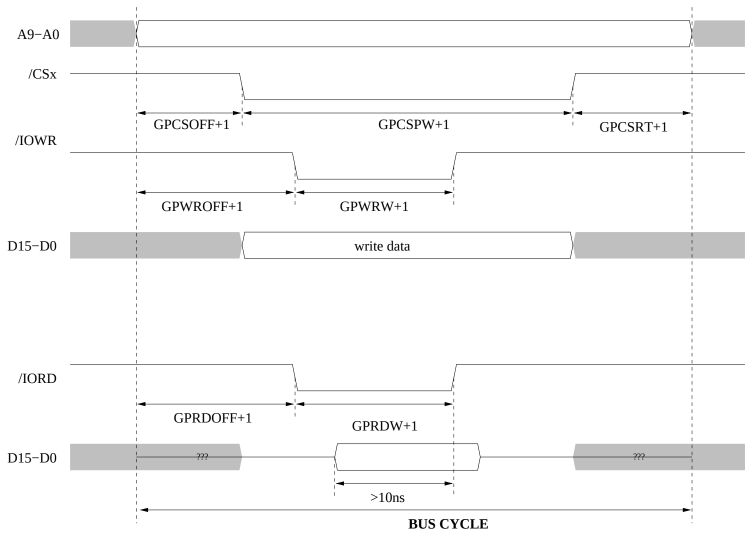


Figure 16: Reduced timing diagram from AMD-SC520 Data sheet [23] pp. 70.

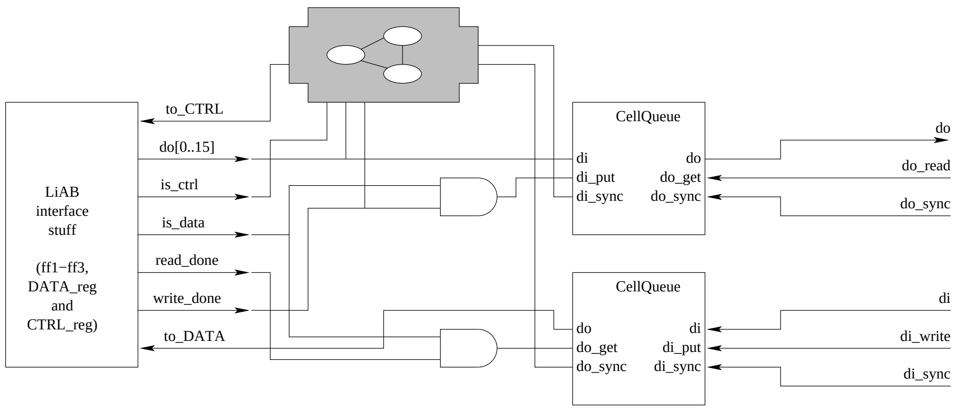


Figure 17: Conceptual overview of the LiAB_Interface.

are only used for testing purposes.

In order to easily produce test vectors for the `LiAB_Interface`, a small C program was developed capable of outputting data in the format supported by `LiAB.vhdl`. This was especially important as an entire cell needs to be transferred to the FPGA. That would mean 512 writes plus the additional control, quite a daunting task. This program can be found in appendix.

4.5.4 LiAB_Interface

The original design for LiAB_Interface (figure 18) included DFF clocked by `/gpiord` and `/gpiowr`. This idea stems from the observation that data is stable around the rising edge of `/gpiord` and `/gpiowr`, and the signals could thus be used to trigger the DFFs. Unfortunally this designs could not be synthezised as LeonardoSpectrum insisted on assigning global clock buffers to `/gpiord`, `/gpiowr` and `gpale` and would not allow me to LOCate the signals to the pins that the PCB was using. Therefore this design was abandoned and the design shown in figure 17 and figure 19 was used.

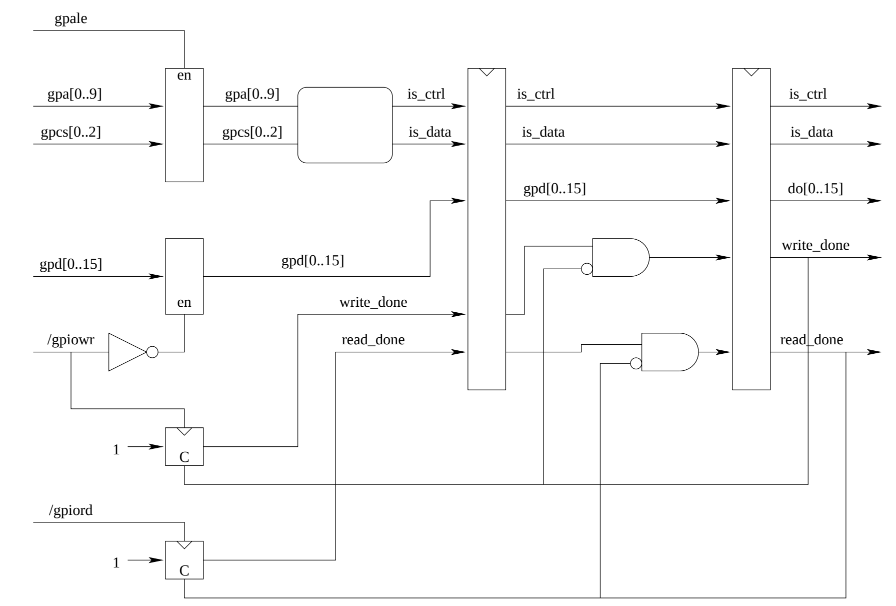


Figure 18: The original LiAB_Interface input stage. Uses gpiord and gpiowr as clocks.

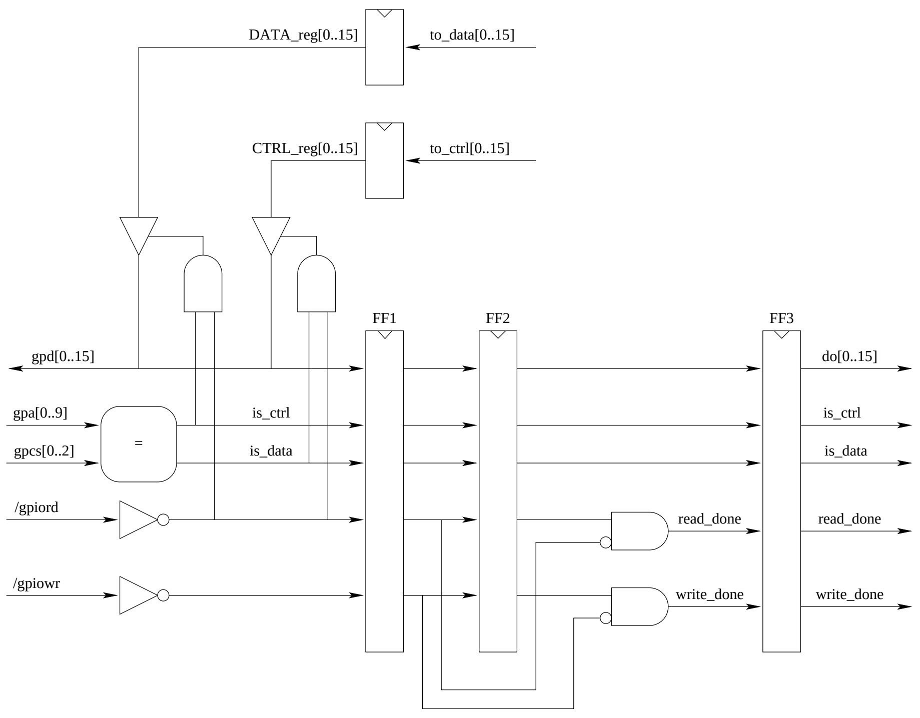


Figure 19: The new input stage clocked only by global clk.

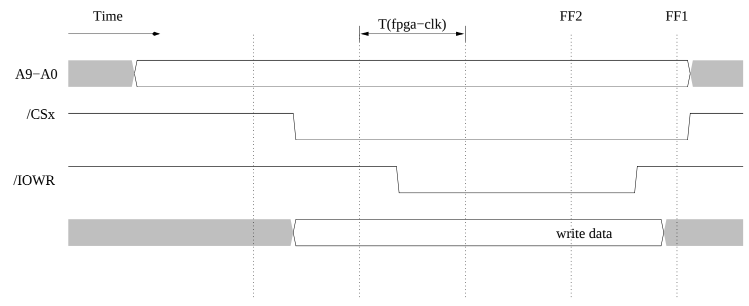


Figure 20: Internal clk in relation to bus signals.

These designs assume that `clk` exceeds 33 MHz as at least one clock edge must fall in the low period of `/gpiowr` or `/gpiord`. It can be seen from figure 19 and figure 20 is that the entry stage detects the rising edge of `/gpiowr` and passes the event on in the `LiAB_Interface`. This should pose no problems as the FPGA is clocked at 100 MHz. Meta stability is more of an issue in this design, but should again pose no problems as $T_{clk} - 2 \times T_{LUT}$ are available for stabilizing signals before entering the inner part of the `LiAB_Interface`.

By always being ahead of the output, we fulfill the 10ns requirement - we prepare what will be read next time in a register ready to drive the bus, hence the delay is only from `/gpiord` to tri-state drivers begin driving the `gpd` pins.

In the `LiAB_Interface`, a small FSM is present. The purpose of this FSM is to handle various communication details such as "this is the first word of a cell", "please drop all cells in my incoming queue". The commands available to this FSM is described in greater detail in section 5.

4.5.5 CellQueue

The `LiAB_Interface` uses `CellQueue`s to store the cells as they arrive word-wise from the LiAB. The `CellQueue`s is a FIFO suited for queueing cells destined for the internal bus or the LiAB. It was originally capable of running with two distinct clocks, as they used the `RAM4B_S16_S16` entities build-in clock boundary, but was later redesigned to use only one global clock. The `CellQueue` entity is capable of burst input, burst output, single-word input and single-word output. The `do_get` and `di_put` inputs on the entity determines the function - if `di_put` is high on rising clock, the word on `di` will be stored on the next free space in the currently filling cell. There are no wait states associated with input - when a cell has been filled and checked, the queue automaticly moves to the next cell and begins filling. Therefore it is possible to continuously input data to a `CellQueue`. Output is a little trickier as the pipeline must be filled before starting output (this was a result of the large routing delays in the FPGA). The `CellQueue` is capable of outputting an entire cell in a burst (one word on each clock cycle), but needs 4 ticks to prepare for the output. This is required to get all registers in the output stage of the queue prepared and to read data ahead in the BRAM. The input side of the BRAM when reading $[n]$ at the output of `CellQueue` is actually $n+2$ and the output of BRAM is $[n+1]$. Thus when it stalls (by leaving `do_get` low), the `CellQueue` needs to capture the output of the BRAM in a temporary register inside the entity as output from BRAM will be $[n+2]$ in the next clock cycle. Thus when leaving `do_get` low, we store in a temporary register and mark that this register is the next output value by adding a FSM inside `CellQueue`. The state machine has the states depicted in figure 22. The large amount of states is a hang-over from debugging the design - I had severe

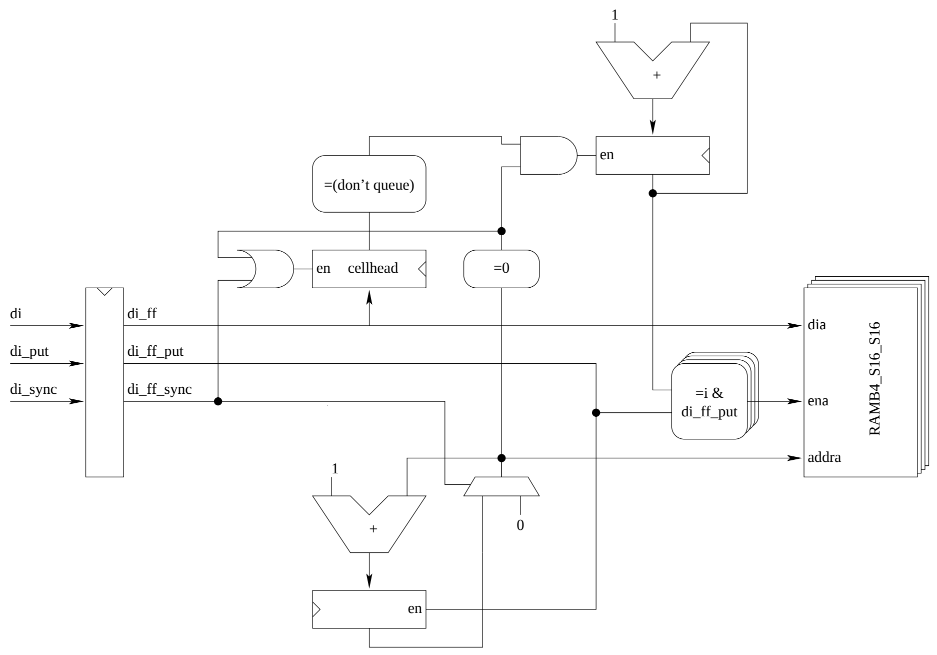


Figure 21: Input side of cell queue.

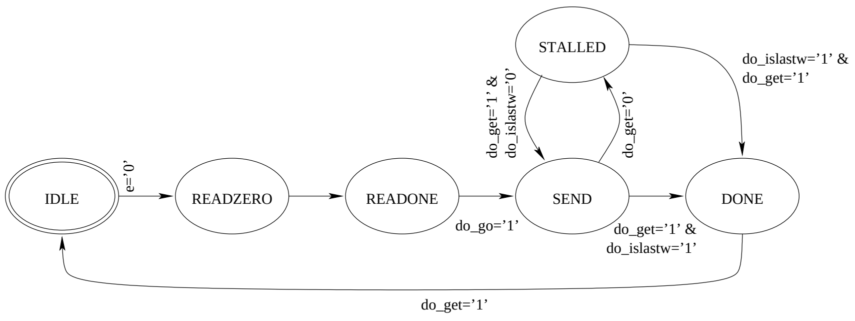


Figure 22: State diagram for the sending side of CellQueue.

problems making the right output at the right time, so a lot of wait-states where added. These remain as the current FSM works and we have, in Reliplan, more than 900 cycles to get ready to transmit due to the TDM nature of the bus.

4.5.6 QuarQueue

To the internal bus, the `QuarQueue` is attached. This queue is the heart of the system as it does all the work. The quarantine queue consists of three distinct functions (as seen on figure 24) - receive cells (RCV), process cells (PRC) and send cells (SND) and a storage for the state of the cell. `QuarQueue` is a reordering queue - it is not guaranteed to output cells in the same order as they arrived.

For each block of memory used to store a cell state, a register is assigned. Each cell must progress through the states depicted in figure 25. Along with the transition between states shown in the figure, an acronym relating to the processing unit responsible for the transition is shown. It is important to note that from any given state, only one processing unit is allowed to move the cell away. This ensures

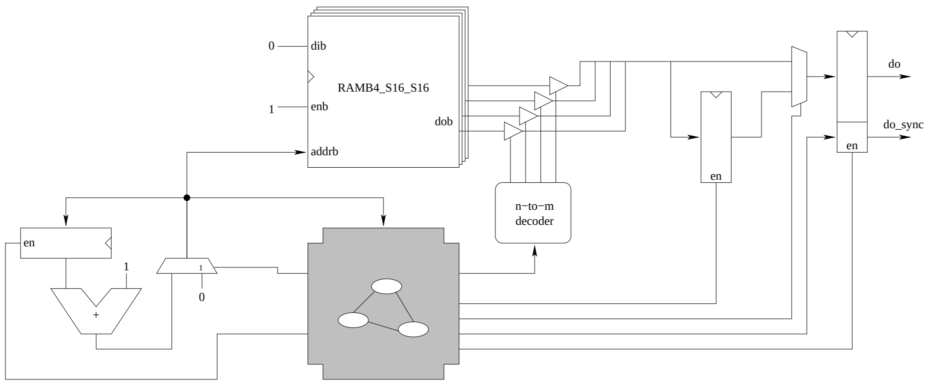


Figure 23: Output side of cell queue.

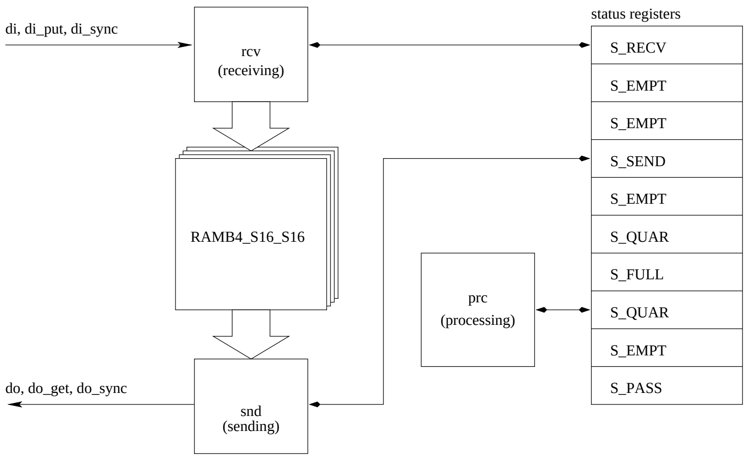


Figure 24: Overview of the quarantine queue.

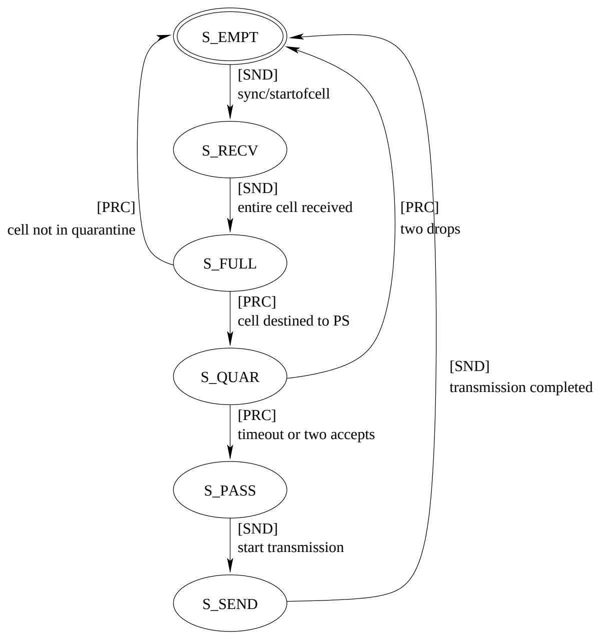


Figure 25: States a cell must progress through and the system components that may move the cells in states.

us that no conflicts will arise.

The cell state registers are implemented as triple-port memory as all three devices needs read-/write access to the cell states. The design of the cell status registers are shown on figure 26). As seen on figure 26, the cell states uses busses for output. Originally, I access the memory with an enable-signal for the cells. This has the unfortunate effect that it is possible for two registers to attempt to drive the bus at the same time. By accessing with a read-position and decoding this value instead, this should be ensured never happen.

Using a bus for output had an unfortunate side effect - it is not immediately possible to tri-state an enumeration and I therefore had to resolve the enum into a `std_logic_vector` as shown in `QuarQueue.vhdl(80)`.

Using a bus for the input to the state register would yield an unfortunate side-effect; if two units where to attempt a write in the same cstat-cell, the result would be undefined as the bus has two drivers. This could be fixed by ensuring in a piece of logic that no wf would ever be asserted if a write conflict exists. This solution has yet to be implemented as normal function of the `QuarQueue` should ensure that this would never happen.

The first unit to process a cell is the RCV unit. It has the contents shown in figure 27 and 28. Two separate functions must be performed in the RCV - scanning for a free cell to fill with cell data and receiving the actual data. The reason for not making a simple state machine here is the need to be able to receive a continuous feed of data from the bus and we must thus be ready with the number of a free cell shortly (worst case is to scan entire buffer for new free cell) after having had the last free cell allocated for a receive buffer. The scanning is performed as a loop around the buffer and takes thus at most n cycles, where n is the number of cells in the quarantine queue, to complete. If no free cell was found, RCV will continue to fill the same block as it already have used - hence it drops the most recently received cell. This feature makes the quarantine queue subject to congestion, which

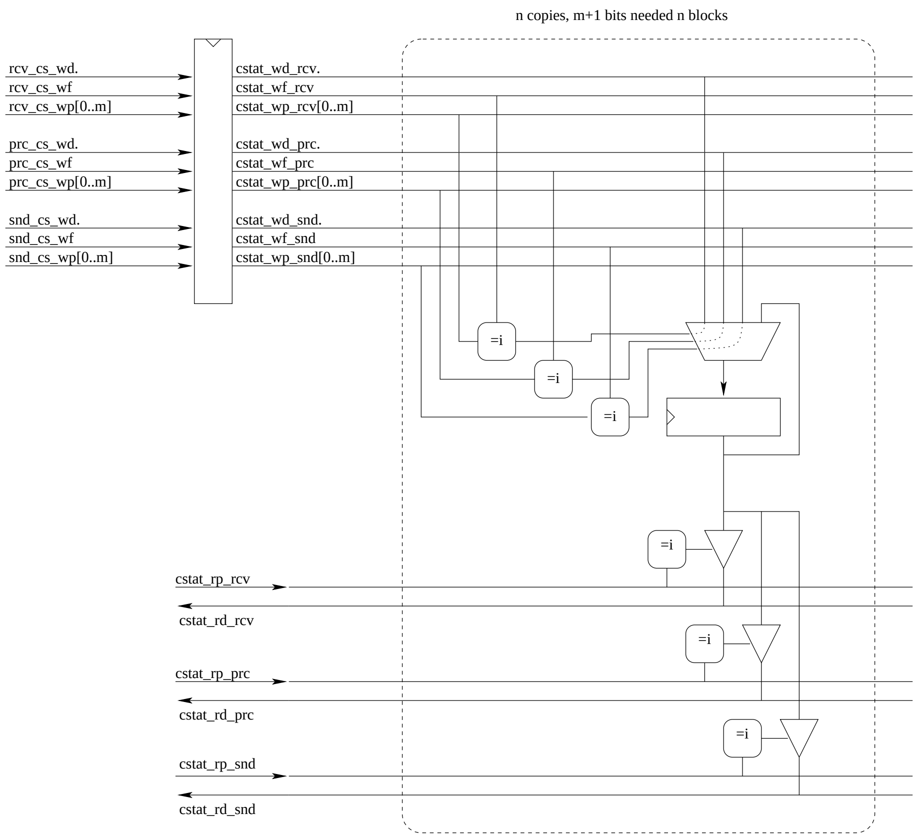


Figure 26: Status register implementation.

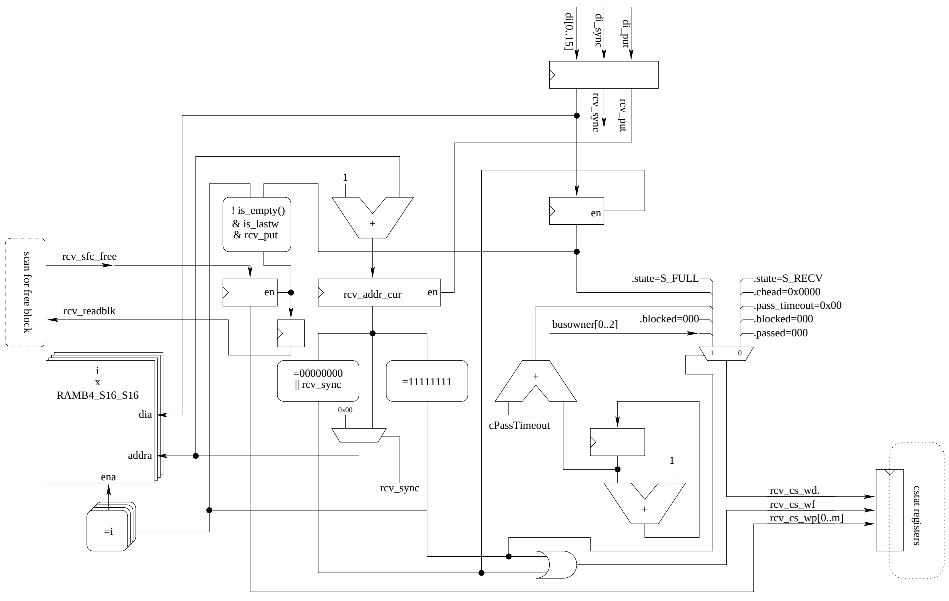


Figure 27: Receiving the cell.

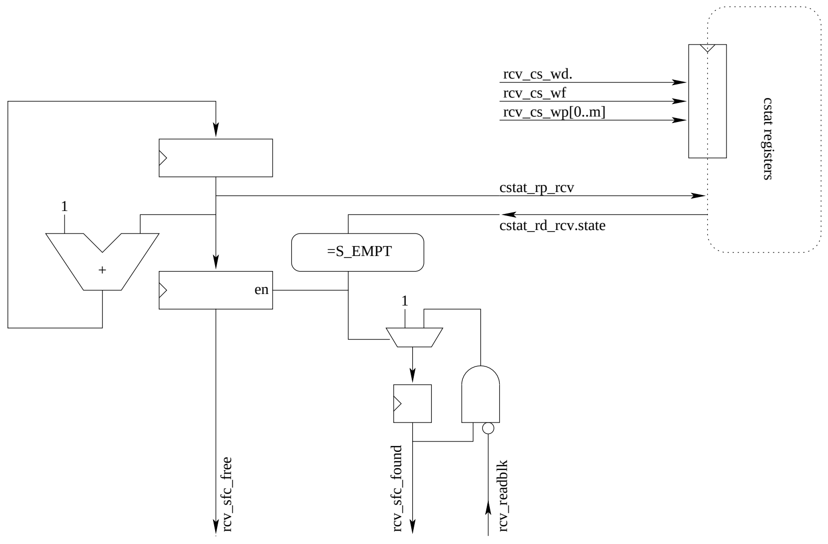


Figure 28: Scanning for a free cell in the quarantine queue.

could be avoided by having counters that count how many cells have been put in quarantine from each LiAB. By counting these, it could be assured that no LiAB could ever congest the queue.

When updating the cell state register when starting receive, RCV notes who is the current bus owner in the `.accept` field. This is used by PRC to update other cell states if the cell received was either an accept or a drop message.

Once RCV is done receiving the cell and the cell has passed initial sorting (not empty), the cell has reached S_FULL. It is then picked up by PRC, which either empties the cell or moves it to S_QUAR (quarantined state) depending on the cells content. If the cell contains a drop or an accept message, the entire queue is searched for cells whose sequence number matches the number in the cell. In case the cells sequence number matches, the appropriate field is updated with the new drop/accept mask. Due to the pipelining of the cstat-registers along with the PRC pipeline, the state machine has been padded with appropriate wait states to allow everything to be updated.

If the cell state is S_QUAR, the PRC will also pick up the cell to check if it has exceeded its timeout value (equal - important later) or if two LiABs has accepted the cell (it is assumed that the one sending the cell always agrees with its content so one has already accepted the cell) it is moved to the S_PASS state to be picked up by SND. If the cell was dropped by two LiABs, the cell is discarded. In this case, the HeartbeatMonitor should be notified but will not be in the current design. The PRC part has been implemented as a state-machine controlling the pipeline for updating. As the diagram containing all connections would be too crowded, the reader is encouraged to read the VHDL source with figure 29 at hand.

The purpose of SND is to send cells that have reached S_PASS. Because no continuous operation is needed of SND (it is not required to be able to continue output if different blocks), it can be implemented as a simple state machine. It moves through the states shown in 31, but unlike `CellQueue`, the `QuarQueue` will only be attached to another cell queue, that supports continuous input, so it needs not such a complex handling of possible waits (ie. get going low during transfer) - it will always run the entire transmission in one fast transfer. The diagram for SND is shown in figure 32. It is important to note that a `QuarQueue`'s output

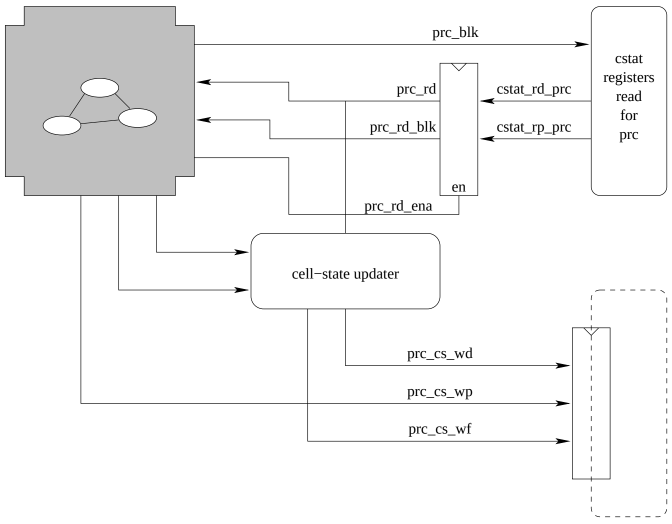


Figure 29: The processing cells pipeline.

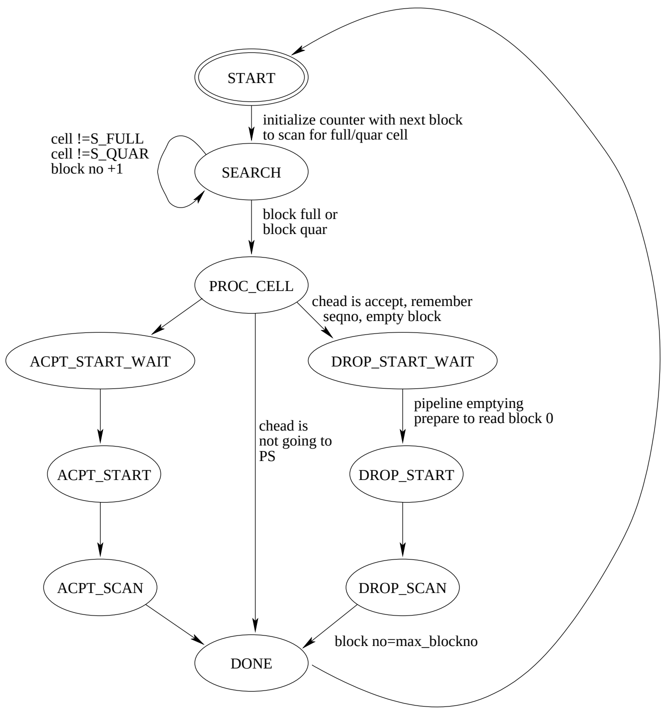


Figure 30: States for the state machine shown in figure 29.

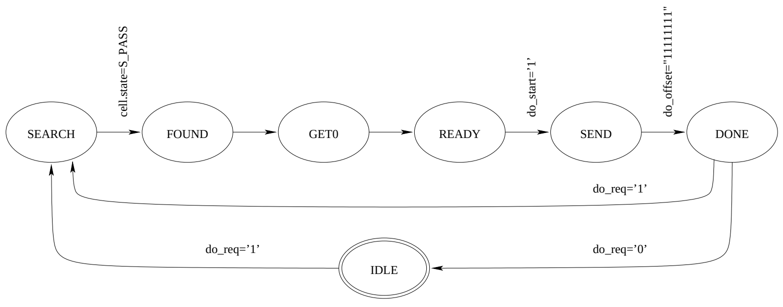


Figure 31: A state diagram for the different states that SND passes through

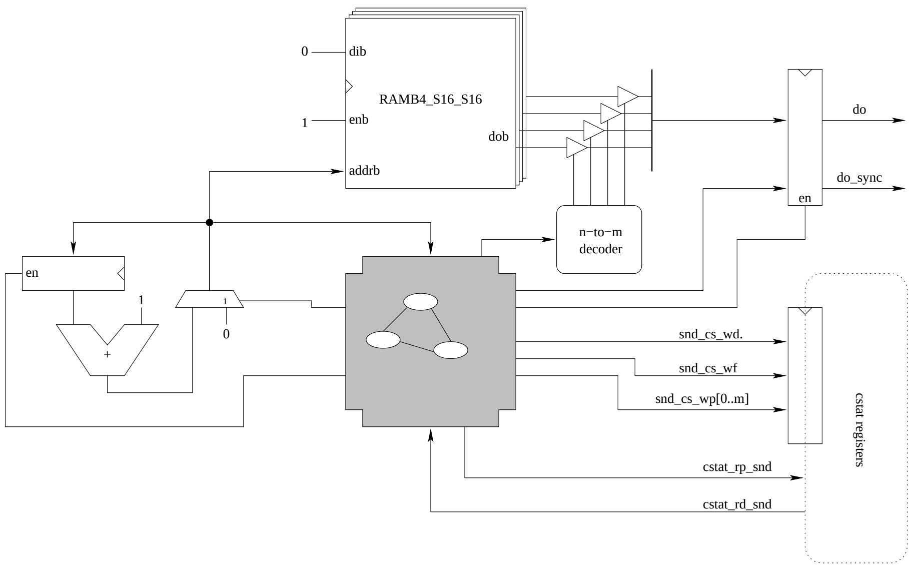


Figure 32: SND part of QuarQueue.

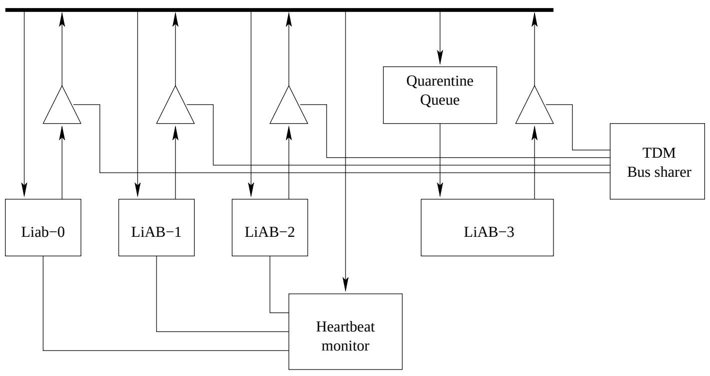


Figure 33: The ReliPlan entity.

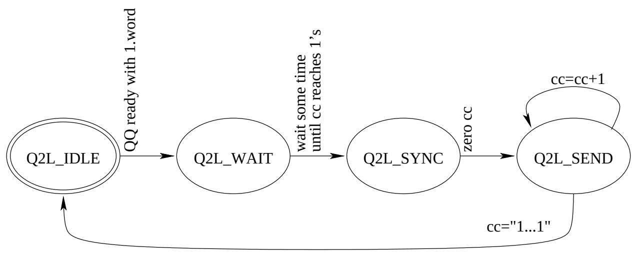


Figure 34: The small state machine used to pass cells from QQ to LI.

can only be attached to something that supports the burst transfer - eg. it is not possible to use it directly towards the LiAB as the LiAB only slowly (compared to internal clock) consumes words. For this reason, the `QuarQueue` is attached to a `LiAB_Interface`-entity.

What is worth noting is that when working with QuarQueue, we have a request signal that is used to ask the `QuarQueue` to start searching for a cell to transmit. When the cell is ready `do_sync` is asserted, hence `do_sync` works as an acknowledge-signal telling the outside that the `QuarQueue` is ready to transmit. `do_get` must be kept high during the entire cell transmission as `QuarQueue`, unlike `CellQueue` does not have the correct handling of a stalled transmission. If someone should happen to deassert `do_get` during transmission, the word following the current, will be lost. This was, as mentioned in 4.5.5, worked around by using an extra register to capture BRAM output during stalling.

4.5.7 Reliplan

We now have the parts needed to build the entire clustering hardware. The parts are put together as depicted in figure 33.

Between the QuarQueue and LiAB-3's input, a small FSM was added in order to handle the proper signalling to the queues. This FSM has the state diagram shown in figure 34 and thus consists mostly of a counter.

4.6 Tool flow

As I spend a significant amount of time figuring out the flows used when using LeonardoSpectrum with Xilinx ISE, I will recapitulate the major points of the

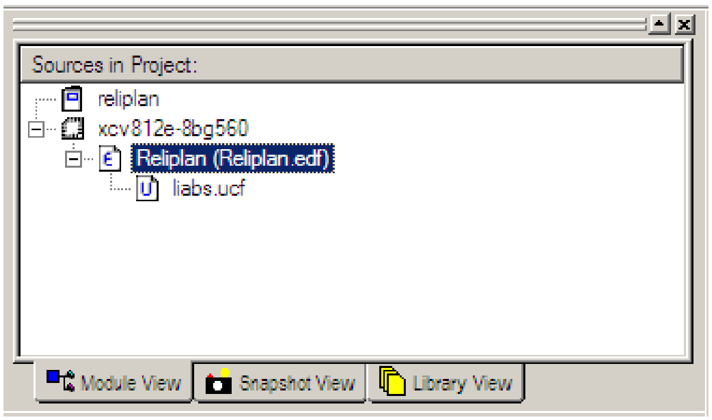


Figure 35: The project tree in Xilinx ISE.

flows here to save future developers time.

LeonardoSpectrum takes the VHDL/Verilog files as input. The listing of files in LS must be sorted accordingly to the use in the design. The output must be written to an EDIF (.edf) file used as input to Xilinx ISE.

The pin mappings are available as both a CSV and an UCF file. The UCF file must be used together with the EDIF file generated by LeonardoSpectrum in Xilinx ISE's Place-and-Route tool (PaR). The new project in Xilinx ISE should consist only of a device (XCV812E-BG560) with an EDIF file, which has an UCF assigned. The project navigator tree is shown in figure 35.

Regarding coding style: Remember to initialize all registers on reset, otherwise LS will complain but not fail. An additional condition may never be applied to `if rising_edge(clk) then`.

4.7 Synthesis

Having designed the needed parts and figured the tool flows out, it is time for some synthesis. For this purpose, we use, for the aforementioned reasons (see 4.5), LeonardoSpectrum coupled with the Xilinx ISE.

4.7.1 Queue sizes

The size of both `CellQueue` and `QuarQueue` are determined by a generic which indicates how many cells the queue must contain. Only powers of 2 are supported, since the wrap-around in adders are implicitly used to limit indexing. For the size of the indexing towards BlockRAMs, a special function in `types.vhdl` is declared - `bits_needed()` - which intentionally returns `positive'high` whenever the number given to it does not match a power of two in the range 2-4096. This should make any synthesis-tool barf over the design (not assured if the FPGA happens to contain significantly more than 4GiBits and the implementation dictates a positive is 32 bits). Otherwise, it returns the bits needed to represent a number with these different values (ie. $ceil(log_2(vals))$).

I expect both `CellQueue` and `QuarQueue` to scale well with the number of cells in the queues. The size of the largest adders in the queues remains 8 bits as offsets into the BRAMs are in the range 0 to 255 and queues would remain below 64 in size. The XCV812E contains 280 blocks of BRAM, so we have plenty of space for queues.

The minimum size of a queue is of course 2 as at least one is needed for filling and one for emptying. Unlike proper VHDL, this is not declared as a "positive range 2 to 280" as it should have been but instead as a "positive".

The default sizes of the queues are respectively 4 and 32 for `CellQueue` and `QuarQueue` - the former probably to little and the later too large, 16 would be a more sensible size. As these values are declared as generics, they can easily be overridden in each instantiation of the entity (in `LiAB_Interface`, they are used in sizes 8 and 16 and in `Reliplan`, `QuarQueue` is used in size 16).

In `LiAB_Interface`, the input from the LiAB arrives so slowly, that the queue in most cases will be emptied before the next cell is received from the LiAB, even if the LiAB writes at full possible speed. The outbound queue in `LiAB_Interface` is more critical. The LiAB is slow (compared to the FPGA) to read data from the queue and the queue therefore needs more room to handle bursts of cells - therefore a queue size of 16-32 on the outbound queue would be more appropriate. Experimentation with synthesizing the design has proved that 16 is easiest to make timing converge.

4.7.2 Timing

As mentioned in section 3, the board was equipped with a 100 MHz crystal, and I therefore insisted that the design should run at 100 MHz. This proved to be a major obstacle as the time needed to route signals around the FPGA was much larger than expected. That meant that the original design where many of the registers shown in the previous sections did not exist turned out to be far too slow. The naive approach to the design reached a f_{clk} of only 50 MHz, by adding entry flip-flops on the `LiAB_Interface`'s, the frequency was boosted significantly towards the goal and reached some 80 MHz. From the timing reports returned from the synthesis tool (such as those depicted in appendix D) and place-and-route, critical path information was extracted and the critical path shortened by reformulating certain statements, eg.

```
do_move_cell <= '1' when do_islastw='1' and e='0' or
                         flush_queue='1' else
                '0';
```

Somehow is synthesized to be slower than

```
do_move_cell <= do_islastw and ((not e) or flush_queue);
```

It is important to note that the timing report returned from LeonardoSpectrum does not include correct wire-delay information, so only the `.twr` timing report from Xilinx should be used.

Fan out also proved to be worth playing around with as the design was slower with an unlimited fanout. This was what lead me to try fiddling around with moving some components in order to reduce fan out. Small increases in speed came from this, but I still needed some 15 MHz to reach the timing goal. This forced me to add additional registers on the output from the queues, though this was an undesired choice as this forced me to handle the quite intricate pipelining issues in the queues - such as the run-away tendency in `CellQueue`, which was an issue when working towards the LiAB, where the normal function requires `do_get` on the `CellQueue` to go low while we wait for the next read from the LiAB. This is also the reason for not implementing the small work-around in `QuarQueue`.

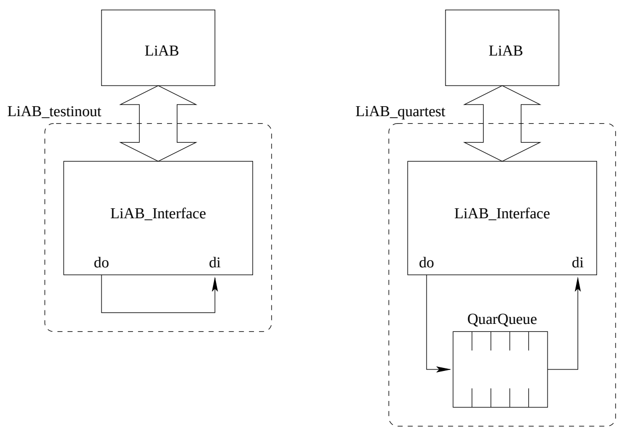


Figure 36: The two simple testbenches.

4.7.3 Testing the synthezised design

For testing the design, two small setups was made - `LiAB_testinout.vhdl` and `LiAB_quartest.vhdl`, which connects a single LiAB (LiAB-0 pins) to the FPGA.

The first test connects a `LiAB_Interface` to itself and is thus a simple loop-back device with the built-in sorting embedded in the `LiAB_Interface`. The internal bus is exported on HP_CONNECTOR_2 and can be monitored. The output is delayed a single cycle as it would otherwise be the critical path of the design and would limit the clock speed reported from Xilinx PaR tool. From the original high-level model, we see that we expect an interrupt 2 shortly after completing the transfer of a cell to the LiAB_Interface. We then read what we recently wrote and can thereby test if any distortion occurred during transfer. The `CellQueue`s embedded in the `LiAB_Interface` performs some sorting of cells arriving on the internal bus and we hence expect certain cells to be dropped. If we were to send a HEARTBEAT message, the message should not appear on the output of the FPGA, likewise with ACCEPT and DROP as these messages are sorted away by the `LiAB_Interface`.

The `LiAB_quartest` connects a `LiAB_Interface` to a `QuarQueue` and again to the `LiAB_Interface`. In this simple design we also have a loop-back functionality like `LiAB_testinout`, but we also have the delay-effect and cell sorting properties embedded in the `QuarQueue`. Basically, the only cells to pass through this design should be the TO_PS messages. This test was implemented using a `QuarQueue` with 16 cells and reached 100.020 MHz after PaR. Reducing the number of cells in the quarantine queue increases the possible speed a bit to 100.1 MHz and 100.2 MHz. We use the same test vectors as above, but our results differentiate as we see some cells get dropped in the `QuarQueue`.

The time required for the PaR increases rapidly from 35 min for 16 cells to $> 3h$ for 32 cells in `QuarQueue`, which, incidentally, does not meet the timing constraints.

4 simple tests was designed in order to see if the testbenches work. The test-vectors was either written by `cell_gen.c` shown in appendix C.

Test-1: Poll LiAB_Interface

1. read ctrl
2. write 0x00 to ctrl
3. read ctrl

4. write 0x01 to ctrl
5. read ctrl

Test vector is then:

```
r 101 0100010000
w 101 0100010000 0000000000000000
r 101 0100010000
w 101 0100010000 0000000000000001
r 101 0100010000
```

Expect to read 0x097 in first and second read, 0x03431 in third read. The purpose of this test is to see if polling works correctly and the `LiAB_Interface` is able to properly run the bus in accordance with what the LiAB expects.

Test-2: Simple cells without sort

1. write CMD_SEND to CTRL to start cell
2. write a TO_PS cell (256 words)
3. wait for an interrupt 2
4. write CMD_RECV to CTRL to ready cell for reading
5. read 256 words

Test vector is then:

```
w 101 0100010000 0000000100000000
w 101 0100010010 0011000001011010
w 101 0100010010 1000111110001000
--- 253 lines with junk data and finally parity ---
w 101 0100010000
i 2
w 101 0100010000 0000000100000001
r 101 0100010010
--- last line repeated 255 times ---
```

Purpose is to see if the testbench is capable of receiving a valid cell, enqueue it, transmit it through the internal bus, enqueue it again in the outbound queue and dequeue it slowly in LiAB speed. Cell is a TO_PS to ensure it goes through both a `CellQueue` and a `QuarQueue`. This especially tests the `CellQueue`CellQueue+s ability to allow low periods of `do_get` and `di_put` - something worth testing as it caused a number of headaches during development.

Test-3: a FROM_PS cell should be removed in QuarQueue but not in CellQueue

1. write 0x10 to CTRL to start cell
2. write a FROM_PS cell (256 words)
3. wait for an interrupt 2
4. write CMD_RECV to CTRL to ready cell for reading
5. read 256 words

Test vector is then:

```
w 101 0100010000 0000000100000000
w 101 0100010010 0010000001011010
--- 254 words junk data and finally parity ---
r 101 0100010000
i 2
w 101 0100010000 0000000100000001
r 101 0100010010
--- last line repeated 255 times ---
```

In the first setup, the cell should be passed immediately to the outbound queue, but in the case of `LiAB\_quartest`, the cell should disappear as the `QuarQueue` is supposed to throw it away.

Test-4: A cell to drop in both cases

1. write 0x10 to CTRL to start cell
2. write a HEARTBEAT cell (256 words)
3. wait for an interrupt 2
4. write CMD_RECV to CTRL to ready cell for reading
5. read 256 words

In both testbenches, the cell must be dropped as it is not destined for anything but the non-existant heartbeat monitor.

```
w 101 0100010000 0000000100000000
w 101 0100010010 0001000000000000
w 101 0100010010 0000000000000000
--- last line repeated 253 times ---
w 101 0100010010 0001000000000000
r 101 0100010000
i 2
w 101 0100010000 0000000100000001
r 101 0100010010
--- last line repeated 255 times ---
```

Test no.	LiAB_testinout	LiAB_quartest
1	OK - cell passed	OK - cell passed
2	OK - cell passed	OK - cell passed after 327us
3	OK - cell passed	OK - dropped cell as expected
4	OK - no interrupt	OK - no interrupt

Table 3: Simulation results of the synthesized testbenches.

Note, that it is practically impossible to view the internal signals of the testbenches since they are named by LeonardoSpectrum and Xilinx and most often has a name like `nx_734`. Therefore, only the outside of the firmware was monitored, which, as noted above, has a flip-flop directly on the internal bus, so it is 100ps delayed always.

4.8 Future Extensions

4.8.1 More cell-sorting in CellQueue

In the current design only empty frames are removed from the queue unless the generic is set to true, in which case heartbeat, accept and drop messages are removed. This feature was used to reduce the traffic to the LiABs, ie. in the outbound queue of each `LiAB_Interface`. A finer grained sorting mechanism could be desired - especially enforcing that only the master is allowed to send some of the messages - this is needed in the inbound queue of `LiAB_Interface`. The most generic way of doing this would be to input a mask to the `CellQueue`, that describes what messages are allowed to be enqueued. This could easily be implemented.

The advantage of monitoring if this entity is master before passing on cells through the queues would be to monitor if a LiAB wrongly assumes it is master and should respond to messages. In case it is not master, a suitable response to not-allowed messages would be to reset the LiAB. Unfortunally, there is not access to the reset pin through the LiAB connectors.

4.8.2 SeqNo assigment

If the current master LiAB malfunctions, it could send messages with duplicate sequence numbers. The problem here is re-identification of the correct frame to drop. The slaves would of course send DROP on the cells SeqNo, but this could also affect a correct frame, which could be dropped unintentionally. This would normally not be a problem as a LiAB would increase its sequence number immediately after transmitting a cell. A method for assigning sequence numbers in the FPGA could be an advantage, but here the problem was modifying the contents of the RAMB in the `CellQueue` before the cell is transmitted on the internal bus - this could happen in the sending part of the `CellQueue`, where the correct sequence number could be forced onto the cell header when appropriate.

Another obstacle would be updating the checksum to reflect the correct sequence number. One solution to this could be to redefine interfaces so the cell header is not included in the checksum, but this defies the purpose of the checksum.

4.8.3 Replace pass-condition in QuarQueue

Currently, a cell moves from S_QUAR to S_PASS only when the current epoch counter is equal to the `pass_timeout` field of the state register. This decision was made as the QuarQueue has 1024 cycles to complete processing of all cells in the QuarQueue. This does not pose a problem if the QuarQueue is set to a small (¡32, in which case the 100 MHz cannot be reached anyways) number of cells. To convince ourselves of this we must consider the time needed to process a single cell. We must finish the processing of a cell before a new cell has been received from the bus, ie. we have 256 cycles to complete the cell processing. The longest processing time is the accept/drop cells as they need to search through the entire QuarQueue for the appropriate cell - for each entry in the queue we spend 1 cycle thanks to the pipeline design, we need to wait for some pipeline empties - that amounts to 5 cycles, thus 37 total. This means that we can safely complete the processing of a cell before the new cell arrives and thus `QuarQueue` never drops behind processing the cells.

In any case, the worst thing that could happen would be the cell would be delayed another timeout period.

4.8.4 QuarQueue overflow signalling and flushing

Flushing is considered dangerous and not implemented as one or two malfunctioning LiABs could severely destroy operation of the cluster. If the flush-signal is just a single bit, the malfunctioning LiABs could easily end up in a state where they signal the FPGA to keep flushing the queue and the system would therefore fail significantly. Therefore, more complex messages must be sent to the `QuarQueue` in order to make it flush. This is one of the basic considerations - nothing should happen by chance but be an intentional operation from the LiABs side. Overflow signals could be a nice thing to add to the QuarQueue but would not really contribute much to the operation of the system. By exposing the `rcv_sfc_found` signal in `QuarQueue` on cell boundaries, an overflow flag could easily be implemented.

4.8.5 QuarQueue and HeartbeatMonitor information exchange

The cluster could be sped significantly up by using the `failed` flags from the Heartbeat monitor to set the timeout in `QuarQueue`. This way, the performance degradation when a LiAB drops out could be avoided by using a shorter timeout when only one or two LiABs are available.

This would unfortunally lengthen the logic delay in RCV of `QuarQueue` and has therefore not been implemented as the timeout epoch would go through additional logic in order to determine the correct value to use.

Sharing information in the other direction is also advantegous. The current design has a problem regarding dropping cells - it would be a significant advantage if a new master was elected whenever a cell was dropped from QQ. This relates to information loss. The `CellQueue` was fitted with a sorting mechanism, that would relieve the slaves from seeing the others communication regarding heartbeats, dropping or accepting cells - drop messages should not be removed from the CQ as they are important in relation to determining if a newly elected master must repeat the last message transmitted. This is a minor fix to CQ (one change in a "when" statement in line 196). This change would put a heavier load on the attached LiABs but would allow a LiAB to determine if a cell was dropped as it must see one besides itself. In the context of verb+Reliplan+, a LiAB does not hear itself as `di_put` is connected to inverted bus-owner.

4.8.6 HeartbeatMonitor should support SOFT_HOVER

Currently, the `HeartbeatMonitor` does not support the `SOFT_HOVER` message - a combination of laziness and time pressure. The fix is not very difficult. The problem is not that difficult - if a hand-over is needed, just refrain from sending heartbeats, and the `HeartbeatMonitor` will timeout (in app. 600ms) doing a hard hand-over.

4.8.7 A checksum more resilient than parity

It turns out that 256 words of the same value actually constitutes a valid cell. This effect was observed in the `LiAB_quartest`, where I could not immediately understand why it seemed to have two cells in the `QuarQueue`, until I realized that 256 words of the same value will pass this parity check. Therefore, a more resilient checksum is needed. Suggested solution (in 10 secs) is to XOR a rotated value instead of the original. This resembles a cyclic redundancy check and would catch the simple error mentioned above. The solution, I took for the above problem was to ensure that the communications bus between the `LiAB_Interface` and the `QuarQueue` would remain "00..00" while waiting for the transfer slot (remember that in `LiAB_quartest`, the bus between QQ and LI was scheduled in blocks of 256 cycles where half of the time, the bus remained idle - the problem was that QQ was allowed to drive the bus all the time, even while waiting for its slot).

Improving the checksum makes it more likely to actually detect errors, hence the need for retransmissions. Currently, the firmware does not support retransmissions, but could easily be fitted to support it by introducing a new input to a `CellQueue` indicating if the transmission was completed correctly and that it may move on.

4.9 Summary

In the FPGA, a quarantine queue was implemented and appropriate interfaces towards the LiABs was built. The FPGA was fitted with a heartbeat monitor, that was capable of monitoring the timely heartbeat messages from each LiAB and capable of pointing out a master for the cluster. The FPGA contained a quarantine queue, that would allow LiAB-0 through LiAB-2 to enter into either triple-redundancy or a slow-functioning mode. The timing goal of 100 MHz was reached after extensive pipelining. Two simple test were devised and simulated but not tested on the board.

5 Software

The LiAB need a driver to interface with the firmware embedded in the Xilinx FPGA. Through section 4, we saw the development of the firmware with emphasis on the hardware details. In this section, we will focus on the FPGAs interface towards the LiAB and what requirements the LiABs on-board driver software needs to fulfill. The firmware against any given LiAB (the `LiAB_Interface` - see 4.5.4) will, from here on, be called Reliplan.

All reads and writes to Reliplan are 16-bit wide and should be performed at the GPBs lower settings of 1 cycle address, 1 cycle data and 1 cycle wait. This will assure Reliplan has enough time to process data.

- `0x[0-9a-fA-F]+`
  base-16 (hex) number
- `0b[0-1]+`
  base-2 (binary) number
- `[0-9]+`
  base-10 (decimal) number

Reliplan has two registers, CTRL and DATA. The location of these registers are given in table 4.

Register	CS(0..2)	A(9..0)
CTRL	0b101	0x110
DATA	0b101	0x112

Table 4: Register locations.

Each register can be either read or written. If a write is performed to CTRL, it is considered a command to the firmware. If a write is done to DATA, it is considered a word of the current cell being built for transmission. If reads are performed from DATA, Reliplan will return data from the cell currently being transmitted to the LiAB. The first operation in a sequence of 256 on DATA should always be preceded with a write to CTRL of either a `CMD_SEND` or `CMD_RECV` to ensure the internal cell offset counter is reset to offset 0 before starting a read/write. If the internal cell offset counter should divert from 0, eg. this is the first write after a crash, Reliplan will just continue beyond the end of the current cell and into the next cell (if any - otherwise returns junk data).

The last paragraph implies that a transfer of a cell may not be interrupted by other read/writes to Reliplan. It is possible to mix reads and writes, but the semantics are so difficult that it is highly disrecommended.

5.1 Commands

In table 5 the commands available for Reliplan are listed. These commands may be written to the CTRL register, and immediately after (as in next read), a return code is ready to be read from CTRL. In table 6 the possible return values can be read.

By regularly writing a `CMD_RECV`, it is possible to poll the interface for new cells, but polling is, in general, considered bad as it consumes CPU time without actually performing any work. Therefore Reliplan supports interrupts, as described in section 5.1.1.

Value	Mnemonic	Description
0x0000	`CMD_ID0`	Put 0x0097 in CTRL
0x0001	`CMD_ID1`	Put 0x3431 in CTRL
0x0002	`CMD_VER`	Major and minor version is put in CTRL
0x0003	`CMD_STAT`	*UNIMPLEMENTED* status in CTRL
0x0100	`CMD_SEND`	Next word written to DATA is first word of cell
0x0101	`CMD_RECV`	Next word read from DATA is first word of cell
0xFFFF	`CMD_RESET`	Reset Reliplan[5]

Table 5: Commands for Reliplan.

Value	Mnemonic	Description
0x0000	`E_SUCCESS`	The command succeeded
0x0001	`E_FULL`	Not possible to start a new cell - queue from LiAB to reliplan already full
0x0002	`E_EMPTY`	Cannot start a new cell read, queue from Reliplan is empty
0x0003	`E_DISCARD`	Cell was discarded
0x0004	`E_CELLOK`	cell was enqueued
0xFFFF	`E_UNIMPL`	Command sent was not understood or not implemented

Table 6: Return values in CTRL for Reliplan.

5.1.1 Interrupts

Reliplan has been connected to the GPIRQ1 and GPIRQ2 pins, whose meaning are respectively ”master” and ”data available”.

If GPIRQ1 is asserted (1), this LiAB is the elected master, if the pin is 0, this LiAB may not attempt to answer messages from the packet switch.

GPIRQ2 tells the LiAB it needs to read a cell from Reliplan and will not be reset to 0 until the queue has been emptied - there is no facility to clear the interrupt after the ISR has been started.

5.2 Checksum

In order to protect against transmission errors from the LiAB to Reliplan, the last word of the cell transmitted must be the even parity of the data words in the cell. This not-too-resilient check should be changed in future versions of Reliplan, which will return a value different from 0x0001 when doing `CMD_VER`. Pseudocode for checksum calculation is shown in figure 37

```
csum = 0x0000
FOR i IN 0 TO 254 DO
  csum = csum XOR cell[i]
cell[255] = csum
```

Figure 37: Pseudocode for checksum calculation

5.3 Communication sequences

5.3.1 Init

Reliplan contains a state machine controlled by the LiAB through the CTRL register located at the address listed in the table below. When the FPGA is reset, a 0x0097 can be read from CTRL. The reason for this init behavior is the need for each LiABs need to poll the devices attached to the system. This behavior makes it easier to find an attached Reliplan device. This behavior is the same as if `CMD_ID0` was written to CTRL. To further assure the presence of a Reliplan device, the LiAB may write `CMD_ID1` to the CTRL register. Soon afterwards, the value 0x3431 can be read from CTRL. If both these test succeed, there is plenty of likelihood that a Reliplan is actually present at the address. By writing `CMD_VER` to the CTRL register, the CTRL register will be set to the current version of the Reliplan in the high byte of the register and a revision level in the low byte. Currently, major=00, minor=01.

The LiAB sometimes needs to reset Reliplan, this can be done by writing `CMD_RESET` to Reliplan. The effect of this is to drop all unsent cells in the queue and reset the interface (usually, the fast transfer of cells will assure the queue is empty).

5.3.2 Send a cell

Start by writing a `CMD_SEND` to CTRL. This will assure that the next word written to DATA will be the first word of a cell. Follow this with the cell header, the 254 words payload and the parity.

5.3.3 Read a cell

Start by writing a `CMD_RECV` to CTRL. Wait a couple of cycles (intentionally imprecise as the Reliplan should be finished before the SC520 can perform the next GPB transaction). Read status from CTRL, if 0x0000, read 256 times from DATA to read entire cell.

5.4 The SC520 on the LiAB

The LiABs used in this project features an AMD Elan SC520 microcontroller - a very versatile x86 compatible microcontroller with many integrated peripherals. Especially the GP bus of this microcontroller is interesting as it is this bus the firmware described in section 4 needs to interact with. The timing of the GPB is fully programmable as described on pg. 69 and ahead in the SC520 data sheet [23] and ch. 13 in the SC520 users manual [24]. To control the bus, the PAR (Programmable Address Registers) must be set up to allow the on-board software to control the bus.

5.5 Transactions

As described in section 4, the firmware supports a set of messages indicated by the upper 4 bits of the first word transferred. These bits describe the destination of the message - if they are a state update, a state dump, state request, destined for the packet switch or if they are from the packet switch. To recapitulate the message types, they are listed in table 7.

The suggested way of programming the cluster is a transaction based approach much like interacting with a transactional database. In these databases, you start a transaction with a `BEGIN`, then you issue your SQL statements completing them

Bits	Mnemonic
0000	EMPTY
0001	HEARTBEAT
0010	FROM_PS
0011	TO_PS
0100	DROP
0101	ACCEPT
1000	GET_STATE
1001	STATE_DUMP
1010	STATE_UPDATE
1111	SOFT_HOVER

Table 7: Cell header values for the contents field of the cell header.

with either a COMMIT or a ROLLBACK. If you roll back, any changes the statements between BEGIN and ROLLBACK will have no effect. If COMMIT ends the sequence, the changes are performed in the database[6]. Likewise in this system - the master starts a transaction by using an unused sequence number in a state update cell and a timeout or an accept works as commit whereas 2 drops works as roll-back.

In each cell header is a sequence number. This sequence number is the ID of the transaction about to take place on the cluster. When the master receives data from the packet switch, it will often need to update its state and send a reply to the packet switch. The master must always precede the messages to the packet switch by a number of state updates, thereby allowing the slave LiABs to build a complete state update and decide, based on listening to the bus and determining if the cells containing the reply to the packet switch was dropped. If the slave LiABs do not hear 2 drop messages they may assume that the master has performed the state update they have collected, and they should therefore commit the state update themselves. If the slaves hear 2 drops (1 besides its own, which it will not hear) they can safely assume the cells sent by the master was dropped from the quarantine queue and the LiAB should hence not commit the update.

5.5.1 Sequences of cells - ”more”-flag

In order to determine if the master LiAB fails in the middle of its transaction, one bit of the ”reserved” field of the cell header is declared the ”more” bit. If this bit is set, the master intends to send additional cells containing data. By using a timer associated with the last cell received, the slaves can determine if they must abort a transaction as the masters last cell had a set more-flag, but no more data arrived in time. This also forces the master to prepare its entire transaction in memory - both the state update and the reply to the packet switch before starting a transaction. The timeout of the heartbeat monitor is 671ms, the quarantine queue times out a transaction after (a currently too small value) 64 epochs $\tilde{6}$40us - so the entire transaction and the time-out values in the LiABs waiting for more cells if the more-flag is set must be below 640us.

If no reply is required to the packet switch, a dummy cell must be issued in order to signify the end of the transaction to the other LiABs. Therefore, the higher level protocol must facilitate the reception of an empty cell.

We may use the reserved field as this field is preserved in the current firmware.

[6]And also, no concurrent access to the database will see any changes before commit - the changes are atomic.

5.5.2 Version numbers

Messages internally in the cluster and messages directed to the packet switch must be equipped with a version number. This is required so that newer releases of the software installed on one of the other members of the cluster can interpret the message sent by older versions of the software currently acting as master. A field must therefore be reserved in the second-layer protocol for this information.

6 Concluding remarks

During the development of the firmware, a number of issues emerged. The most important part of these was the unexpected reason for the longest path in the digital design. I initially expected only logic delay to be a problem, but the logic routes were, in my opinion, so short they should not pose a major obstacle. They did not, unlike the routing of the signal around the FPGA. The signal propagation between the different functions was so high that extensive pipelining was required in order to shorten the paths. This is the reason for the output registers in both `QuarQueue` and `CellQueue`, even though these registers are connected only through a tri-state driver to the internal bus.

Another major issue encountered was the wrong synthesis tool - Precision RTL that failed to properly synthesize the busses. This was worked around by simply choosing to use LeonardoSpectrum instead.

The board completed initial testing (a simple counter outputting an increasing number on the HP-connectors) and the firmware was synthesized and the timing model from Xilinx ISE was tested against two simple sequences of cells.

The driver for the LiABs was never completed as time spendt on making the firmware work exceeded the expected by a large factor, Likewise implementation of the TCP stack being an extension on the driver, was never initiated.

References

[1] BTexact: *Carrier requirements of core IP routers*, BTexact (2002)

[2] BTexact: *Reliable routing and forwarding*, BTexact (2002)

[3] Hadriel Kaplan: *NSR™Non-Stop Routing Technology*, Avici Systems Inc. (2002)

[4] Aman Shaikh, Rohit Dube, Anujan Varma: *Avoiding Instability during Graceful Shutdown of OSPF*, IEEE Infocom (2002)

[5] Alberto Leon-Garcia and Indra Widjaja: *Communication Networks*, McGraw-Hill Higher Education (2000)

[6] Christian Huitema: *Routing in the Internet, 2nd ed.*, Prentice Hall PTR, Upper Saddle River, NJ 07458 (2000)

[7] G. Malkin: *RFC-1723: RIP Version 2 - Carrying Additional Information*, (November 1994)

[8] Y. Rekhter, T. Li: *RFC-1771: A Border Gateway Protocol 4 (BGP-4)*, (March 1995)

[9] F. Baker: *RFC-1812: Requirements for IPv4 Routers*, (June 1995)

[10] J. Moy: *RFC-2328: OSPF Version 2*, (April 1998)

[11] J. Moy, P. Pillay-Esnault, A. Lindem: *RFC-3623: Graceful OSPF Restart*, (November 2003)

[12] Xilinx: *DS025-2: Virtex™-E 1.8V Extended Memory Field Programmable Gate Arrays, v2.2*, Xilinx (September 10, 2002)

[13] Xilinx: *DS026: XC18V00 Series In-System Programmable Configuration PROMs, v5.0.1*, Xilinx (July 20, 2004)

[14] Xilinx: *XAPP058: Xilinx In-System Programming Using an Embedded Microcontroller, v3.1*, Xilinx (June 25, 2004)

[15] Xilinx: *XAPP138: Virtex FPGA Series Configuration and Readback, v2.7*, Xilinx (July 11, 2001)

[16] IEEE Computer Society: *1076-2002: Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronic Engineering (2002)

IEEE Computer Society: *1076-1993: Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronic Engineering (1993)

IEEE Computer Society: *1076-1987: Standard VHDL Language Reference Manual*, The Institute of Electrical and Electronic Engineering (1987)

[17] Carl Carmichael: *XAPP197: Triple Module Redundancy Design Techniques for Virtex FPGAs*, Xilinx (November 1, 2001)

[18] John Lach, William H. Mangione-Smith, and Miodrag Potkonjak: *Enhanced FPGA Reliability Through Efficient Run-Time Fault Reconfiguration* IEEE transactions on Reliability, Vol. 49, No. 3 (September 2000)

[19] Vinita Singhal and Robert Le: *XAPP240: High-Speed Buffered Crossbar Switch Design Using Virtex-EM Devices*, Xilinx (March 14, 2000)

[20] William J. Dally, Larry R. Dennison, David Harris, Kinhong Kan and Thucydides Xanthopoulos: *Architecture and Implementation of the Reliable Router*, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, Cambridge Massachusetts

[21] Suresh Chalasani and Rajendra Boppana: *Fault-Tolerance with Multimodule Routers*, IEEE (1996)

[22] Mikael Dich: *Hardware Reference for the Linux In A Box (LIAB), ver.3*, LiAB Electronics ApS (March 2001)

[23] AMD: *AMD Elan™SC520 Microcontroller Data Sheet (preliminary)*, order #22003, Advanced Micro-Devices (2001)

[24] AMD: *AMD Elan™SC520 Microcontroller User's Manual*, order #22004B, Advanced Micro-Devices (2001)

[25] AMD: *AMD Elan™SC520 Register Set Manual*, order #22005B, Advanced Micro-Devices (2001)

[26] D. Kalinsky Associates: *Design Patterns for High Availability*, `http://www.kalinskyassociates.com/Wpaper6.html` (2004)

[27] James S. Plank, Micah Beck, Gerry Kingsley, Kai Li: *Libckpt: Transparent Checkpointing under Unix*, University of Tennesse (????)

[28] Mark E. Staknis: *Sheaved Memory: Architectural Support for State Saving and Restoration in Paged Systems*, Department of Industrial Engineering and Information Systems, Shell Engineering Center, Northeastern University, Boston, Massachusetts (????)

[29] Hackers ass.: *The Jargon File 4.4.7*, `http://www.catb.org/~esr/jargon/` (Dec.2003)

A Pin mappings

All pin-mappings are included on the attached CD-ROM in `/rip_pins/` as both a CSV file and an UCF file suited for copy'n'paste to your projects UCF file. To have a hardcopy just-in-case, the pin mappings are printed here. Along with the files, the simple Perl-script shown in figure 4 is included for fast translation to other formats (a slightly modified script made the tables showed below).

Clock and resets.

Net	Loc.	Dir.
`clk`	AL17	Input
`reset`	C30	Input

Leds; 3 banks of 4 leds.

Net	Loc.	Dir.
`led0(0)`	AD4	Output
`led0(1)`	AE3	Output
`led0(2)`	AC5	Output
`led0(3)`	AE1	Output
`led1(0)`	AD3	Output
`led1(1)`	AC4	Output
`led1(2)`	AB5	Output
`led1(3)`	AC3	Output
`led2(0)`	AE33	Output
`led2(1)`	AC29	Output
`led2(2)`	AE32	Output
`led2(3)`	AD30	Output
`led3(0)`	AE31	Output
`led3(1)`	AF32	Output
`led3(2)`	AD29	Output
`led3(3)`	AE30	Output

Dipswitches; two groups of 6 and 3 switches.

Net	Loc.	Dir.
`dipsw1`	L32	Input
`dipsw2`	M31	Input
`dipsw3`	L33	Input
`dipsw4`	M29	Input
`dipsw5`	L31	Input
`dipsw6`	M30	Input
`dipswa`	AM31	Input
`dipswb`	AK28	Input
`dipswc`	AL30	Input

The four HP connectors and their FPGA supplied clocks.

Net	Loc.	Dir.	Net	Loc.	Dir.
hp0_clk1	U31	Output	hp1_clk1	U29	Output
hp0_data(15)	U32	Output	hp1_data(0)	U33	Output
hp0_data(14)	T32	Output	hp1_data(1)	V32	Output
hp0_data(13)	T30	Output	hp1_data(2)	V31	Output
hp0_data(12)	T29	Output	hp1_data(3)	V29	Output
hp0_data(11)	T31	Output	hp1_data(4)	V30	Output
hp0_data(10)	R33	Output	hp1_data(5)	W33	Output
hp0_data(9)	R31	Output	hp1_data(6)	W31	Output
hp0_data(8)	R30	Output	hp1_data(7)	W30	Output
hp0_data(7)	R29	Output	hp1_data(8)	W29	Output
hp0_data(6)	P32	Output	hp1_data(9)	Y32	Output
hp0_data(5)	P31	Output	hp1_data(10)	Y30	Output
hp0_data(4)	P30	Output	hp1_data(11)	AA33	Output
hp0_data(3)	P29	Output	hp1_data(12)	Y29	Output
hp0_data(2)	M32	Output	hp1_data(13)	AA32	Output
hp0_data(1)	N31	Output	hp1_data(14)	AA31	Output
hp0_data(0)	N30	Output	hp1_data(15)	AA30	Output

Net	Loc.	Dir.	Net	Loc.	Dir.
hp2_clk1	U3	Output	hp3_clk1	U1	Output
hp2_data(0)	M2	Output	hp3_data(0)	U2	Output
hp2_data(1)	N4	Output	hp3_data(1)	U4	Output
hp2_data(2)	N3	Output	hp3_data(2)	V2	Output
hp2_data(3)	N2	Output	hp3_data(3)	V4	Output
hp2_data(4)	P5	Output	hp3_data(4)	V5	Output
hp2_data(5)	P4	Output	hp3_data(5)	V3	Output
hp2_data(6)	P3	Output	hp3_data(6)	W1	Output
hp2_data(7)	P2	Output	hp3_data(7)	W3	Output
hp2_data(8)	R5	Output	hp3_data(8)	W4	Output
hp2_data(9)	R4	Output	hp3_data(9)	W5	Output
hp2_data(10)	R3	Output	hp3_data(10)	Y3	Output
hp2_data(11)	R1	Output	hp3_data(11)	Y4	Output
hp2_data(12)	T4	Output	hp3_data(12)	AA1	Output
hp2_data(13)	T5	Output	hp3_data(13)	Y5	Output
hp2_data(14)	T3	Output	hp3_data(14)	AA3	Output
hp2_data(15)	T2	Output	hp3_data(15)	AA4	Output

Net	Loc.	Dir.
liab0_io(0)	C9	InOut
liab0_io(1)	A9	InOut
liab0_io(2)	C10	InOut
liab0_io(3)	E11	InOut
liab0_io(4)	D11	InOut
liab0_io(5)	B10	InOut
liab0_io(6)	C11	InOut
liab0_io(7)	B11	InOut
liab0_io(8)	D12	InOut
liab0_io(9)	A11	InOut
liab0_io(10)	E13	InOut
liab0_io(11)	C12	InOut
liab0_io(12)	D13	InOut
liab0_io(13)	C13	InOut
liab0_io(14)	E14	InOut
liab0_io(15)	A13	InOut

Net	Loc.	Dir.
liab0_data(0)	F3	InOut
liab0_data(1)	F5	InOut
liab0_data(2)	B3	InOut
liab0_data(3)	F4	InOut
liab0_data(4)	C1	InOut
liab0_data(5)	G5	InOut
liab0_data(6)	E3	InOut
liab0_data(7)	D3	InOut
liab0_data(8)	D2	InOut
liab0_data(9)	G4	InOut
liab0_data(10)	H5	InOut
liab0_data(11)	E2	InOut
liab0_data(12)	H4	InOut
liab0_data(13)	G3	InOut
liab0_data(14)	G1	InOut
liab0_data(15)	J5	InOut

Net	Loc.	Dir.
liab0_addr(0)	A2	Input
liab0_addr(1)	D6	Input
liab0_addr(2)	E7	Input
liab0_addr(3)	A3	Input
liab0_addr(4)	C5	Input
liab0_addr(5)	B4	Input
liab0_addr(6)	A4	Input
liab0_addr(7)	D7	Input
liab0_addr(8)	C6	Input
liab0_addr(9)	B5	Input
liab0_n_cs0	A5	Input
liab0_n_cs1	E8	Input
liab0_n_cs2	D8	Input
liab0_n_iord	C7	Input
liab0_n_iowr	A6	Input

Net	Loc.	Dir.
liab0_ale	B7	Input
liab0_pciclk	D17	Input
liab0_irq0	A15	Output
liab0_irq1	C16	Output
liab0_irq2	E16	Output
liab0_rdy	D16	InOut
liab0_n_bhe	B16	Input
liab0_n_dbufoe	B17	Input
liab0_n_iocs16	C17	Input

Net	Loc.	Dir.
`liab1_io(0)`	G30	InOut
`liab1_io(1)`	F29	InOut
`liab1_io(2)`	D31	InOut
`liab1_io(3)`	F30	InOut
`liab1_io(4)`	C33	InOut
`liab1_io(5)`	G29	InOut
`liab1_io(6)`	E30	InOut
`liab1_io(7)`	E31	InOut
`liab1_io(8)`	D32	InOut
`liab1_io(9)`	F31	InOut
`liab1_io(10)`	H29	InOut
`liab1_io(11)`	E32	InOut
`liab1_io(12)`	E33	InOut
`liab1_io(13)`	G31	InOut
`liab1_io(14)`	F33	InOut
`liab1_io(15)`	J29	InOut

Net	Loc.	Dir.
`liab1_data(0)`	B26	InOut
`liab1_data(1)`	C25	InOut
`liab1_data(2)`	D24	InOut
`liab1_data(3)`	B25	InOut
`liab1_data(4)`	E23	InOut
`liab1_data(5)`	A25	InOut
`liab1_data(6)`	D23	InOut
`liab1_data(7)`	B24	InOut
`liab1_data(8)`	E22	InOut
`liab1_data(9)`	C23	InOut
`liab1_data(10)`	A23	InOut
`liab1_data(11)`	D22	InOut
`liab1_data(12)`	E21	InOut
`liab1_data(13)`	B22	InOut
`liab1_data(14)`	D21	InOut
`liab1_data(15)`	C21	InOut

Net	Loc.	Dir.
`liab1_addr(0)`	D30	Input
`liab1_addr(1)`	E28	Input
`liab1_addr(2)`	D29	Input
`liab1_addr(3)`	D28	Input
`liab1_addr(4)`	A31	Input
`liab1_addr(5)`	E27	Input
`liab1_addr(6)`	C29	Input
`liab1_addr(7)`	B30	Input
`liab1_addr(8)`	D27	Input
`liab1_addr(9)`	E26	Input
`liab1_n_cs0`	B29	Input
`liab1_n_cs1`	C28	Input
`liab1_n_cs2`	D26	Input
`liab1_n_iord`	C27	Input
`liab1_n_iowd`	A27	Input

Net	Loc.	Dir.
`liab1_ale`	E25	Input
`liab1_irq0`	B20	Output
`liab1_irq1`	E19	Output
`liab1_irq2`	D19	Output
`liab1_rdy`	C19	InOut
`liab1_n_bhe`	A19	Input
`liab1_n_dbufoe`	D18	Input
`liab1_n_iocs16`	E18	Input
`liab1_pciclk`	A17	Input

Net	Loc.	Dir.
liab2_io(0)	AJ23	InOut
liab2_io(1)	AN26	InOut
liab2_io(2)	AL24	InOut
liab2_io(3)	AK23	InOut
liab2_io(4)	AJ22	InOut
liab2_io(5)	AL23	InOut
liab2_io(6)	AM24	InOut
liab2_io(7)	AK22	InOut
liab2_io(8)	AM23	InOut
liab2_io(9)	AJ21	InOut
liab2_io(10)	AN23	InOut
liab2_io(11)	AK21	InOut
liab2_io(12)	AM22	InOut
liab2_io(13)	AJ20	InOut
liab2_io(14)	AL21	InOut
liab2_io(15)	AN21	InOut

Net	Loc.	Dir.
liab2_data(0)	AH33	InOut
liab2_data(1)	AE29	InOut
liab2_data(2)	AF30	InOut
liab2_data(3)	AH32	InOut
liab2_data(4)	AF29	InOut
liab2_data(5)	AH31	InOut
liab2_data(6)	AJ32	InOut
liab2_data(7)	AG30	InOut
liab2_data(8)	AK32	InOut
liab2_data(9)	AJ31	InOut
liab2_data(10)	AG29	InOut
liab2_data(11)	AH30	InOut
liab2_data(12)	AK31	InOut
liab2_data(13)	AJ30	InOut
liab2_data(14)	AH29	InOut
liab2_data(15)	AL33	InOut

Net	Loc.	Dir.
liab2_addr(0)	AJ27	Input
liab2_addr(1)	AL28	Input
liab2_addr(2)	AN31	Input
liab2_addr(3)	AL29	Input
liab2_addr(4)	AK27	Input
liab2_addr(5)	AN28	Input
liab2_addr(6)	AJ26	Input
liab2_addr(7)	AM30	Input
liab2_addr(8)	AM29	Input
liab2_addr(9)	AK26	Input
liab2_n_cs0	AJ25	Input
liab2_n_cs1	AN29	Input
liab2_n_cs2	AK25	Input
liab2_n_rd	AL26	Input
liab2_n_wr	AL25	Input

Net	Loc.	Dir.
liab2_ale	AJ24	Input
liab2_irq0	AM20	Output
liab2_irq1	AK19	Output
liab2_irq2	AL19	Output
liab2_rdy	AN19	InOut
liab2_n_bhe	AJ18	Input
liab2_n_dbufoe	AK18	Input
liab2_n_iocs16	AL18	Input
liab2_pciclk	AJ17	Input

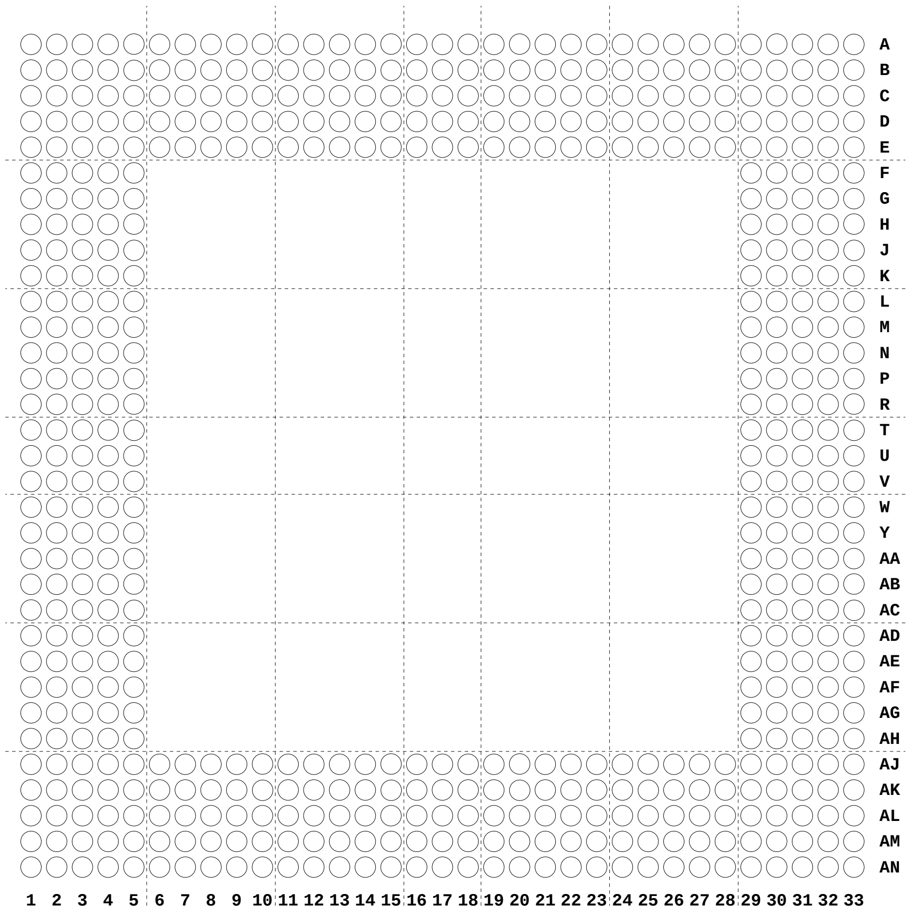
Net	Loc.	Dir.
liab3_io(0)	AG2	InOut
liab3_io(1)	AE4	InOut
liab3_io(2)	AH1	InOut
liab3_io(3)	AE5	InOut
liab3_io(4)	AH3	InOut
liab3_io(5)	AF4	InOut
liab3_io(6)	AJ1	InOut
liab3_io(7)	AJ2	InOut
liab3_io(8)	AF5	InOut
liab3_io(9)	AG4	InOut
liab3_io(10)	AK2	InOut
liab3_io(11)	AK3	InOut
liab3_io(12)	AJ3	InOut
liab3_io(13)	AG5	InOut
liab3_io(14)	AL1	InOut
liab3_io(15)	AH4	InOut

Net	Loc.	Dir.
liab3_data(0)	AL9	InOut
liab3_data(1)	AM9	InOut
liab3_data(2)	AK10	InOut
liab3_data(3)	AN9	InOut
liab3_data(4)	AL10	InOut
liab3_data(5)	AM10	InOut
liab3_data(6)	AJ11	InOut
liab3_data(7)	AL11	InOut
liab3_data(8)	AJ12	InOut
liab3_data(9)	AN11	InOut
liab3_data(10)	AK12	InOut
liab3_data(11)	AL12	InOut
liab3_data(12)	AM12	InOut
liab3_data(13)	AK13	InOut
liab3_data(14)	AL13	InOut
liab3_data(15)	AM13	InOut

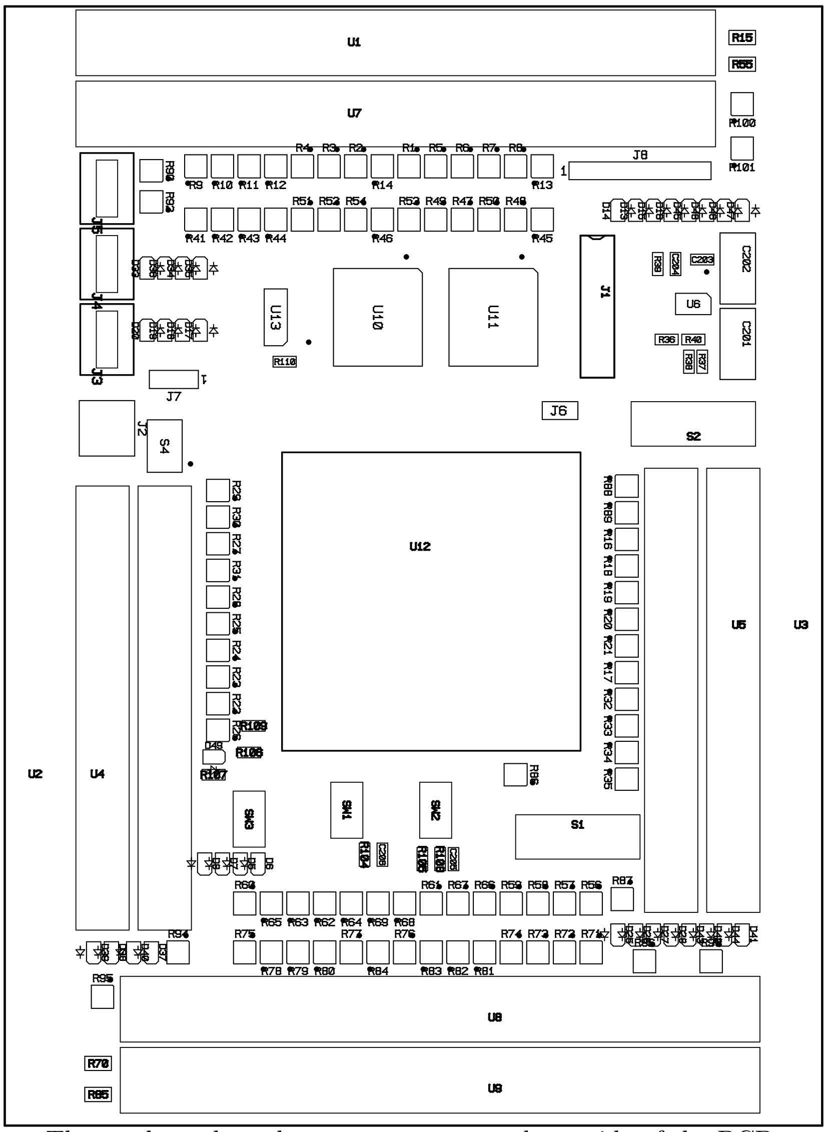
Net	Loc.	Dir.
liab3_addr(0)	AL4	Input
liab3_addr(1)	AJ6	Input
liab3_addr(2)	AK5	Input
liab3_addr(3)	AN3	Input
liab3_addr(4)	AL5	Input
liab3_addr(5)	AJ7	Input
liab3_addr(6)	AM4	Input
liab3_addr(7)	AM5	Input
liab3_addr(8)	AK6	Input
liab3_addr(9)	AK7	Input
liab3_n_cs0	AL6	Input
liab3_n_cs1	AM6	Input
liab3_n_cs2	AN6	Input
liab3_n_rd	AK9	Input
liab3_n_wr	AL7	Input

Net	Loc.	Dir.
liab3_ale	AJ8	Input
liab3_irq0	AK15	Output
liab3_irq1	AL15	Output
liab3_irq2	AM16	Output
liab3_rdy	AL16	InOut
liab3_n_bhe	AJ16	Input
liab3_n_dbufoe	AK16	Input
liab3_n_iocs16	AN17	Input
liab3_pciclk	AM17	Input

And because I did not relate to the pin values, a floorplan for a BG-560 is included here - suddenly the values in the previous listing makes sense....sort of...



B PCB components



The mask used to place components on the upside of the PCB.

C Cell generator - cell_gen.c

```
#include <stdio.h>
#include <stdlib.h>

#define IO_CTRL 0x110
#define IO_DATA 0x112
#define CMD_START_CELL 0x0100
#define CMD_RECV 0x0101

typedef short int uint16; /* MUST be a 16-bit unsigned value! */

void to_bin(uint16 v, char *str, char bits) {
  int j=0;
  str += bits;
  *str='\0';
  for (j=0;j<bits;j++) {
    str--;
    if (v&0x0001)
      *str='1';
    else
      *str='0';
    v >>= 1;
  }
}

/* update buffer with checksum */
void checksum(uint16* buffer) {
  uint16 checksum=0;
  int lp1;

  for (lp1=0;lp1<255;lp1++) {
    checksum ^= *buffer;
    buffer++;
  }
  *buffer = checksum;
}

/* output a sequence transmitting a cell to the FPGA
 * CMD_START_CELL -> [CTRL]
 * cell -> [DATA]
 * read [CTRL]
 */
void write_cell(uint16* buffer) {
  int lp1;
  char *mem = calloc(1,11);
  char *dat = calloc(1,17);
  to_bin(IO_CTRL,mem,10);
  to_bin(CMD_START_CELL,dat,16);
  printf("w 101 %s %s\n",mem,dat);
  to_bin(IO_DATA,mem,10);
  for (lp1=0;lp1<256;lp1++) {
    to_bin(*buffer,dat,16);
    printf("w 101 %s %s\n",mem,dat);
    buffer++;
  }
  to_bin(IO_CTRL,mem,10);
  printf("r 101 %s\n",mem);
  free(mem);
  free(dat);
}


int main(int argc, char** argv) {
  /* a buffer containing a cell */
  uint16 *buffer = calloc(sizeof(uint16),256);
  int lp1;
  char* cmd;

  if (sizeof(uint16)!=2) {
    printf("Compile error - sizeof(uint16)=%d and not 2 as it is supposed to!\n",
           sizeof(uint16));
    return 1;
  }

  srand(time());
  argv++;

  while (cmd = *argv) {
    argv++;
    memset(buffer,0,512);
    printf("# %s\n",cmd);
    if (strcmp("rd",cmd)==0) {
      char mem[11];
      char dat[17];
      to_bin(IO_CTRL,mem,10);
      to_bin(CMD_RECV,dat,16);
      printf("w 101 %s %s\n",mem,dat);
      to_bin(IO_DATA,mem,10);
      for (lp1=0;lp1<256;lp1++)
        printf("r 101 %s\n",mem);
    } else

    if (strcmp("empty",cmd)==0) {
      buffer[0]=0x0000;
      checksum(buffer);
      write_cell(buffer);
    } else

    if (strcmp("hb",cmd)==0) {
      buffer[0]=0x1000;
      checksum(buffer);
      write_cell(buffer);
    } else

    if (strcmp("tops",cmd)==0) {
      buffer[0]=0x305a;
      for (lp1=1;lp1<256;lp1++)
        buffer[lp1]=rand()&0xffff;
      checksum(buffer);
      write_cell(buffer);
    } else

    if (strcmp("drop",cmd)==0) {
      buffer[0]=0x40aa;
      checksum(buffer);
      write_cell(buffer);
    } else

      if (strcmp("accept",cmd)==0) {
      buffer[0]=0x505a;
      checksum(buffer);
      write_cell(buffer);
    } else

    if (strcmp("sdmp",cmd)==0) {
      buffer[0]=0x9001;
      for (lp1=1;lp1<256;lp1++)
        buffer[lp1]=rand()&0xffff;
      checksum(buffer);
      write_cell(buffer);
    }
  }
}
```

D Timing reports

An example of a timing report from LeonardoSpectrum. This is the output from `LiAB_testinout` which states that $129MHz$ is max speed and critical path seems to be the address match to bus driver.

```
                        Clock Frequency Report

      Clock                : Frequency
      ------------------------------------

      clk                  : N/A
      clk_int              : 129.4 MHz


                        Critical Path Report


Critical path #1, (path slack =  2.3):

NAME                                                        GATE                  ARRIVAL                LOAD
--------------------------------------------------------------------------------------------------------
liab0_gpa(8)/                                                           0.00  0.00 up              1.02
liab0_gpa_ibuf(8)/I                                         IBUF        0.00  0.00 up              0.00
liab0_gpa_ibuf(8)/O                                         IBUF        1.03  1.03 up              1.16
liab0_notri/nx162/I3                                        LUT4        0.00  1.03 up              1.02
liab0_notri/nx162/O                                         LUT4        0.74  1.77 up              1.02
liab0_notri/is_data/I3                                      LUT4        0.00  1.77 up              1.16
liab0_notri/is_data/O                                       LUT4        0.79  2.55 up              1.16
liab0_nx20/I1                                               LUT2        0.00  2.55 up              1.16
liab0_nx20/O                                                LUT2        0.79  3.34 up              1.16
nx364/I0                                                    LUT1        0.00  3.34 up              3.12
nx364/O                                                     LUT1        1.52  4.86 up              3.12
liab0_ix1519/T                                              BUFT        0.00  4.86 up              0.00
liab0_ix1523/O                                              BUFT        0.75  5.61 up              1.02
liab0_gpd_bdbuf(15)/I                                       IOBUF       0.00  5.61 up              0.00
liab0_gpd_bdbuf(15)/IO                                      IOBUF       2.12  7.73 up              1.02
liab0_gpd(15)/                                                          0.00  7.73 up              0.00
data arrival time                                                             7.73

data required time  (default specified)                                       10.00
--------------------------------------------------------------------------------------------------------
data required time                                                            10.00
data arrival time                                                             7.73
                                                                           ----------
slack                                                                        2.27
--------------------------------------------------------------------------------------------------------
```

And the corresponding report from Xilinx ISE:

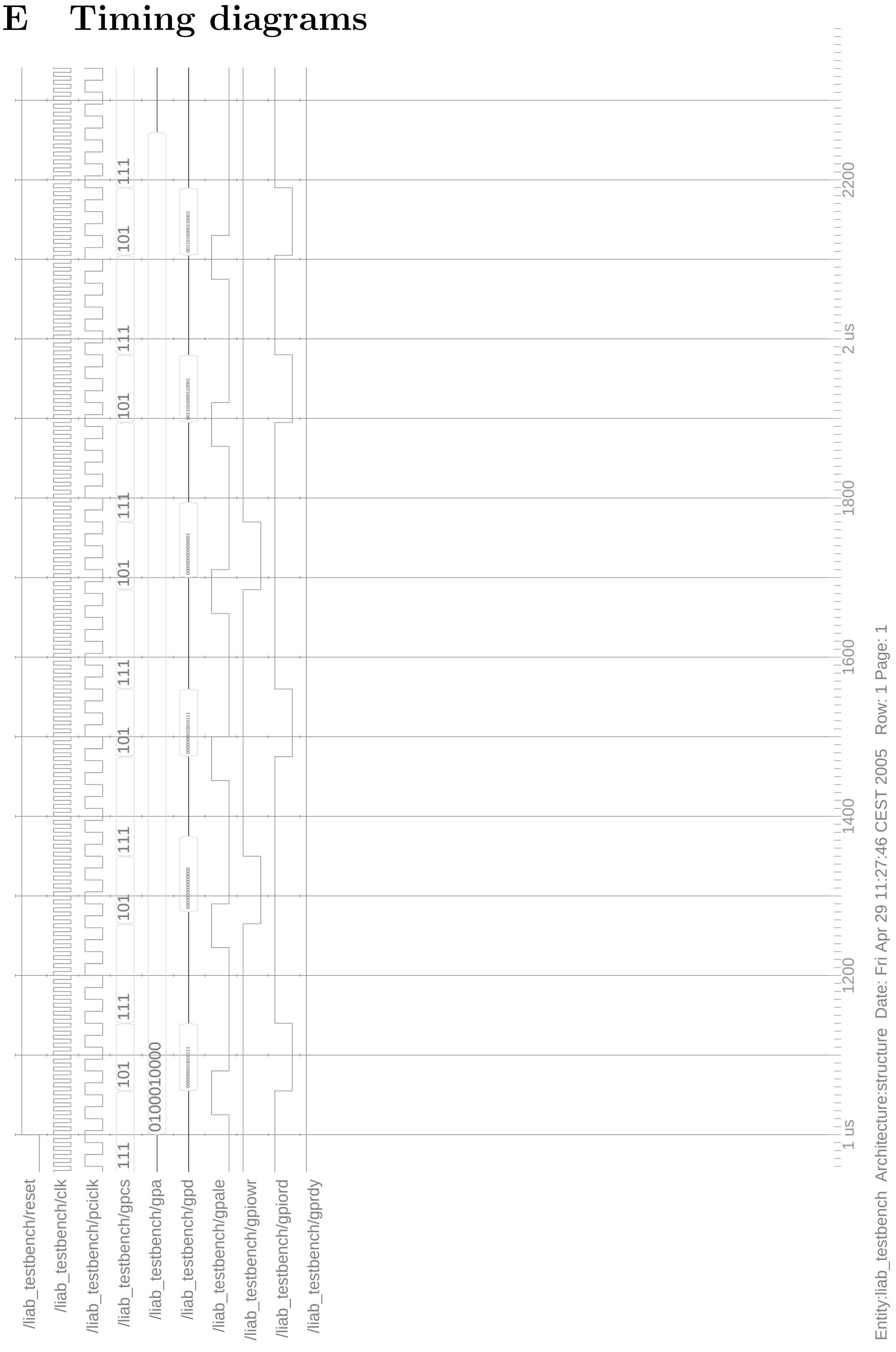
```
--------------------------------------------------------------------------------
Release 6.2.03i Trace G.28
Copyright (c) 1995-2004 Xilinx, Inc.  All rights reserved.

C:/Xilinx/bin/nt/trce.exe -intstyle ise -e 3 -l 3 -xml LiAB_testinout
LiAB_testinout.ncd -o LiAB_testinout.twr LiAB_testinout.pcf


Design file:              LiAB_testinout.ncd
Physical constraint file: LiAB_testinout.pcf
Device,speed:             xcv812e,-8 (PRODUCTION 1.69 2003-12-13)
Report level:             error report

Environment Variable      Effect
--------------------      ------
NONE                      No environment variables were set
--------------------------------------------------------------------------------

INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths
   option. All paths that are not constrained will be reported in the
   unconstrained paths section(s) of the report.

================================================================================
Timing constraint: TS_clk_0 = PERIOD TIMEGRP "xmplr_clk"  10 nS   HIGH 50.000000 % ;

 9159 items analyzed, 0 timing errors detected. (0 setup errors, 0 hold errors)
 Minimum period is   9.865ns.
--------------------------------------------------------------------------------


All constraints were met.


Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)

Clock to Setup on destination clock clk
---------------+---------+---------+---------+---------+
               | Src:Rise| Src:Fall| Src:Rise| Src:Fall|
Source Clock   |Dest:Rise|Dest:Rise|Dest:Fall|Dest:Fall|
---------------+---------+---------+---------+---------+
clk            |    9.865|         |         |         |
---------------+---------+---------+---------+---------+


Timing summary:
---------------

Timing errors: 0  Score: 0

Constraints cover 9159 paths, 0 nets, and 4023 connections

Design statistics:
   Minimum period:   9.865ns (Maximum frequency: 101.368MHz)


Analysis completed Mon Apr 11 10:37:14 2005
--------------------------------------------------------------------------------

Peak Memory Usage: 83 MB
```

Where you note that LS was a bit optimistic. What you will also note is that Xilinx will not output a critical path unless a timing violation occured, in which case it will dump only a few of the violating paths.

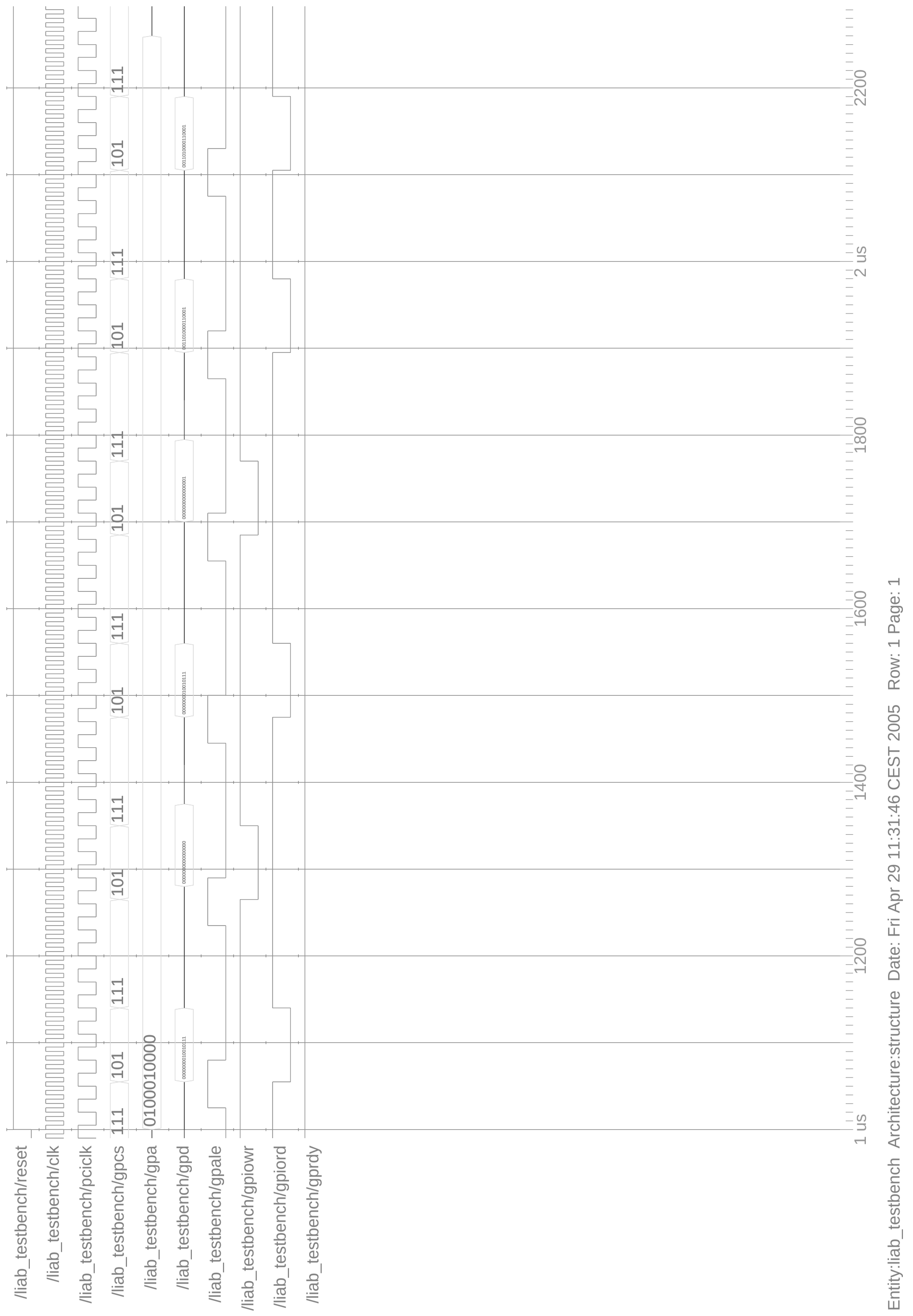
Using the area report from LS, we get an idea of the size of the design:

```
*******************************************************

Cell: LiAB_testinout    View: structure    Library: work

*******************************************************


 Cell                                             Library  References     Total Area

BUFGP                                             xcve     1 x      1      1 BUFGP
BUFT                                              xcve    32 x      1     32 LUTs
FDC                                               xcve    24 x      1     24 Dffs or Latches
GND                                               xcve     1 x      1      1 GND
IBUF                                              xcve    17 x      1     17 IBUF
IOBUF                                             xcve    16 x      1     16 IOBUF
LUT1                                              xcve     4 x      1      4 Function Generators
LUT1_L                                            xcve     8 x      1      8 Function Generators
LUT2                                              xcve     3 x      1      3 Function Generators
LUT2_L                                            xcve    16 x      1     16 Function Generators
LUT4                                              xcve     4 x      1      4 Function Generators
LiAB_Interface_0_0100010000_0100010010_101_notri  work     1 x     24     24 MUXF5
                                                                   24     24 RAMB4_S16_S16
                                                                  356    356 Function Generators
                                                                   30     30 XORCY
                                                                   44     44 MUX CARRYs
                                                                   28     28 MULT_AND
                                                                  270    270 Dffs or Latches
                                                                  384    384 LUTs
                                                                    3      3 VCC
                                                                    3      3 GND
                                                                  339    339 gates
MUXCY_L                                           xcve    22 x      1     22 MUX CARRYs
OBUF                                              xcve    88 x      1     88 OBUF
VCC                                               xcve     1 x      1      1 VCC
XORCY                                             xcve    24 x      1     24 XORCY

 Number of ports :                     124
 Number of nets :                      320
 Number of instances :                 262
 Number of references to this view :     0

Total accumulated area :
 Number of BUFGP :                       1
 Number of Dffs or Latches :           294
 Number of Function Generators :       391
 Number of GND :                         4
 Number of IBUF :                       17
 Number of IOBUF :                      16
 Number of LUTs :                      416
 Number of MULT_AND :                   28
 Number of MUX CARRYs :                 66
 Number of MUXF5 :                      24
 Number of OBUF :                       88
 Number of RAMB4_S16_S16 :              24
 Number of VCC :                         4
 Number of XORCY :                      54
 Number of gates :                     370
 Number of accumulated instances :    1427
***********************************************
Device Utilization for v812ebg560
***********************************************
Resource                Used    Avail   Utilization
-----------------------------------------------
IOs                     123     404      30.45%
Function Generators     391     18816     2.08%
CLB Slices              196     9408      2.08%
Dffs or Latches         294     20028     1.47%

-----------------------------------------------
```

E Timing diagrams



Test-1 from the unsynthesized design. Important to note is the fact that our LiAB-emulator follows protocol and we actually read 00000000 10010111 first and second, followed by two reads of 01110100 01110001.



Test-1 from the synthesized design. Important to note is the fact that our LiAB-emulator follows protocol and we actually read 00000000 10010111 first and second, followed by two reads of 01110100 01110001.

F CD-ROM with sources

The CD-ROM contains:

- `/rip_pins/`
  CSV and UCF files
- `/rip_fpga/`
  VHDL code for the design described in section 4
- `/pcb_adders/`
  A very simple design consisting of an adder outputted to the HP-connectors to see if the FPGA can be programmed. There was also some blinkenlichten that did not work.
- `/pcb_liab0/`
  The `LiAB_testinout` entity synthesized and PaR. Contains a Xilinx project ready to load and program.
- `/pcb_liab0quar`
  The `LiAB_quartest` entity synthesized and PaR. Contains a Xilinx project ready to load and program.
- `/pcb_reliplan_8`
  Synthesized version of Reliplan with 8 entries in the quarqueue. Barely meets timing req.
- `/pcb_reliplan_16`
  ynthesized version of Reliplan with 8 entries in the quarqueue. Does not meet timing req.