Department of Computer Science
Embedded Systems Group

Master Project

Implementation of SCAD processor

Julius Roob
Mahircan Gül
Maisum Haider
Subash Kannoth

supervised by
Tripti Jain

April 30, 2016

Abstract
Write Abstract here

Contents

Todo list

1 Instruction Set Architecture

Like the Transport Triggered Architecture (TTA), the main feature of the SCAD machine instruction set is the move instruction. Moves happen from the output buffer of one Functional Unit (FU) to the input of another. Those moves have an order given by the program, and all parallel or out-of-order execution is done transparently by the hardware.

While all operations are performed by moving data between functional units, some initial data is required, for example the addresses of where inputs and results are located in memory. Two possible means of getting that initial data to the functional units were considered in design. Those two were to either have dedicated FIFOs for inputs and outputs of the processor and program, or extend the ISA by instructions to load values for the data network. For the first approach, all constants that are required for a program to run have to be made available through one or more FIFOs or ROMs that outputs data into the data network when a corresponding move instruction is issued. The second, which is inspired by the bachelor thesis of Sebastian Schumb [4], is to add at least one instruction to load immediate values.

To make this design as simple to implement as possible, it was decided to have the control unit be part of the data network like the FUs, and both load immediate values into an output buffer, and take branch conditions from and input buffer. This is explained further in Section **??**.

Add Section reference

So, aside from the mandatory move instruction, the ISA has instructions for loading immediate values, jumping to fixed addresses and branching. An overview of those instructions is given in Figure 1.

Example programs can be found in the Appendices A and B.

instruction	semantics
move src, dest	Move data from an output buffer to an input buffer by sending this instruction to the corresponding functional units. Data move will asynchronous
jump address	Set PC to address.
immediate data	Place data into output buffer of control unit.
branch address	A no-op when the first data in the input buffer is a 0, jump to address otherwise. Will wait for data to arrive when there is none.

Figure 1: SCAD Architecture Instructions

2 Move Instruction Bus

2.1 2-Phase Commit

To take into account both stalls of source and destination functional units, the control unit sends move instructions in two phases, both of which are indicated by a rising edge of the ”valid” flag. First, with phase low, the functional units only check whether there is space in the corresponding input and output buffers. When stalls are asserted, the control unit waits some time until retrying. When no functional unit stalls, the phase being high signals a ”write”.

Either the text, the diagram and/or the implementation need to be adapted (not in sync).

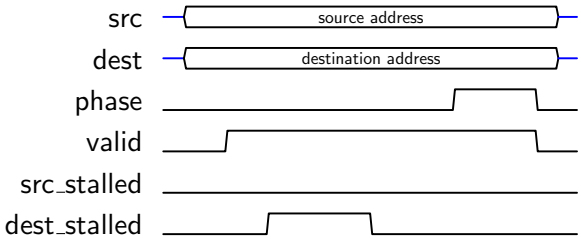

Figure 2: 2-Phase Commit on Move Instruction Bus with Destination Stalling

3 Control Unit and Data Network

remove this section? -julius

To keep the MIB simple, we will have the control unit send immediate values, and receive branch conditions through the data network. While broadcasting immediate values through either the MIB or a separate bus may be the faster alternative, having the control unit send them through the data network keeps the architecture cleaner.

better formulation please

4 Data Trasport Network - DTN

The Data Trasport Network is the core part of the SCAD architecture. The 32 *Fuction-Units* are identified by its physical address in the architecture which ranges from 0 to 31. Each *Function-Unit* has output that can send the *Data-packets* to any other *Function-Unit* in a given point of time. The *Data-Packet* structure is shown in the Figure 3. The intention is to get all the *Data-packets* deliverd to the corresponding target *Function-Units* in minimum time, possibly constant time. A constant time means the time taken is deterministic. It can be several clock cycles and it will be the same for any permutations of set if input addresses.

A *Cross-Bar* switch network is a possible solution with a gurantee of constant time delivery of *Data Packets* to the target, but we have to compramise the resource consumption which makes it fairly complicated for 32 *Function-Units*. In other words the $32X32$ cross connection itself will consume a major part of the resources of the complete system, which is not efficient. Another possible solution for this is to use memory mapped bus. But in this case, again it does not guarantee a constant time delivery to the *Function-Units*, since there will be cases where we have to prioritize the *Data-Packets* based in addresses possibly by using an Arbeiter.

As a tradeoff between resource consumption and time to deliver a *Data-Packet*, a Bitonic Network and a Beneš Network are indentified as good choices among many parallel sorting networks. We implemented the DTN with a Bitonic as well as with a Beneš Network. A Bitonic Network sort any possible permutations in of input in constant time which makes it an excellent choice as a network router. Basically all the *Fuction-Units* can be connected to each outputs of a *Bitonic sorter* with respect to the physical address and thus Bitonic sorter acts network router. Before going deep on the actual implementation we would like to give a short introduction about Bitonic sorter.

4.1 Bitonic Network

A Bitonic Sort[1] is a comparison based parallel sorting algorithm. A random input sequence in first converted into a Bitonic sequence , which monotonically increases and then decreases thus the name

Figure 3: Data Packet struture

Bitonic. The rotations applied to a Bitonic sequence is also Bitonic.The random input is coversion to Bitonic sequence is achieved with a *Bitonic-Splitter*.
Let

$$S = \langle x_0, x_1, ..x_{n-1} \rangle$$

be Bitonic sequence such that

$$x_0 \leq x_1 \leq ... \leq x_{n/2-1} \quad \text{and} \quad x_{n/2} \leq x_{n/2+1} \leq ... \leq x_{n-1} \quad \text{holds}$$

Consider the following subsequences

$$S_1 = \langle min(x_0, x_{n/2}), min(x_1, x_{n/2+1}), .., min(x_{n/2-1}, x_{n-1}) \rangle$$

$$S_2 = \langle max(x_0, x_{n/2}), max(x_1, x_{n/2+1}), .., max(x_{n/2-1}, x_{n-1}) \rangle$$

with the following property.

$$\forall_x \forall_y . x \in S_1 \wedge y \in S_2 \quad x < y$$

Both S_1 and S_2 are Bitonic. A sorted sequence is produced as a result of applying S_1 and S_2 recursively. The above procedure is called *Bitonic Split*. Firstly *Bitonic Split* is performed in the input random sequence which trasforms any given sequence to a Bitonic sequence which is then fed to the *Bitonic Merge* network. The *Bitonic Merge* converts the splitted sequence to sorted sequence. sequence. A 16 input Bitonic sorter configuration is shown in the Figure 4. In our case we expand the same configuration to a 32 input sorter . The outputs z_0 to z_{31} will be connected to the corresponding *Function-Units* with respect to the physical address. The arrow indicates a Comparator. The up-down arrow indicates an ascending comparator and the down-to up indicates a descending one. Both the comparator element configurations are shown in Figure 5. An N input Bitonic Network consists of $O(N.log_2(N)^2)$ comparators and has a combinatorial depth of $O(log_2(N)^2)$.

4.1.1 Bitonic Network as Network Router

The Bitonic Network is only a sorting network, does not have enough intelligence to act it as a network router in practical cases. Everything works well when all *Function-Units* are ready to send the *Data-Packets* to its target. But this is not the case most of the times. Many *Function-Units* can still be in a state where its outputs are not ready, at the same time some of the *Function-Units* are ready with their outputs. In this case Bitonic Network can fail in routing the *Data-packets* to the right destinations because all that are not ready can feed an unknown *Target-Address*. Two solutions were identified as the following.

4.1.2 Bitonic Network with additional Routers

This is a proposed solution before realizing the Bitonic-Banyan [2](Batcher's Banyan) network. In this case the invalid *Target-Address* problem is sorted out by adding and two extra stages at the input

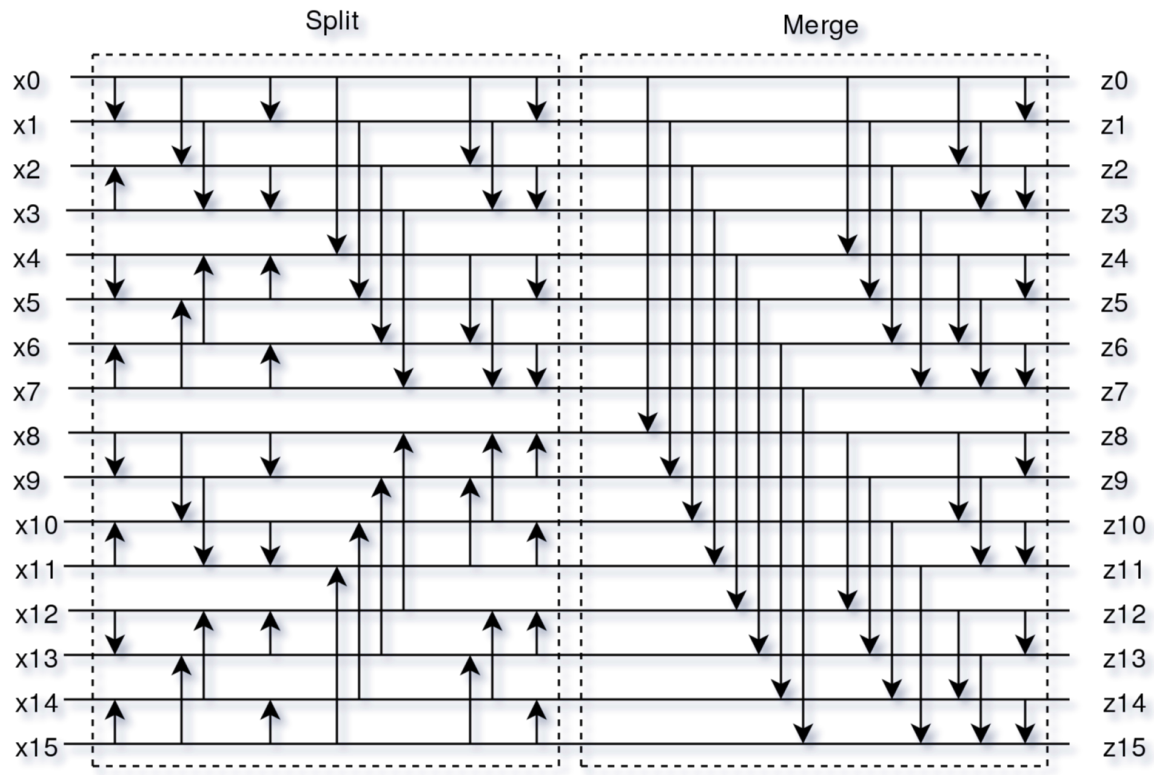

Figure 4: A 16 input BitonicNetwork

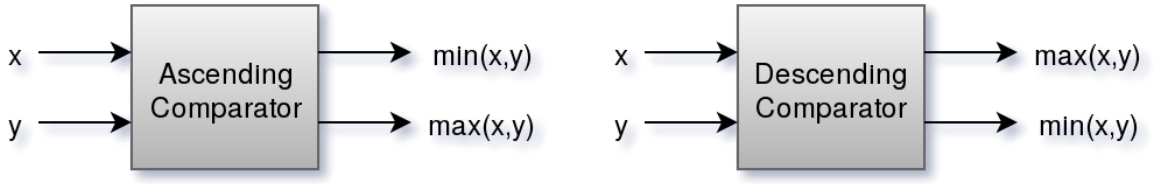

Figure 5: Basic 2 x 2 comparator elements of a Bitonic sorter

and output of the Bitonic Sorter. At input we add an *Address-Resolver* module for each comparators. In the architecture an invalid addresses is identified by a *VALID* bit in the *Data-Packet* (Figure 3). All *Functional-Unit* writes a *LOW* to the *VALID* bit of its output packets in every clock cycle unless an packet is ready. In this way the network can interpret the validity of the *Data-Packet* and act accordingly. The *Address-Resolver* feeds the input of each comparator with a predefined hardcoded address according to the position of the comparator. Figure 6 shows a implementation of the Address-Resolver. The *Address-Resovler* is a simple combinatorial multiplexer which switches the address based on the *VALID* bit of address and is connected to all the N comparators of input. For an N input Bitonic network the hardcoded address is defined as $N-1+C$, where C is the position of the the comparator which ranges from 1 to N. This ensures the comparator is fed with an address which is greater that $N-1$, which enables the sorter to work even if some of the inputs are not ready. In other words the *Address-Resovler* switches the input of the Bitonic network to valid address at the instant when the *VALID* bit is *HIGH*. One more

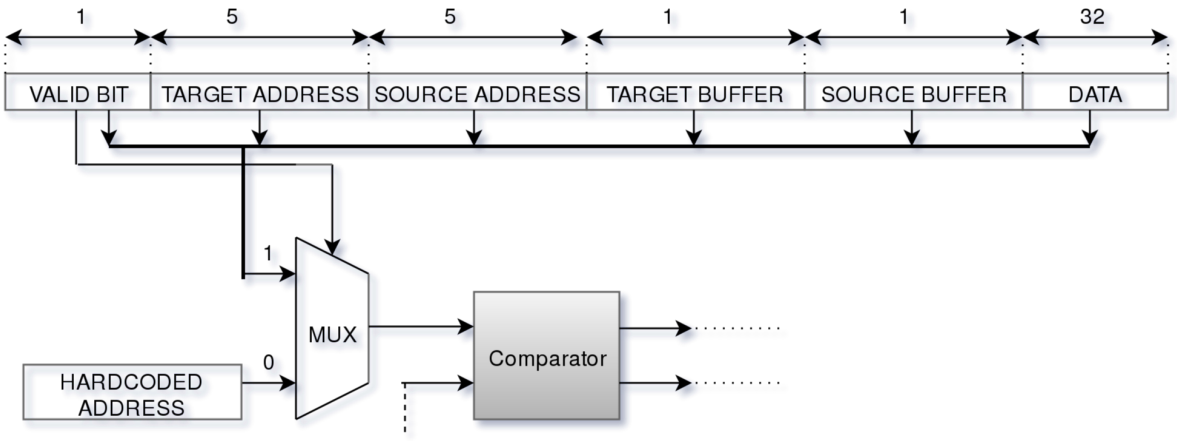

Figure 6: Address resolution

problem that to be resolved still is the output of the network. The sorter will sort the *Data-Packets* with all the hardcoded addresses positioned at last of the address sequence but still the actual addresses can be routed into wrong destinations. For eg: If the input is an addresse sequence $\langle 31, X_1, X_2, .., X_{31} \rangle$ (where X_i indicates invalid address) which results in the output address sorted sequence of $\langle 31, 32, 33, .., 63 \rangle$. The routing is wrong since the 31 is routed to target 0. To resolve this we have to use a stage of routers at the output of the network. The routers have forward and reverse routing path and *Data-packet* is routed to either forward or reverse path based on the distance to the target. The distance to each destination is hardcoded in a routing table which enables faster decision making. The configuration is shown in Figure 7 .The drawback is a *stall* signal is required to stop the network to take new Data-packets until the old ones are delivered to the corresponding target Functional-Units. The worst case time for a packet to reach the target from the router network will be $N/2$ .In other words in the worst case we have to stall network for $N/2$ clock cycles without doing anything useful in that time. The best case would be when all the inputs are valid. Besides the circuit is sequencial and the *stall* is active which degrades the DTN performance. Also a router element is a complicated state machine which consumes considerable amount of resources when we have 32 instances of the same. This fact lead us to use a Bitonic-Banyan[2] cascaded network which resolves this address permutation problem in constant time.

4.1.3 Bitonic-Banyan Network

We utilize the self routing property of a Batcher-Banyan network [2] when the invalid Target-addresses are found at the input. The configuration contains a Bitonic Sorting Network cascaded with a Banyan routing network. As in the case of a Bitonic Network we use the 2 X 2 switching element(comparator) but the connection state of this element is determined by the destination tags of the input, in our case the Target address. A BitonicNetwork can is capable or realizing arbritary permutations of the inputs. But still the routing is not proper with incomplete Data-packet at the inputs. Here comes the use of Banyan Network which can route the sorted outputs to appropriate Target addresses. As mentined before, special kind of modified switching elements are used which is quite differant from the configuration of normal Bitonic comparators. The switching elements are added with some extra logic to route the larger target address Data-packets to the direction pointed by the arrow. Fig.6 depicts the switch configuration in different possible scenarios. An 'X' indicates incomplete inputs. This switch basically moves all the X

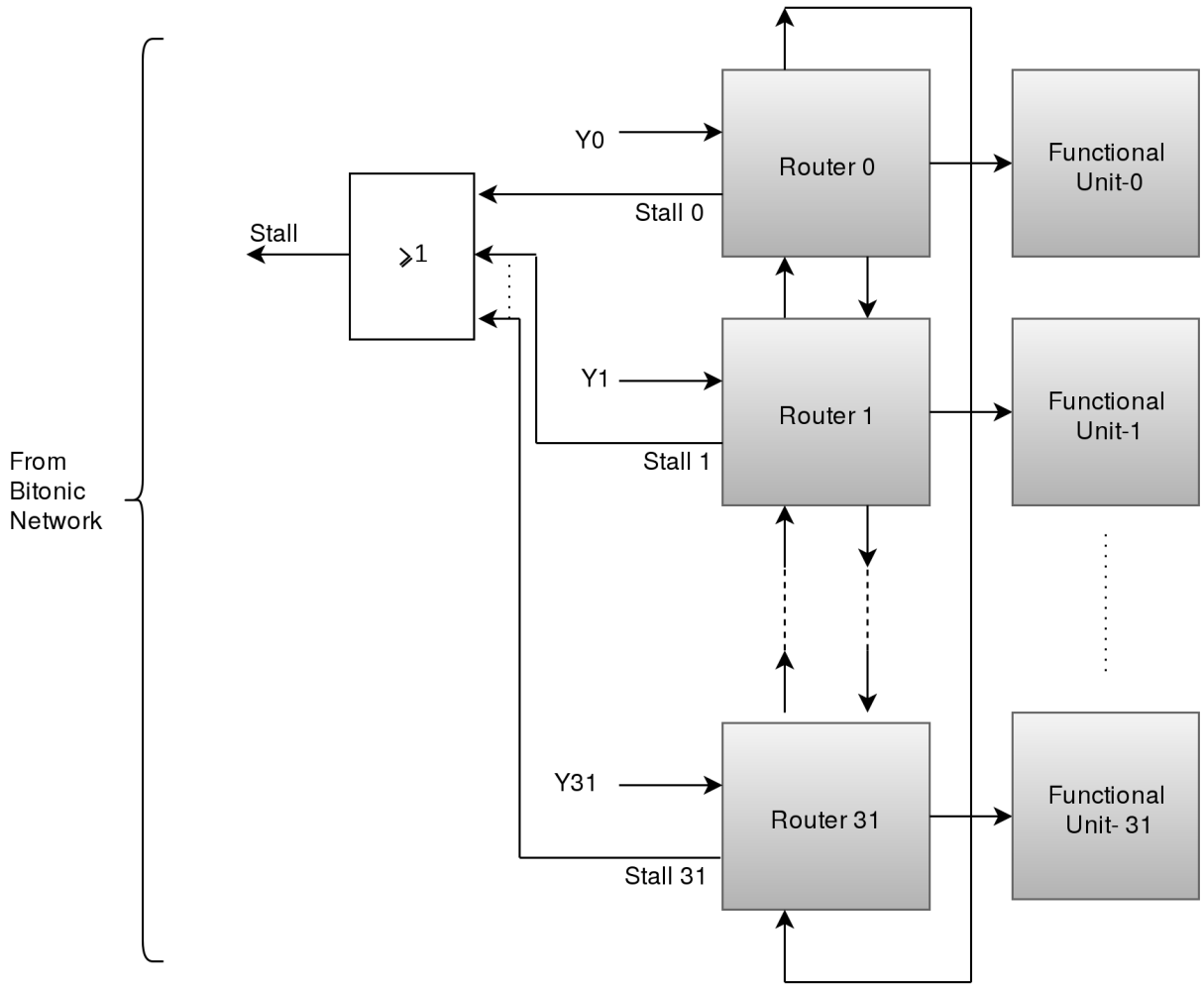

Figure 7: Router Network

inputs to the bottom of the list. Thus for any input sequence of Data-Packets with $U_0, U_1, .., Ur-1$ are the invalid addresses, the Bitonic network will produce an output sequence of $A_0, A_1, .., A_K, .., U, U, ..U$, where $K = 32 - r - 1$. Figure 8 shows the modified switching element for the Bitonic Network of the SCAD machine. The Normal-switch is an ascending comparator which is shown in Fig.5 and the Modified swich the one which is shown in Figure 9. which handles our situation. The next stage is a Banyan Network which takes up the sorted sequences by the BitonicNetwork and route to the proper destination.

It is proven [2] that an Banyan network can completely route a sorted list when the incomplete inputs appear either at high end or low end of the list. We have already moved all the incomplete Data-Packets to the lower end of the sequence with the help of modified switch. An 8 input Banyan Network configuraion is shown in Figure 10. We simply expand the same to 32. The Data-Packets in the Banyan Network is based on the i^{th} bit of the Target-Adress, where i is the stage index. For an N input Banyan Network we have $log_2(N)$ stages and combinatorial circuit depth.A Banyan Newtwork is collision free if the input sequences are sorted ascending thus in our case gurantees the delivery of Data-Packets at proper

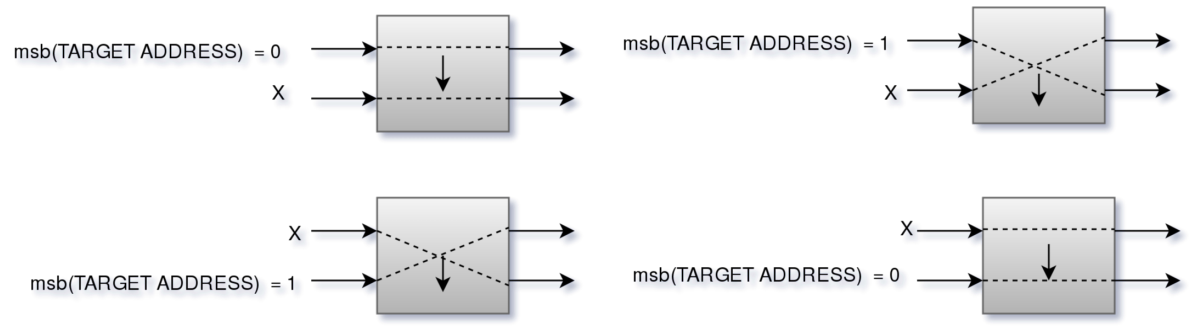

Figure 8: Modified switches for handling incomplete Data-packets in Bitonic Network

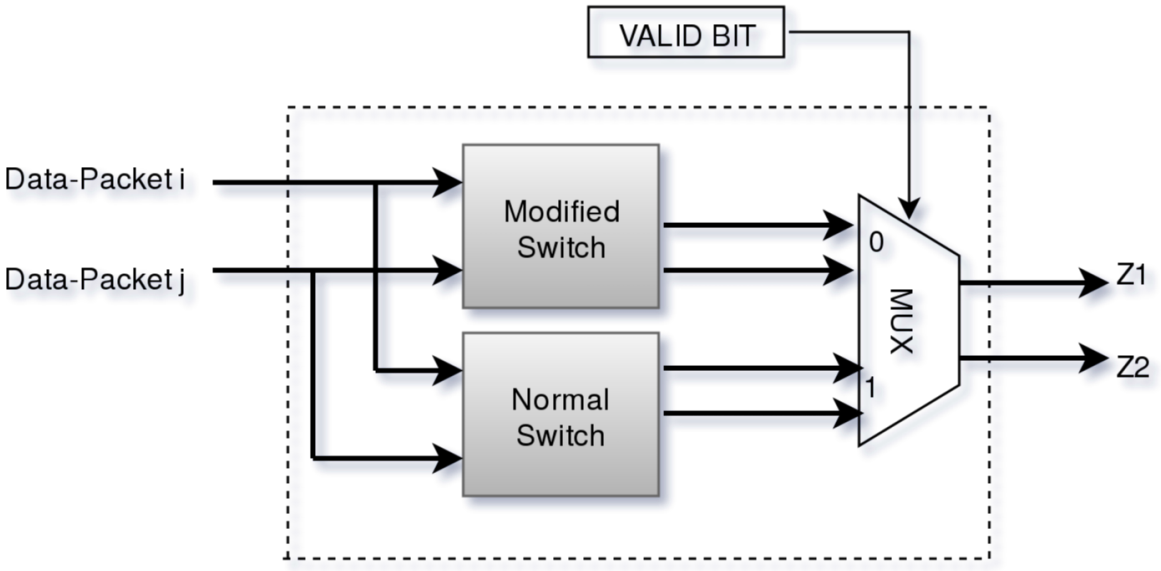

Figure 9: Swiching element of Bitonic Network

destination since the Bitonic Network in the previous stage will already produce an acending sequence. In our case the sequence can also have invalid Data-Packets which is in the tail of the sequence. Different scenarios in handling of incomplete input for a Banyan switch is shown in Figure 11. A 'X' corresponds to incomplete inputs. The valid inputs are the bit position of the stage i of the Target-Address. Beased on this bit the Data-packet is routed up or down. It has been proven [3] that for $N > 8$ the latency due to combinatorial depth of a Bitonic Network is more that when it is piplelined. So we have implemented 5 stage pipleline for th Bitonic Network and a 5 stage for the Banyan Network.Altogether for the SCAD machine the synchronous Bitonic-Banyan version of DTN is shown in Figure 12.

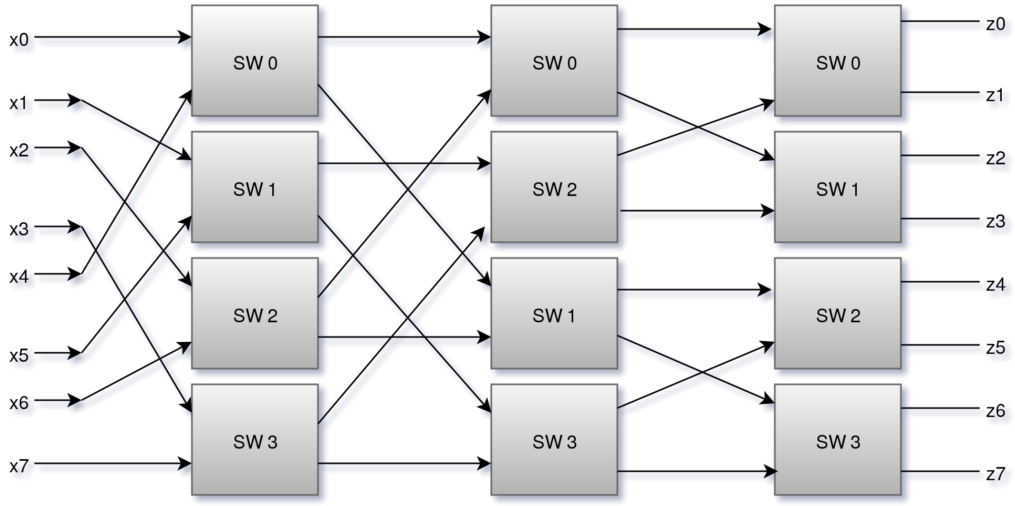

Figure 10: 8 input Banyan network

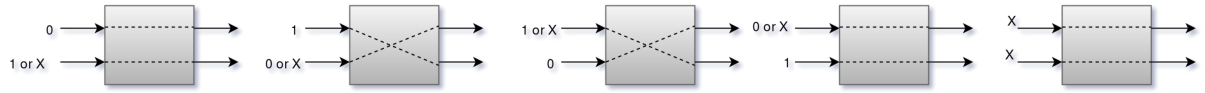

Figure 11: Modified switches for handling incomplete Data-packets in Banyan Network

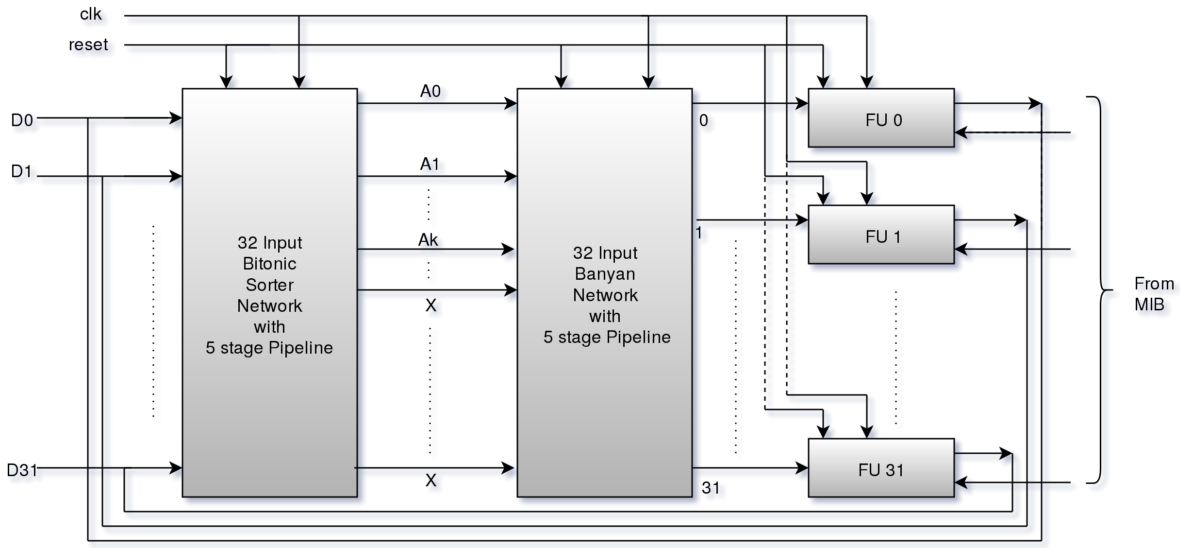

Figure 12: DTN of SCAD machine

4.2 Beneš Network

The Beneš network is an extension of the Banyan network. It solves the problem that a Banyan network is not able to produce all possible permutations by mirroring the Banyan network and connecting the

two. An example for a Beneš network with 8 inputs is given in Figure 13.

The two halves of the network are different in the routing procedure employed. For the output half, the destination of a data packet defines the exact setting of every switch on its path, because there always is only one possible choice. The input half is responsible for conflict avoidance, which has a significantly higher complexity due to the fact that it depends on all inputs of one column of the network.

For implementation, the recursive structure given in Figure 14 was used. A network with 2^n inputs is defined as $benes(n,n)$, where the first parameter gives the size of the specific network defined, and the second is used during recursion to remember the size of the whole network.

For this structure, the routing decision of the output switches is based on just bit $k-1$, where a 0 requires the packet to go up and 1 to go down.

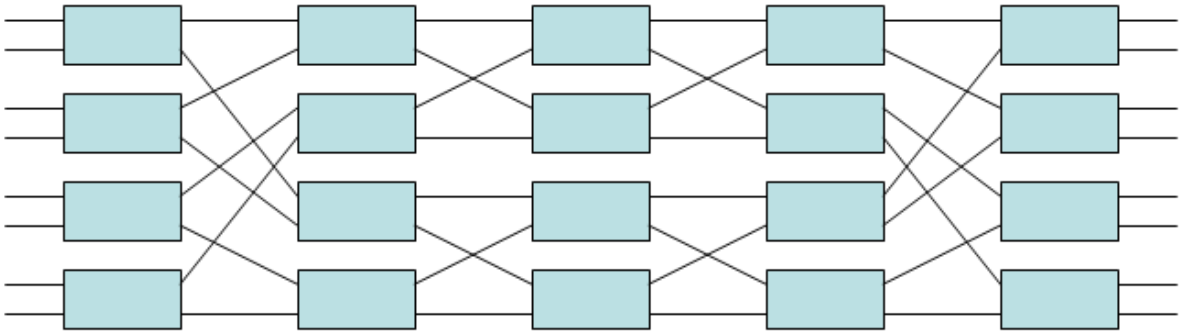

Figure 13: The basic structure of a Benes Network
(source: https://commons.wikimedia.org/wiki/File:Benesnetwork.png)

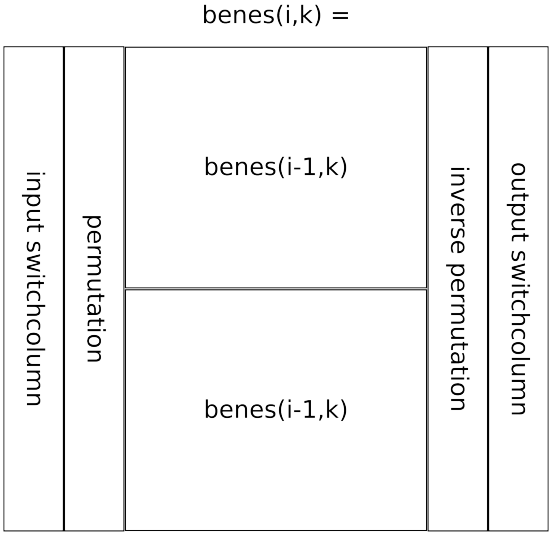

Figure 14: recursive definition of a benes network

5 Future Work

Having duplication as a seperate functional unit may cause it to be a significant bottleneck. One feasible solution to this is the extension of the move instruction by an additional ”non-destructive” move, where data sent is kept in the output buffer of the sender.

at least two more ideas

6 Bibliography

References

[1] K. E. Batcher (1968): *Sorting networks and their applications*. in 1968 Spring Joint Computer Conf., AFIPS Proc., vol. 32, pp. 307-314.

[2] Madihally. J. Narasimha (1988): *The Batcher-Banyan Self-Routing Network: Universality and Simplification*. IEEE TRANSACTIONS ON COMMUNICATIONS, VOL. 36, NO. 10, OCTOBER 1988.

[3] Gustavo Alonso Rene Mueller, Jens Teubner (2012): *Sorting networks on FPGAs*. The VLDB Journal February 2012, Volume 21, Issue 1, pp 1-23.

[4] S. Schumb (2015): *Hardware Generation for Transport Triggered Architectures*. Master's thesis, Department of Computer Science, University of Kaiserslautern, Germany. Bachelor.

Appendices

A Memory Access and Branch

Basic example for memory access and branching:

```
1  // Basic function:
2  // *result = *op1 == *op2 ? 33 : 27;
3
4  // Load operands from memory
5  immediate <op1_address> // op1_address into the control unit output
6  move ctrl.o0, load.i0
7  immediate <op2_address>
8  move ctrl.o0, load.i0
9
10 // Send result destination to "store" function unit
11 immediate <result_address>
12 move ctrl.o0, store.i0
13
14 // Send parameters to compare unit
15 move load.o0, cmp.i0
16 move load.o0, cmp.i1
17
18 move cmp.o0, ctrl.i0 //move to control unit input for branch
19 branch yes // branch to yes if control unit input != 0
20 no:
21   immediate 27
22   jump both
23 yes:
24   immediate 33
25 both:
26   move ctrl.o0, store.i1 // move to data input of the store unit
```

B Fibonacci

```
1  setup:
2    immediate 0
3    move ctrl.o0 duplication.i0
4
5    immediate N
6    move ctrl.o0 add.i0
7    move duplicaton.o0, add.i1
8
9    immediate 1
10   immediate 0
11   move ctrl.o0 add.i0
12   move ctrl.o0 add.i1
13
14 loop:
15   // Loop invariant: * output buffer of add contains: (n-i)
16   //                                                  then fib(i)
17   //                 * output buffer of duplication contains: fib(i-1)
18
19   immediate -1
20   move ctrl.o0, add.i1
21
22   move duplication.o0, add.i1
23   // add.i1: -1, fib(i-1)
24   // add.o0: (n-i), fib(i)
25
26   move add.o0, duplication.i0
27   // add.i1: -1, fib(i-1)
28   // add.o0: fib(i)
29   // duplication.o0: (n-i), (n-i)
30
31   move duplication.o0 ctrl.i0
32   // add.i1: -1, fib(i-1)
33   // add.o0: fib(i)
34   // duplication.o0: (n-i)
35   // ctrl.i0: (n-i)
36
37   move duplication.o0 add.i0
38   // add.i1: fib(i-1)
39   // add.o0: fib(i), (n-i-1)
40   // ctrl.i0: (n-i)
41
42   move add.o0, duplication.i0
43   // add.i1: fib(i-1)
44   // add.o0: (n-i-1)
45   // duplication.o0: fib(i), fib(i)
46   // ctrl.i0: (n-i)
47
48   move duplication.o0, add.i0
49   // add.o0: (n-i-1), fib(i+1)
```

```
50    // duplication.o0: fib(i)
51    // ctrl.i0: (n-i)
52
53    // takes ctrl.i0 as branch condition
54    branch loop
55
56  finished:
57    // TODO: write values to result place in ram?
```