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

May 3, 2016

Abstract

Current processor architectures offer instruction-level parallelism by duplicating processing resources and handling both instruction and data distribution from a central controller. Although this scheme offers high throughput on non-dependent instructions, existence of central register file cripples the performance on data dependent workloads where several consecutive instructions contend registers, which results in serialization of the machine code and stalls on controller.

In this work, we implement *Synchronous-data asynchronous dataflow (SCAD[1])* architecture, where instructions are issued in order but data movement is handled by processing elements that are connected to each other via data network. Similar to out-of-order superscalar processors, SCAD architecture enforces dataflow order of instructions. However, fundamental difference is the decentralized hazard resolving mechanism, in which the controller does not halt issuing instructions as long as there are enough processing elements available. Instructions are issued in-order (synchronously), while operands and results are moved out-of-order (asynchronously). Data dependencies within a block of operations are resolved with a VLIW-like mechanism without incurring register file contention.

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 [5], 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.

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”.

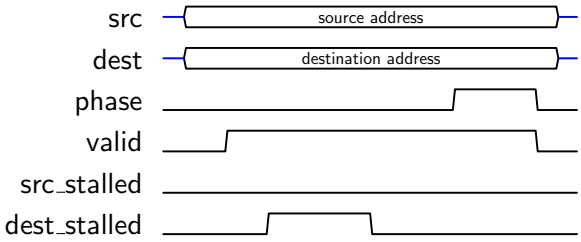


Figure 2: 2-Phase Commit on Move Instruction Bus with Destination Stalling

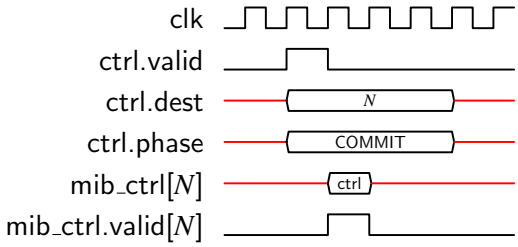
3 MIB Controller

MIB controller is the bridge between control unit and FU network. It establishes communication for both control and status signals, where the former is routed from controller to target and latter is from target to controller, both unidirectionally. Table 1 describes input/output ports.

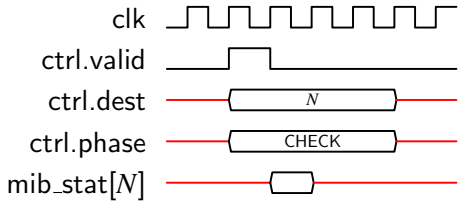
name	**direction**	**type**	**description**
ctrl	input	mib_ctrl_out	control packet from controller. *dest.fu* signal is used to select target. ctrl signal is steered to destination port without modification.
stat	output	mib_stalls	status packet from target. *dest.fu* signal of *ctrl* determines the address of target in which its status to be read.
mib_ctrl	output	mib_ctrl_bus	control signal group. Every FU in network is connected to the bus in ascending order of their addresses.
mib_stat	input	mib_status_bus	status signal group. Status output of FUs in network is connected through corresponding signal

Table 1: MIB ports

Assuming target address N, Figures 3a and 3b illustrate control and status packet transfers from controller to target, and from target to controller, respectively. Since availability of both source and destination is checked by controller before performing control operations, packet is held at the bus for a cycle duration only.



(a) Control packet to target



(b) Status packet from target

Figure 3: Control and status packet transfer

Design is straightforward, consisting of $1 \times MAX\_FUS$ de-multiplexer for controller to network routing and $MAX\_FUS \times 1$ multiplexer for opposite direction. Target functional unit is selected by destina-

tion address field of MIB control input, regardless of the operation type. Due to lack of tri-state ic in FPGA fabric to be used, there is many-to-one relationship for status signals between functional units and control unit. Output of every unit is encoded in bus controller using destination address from control unit as line select. However, the opposite is not necessarily true, since a single line from controller for control packages can be snooped by functional units. For simplicity, we prefer using separate lines for control messages as well, by steering packet from controller to target line using destination address as line select. Bus snooping for control messages will be featured after validating main functionality. Figure 4 shows structure of bus controller.

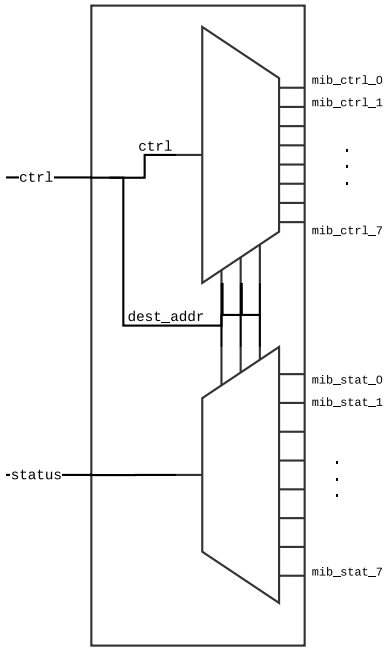


Figure 4: MIB structure

4 Control Unit and Data Network

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.

5 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 5. 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.

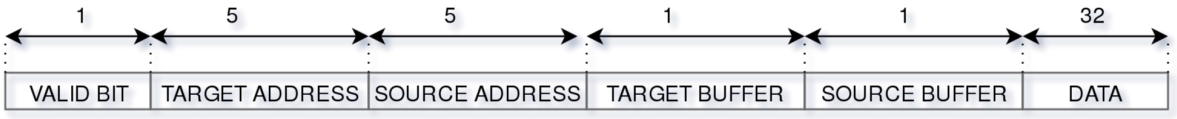


Figure 5: Data Packet struture

5.1 Bitonic Network

A Bitonic Sort[2] 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 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 6. 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

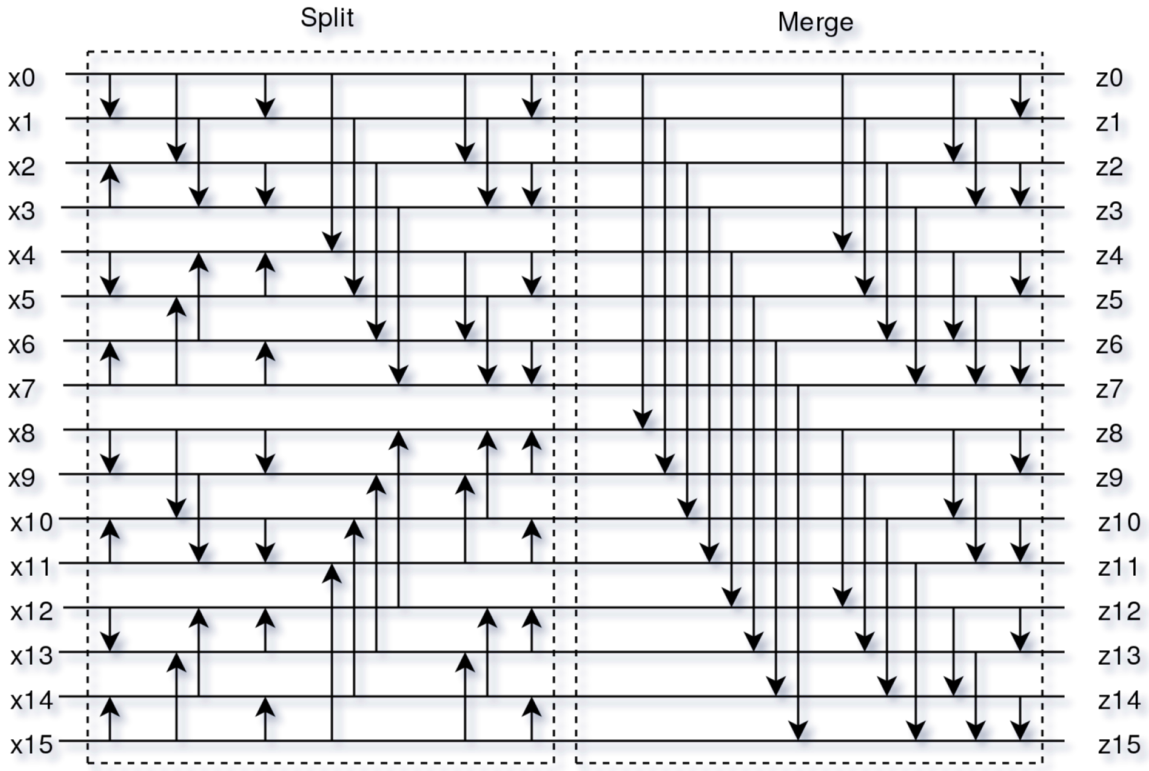


Figure 6: A 16 input BitonicNetwork

comparator and the down-to up indicates a descending one. Both the comparator element configurations are shown in Figure 7. An N input Bitonic Network consists of $O(N.log_2(N)^2)$ comparators and has a combinatorial depth of $O(log_2(N)^2)$.

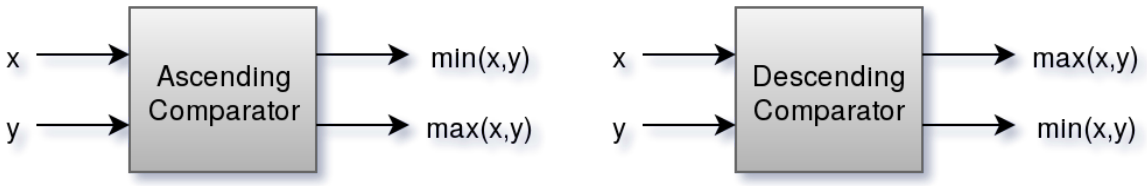


Figure 7: Basic 2*x*2 comparator elements of a Bitonic sorter

5.1.1 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.

5.1.2 Bitonic Network with Routers

This is a proposed solution before realizing the Bitonic-Banyan [3](Batcher's Banyan) network. In this case the invalid *Target-Address* problem is sorted out by adding and two extra stages at the input and output of the Bitonic Sorter. At input we add an *Address-Resolver* module for each comparators. In the architecture an invalid addresses[1] is identified by a *VALID* bit in the *Data-Packet* (Figure 5). 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 8 shows a implementation of the *Address-Resolver*. The *Address-*

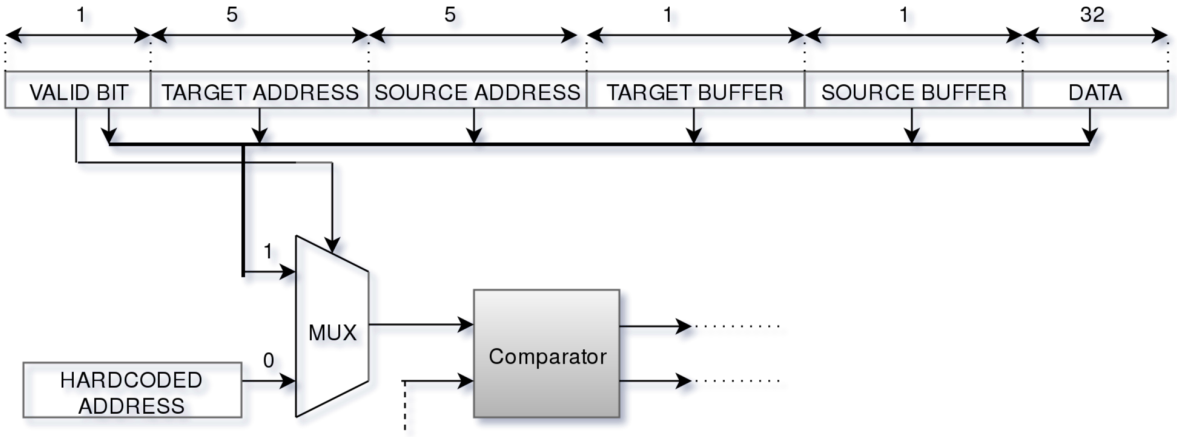


Figure 8: Address resolution

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 comparators are fed with an addresses which is greater that $N-1$, which enables the sorter to work even if some of the inputs are invalid. 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 problem that to be resolved still is the sorted output sequence 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 address sequence $\langle 31, X_1, X_2, .., X_{31} \rangle$ (where X_i indicates invalid address) which results in the sorted output 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 sequencial routers at the output of the network.

[1]An invalid address here means when the a *Function-Unit* has no outputs ready at a given point of time to send to another *Function-Unit*, which can result in any address at the input of the DTN. The *VALID* bit is asserted to a *LOW* in every clock cycle by the *Function-Unit* when not *Data-Packets* are ready.

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 9 .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 *Function-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. This is because the *stall* creates backpressure which will be propegated to all the *Function-Units* and the *Control-Unit*.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[3] cascaded network which resolves this address permutation problem in constant time.

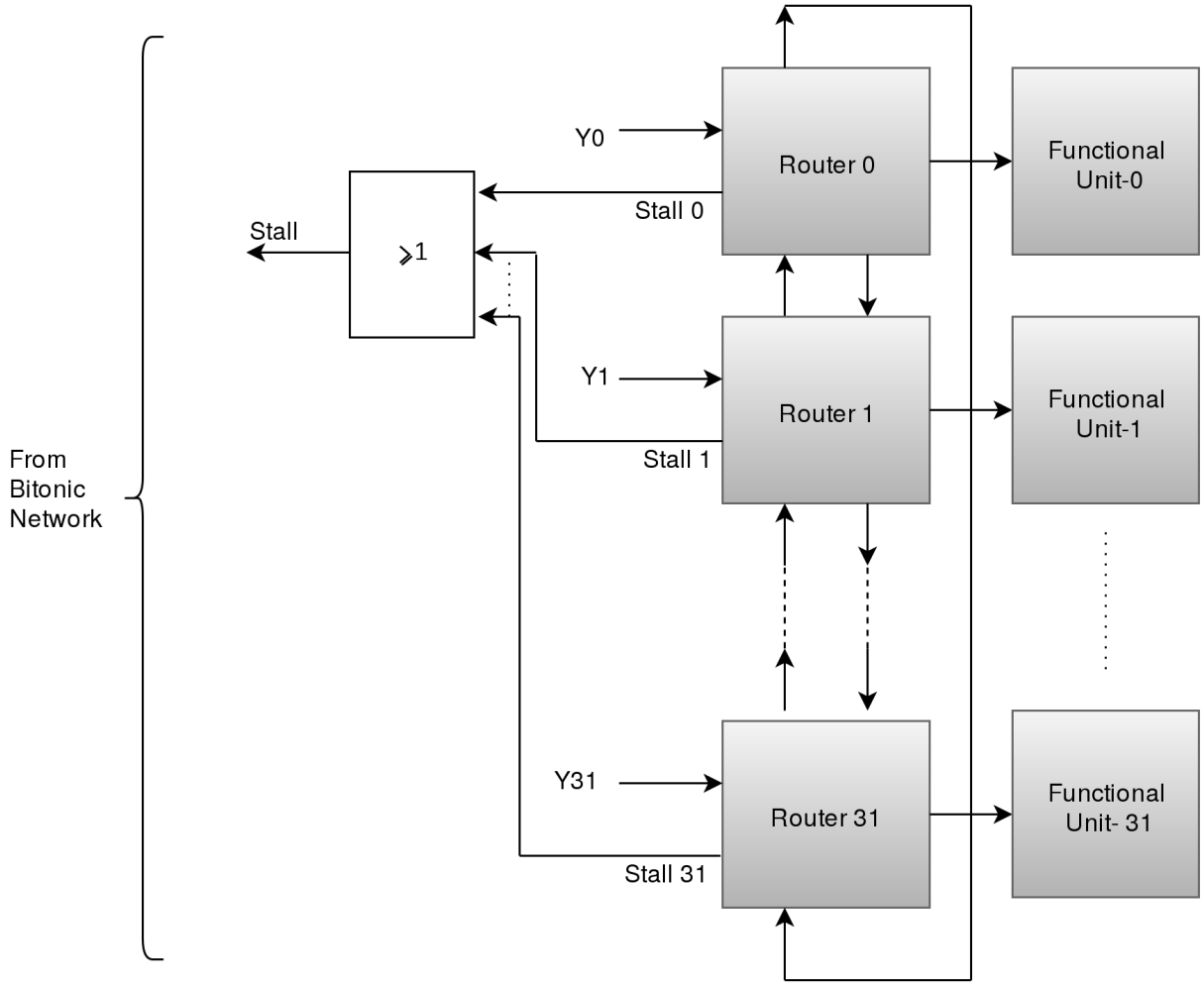


Figure 9: Router Network

5.1.3 Bitonic-Banyan Network

We utilize the self routing property of a Bitonic-Banyan[2] network [3] 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 $2X2$ 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 Bitonic network is capable or realizing arbritary permutations of the inputs. But still the routing is not proper with invalid *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. Figure 10 depicts the switch configuration in different possible scenarios.

An X indicates incomplete inputs. This switch basically moves all the Xs from the input sequence to the bottom of the sequence. Thus for any input sequence of Data-Packets with a set of the invalid addresses $\{U_0, U_1, .., U_{r-1}\}$, the Bitonic network will produce an output sequence of $\langle A_0, A_1, .., A_K, .., U, U, .., U \rangle$, where $K = 32 - r - 1$. Figure 10 shows the modified switching element for the Bitonic network of the DTN. The *Normal-Switch* is an ascending comparator which is shown in Figure 7 and the *Modified-Switch* the one which is shown in Figure 11. which handles the invalid address situation. The next stage

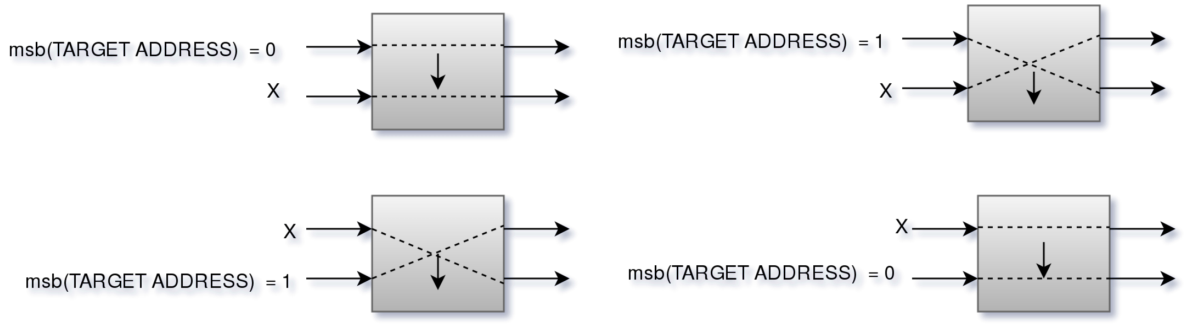


Figure 10: Modified switch handles different scenarios in Bitonic network

is a Banyan Network which takes up the sorted sequences by the Bitonic network and route to the proper destination. Note that a shuffle permutaion is added at the input of the Banyan network. This is shown in Figure 12 for an 8 input Banyan network configuration. The input shuffle permutation is such that two destination tags (*Target-Address* in our case) are different in MSB. We simply expand the same to 32. It is proven [3] that an Banyan network with a the given input shuffle permutation as shown in 12 can completely route a sorted sequence when the incomplete inputs appear either at high end or low end of the list without any conflicts. We have already moved all the incomplete *Data-Packets* to the lower end of the sequence with the help of modified switch. The *Data-Packets* in the Banyan network is based on the i^{th} bit of the *Target-Address*, where i is the stage index. For an N input Banyan network we have $log_2(N)$ stages and combinatorial circuit depth. Also a Banyan newtwork is collision free if the input sequences are sorted ascending thus strongly guaranteeing the delivery of *Data-Packets* at proper destination. 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 13. A X corresponds

[2]A Bitonic-Banyan cascaded network is called a Batcher-Banyan network in the reference paper[3]. We call it as Bitonic-Banyan network all over this paper to give an intuitive meaning how the network is constructed.

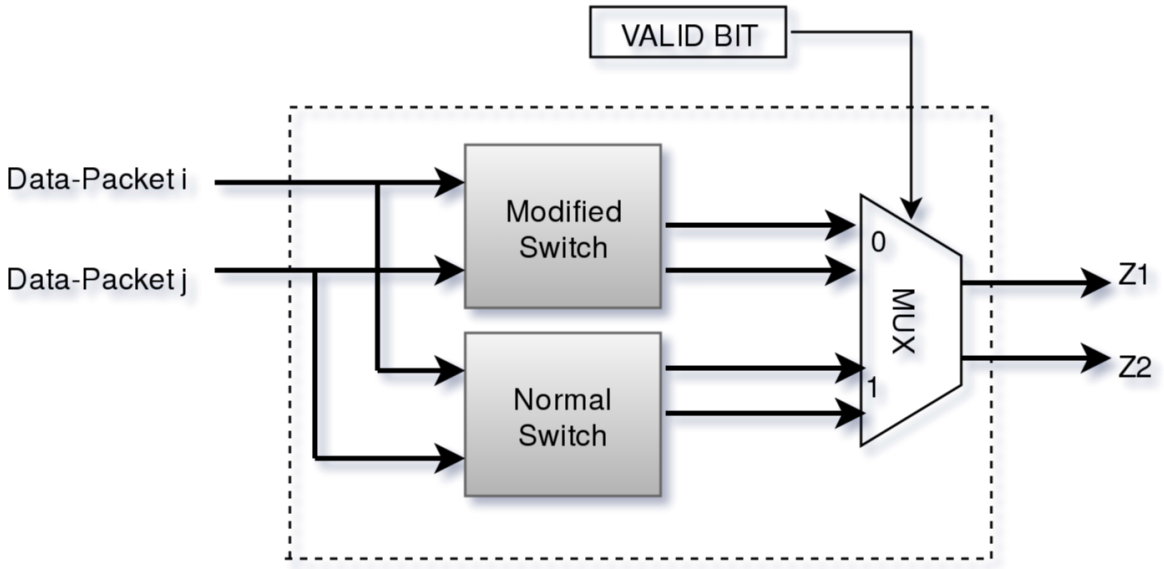


Figure 11: Switching element of Bitonic network

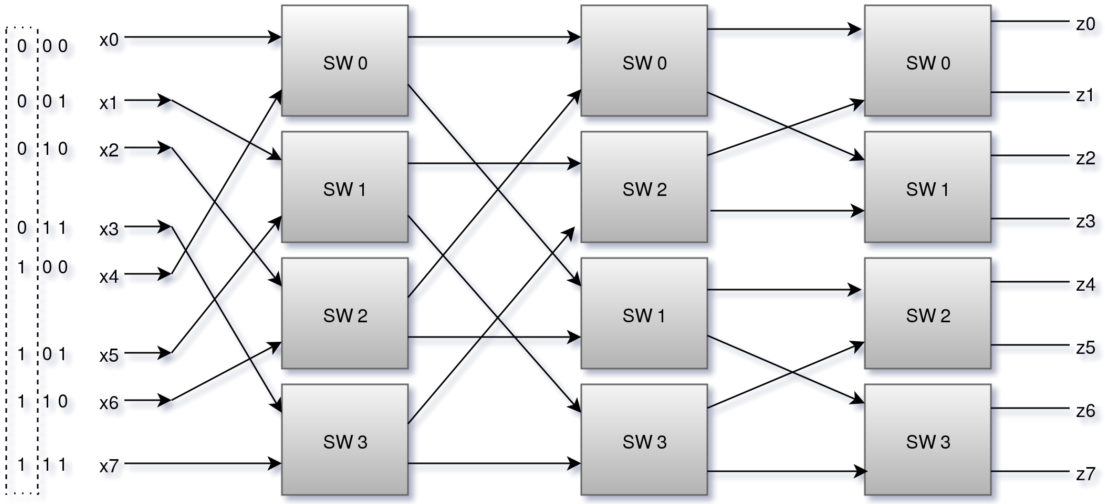


Figure 12: An 8 input Banyan network

to incomplete inputs. The valid inputs are the bit position of the stage *i* of the *Target-Address*. Based on this bit the *Data-packet* is routed up or down. It has also verified with benchmarks [4] that for $N > 8$ the

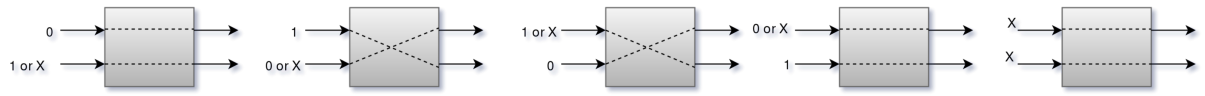


Figure 13: Different scenarios in Banyan network switch

Site Type	Bitonic with routers (%)	Bitonic-Banyan (%)
Slice LUTs	30.85	37.65
LUT as Logic	30.85	37.65
LUT as Memory	0.00	0.00
Slice Registers	11.12	13.53
Register as Flip Flop	11.12	13.53
Register as Latch	0.00	0.00
F7 Muxes	0.00	0.00
F8 Muxes	0.00	0.00

Table 2: Resource utilization of DTN with Bitonic network

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 arhitecture the synchronous *Bitonic-Banyan* version of DTN is shown in Figure 14.

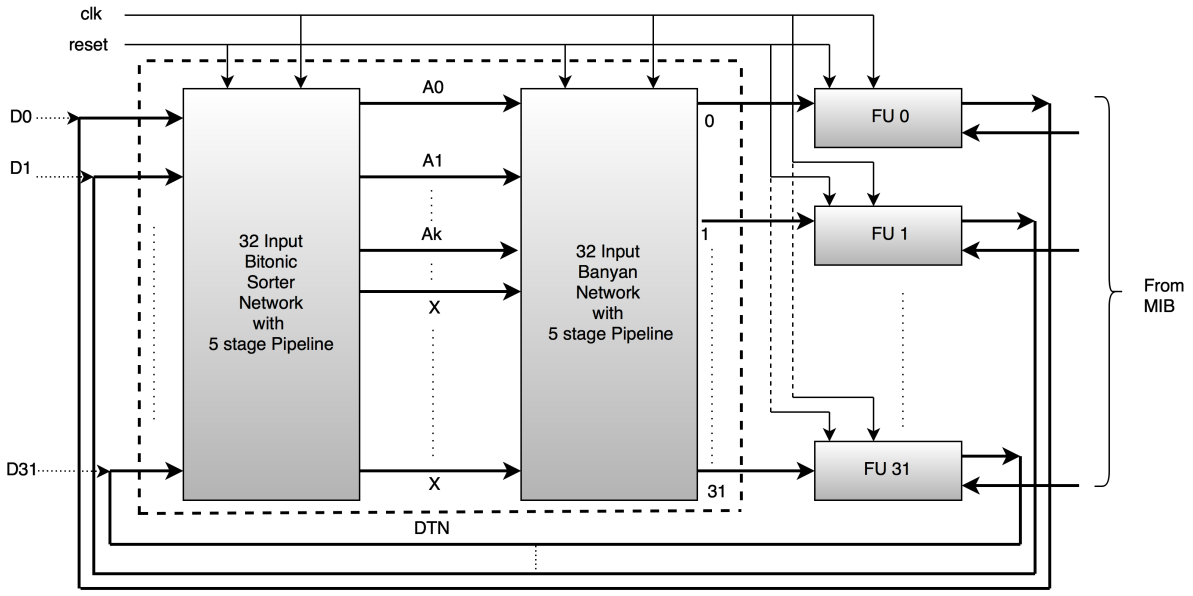


Figure 14: DTN of SCAD arhitecture with Bitonic-Banyan network

5.1.4 Resource utilization

The resource ulitzation on the FPGA shows significant advantage on Bitonic-Banyan over the sequencial implementation with routers. The table 2 shows a comparison table for both the implementation. It is clear to see that the Bitonic-Banyan consumes not much than a the sequencial version , but still has benifit in terms of constant time in delivering *Data-Packets* to its destination.

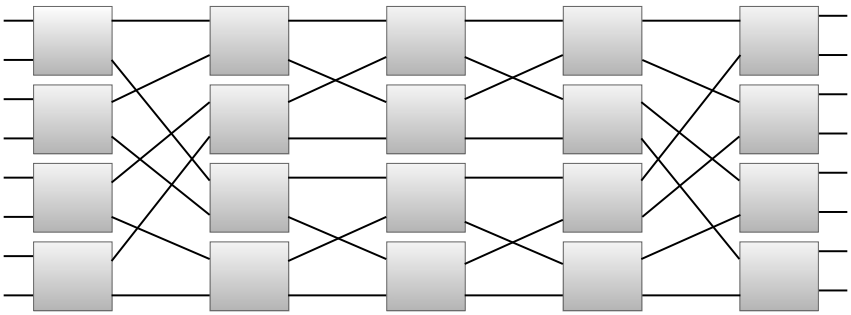


Figure 15: The basic structure of a Beneš network

5.2 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 15.

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.

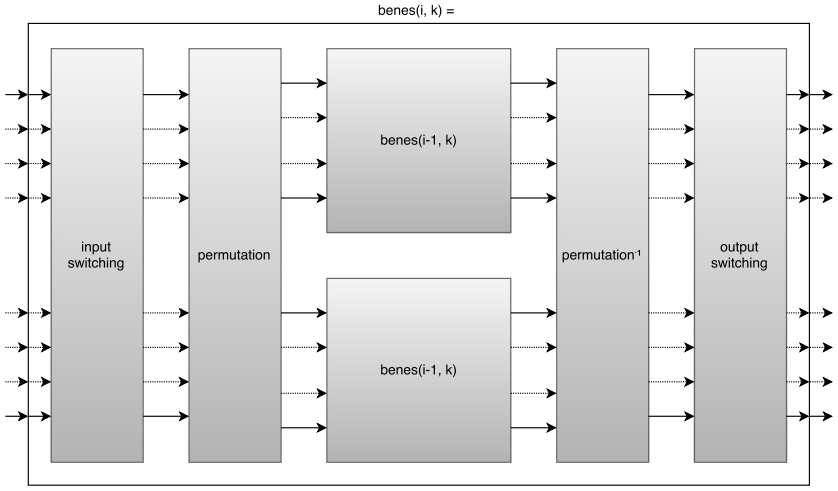


Figure 16: Recursive definition of a Beneš network

For implementation, the recursive structure given in Figure 16 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.

Together with the necessary registers for a pipelined network, the structure of those output switches is given in Figure 17. The second half of the Beneš network, the stalling Banyan, is built using only these switches, and requires only a minimal amount of logic (see Section 5.2.2) for a functionally complete

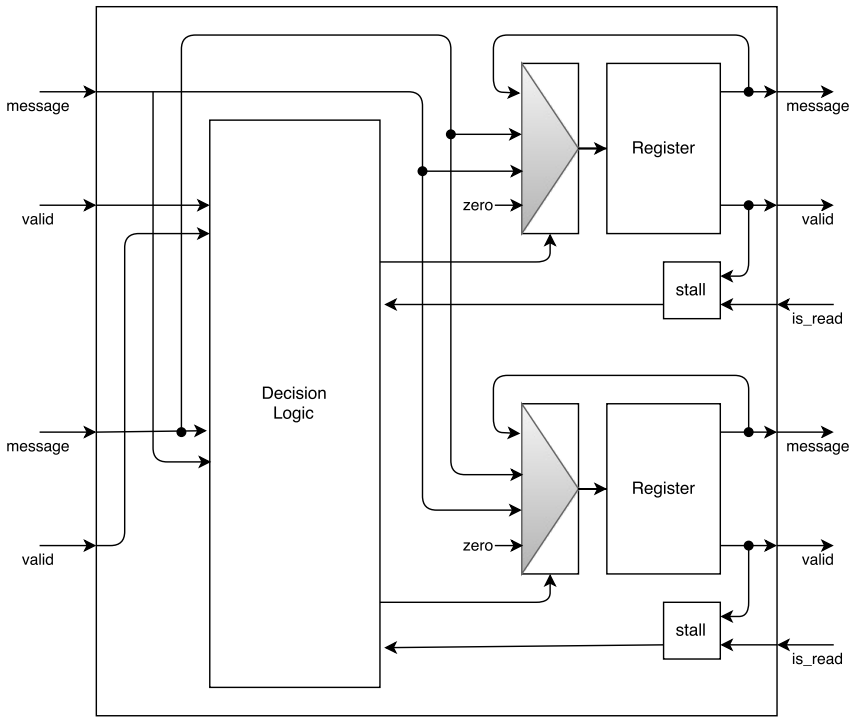


Figure 17: Structure of stalling banyan network switches

DTN.

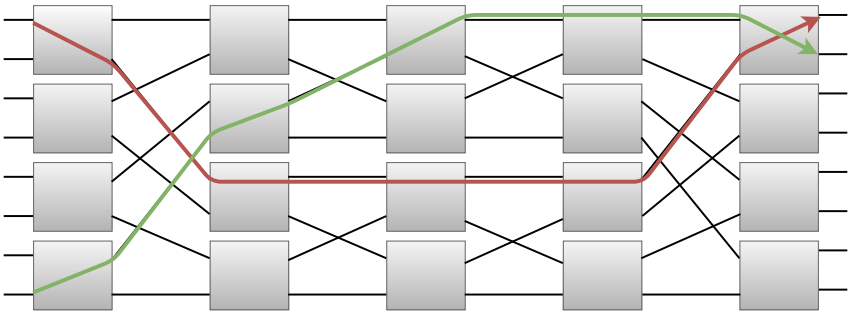


Figure 18: Paths of two outermost-layer companions

Causing more complexity, the routing decisions for the input switches of the Beneš network depend on all inputs of the network. This is due to the fact that each pair of messages comes to an output switch has to arrive from two different subnetworks. An example illustrating this is shown in Figure 18. For each destinationation address there is a ”companion” address that is routed to the same output switch.

To handle these dependencies properly, the input switch column of the recursive benes network uses reservation signals, where specific destinations are reserved by the switches sending packets to them so they are sent only once. This is shown in Figure 19, along with the lines for reservations from stalled packets, which need to have the highest priority because they can not be held back in the current architecture. A detailed overview of an input switch is given in Figure 20.

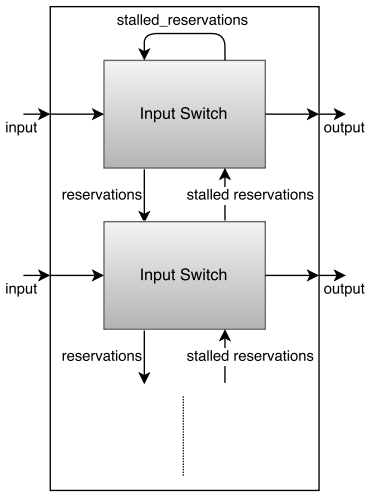


Figure 19: Structur of Beneš network input switch column

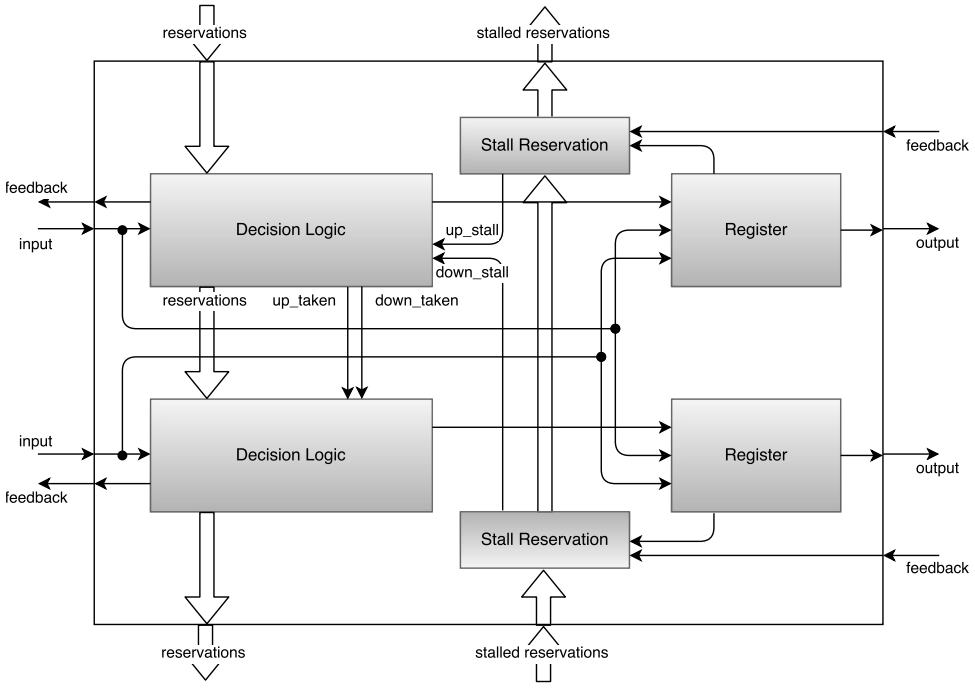


Figure 20: Functional overview of Beneš input switch

As a more simple solution, a lookup table for the first stage of a 32×32 benes network would require around 2.92×10^{48} bytes.

5.2.1 Message Ordering

For each packet from source s and destination d and for the program order of move instructions $m_n(s_n, d_n) \prec m_{n+1}(s_{n+1}, d_{n+1})$, and delivery order of network $m_n \lhd m_k$, the following constraint has to hold:

$$(m_n(s_n, d_n) \prec m_k(s_k, d_k)) \wedge (s_n = s_k \wedge d_n = d_k) \\ \Rightarrow (m_n(s_n, d_n) \lhd m_k(s_k, d_k))$$

This is necessary since the input buffers cannot distinguish between different packets that have equal sources and destinations.

With the recursive structure of the Beneš network and possible stalling at each input step, this constraint can not be guaranteed and is a case for which the network might work incorrectly. This does not apply to the Banyan part of the network, since relevant message pairs take the same path and do not overtake each other.

5.2.2 Resource Utilisation of the Beneš Network

Site Type	Benes(%)	Stalling Banyan(%)
Slice LUTs	51.23	13.86
LUT as Logic	51.23	13.86
LUT as Memory	0.00	0.00
Slice Registers	12.28	06.61
Register as Flip Flop	12.28	06.61
Register as Latch	0.00	0.00
F7 Muxes	1.53	0.00
F8 Muxes	0.63	0.00

Figure 21: Resource utilization of DTN with Benes network

As shown in Figure **??**, the Beneš network takes significantly more logic than all other DTNs. This, together with the high circuit depth required, make this implementation an unsuitable candidate for the SCAD architecture data network.

The stalling banyan network may be considered for applications where a small DTN is required.

6 Buffers

Temporary data storage within functional units is implemented with buffers. Design of input and output buffers are un-identical; *data packets* from DTN and *reservations* from MIB, together forming an operand, has to be handled in input buffer whereas output buffer simply stores the results.

A reservation r_i is represented by source field address of an MIB packet. Once an address is reserved, buffer snoops DTN for valid data packages d_j and notifies operand availability if and only if $r_i = d_j.src$, i. e. source field of incoming data matches with reservation. Subscript for packets represent cycle of arrival.

Reservations on input are performed with FIFO ordering for retaining actual sequence of operations; reservation r_i is executed before r_{i+1} where i denotes arrival time, given that data is ready for the preceding reservation. Whenever data is available for a reservation, *available* signal is asserted for FU to read the data from head of buffer. Next two sections explain input and output buffers in detail.

6.1 Input buffer

Input buffer forms an operand when DTN has valid data and its source address (routed from FU) matches with any of the reserved addresses inside buffer. Clearly, in every cycle a search needs to be performed amongst reserved addresses for d_j. For that purpose, two structures are used namely *look-up-table (lut)* for searching and *buffer (buf)* for operand storage. Figure 22 shows these components.

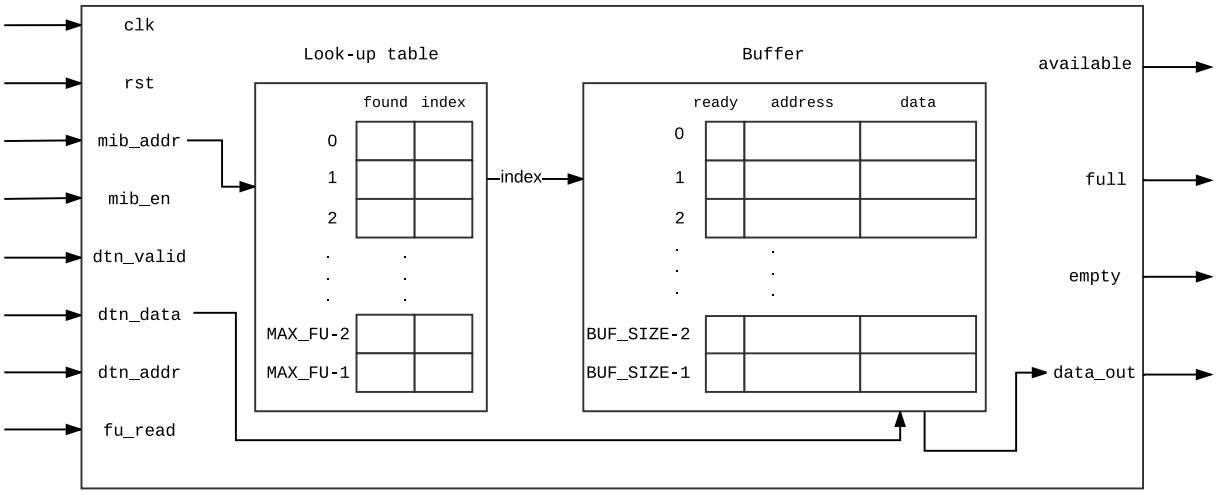


Figure 22: Input buffer structure

For managing the buffer, variables *head*, *tail*, and *num_elements* are used. *head* points to first element of the buffer and is incremented when a read is performed. *tail* points to last element and is incremented on every write. *num_elements* keeps track of number of elements in the buffer and is incremented or decremented in case of write and read respectively. Since buffer is single channel, these signals are not subject to concurrent updates. Input/output ports are described in 3.

lut is an array of 2-tuple $(found, index)$ with size MAX_FUS, where found is a 1-bit signal and index is a logic vector of size $log_2 BUF\_SIZE$. For index i, *found* states where address i is reserved in buffer. Source address of the MIB control packet is used as index to table for marking the corresponding entry in case of a reservation r_i by marking lut$[r_i]$ $(found = 1, index = tail)$. For example, search for incoming DTN packet d_j can be performed by checking $lut[d_j.src].found \stackrel{?}{=} 1$. If found, $lut[d_j.src].index$ can be used for indexing the buffer.

buffer is an array of $(ready, address, data)$ with size BUF_SIZE, where ready is a 1-bit signal, and address and data is logic vector of matching size. *ready* is set whenever a data and address pair for a reservation gets available. In case of a reservation r_i, after marking $found$ in *lut*, $buf[lut[r_i].index].address$ is set to r_i without modifying *data* ir *ready* fields. Continuing with the previous example, arrival of data package with $d_j.src = r_i$ triggers modifying corresponding buffer entry as $buf[lut[d_j.src].index] = (ready = 1, address = r_i, data = d_j.data)$.

Since FIFO ordering of reservation arrival is enforced, only *head* of the buffer can be read in case it is ready. This means that ready entries superseded by a non-ready entry cannot be read and processesd by functional unit. Inspection and simulation of such cases has crucial importance for performance evaluation of the architecture.

name	**direction**	**type**	**description**
mib_addr	input	logic_vector(FU_ADDRESS_W)	address to be reserved
mib_en	input	logic	enable strobe for address reservation
dtn_valid	input	logic	valid signal from DTN
dtn_data	input	logic_vector(FU_DATA_W)	data from DTN
dtn_addr	input	logic_vector(FU_ADDRESS_W)	address from DTN
fu_read	input	logic	read enable from functional unit. Data at the buf[head] is written to the output when this signal is asserted
available	output	logic	signals that an address/data pair is available at buf[head]. fu_read is assumed to be set only when available is high
full	output	logic	buffer full signal. Any reservation attempt on a non empty buffer is ignored
empty	output	logic	buffer empty signal. This signal is obsolete and can be removed safely
data_out	output	logic_vector(FU_DATA_W)	data output to functional unit. This signal is stable during a clock cycle after fu_read is set

Table 3: Input buffer ports

6.2 Output buffer

For storing results of operations inside FU, a single-channel FIFO buffer is used. It is a simple design that performs read/write and asserts empty/full. Every operation takes single cycle. Top-level scheme and port descriptions can be found in Figure 23 and Table **??**, respectively.

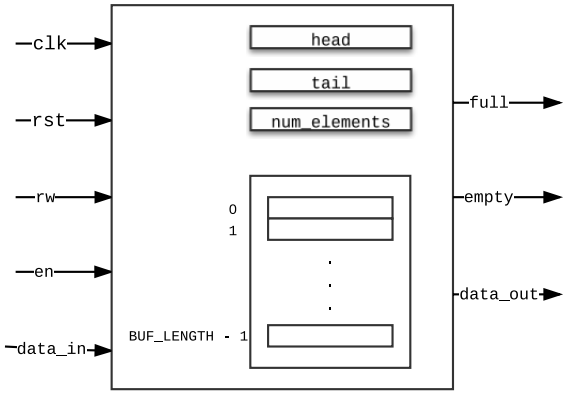


Figure 23: FIFO buffer structure

name	direction	type	description
rw	input	logic	when low, data is read from buf[head], otherwise input is written to buf[tail]
en	input	logic	enable strobe. rw is valid only when this signal is high
data_in	input	logic_vector(FU_DATA_W)	data input from functional unit
full	output	logic	buffer full signal. Writes are ignored when it is high
empty	output	logic	buffer empty signal. When set, read operations cannot be performed on controlling functional unit.
data_out	output	logic_vector(FU_DATA_W)	data output to functional unit

Table 4: FIFO buffer ports

7 Functional Units(FU)

FUs are fundamental processing elements in this architecture. Connected to controller and rest of the units via MIB and DTN, each FU has the capability of storing operands/results and performing single operation. FUs are classified into *Arithmetic-logic unit (ALU)* and *memory* depending on the type of processing performed. Next, design for both subsystems is explained.

7.1 Arithmetical operations

Arithmetical operations take two operands and produce a single result. For that purpose, it consist of two input and one output buffers. As described in Table 5, *index* field of *mib_inp* selects the input buffer to perform reservation. DTN input is directly connected to input buffers, since snooping capability is implemented within them.

One of the features of SCAD architecture is the ease of plugging-in new operators without rigorous attempts for data-path modification. Hence, operation classes encapsulate the operators by using a well-defined interface. Either single-cycle, multi-cycle or pipelined, components implementing the interface can easily be hooked inside a FU, by only designing operator without making any effort on bus connections or buffer modification. Thus, we represent design of arithmetical functional units in two parts by first explaining generic interface and then top-level functional unit design.

name	direction	type	description
mib_inp	input	mib_ctrl_out	control signal, including address to be reserved and target buffer index
status	output	mib_stalls	specifies whether input/output buffers are available for reservation.
ack	input	logic	acknowledgment from interconnection network to indicate safe removal of current output entry from buffer
dtn_data_in	input	data_port_sending	input from DTN
dtn_data_out	output	data_port_sending	output to DTN

Table 5: Input buffer ports

name	direction	type	description
op1	input	logic_vector(FU_DATA_W)	operand from first buffer
op2	input	logic_vector(FU_DATA_W)	operand from second buffer
en	input	logic	when set, operation *op* is performed using operands
busy	output	logic	functional unit does not send new operands when unit is busy. Is useful only when operation to be performed requires stalling. Pipelined designs can be implemented by keeping it low
valid	output	logic	signals stable output value
res	output	logic_vector(FU_DATA_W)	result of operation

Table 6: Input buffer ports

7.1.1 Generic operation

Operator component of arithmetic FU uses ports defined in 6. Control logic from FU sets *en* signal when both buffers have their data available. When *valid* is asserted, result of operation is written to *res* for FU to read into its output buffer. Pipelined or multi-cycle designs can be realized by keeping *en* and *valid* signals high by changing operands in every cycle. FU considers stalls on both input and output buffers when driving *en* signal. Anyways, only restriction is to prevent having *valid* signal asserted when *en* is low to avoid overflowing output buffer with stale results. Figure 24 illustrated operator structure.

Regardless of operation type (addition, multiplication, string comparison!), components adhering to the interface can simply be attached to the interconnection network. Abstracting functionality from architectural details results in easy additions to high-level instruction set.

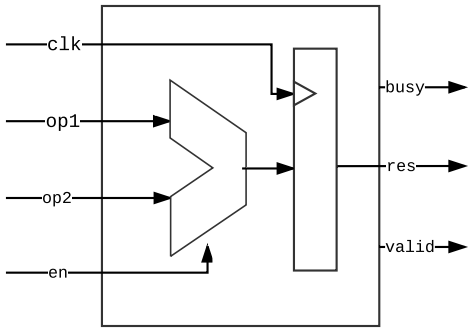


Figure 24: Arithmetic operator structure

7.1.2 FU design

Although off-loading snooping logic to input buffers reduces complexity of FU, task of managing read-/write signals amongst buffers causes non-negligible area cost in design. Main task performed by FU is to route the available operands from input buffers into operator and then write the result to output buffer.

FU also checks incoming MIB packets' source address to decide if it should supply operands to the target FU. Assuming address N for a FU, arrival of a MIB packet with $mib\_inp.src = N$ triggers an assemble phase where data at *head* of output buffer out_buf is used for assembling a new DTN packet $d_j = (valid = 1, src = N, dest = mib\_inp.dest, data = out\_buf[head])$ to be sent to destination

via interconnection network. Even though it is not implemented in this version, FU should also check *ack* signal coming from input of interconnection network and keep its output stable until it is asserted. Reason for such mechanism is the presence of DTN packets destined for same output, which are solved by adding FIFO buffers to the input of interconnection network for arbitrating access to same destination. In case when network buffer where FU is sending data through gets full, output should be held stable by FU until buffer has an empty space (until *ack* is high), in which it is hard to derive an upper bound on number of cycles until arrival of acknowledgment. Due to the initial state of design and low probability of such event occurring, we simply discarded usage of *ack* signal. Figure 25 shows top-level arithmetic FU.

In the figure, red lines belong to MIB, blue lines ones to DTN and yellow lines to internal data connections. As mentioned in 7.1.1, FU waits for availability of data from both buffers. When available, a read is performed on both buffers. Then FU routes the operands to operator and waits for completion. Once done, result is written into output buffer.

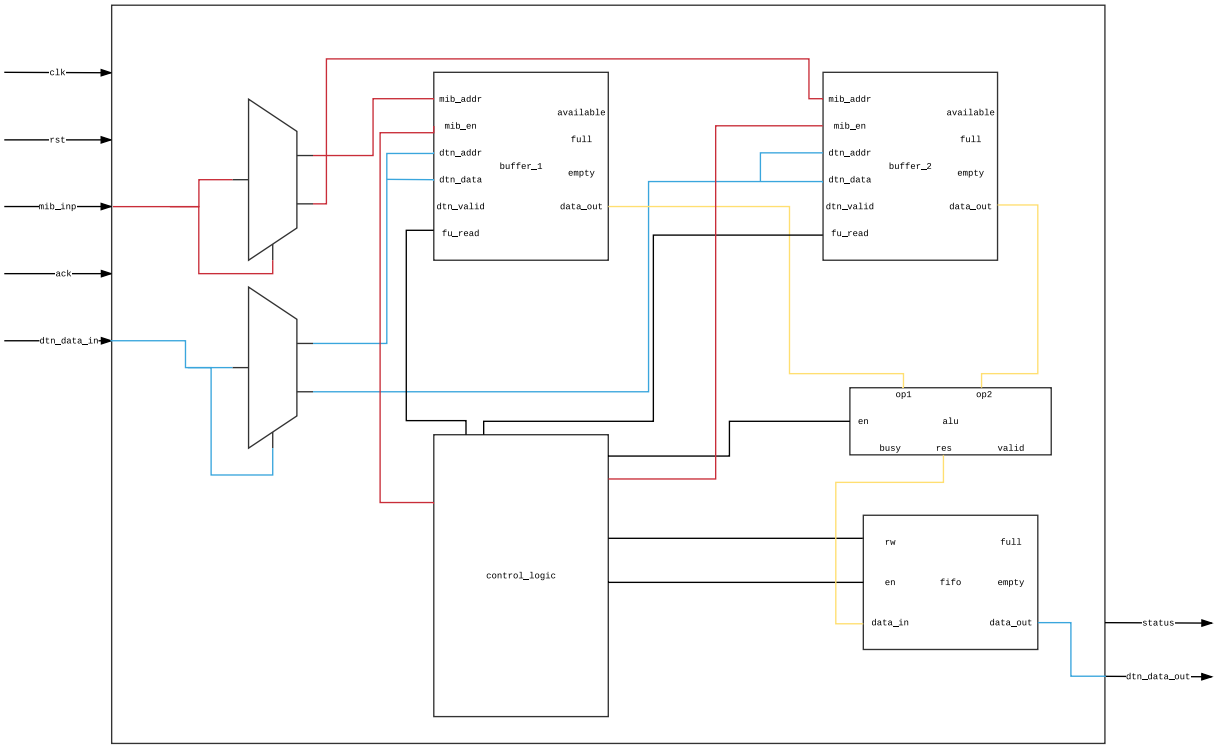


Figure 25: Arithmetic operator structure

7.2 Memory operations

Memory access is done via load/store functional units. For initial version, RAM is implemented using on-chip BRAM units, as an array of logic vectors, though it is not far from real design since caches should be introduced in some point anyways. Memory is divided into banks where each bank is shared by a pair of load/store units. This way, memory is simply segmented into regions and compiler/linker is responsible for translation from a flat address space to segmented addresses, since load/store units can access only to local RAM and bank. Figure 26 shows an overview of subsystem.

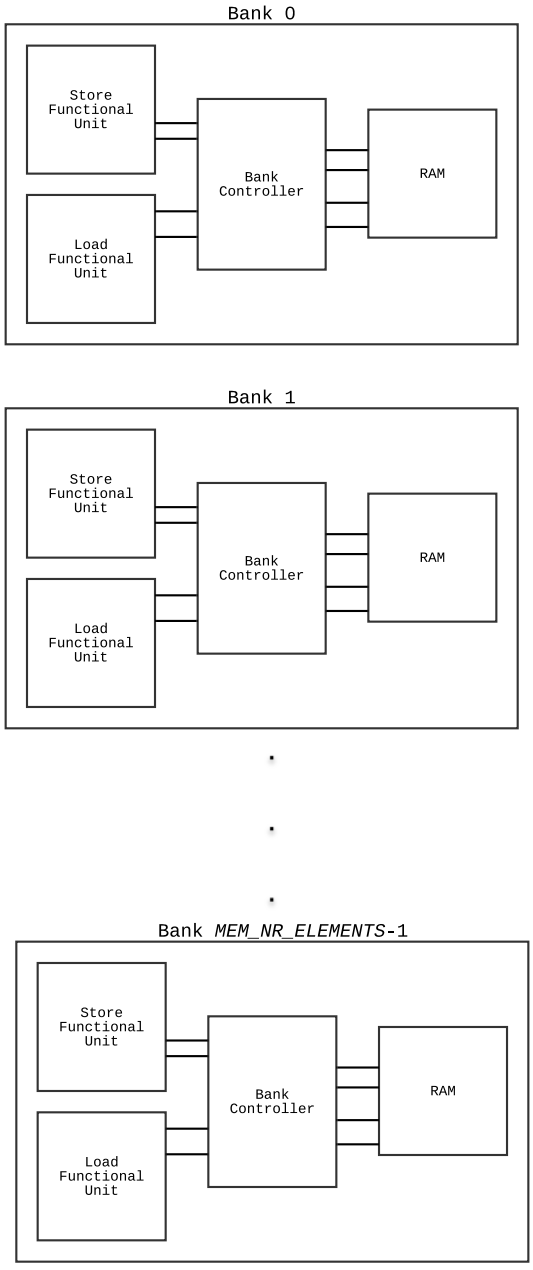


Figure 26: Arithmetic operator structure

Parameters defined in glossary section **??** represents size, word-length and number of banks for memory system. Next, design of load and store FUs are explained. Figures of MIB and DTN signals are omitted since they are identical to those used in arithmetic FUs.

7.2.1 RAM

Main memory blocks are designed to infer synchronous read/write, dual-port, word-addressed RAM using BRAM blocks on fabric. With usage of acknowledgment signals for request, it resembles a synchronous master/slave bus protocol. Enable strobes for read/write are not driven directly by functional units, but via *bank controller*, so RAM block does not check for conflicting read/write addresses.

Every access takes single cycle. Following read/write, respective acknowledgment signal is set to notify the functional unit. Table 7 includes detailed port definitions.

name	**direction**	**type**	**description**
re	input	logic	read enable strobe
we	input	logic	write enable strobe
r_addr	input	logic_vector(MEM_BANK_ADDR_LENGTH)	address to read
w_addr	input	logic_vector(MEM_BANK_ADDR_LENGTH)	address to write
data_in	input	logic_vector(MEM_WORD_LENGTH)	data to write
r_ack	output	logic	read acknowledgment signal. It is set when *data_out* is stable
w_ack	output	logic	write acknowledgment signal. Used for indicating write completion
data_out	output	logic_vector(MEM_WORD_LENGTH)	data read from address *r_addr*

Table 7: Input buffer ports

7.2.2 Memory bank

A memory bank consists of a bank controller and a RAM unit. Load and store units cannot send commands to RAM module, but they request access from bank controller. This way, conflicts on read/write to same memory location at same cycle can be controlled. This module does not enforce any memory consistency model; only precaution is when a same address is to be accessed by multiple unit, store is given preference and load is stalled for one cycle. Otherwise, since memory is dual-channel reads/writes can be issued in same cycle.

Even in presence of memory barrier instructions in high-level language, forcing a weak consistency model in rather *chaotic* movement of data is a challenging task since order of memory accesses is not preserved despite having ordered reservations on FUs.

Data lines are not routed through bank but connected directly to RAM. Since there is no asynchronous read on memory, this does not cause any side-effects. Acknowledgment signals are directly connected to FUs as well. Port description can be found in Table 8.

name	direction	type	description
FU Signals			
re	input	logic	read enable strobe from load FU
we	input	logic	write enable strobe from store FU
r_addr	input	logic_vector(MEM_BANK_ADDR_LENGTH)	read address from load FU. Used for storing in register in case of conflicts
w_addr	input	logic_vector(MEM_BANK_ADDR_LENGTH)	write address from store FU. Used for comparing with load address to check conflict
busy	output	logic	FUs do not send further requests when this signal is set
Memory signals			
re_out	output	logic	read enable strobe to RAM
we_out	output	logic	write enable strobe to RAM

Table 8: Input buffer ports

7.2.3 Load/store unit

Load Load unit has single input and single output buffer where input is used for storing memory address to be loaded and output is used for holding read data coming from memory. Table 9 describes the input/output ports. Load unit has connections to both FU and bank controller where FU provides address to be read and load unit manages memory access transaction by setting commands to bank controller and checking acknowledgment from memory.

name	direction	type	description
FU Signals			
operand	input	logic_vector(MEM_BANK_ADDR_LENGTH)	operand from FU. Used as address on memory access
busy	output	logic	indicates a memory transaction is on progress
valid	output	logic	asserted when data loaded from memory is stable
res	output	logic_vector(MEM_WORD_LENGTH)	output of memory access. Used by FU
Memory signals			
res	input	logic_vector(MEM_WORD_LENGTH)	output of memory access
ack	input	logic	acknowledgement from memory. When set, *res* is stable and can be read into internal register
addr	output	logic_vector(MEM_BANK_ADDR_LENGTH)	address to load
re	output	logic	read enable strobe to memory bank controller

Table 9: Load unit ports

Store Store unit consists of two input buffers. Unlike other types of units, it does not have an output buffer, since store operation does not have output. Two inputs buffers are used for storing address/data pairs provided by DTN. Depending on the addressing modes, it is likely that address might be result of a computation, as well as an immediate from controller. Immediate operands are converted into DTN packets by control unit, so snooping DTN for write address works in both addressing modes.

name	direction	type	description
FU Signals			
fu_address	input	logic_vector(MEM_BANK_ADDR_LENGTH)	address to write
fu_data	input	logic_vector(MEM_WORD_LENGTH)	data to write
busy	output	logic	indicates a memory transaction is on progress
valid	output	logic	asserted when data loaded from memory is stable
Memory signals			
ack	input	logic	acknowledgement from memory. Used for indicating write completion
addr	output	logic_vector(MEM_BANK_ADDR_LENGTH)	address to load
data	output	logic_vector(MEM_WORD_LENGTH)	data to load
we	output	logic	write enable strobe to memory bank controller

Table 10: Store unit ports

7.2.4 FU design

Functional units encapsulating load and store components are similar to arithmetic FUs. Only difference is the number of input/output buffers and existence of connection between internal load/store units and RAM/bank controller components. Without MIB and DTN signals, load/store FUs are illustrated in Figure 27 and Figure 28, respectively.

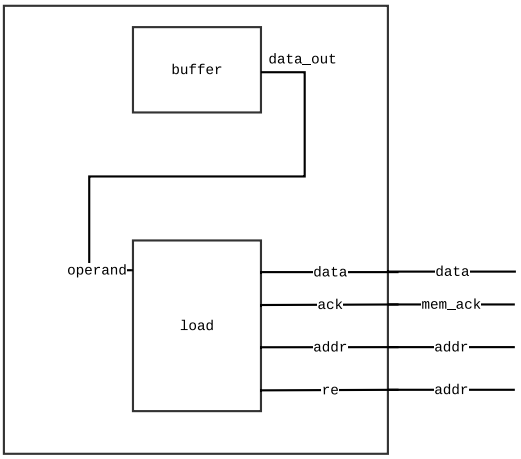


Figure 27: Load FU

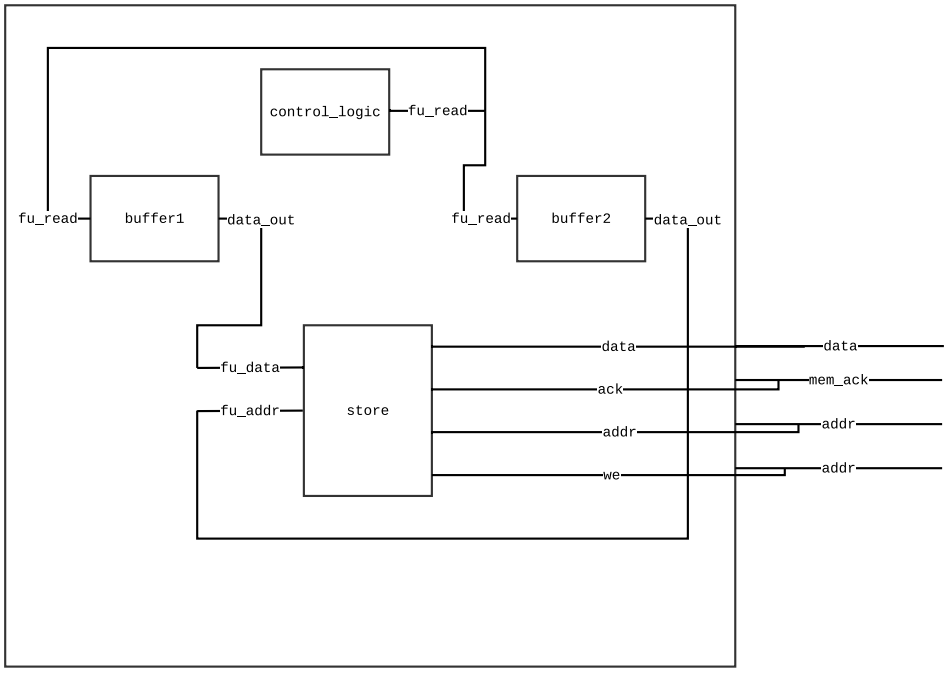


Figure 28: Store FU

8 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.

Currently, load and store operations to same address in a given cycle is resolved by serializing the instructions such that write is given precedence over read. Evidently, this scheme does not provide any form of guarantees on memory access ordering. A future work is to enforce a memory consistency model, which can be implemented in memory bank controllers.

9 Bibliography

References

[1] Tripti Jain Anoop Bhagyanath & Klaus Schneider (2015): *A Time-Predictable Model of Computation*. IEEE Real-Time Systems Symposium.

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

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

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

[5] 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?
```