A dataflow architecture for beamforming operations

Msc Assignment

by

Rinse Wester

Supervisors:
dr. ir. André B.J. Kokkeler
dr. ir. Jan Kuper
ir. Kenneth Rovers
Anja Niedermeier, M.Sc
ir. André W. Gunst
dr. Albert-Jan Boonstra

Computer Architecture for Embedded Systems
Faculty of EEMCS
University of Twente

December 10, 2010

Abstract

As current radio telescopes get bigger and bigger, so does the demand for processing power. General purpose processors are considered infeasible for this type of processing which is why this thesis investigates the design of a dataflow architecture. This architecture is able to execute the operations which are common in radio astronomy.

The architecture presented in this thesis, the FlexCore, exploits regularities found in the mathematics on which the radio telescopes are based: FIR filters, FFTs and complex multiplications. Analysis shows that there is an overlap in these operations. The overlap is used to design the ALU of the architecture. However, this necessitates a way to handle state of the FIR filters.

The architecture is not only able to execute dataflow graphs but also uses the dataflow techniques in the implementation. All communication between modules of the architecture are based on dataflow techniques i.e. execution is triggered by the availability of data. This techniques has been implemented using the hardware description language VHDL and forms the basis for the FlexCore design. The FlexCore is implemented using the TSMC 90 nm technology.

The design is done in two phases, first a design with a standard ALU is given which acts as reference design, secondly the Extended FlexCore is presented. The Extended FlexCore incorporates the ALU which exploits the regularities found in the mathematics. The ALU of the Extended FlexCore is able to perform a four point FIR filter, a complex multiplication or an FFT butterfly operation in a single clock cycle. The Extended FlexCore uses an Explicit State Store (ESS) to handle stateful operations like a four point FIR filter.

The standard FlexCore and the Extended FlexCore are compared by executing a FIR filter, FFT and complex multiplications. The comparison shows that the Extended FlexCore is significantly more energy efficient per operation than the reference FlexCore.

Finally, an indication of the energy efficiency of the Extended FlexCore is given in comparison with other architectures. It is shown that the FlexCore lies, in terms of energy per operation, between the ASICs and the general purpose ARM processor.

Contents

1

Introduction

A way to summarise the developments in radio astronomy is "the bigger, the better". Especially radio telescopes like LOFAR reach sizes over 1500 kilometer across[1],[2]. An even bigger telescope, SKA, is planned to be operational in 2023. SKA will have a total collecting area of 1 square kilometer and has a diameter of more than 3000 kilometer[3],[4]. All these telescopes use a huge number of antennas to receive the signals. All these signals are combined into sky pictures by a technique called beamforming.

With the increase in size and number of antennas, so does the demand for processing power to handle all the data coming from these antennas. Conventional processors like von Neumann architectures are considered infeasible for these types of applications in terms of processing power and energy consumption which is why the radio astronomy community uses ASICs and FPGAs [5],[6],[7].

In this thesis, an implementation of the dataflow architecture proposed by Kenneth Rovers [8], the FlexCore, is presented. Parts of the algorithms used in radio astronomy are used to evaluate the FlexCore to see whether dataflow architectures are suitable for applications like radio astronomy.

1.1 Research goals

The goal of this thesis is to implement the FlexCore using the hardware description language VHDL. After that, the design should be synthesized using ASIC tooling such that numbers like area and power consumption can be extracted. The FlexCore is a dataflow architecture where execution is triggered by the availability of data instead of a program counter (dataflow execution). The corresponding models used for analysis of execution based on availability is called dataflow analysis.

The FlexCore itself also uses data triggered execution for all the internal modules of the design. The FlexCore can therefore also be considered a dataflow graph. All the connections in this graph (connections between the modules of the architecture) use buffers with feedback for communication. This feedback prevents overflows of the buffers and is called backpressure[9]. Before implementing the FlexCore, dataflow graphs with backpressure should

be implemented using VHDL first. This will form the basis on which the design of the whole FlexCore is based. The corresponding research question is: How can dataflow graphs with backpressure be implemented in VHDL and what will be the lower bound of the buffer sizes?

The reason for trying to combine dataflow architectures and beamforming is that dataflow programs lie closer to the mathematical description of the program than for example von Neumann architectures. However, dataflow architectures introduce overhead due to the fine grain parallelism[10]. An important way of increasing energy efficiency is exploiting locality of reference[11]. The main research question for this thesis is: How can the granularity of dataflow execution be increased by exploiting locality of reference in the LOFAR beamforming application?

1.2 Thesis structure

Background information is given in chapter 2 about the concepts that will be used throughout this thesis. First an introduction on dataflow graphs will be given, followed by how these are executed in dataflow architectures. Finally, more information is given on beamforming and how this is implemented on the LOFAR radio telescope.

As dataflow principles are the major part of the design of the FlexCore, they are first implemented in hardware using the hardware description language VHDL. Chapter 3 shows how dataflow graphs are implemented using VHDL and how much buffering of data is required. In order to exploit locality of reference by increasing the granularity, the mathematical operations used for radio astronomy are investigated in chapter 4. Regularities in the algorithms of beamforming are used to merge the basic components of these algorithms together into a single module of the processor.

In order to evaluate these ideas, two implementations of the FlexCore are made. The first implementation, referred to as standard FlexCore, is a standard dataflow architecture as described in literature[12]. The standard FlexCore is used as reference implementation to evaluate the Extended FlexCore which incorporates the regularities found in the beamforming application. Chapter 5 starts with the implementation of the standard FlexCore and elaborates on the techniques used to design this processor (dataflow). In section 5.3 the Extended FlexCore is presented exploiting the mathematical transformations from chapter 4.

Both designs have been implemented using 90 nm technology. Also several algorithms have been executed on both designs such that an indication of power usage can be given. The results of this can be found in chapter 6 and are compared to other architectures in chapter 7.

Finally the results are discussed in chapter 8 followed by the conclusion in chapter 9.

2

Background

Before diving into dataflow graphs, dataflow machines and beamforming, some background information about dataflow graphs, dataflow architectures and beamforming is needed. This chapter gives the background information on which the rest of this thesis is based. First a basic explanation on dataflow graphs is given in section 2.1. This is used in section 2.2 where the inner working of dataflow machines is explained. Section 2.3, gives the information on beamforming which is the application of which parts should run on the FlexCore. Finally, section 2.4, explains the tool flow for building an ASIC.

2.1 Dataflow graphs

Data Flow Graph (DFG)s are mathematical representations of programs[13]. Execution of these programs is not driven by a sequence of instructions but by the availability of data. The most well known type of dataflow graphs are Synchronous Data Flow (SDF) graphs[14]. Dataflow graphs are the basis for the work presented in this thesis.

The operations that should be performed during execution of a dataflow graph are represented by *nodes*. Nodes can be of any granularity, ranging from simple operations like addition and multiplication to complete FFT operations and processors. Nodes in a dataflow graph are connected by *arcs* which do not only represent the dependencies between nodes but which are also the locations where data is stored. The packets containing data are so called *tokens*. Figure 2.1 shows the terminology displayed graphically.

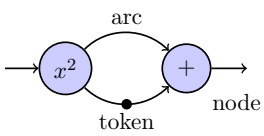


Figure 2.1: Simple dataflow graph for $x^2 + x^2$

The execution of a dataflow node is triggered by the availability of tokens. When all required inputs for a node are available the node becomes *enabled*.

An enabled node consumes tokens on the input(s) and produces tokens on the output(s). This process is called *firing*. Only enabled nodes may fire which is called the *firing rule* i.e. the firing rule states that a node may only execute when all required tokens are available on the input(s).

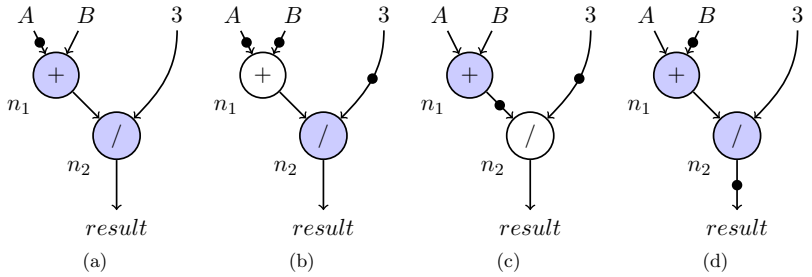


Figure 2.2: Firing rules in dataflow graphs

Figure 2.2 shows four events during execution of a dataflow graph created from the formula $result = \frac{A+B}{3}$. Node n_1 (the adder) is the first to receive a token but is not yet enabled because it requires two tokens. At a later time (figure 2.2b) the second token has arrived as well and node n_1 becomes enabled. Node n_2 (the divider) also receives a token but remains disabled until n_1 has produced its result. Figure 2.2c shows the time where n_1 has produced a result and therefore enables node n_2. When n_2 fires it consumes the tokens on the inputs and produces a token containing the result (figure 2.2d).

A graph whose nodes only consume and produce a single token per input and output during firing are called Homogeneous Synchronous Data Flow (HSDF) graphs [15]. Homogeneous dataflow graphs are a subset of Synchronous Data Flow (SDF) graphs[14] which can produce and consume several tokens at once. The work presented in this thesis however only involves Homogeneous Synchronous Data Flow (HSDF) graphs.

The time between two events in a Synchronous Dataflow Graphs can be arbitrary. A pure implementation of such graphs therefore does not require a global clock because all the synchronization is enforced by the firing rules. More information can be found in the literature report which is included in appendix B. More theoretical information on dataflow graphs can be found in [15] and [16].

2.2 Dataflow architectures

The machines able to directly execute dataflow graphs are called dataflow machines. These machines use the firing rule as explained in the previous section to start execution of nodes. The first dataflow machine was developed by Dennis at MIT[17] and is called the MIT static dataflow processor.

Dataflow machines are usually divided in two groups, the static dataflow machines and dynamic dataflow machines. In static dataflow machines, the DFG being executed does not change. In a dynamic dataflow machine however, the DFG is able to change during execution. Dynamic dataflow machines are able to perform more advanced features like procedure calls. More information about different dataflow machines can be found in the literature report in appendix B and in [12].

The architectures presented in this thesis use principles from both static and dynamic dataflow machines. Figure 2.3 shows the general structure of a static dataflow machine. The dataflow graph is usually stored in a special memory. This memory also supports storage of tokens. As can be seen in figure 2.3 every incoming token from the left triggers the *enabling unit* to detect whether a node from the DFG becomes enabled. If this is not the case, the token will be stored in the memory. When a node becomes enabled, i.e. there is a match, both operands and instruction for that particular node are combined in a packet and sent to the *functional unit* which executes the instruction with the given operands. After the instruction has been completed, the result is sent to the enabling unit again. Resulting tokens may enable other nodes from the DFG which completes the cycle.

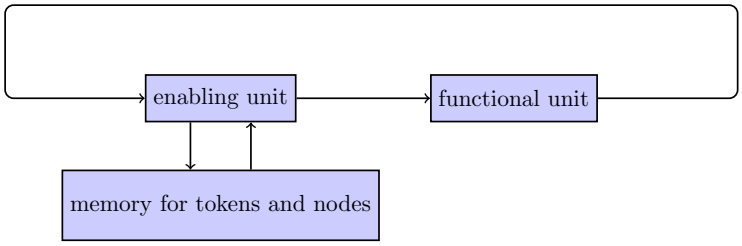


Figure 2.3: Basic structure of a static dataflow machine.

The matching procedure in static dataflow machines requires special hardware for every possible node in the DFG. The hardware has three registers: one for the instruction and two for the operands (the in-degree of nodes is usually two). When both operand registers contain a value, a match is found. Static dataflow machines like the MIT static dataflow machine therefore have a matching circuit for every node[17]. More information about static dataflow machines can be found in the literature report in appendix B.

Soon after the static machines came the dynamic versions which used general memories as found in von Neumann architectures. The architectures were however inefficient in matching because they used hashing functions for that [18]. This resulted in low utilization of the functional unit and therefore degraded the performance of the whole processor. By using a so called Explicit Token Store (ETS)[19] the slow hashing parts could be removed. Both implementations of the FlexCore use an ETS for matching of tokens.

ETS

A more efficient way of matching instead of using hashing functions is an Explicit Token Store (ETS) which was introduced in a machine called the Monsoon[19]. The ETS was specifically designed for efficient execution of procedure calls. The central idea behind ETS is to allocate large blocks of memory for complete procedures but let the addressing details of single variables in the program be determined at compile time. Both FlexCores do not support procedure calls but do make use of the addressing techniques that should be applied at compile time. Therefore this explanation will only focus on the addressing part, a complete view of the ETS can be found in literature report of appendix B.

The addressing for an ETS is performed by the compiler. Every node in the dataflow graph is assigned a unique address. This address is then used for both the instruction in the program memory and tokens that have to be stored before a match occurs. Consider the following dataflow graph.

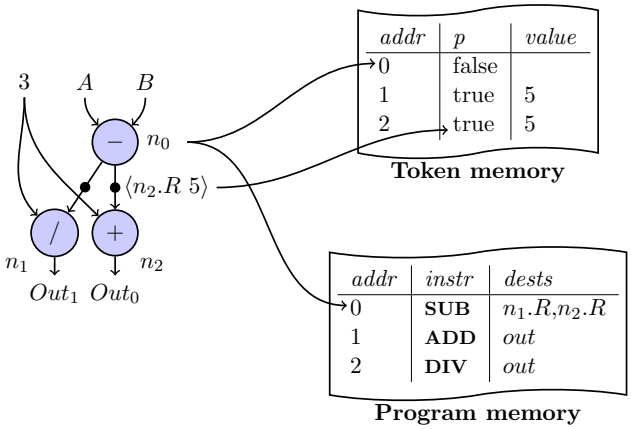


Figure 2.4: Example of ETS-principle.

Figure 2.4 shows an example dataflow graph with the corresponding memory contents of a dataflow machine. All nodes of the DFG are assigned a unique address that is used in both the token memory and the program memory. The program memory contains the instruction that corresponds to the operation in the DFG and a set of destinations. Node n_0 for example is assigned address 0 and has two destination nodes (right input of n_1 and n_2) which use the produced result.

When a token is sent to an input of a node, the address of that node is used to select an element from the token memory. The field p from that element represents the presence bit. This bit indicates whether the node, corresponding to the address of the incoming token, has one operand available on one of the inputs. When this is the case, the bit should be set to $true$. When there are no tokens available, the bit should be $false$. When the bit

is $false$ no token is on any input so the incoming token should be stored in the token memory(see address 1 and 2). If the bit was set to $true$ it means that the incoming token causes a match. The value of the previously stored token should then be fetched from the token store. Both the incoming token and the one from the token store form the operands for the instruction that is addressed by the incoming token. The instruction has a set of destinations to which the result is sent. These resulting tokens may enable other nodes which completes the cycle.

2.3 Beamforming

Beamforming is a technique used in radio astronomy to combine signals from several antennas. Using this technique, a much better directivity can be achieved. Signals which would normally be undetectable due to noise, can now be received if enough antennas are used. Beamforming uses the fact that the antennas are separated from each other by a certain distance. Signals from a certain angle therefore do not arrive at the same time. Adding the proper delay to the received signals makes the system directive. Figure 2.5 shows this process graphically.

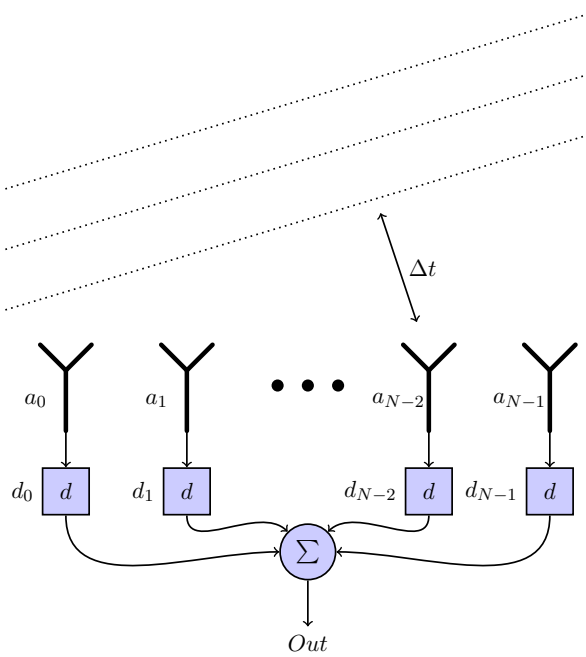


Figure 2.5: Beamforming

Figure 2.5 shows an array of N antennas which are combined with delays $d_0 \ldots d_{N-1}$. Signals (the wavefront shown with dashed lines) arrive first at antenna a_0 then at a_1 and eventually at a_{N-1}. By selecting the delays

such that all received signals are in phase again, the sum of all signals has a much better signal quality than a signal from a single antenna. There are several techniques to perform beamforming[20] and to implement the delays as shown in figure 2.5. LOFAR uses frequency beamforming to implement these delays[2].

From Fourier theory it is known that a delay can be implemented by multiplying the Fourier transform of the signal with a complex phase shift (equation (2.1)). More information on Fourier theory can be found in [21],[22].

$$f(t-\tau) \quad \leftrightarrow \quad \hat{f}(\omega)e^{-i\omega\tau} \tag{2.1}$$

$f(t)$ is a signal in the time domain being delayed with time τ. The time domain signal $f(t)$ has a corresponding signal $\hat{f}(\omega)$ in the frequency domain. $\hat{f}(\omega) = \mathcal{F}(f(t))$ i.e. $\hat{f}(\omega)$ can be found by taking the Fourier transform of the time domain signal $f(t)$. The signal $f(t)$ is now delayed by multiplying it with a complex phase shift $e^{-i\omega\tau}$.

Delaying a signal by using a phase shift only works perfectly for sinusoid functions. However when the bandwidth of the signal $f(t)$ is small enough, the phase shift can still be used. To determine if the bandwidth of the signal is "small enough", a measure called the Fractional Bandwidth (FB)[23] is used. The Fractional Bandwidth is a number that gives a comparison between the bandwidth of an incoming signal and the center frequency of that signal. When this number is less than 1%, the signal is considered narrowband and the phaseshift can therefore be applied to implement delays. The formula to calculate the Fractional Bandwidth is given in equation (2.2).

$$FB = \frac{f_h - f_l}{\frac{f_h+f_l}{2}} < 0.01 \tag{2.2}$$

Where f_h is the highest frequency occurring in the signal and f_l is the lowest frequency. The numerator $f_h - f_l$ is the bandwidth and denominator $\frac{f_h+f_l}{2}$ is the center frequency.

Now consider a narrowband signal $f(t)$ which is concentrated around frequency ω_0, by using the delay property of equation (2.1) the $f(t)$ becomes:

$$\mathcal{F}(f(t)-\tau) = \hat{f}(\omega) * e^{-i\omega\tau}$$

Because $f(t)$ is a narrowband signal, the ω of the complex exponent in the frequency domain can be replaced with a constant ω_0 which is the center frequency of the narrow band signal $f(t)$. The whole complex exponent in the frequency domain is now constant and can be moved to the time domain:

$$\hat{f}(t) * e^{-i\omega\tau} = \hat{f}(t) * \underbrace{e^{-i\omega_0\tau}}_{constant} \quad \rightarrow \quad f(t-\tau) = f(t) * e^{-i\omega_0\tau}$$

Concluding, a narrowband signal can be delayed by multiplying the signal with a constant complex number.

Beamforming in LOFAR

The signals that LOFAR receives are not narrowband and simply multiplying the signal with a complex number therefore doesn't work. Instead, for every antenna, the signal is first split into 1024 spectral components. Each of these components now has a relatively small bandwidth compared to their frequency i.e. the Fractional Bandwidth is less then 1% and complex multiplications for phase shifts per band are therefore possible.

Splitting the signal into spectral components is implemented by a so called filterbank. A filterbank is a FIR filter combined with an FFT to derive the spectrum of a signal coming from the antenna. The filter first preprocesses the signal such that unwanted signals are filtered out. The filtered signal is then fed to a 1024 point FFT which calculates the spectrum of the signal. Every component of this spectrum is then multiplied with a complex number which implements the phase shift. This process is shown in figure 2.6.

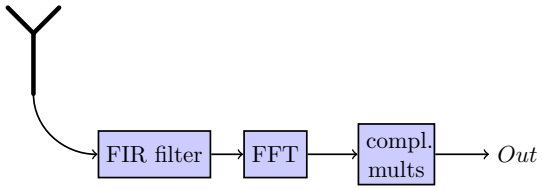


Figure 2.6: implementing delay by FFT

This implementation however suggests that for every sample from the antenna, a whole new FFT should be calculated. The FFT is a block based operation which means the the input signal is split into blocks and the FFT is applied to every block. The FFT in LOFAR is therefore executed once in every 1024 input samples (recall that the length of the FFT is 1024 points). The FIR filter in front of the FFT can also be optimized because not all samples have to be filtered completely. A more efficient architecture which combines the filter and FFT is called a polyphase filterbank[22] which exploits the fact that the FFT is block based. Figure 2.7 shows the LOFAR polyphase filterbank.

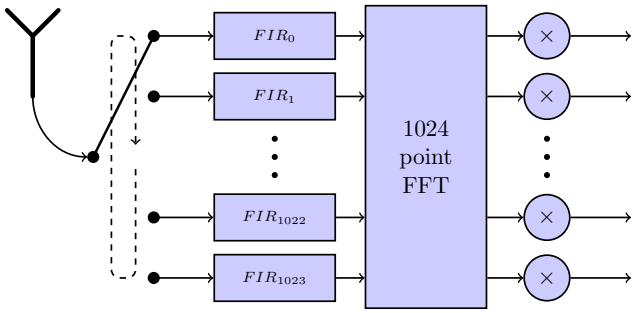


Figure 2.7: Polyphase filterbank

Figure 2.7 shows that the filter of figure 2.6 is split into several components. Each of these components is again a standard FIR filter but with a different set of coefficients. The set of filters is fed using a big switch. This switch distributes the samples of the input signal among the filters which send their results to the FFT which is still the same as the one of figure 2.6. The FFT is now executed when a whole block of 1024 samples is complete. That is also the moment where the big switch starts at the begin position again. The results from the FFT are forwarded to the set of complex multiplications which implement the phase shifts. More information on beamforming of LOFAR can be found in the work of Gerard Bos[24] where he made a mapping of the LOFAR application to a multicore SoC.

2.4 Tool flow

Creating an Application-specific Integrated Circuit (ASIC) of the FlexCore consists of several steps and requires several tools. First the FlexCore is implemented using VHDL and compiled and simulated using ModelSim[25]. When the design behaves correctly, it is synthesized with Synopsis Design Compiler[26] which translates the VHDL description of the architecture into cells which will be used in the ASIC. The design is synthesized using the TSMC 90 nm low power cell library. To verify whether synthesis is successful, the design is simulated again using ModelSim.

The collection of cells produced by synthesis are physically positioned and connected using the place and route tool Cadence Encounter[27]. The result contains a full description of the ASIC for production. Again, the result of place and route is verified using ModelSim. The last step is to determine power consumption. This is done using Synopsis Primetime[28] which uses all the signal changes from simulation, the power information from the cell library and the wire information from place and route to calculate the expected power consumption.

Another term that will show up in this thesis is clock gating. Clock gating is a technique to save dynamic energy consumption in an ASIC[29]. Clock-gating is applied to flipflops where the clock is disabled on a group of flipflops when no state change occurs. The tooling recognizes state changes based on a signal, for example a write enable, and adds a clock gate. The clock is disabled when the enable signal is *false* and enabled when *true*.

3

Dataflow graphs in VHDL

The main goal of this thesis is to design a dataflow architecture, the FlexCore. The FlexCore itself can be seen as a dataflow graph. By describing the architecture as a dataflow graph, the design of the nodes representing the modules of the processor should become easier. By applying the rules from dataflow, the synchronization of data should also be easier to implement. Before building a complete processor using dataflow graphs, the principles of dataflow (firing based on availability of data) and backpressure are implemented using VHDL[30]. This chapter shows how dataflow graphs with backpressure can be implemented in VHDL.

3.1 Dataflow graphs with backpressure

Dataflow graphs allow an infinite number of tokens to be stored on the arcs. This is in hardware not feasible because buffers are always finite in size. To prevent overflows resulting in data loss, a technique called backpressure is used to implement arcs with a finite amount of storage. This section describes how backpressure and buffering using FIFOs can be implemented and section 3.2 shows how a dataflow graph can directly be implemented in VHDL. All nodes of the dataflow graphs described in this thesis consume/produce only one token per execution on an input/output. All graphs are therefore Homogeneous Synchronous Data Flow (HSDF) graphs which is sufficient for the design of the FlexCore.

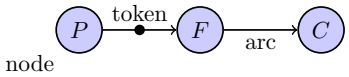


Figure 3.1: Simple DFG

Consider a simple dataflow graph without backpressure as depicted in figure 3.1 containing three nodes connected by two arcs. In reality all arcs allow only a limited number of tokens to be stored. However the producer (node P) receives no feedback from the arc connecting node P and F and therefore always assumes that there is storage available. When node F runs

at a slower rate than node P, the arc in between will eventually overflow and data will be lost. The firing rule of a dataflow node should therefore not only be dependent on the availability of incoming tokens but also on the space available on the arc to which outputs are connected. The producer therefore feels "pressure" which limits the production rate.

By introducing feedback about the available space on the arcs, the nodes producing data are restricted in the number of tokens that can be produced and consumed. The feedback signal, called *full*, indicates whether there is space on the arc. Tokens may only be produced when there is space on the arc. When there is no space available anymore the full signal is asserted and the producing node should stop producing tokens. The firing rule of the nodes then states that a node may only fire if all required inputs are available and if there is space available on all arcs where tokens will be produced.

The firing rule is encoded in a state machine containing two states. Every dataflow node contains such a statemachine. When the firing rule is not satisfied, a node should be in the *waiting* state. When the rule is satisfied, the node goes into the *processing* state. Note again that all nodes produce and consume only one token per in or output, as the graphs are HSDF graphs. Figure 3.2 shows this state machine.

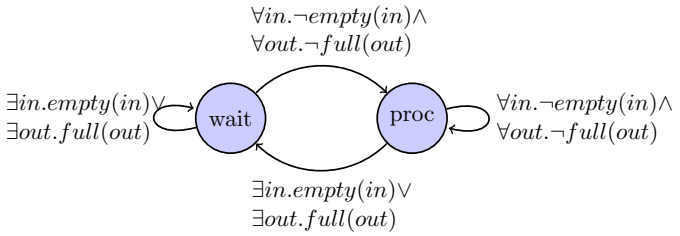


Figure 3.2: State machine implementing the firing rule

The initial state of the node is the *wait* state and the processing state is denoted with *proc*. The node may only go to or stay in the processing state when the firing rule is satisfied. All required inputs must have data available on the arcs and all arcs connected to the outputs may not be full : $\forall in.\neg empty(in) \wedge \forall out.\neg full(out)$.

3.2 Implementation

Figure 3.3 shows the implementation of the dataflow graph shown in figure 3.1 with backpressure. A node can send a token by placing the value on the data channel and asserting the *write* signal. Based on the value of the *full* signal the node may start sending a token. Tokens are stored in FIFOs inside the destination-node which are the implementation of arcs in dataflow graphs.

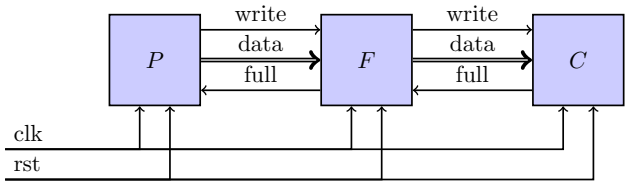


Figure 3.3: Data Flow Graph implementation with backpressure signals

The FIFO has data and command signals for both reading and writing tokens. The FIFO is a purely synchronous system[31] i.e. all read and write operations are executed at the rising edge of the clock signal. There are also 4 status-signals available which are used to generate the feedback-signals to the nodes. Figure 3.4 shows a FIFO with all the signals.

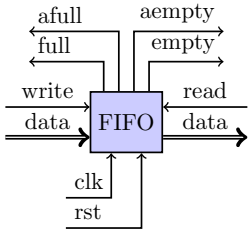


Figure 3.4: FIFO implementation

The *write* and *read* signals are synchronous command-signals for reading and writing tokens. These commands are only executed at the rising edge of the clock. The *full* and *empty* signals are asserted when the FIFO is full or empty. The signal *afull* is asserted by the FIFO when there is only one place available(almost-full). A similar signal is asserted when the FIFO is almost empty (*aempty*). The *empty* signals are used in the firing rule to check the availability of tokens on the input i.e. a node may not execute if any of the required arcs is empty. Note that *full* and *empty* cannot be omitted because the FIFO may become completely full or empty. This is caused by backpressure when a dataflow node does not read until an arc is completely filled, or does not write until an arc is completely empty. Without the *full* and *empty* signals of the FIFO, the next state cannot be predicted correctly and the Data Flow Graph deadlocks.

Arcs from dataflow graphs are implemented using the mentioned FIFO combined with two additional modules called *next state predictors*. These state predictors predict, based on the status signals of the FIFO and the read and write signals, what the next state of the node should be. The full and the empty signals are predicted by the Full Predictor (FP) and Empty Predictor (EP) respectively. All status signals from the FIFO, the *full* and *empty* signals and the read and write are single bit signals. The number of

bits for the data signals depends on what type of data should be stored on the arcs. Figure 3.5 shows the implementation of the arc.

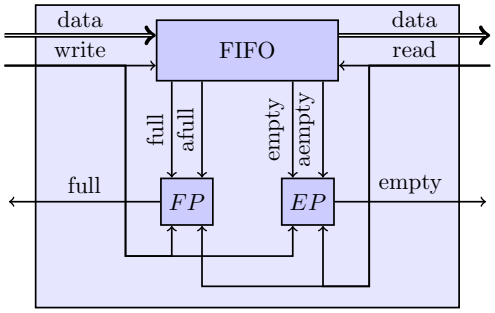


Figure 3.5: Arc with next-state prediction

The arc depicted in figure 3.5 shows the two predictors FP and EP. These are pure combinatorial blocks which means that the full and empty signals are valid before the next rising edge of the clock. These signals are then used by the state machines of the nodes to implement the firing rule. The expression of equation (3.1) predicts when the FIFO is full and equation (3.2) predicts when it is empty.

$$full = (\neg read \wedge full) \vee (\neg read \wedge write \wedge afull) \tag{3.1}$$

$$empty = (\neg write \wedge empty) \vee (\neg write \wedge read \wedge aempty) \tag{3.2}$$

Every node in the dataflow graph implemented in VHDL, is composed of an arc on every input, a combinatorial block which performs the operation and a state machine which implements the firing rule with backpressure. Figure 3.6 shows the components in a complete dataflow node. All the inputs are constructed using arcs such that tokens can be stored. Note that tokens are now stored inside of the dataflow node instead of on the edge in between two dataflow nodes. On arcs, backpressure guarantees that tokens cannot be lost and is implemented using the *full* signals. The firing rule with backpressure is implemented using a simple state machine shown in figure 3.2. This state machine uses the *empty* signals from all the input arcs and the *full* signals from the destination node as control signals.

The operation of a dataflow node is implemented using a combinatorial circuit (denoted with *Comb. circuit* in figure 3.6) which can be anything like addition, multiplication, subtraction etc. Applications can be implemented by connecting the VHDL implementations of the nodes with signals together. As the arcs are implemented inside of the dataflow nodes, synchronization is performed automatically.

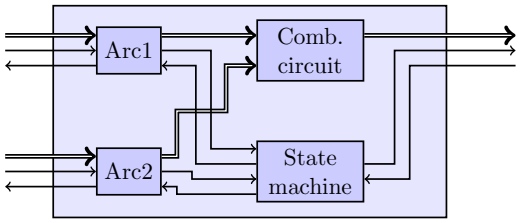


Figure 3.6: Components in dataflow node with backpressure

3.3 FIFO-size for full throughput

Consider the implementation of a simple dataflow graph containing two nodes shown in figure 3.7. The size of the FIFO contained within the arc can be one element. Both producer and consumer change their state based only on the state of the arc. During startup of the system, the arc is empty and both state machines (SM_1 and SM_2) are in the waiting state. The producer makes a transition to the processing state because the arc is empty. The consumer remains in the waiting state because the arc contains no tokens yet. After a rising edge of the clock, the predictors in the arc predict that the producer should go to the waiting state and the consumer should go to the processing state. After yet an other rising edge, the predictors produce the reverse prediction, now the consumer should wait while the producer should go to the processing state. The effective performance of a node containing a combinatorial function is therefore one token per two clock cycles. This is because there is only one position available and both nodes can not read and write at the same time. Although it is possible to achieve full performance with only a single position in the arc, it requires a combinatorial path through all nodes. An example is a pipeline, the whole pipeline should stall if there is no storage available anymore.

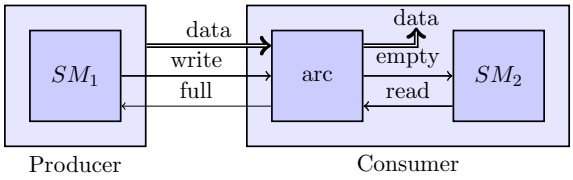


Figure 3.7: Dataflow graph for full performance

Figure 3.8 shows the timing behavior of the dataflow graph shown in figure 3.7. Before the first rising edge of the clock, both state machines are in the waiting state. During the rising edge, the state machine of the producer (SM_1) goes to the processing state (*proc*) because the arc is not full. The state machine of the consumer (SM_2) remains in the waiting (*wait*) state as long as the arc is empty. When the producer is in the *proc* state, the predictor

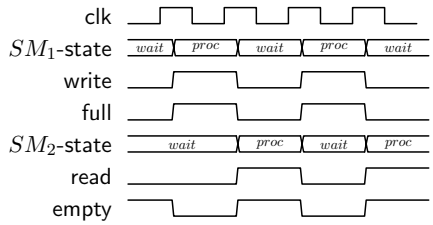


Figure 3.8: Toggling behavior when size of FIFO is 1.

in the arc informs the producer to go to the *wait* state again because the arc will be full in the next state. After the second rising edge of the clock, the consumer is in state *proc*. The empty prediction in the arc makes sure that, during the third rising edge of the clock, SM_2 goes to state *wait* again.

Because the FIFO can contain at most one token, both the producer and consumer have to wait before the arc is available for them. This results in the toggling behavior as shown in figure 3.8. The arc is the bottleneck because it will be full after a single write. The maximum throughput is therefore restricted to one token per two clock cycles.

By increasing the size of the FIFO to two, the toggling disappears. The full-signal generated by the full-predictor now remains *false* because the consumer starts reading during the writing of the second token. Writing a token to the arc and reading a token at the same time has no effect on the number of tokens stored in the arc. The producer and consumer can therefore continue at full speed of one token per clockcycle. Figure 3.9 shows that after one clock cycle both the *full* and *empty* signal remain low. Both SM_1 and SM_2 can therefore remain in the *proc* state which results in full performance.

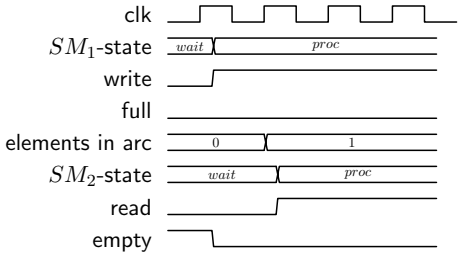


Figure 3.9: Full throughput when size of FIFO is 2.

Appendix A shows an example dataflow node that includes all the concepts explained in this chapter. The code of the ALU for the standard FlexCore (the first implementation of the FlexCore architecture) is given, which includes the arcs for buffering of tokens on the input and the firing rule.

4

Analysis of algorithms

A shown in the chapter with the background information, the beamformer used in LOFAR consists of three major algorithms, Fast Fourier Transformation (FFT)s, complex multiplications and Finite Impulse Response (FIR)-filters. In chapter 6, these algorithms are used to analyse the performance of both implementations of the FlexCore. First, this chapter explores the regularities that can be found in these algorithms. The goal is to exploit locality of reference by reducing the communication overhead caused by the matching procedure as explained in the chapter on background information (chapter 2). The overlap among the three algorithms is used to design an Arithmetic Logic Unit (ALU) which is able to perform the three algorithms. The three algorithms are analysed and than combined into a single dataflow graph which will form the major part of the ALU.

4.1 Complex multiplication

Complex multiplications are used in both the FFT and phase shifts. Consider a complex multiplication $Z = Z_1 \times Z_2$ where $Z_1 = a + ib$ and $Z_2 = c + id$ are both complex numbers. By writing down the multiplication in the canonical form(equation (4.1)), the number of real valued operations can be found.

$$Z = Z_1 \times Z_2 \Rightarrow (a+ib)\times(c+id) = ac+iad+ibc-bd = ac-bd+i(ad+bc) \quad (4.1)$$

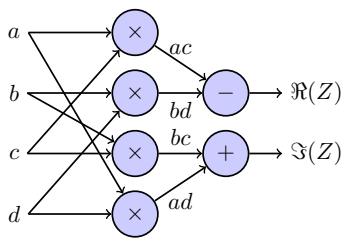


Figure 4.1: DFG of complex multiplication

Both the real part $\Re(Z) = ac - bd$ and the imaginary part $\Im(Z) = ad + bc$ require two real valued multiplications which makes a total of four multiplications, one addition and one subtraction. The corresponding DFG is depicted in figure 4.1.

4.2 FFT

As shown in the background information of chapter 2 and in [24], the Fast Fourier Transformation (FFT) is the algorithm which splits the spectrum in small parts such that phase shifts can be easily implemented. An FFT is a computationally efficient implementation ($Nlog_2(N)$ operations for N input samples instead of N^2) of the Discrete Fourier Transform (DFT). In this section, the FFT is derived from the definition of the DFT. From the FFT the basic building block, called a butterfly operation[21], is derived which is the smallest FFT possible. The butterfly operation will be combined with a partial FIR filter and the complex multiplication in section 4.4 to form an ALU which can execute them all.

First consider the definition of the DFT shown in equation (4.2). This function operates on blocks of N samples from which a spectrum of N points is calculated.

$$\hat{X}_k = \sum_{n=0}^{N-1} x_n e^{-\frac{2\pi i}{N}nk} = \sum_{n=0}^{N-1} x_n W^{\frac{kn}{N}}, \quad \text{where } k = 0, \ldots, N-1 \tag{4.2}$$

Where the complex factor $W^{\frac{kn}{N}}$ is called a twiddle factor[21]. The sum of equation (4.2) can be split in an odd and even part:

$$\hat{X}_k = \sum_{m=0}^{N/2-1} x_{2m} W^{\frac{k2m}{N}} + \sum_{m=0}^{N/2-1} x_{2m+1} W^{\frac{k(2m+1)}{N}} \tag{4.3}$$

By looking only at the first part of the spectrum and using the identity $W^{\frac{k2m}{N}} = W^{\frac{km}{N/2}}$, the definition of the FFT can be found as shown in equation (4.4).

$$\hat{X}_k = \sum_{m=0}^{N/2-1} x_{2m} W^{\frac{km}{N/2}} + W^{\frac{k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} W^{\frac{mk}{N/2}}, \quad k = 0, \ldots, \frac{N}{2} - 1 \tag{4.4}$$

The second half of the spectrum is given by equation (4.7) using the following equalities:

$$W^{\frac{m(k+N/2)}{N/2}} = W^{\frac{mN/2}{N/2}} W^{\frac{mk}{N/2}} = W^{\frac{mk}{N/2}} \tag{4.5}$$

$$W^{\frac{k+N/2}{N}} = W^{\frac{N/2}{N}} W^{\frac{k}{N}} = -W^{\frac{k}{N}} \tag{4.6}$$

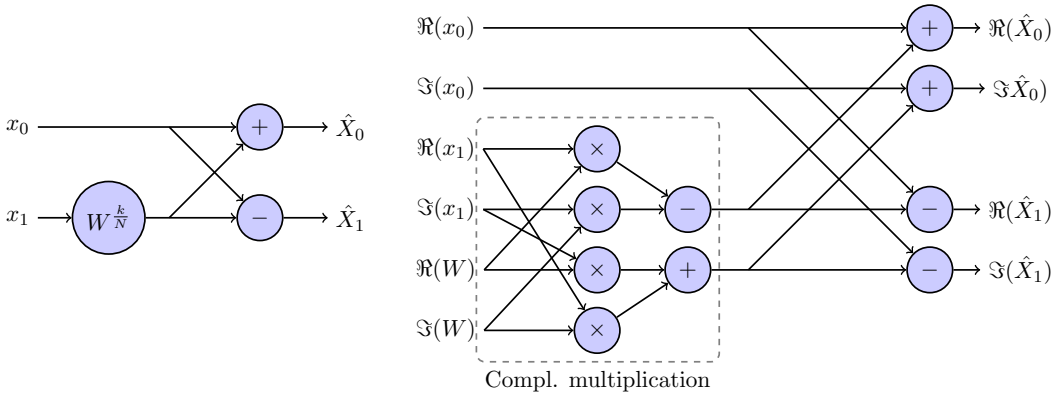
$$\hat{X}_{k+N/2} = \sum_{m=0}^{N/2-1} x_{2m} W^{\frac{km}{N/2}} - W^{\frac{k}{N}} \sum_{m=0}^{N/2-1} x_{2m+1} W^{\frac{mk}{N/2}}, \quad k = 0, \ldots, \frac{N}{2} - 1 \quad (4.7)$$

By using $N = 2$ in equation (4.4) and equation (4.7), the smallest FFT possible, the butterfly operation, can be found. This gives two complex results $\hat{X}_0$ and $\hat{X}_1$.

$$\hat{X}_0 = \sum_{m=0}^{2/2-1} x_{2m} W^{\frac{km}{2/2}} + W^{\frac{k}{N}} \sum_{m=0}^{2/2-1} x_{2m+1} W^{\frac{mk}{2/2}} = x_0 + W^{\frac{k}{N}} x_1 \quad (4.8a)$$

$$\hat{X}_1 = \sum_{m=0}^{2/2-1} x_{2m} W^{\frac{km}{2/2}} - W^{\frac{k}{N}} \sum_{m=0}^{2/2-1} x_{2m+1} W^{\frac{mk}{2/2}} = x_0 - W^{\frac{k}{N}} x_1 \quad (4.8b)$$

This formula can be directly translated into a dataflow graph as shown in figure 4.2a which gives the butterfly structure. Because the twiddle factor can be implemented using a complex multiplication, the graph of figure 4.1 is reused. Figure 4.2b shows the butterfly structure for real valued signals using the graph for complex multiplications. The total number of operations required to execute a butterfly operation is four multiplications, three additions and three subtractions.



(a) Butterfly of complex signals

(b) Butterfly of real valued signals

Figure 4.2: Complex and real valued dataflow graph og the butterfly operation

4.3 FIR filter

The FFT in the filterbank is preceded by FIR filters which enhance the signals before being processed by the FFT. Combined with downscaling, a filterbank

is constructed[22],[24]. A FIR filter is a linear combination of the current and previous samples as shown in the following recurrence equation:

$$y_n = \sum_{i=0}^{N-1} C_i \times x_{n-i} \tag{4.9}$$

N is the number of filter taps and $N-1$ is the order of the filter. Analysis of FIR filters is usually done in the Z domain where a FIR filter is expressed as a polynomial. Equation (4.10) shows the polynomial of a FIR filter.

$$H(z) = C_{N-1} * Z^{N-1} + C_{N-1} * Z^{N-1} + ... + C_1 * Z^{-1} + C_0 \tag{4.10}$$

This function is called the transfer function of a filter and it shows the sum from equation (4.9) implemented in the Z domain. Every term $C_n * Z^{-n}$ represents a coefficient being multiplied with a delayed input sample. Multiplying a signal with Z^{-n} corresponds to delaying the input signal with n samples. Equation (4.10) can be translated into a dataflow graph as shown in figure 4.3 which is called the standard form [21].

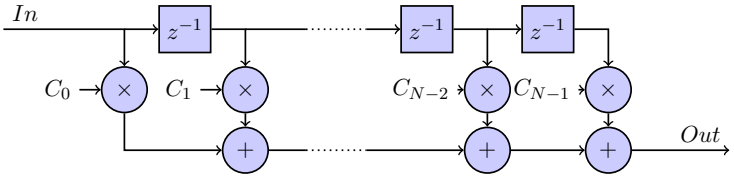


Figure 4.3: FIR Filter

Any valid mathematical rearrangement of the transfer function of equation (4.10) gives the same filter response but results in another structure of the dataflow graph. The rearrangement shown in equation (4.11) of the transfer function results in the dataflow graph of figure 4.4 which is called the transposed form[21].

$$H(z) = C_0 + Z^{-1}(C_1 + Z^{-1}(C_2 + ... + Z^{-1}(C_{N-2} + Z^{-1}C_{N-1})...)) \tag{4.11}$$

The advantage of the transposed form is that the longest combinatorial path is only a combination of one multiplier and one adder. The longest combinatorial path in the standard form starts at the input, passes trough the first multiplier and then passes trough all adders on the bottom. The combination of a multiplication, adder and register of the transposed form are the basic building blocks of filters: a filter tap. The transposed form also shows overlap with the complex multiplication of figure 4.1 which is used to merge the three algorithms to one single graph as shown in section 4.4.

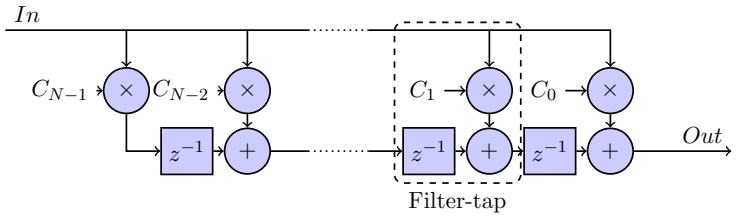
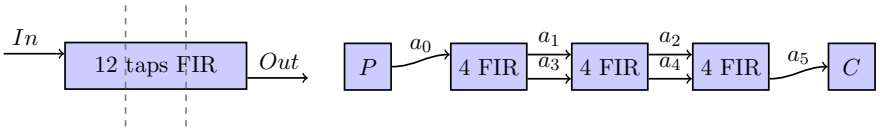


Figure 4.4: FIR Filter in transposed form

Partitioning and sequential execution

By partitioning the filter into smaller slices, they can be executed sequentially. This process is called folding [32]. The size of each slice is chosen to be 4 taps such that the number of multiplications is the same as for complex multiplications. Every tap consists of a multiplier and adder i.e. a single slice consist of 4 multipliers and 4 adders. The number of multiplications is therefore the same for filtering, complex multiplication and the butterfly. An example of a sliced FIR filter is shown in figure 4.5.



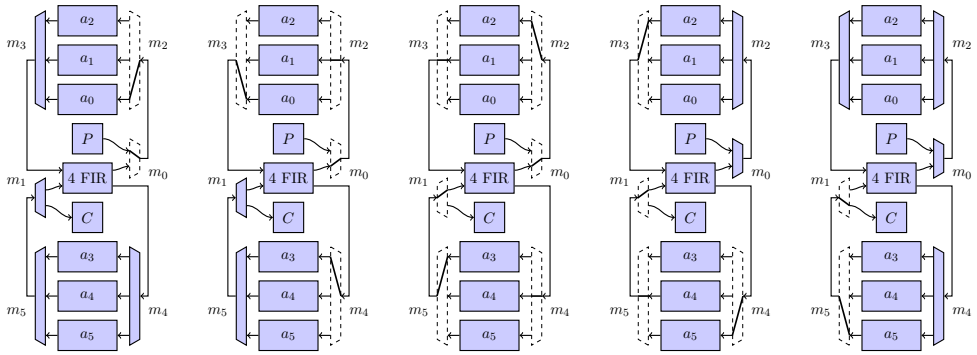
(a) 12 taps FIR (b) Sliced FIR

Figure 4.5: Slicing of FIR filter

The 12-tap FIR filter shown in figure 4.5a is partitioned in $\lceil \frac{12}{4} \rceil = 3$ slices as shown in figure 4.5a. These slices are executed sequentially by an ALU which is able to execute a single slice at once. The results of a slice are forwarded to the next slice or the output. The input data for a slice comes either from another slice or from the input of the filter.

Figure 4.6 shows the flow of tokens in the sequentialized filter. Every arc in the dataflow graph of figure 4.5b is translated into a buffer a_n of figure 4.6. In order to support two incoming and two outgoing streams of data for the four taps FIR, the arcs are divided into two groups (a_{0-2} and a_{3-5}) that can be used in parallel. A single multiplexer is used to select between the producer P and the FIR which are the only producers for a_{0-2}. The data from group a_{3-5} is forwarded to the lower input of the FIR or the consumer C. The upper input of the FIR accepts only data from a_{0-2} and the lower output only produces data for a_{3-5}, this all follows from the dataflow graph of figure 4.5b.

The execution of the dataflow graph of figure 4.5b is performed in 5



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4 (e) Phase 5

Figure 4.6: Sequential execution of FIR filter.

phases[1], one phase for every node. During phase 1, only the producer P is active and sends data to arc a_0 via multiplexers m_0 and m_2. Note that P may send any number of tokens to the arc as long as it fits in the arc and when all other nodes in the graph consume and produce the same number of tokens. In phase 2 (figure 4.6b), the tokens sent into a_0 are used by the first filter slice.

During phase 2 the first filter slice is active which consumes all tokens in arc a_0 (selected by multiplexer m_3) and produces tokens in arc a_1 and a_3. As can be seen in figure 4.6b, a_1 and a_3 are selected by multiplexers m_2 and m_4. m_0 now selects the upper output of the FIR slice instead of the producer P. Note that the lower input of the slice is not shown here but it can be fed with a stream of zeros.

During phase 3 a FIR slice is executed which requires two input arcs and two output arcs. Again, the input arcs are selected by m_3 and m_5 and the output tokens are forwarded by m_2 and m_4. Corresponding to the DFG of figure 4.5b, tokens are consumed from a_1 and a_3 and and the resulting tokens are sent to arc a_2 and a_4 such that the last FIR slice can consume these again.

The tokens produced in phase 3 are consumed by the last filter slice in phase 4 (figure 4.6b). Multiplexers m_3 and m_5 are now selecting a_2 and a_4 which contain the tokens produced in the previous phase. These tokens are now consumed by the slice which produces output tokens which are sent to a single arc, a_5. During the last phase, figure 4.6e, multiplexers m_1 and m_5 are configured such that tokens from a_5 can be consumed by the consumer C.

Every time a new slice is scheduled the corresponding set of 4 coefficients has to be supplied too. Although the flow of tokens can be implemented like shown in figure 4.6, the filter-state is not preserved. Every slice contains

[1]Note that phase 1 and 5 can be combined in a single phase as no multiplexers and arcs are used in both phases

four delay elements (z^{-1}) which contain intermediate results from previous slice executions. This means that every slice has a corresponding state which has to be loaded before it may consume tokens and execute. The ALU that executes the FIR operation therefore must be able to load the complete state of the slice first. By adding a multiplexer in front of every register (z^{-1}), the state can be loaded using the $StateIn$ inputs as depicted in figure 4.7. The state comes from a module outside of the ALU which will be introduced in section 5.3.

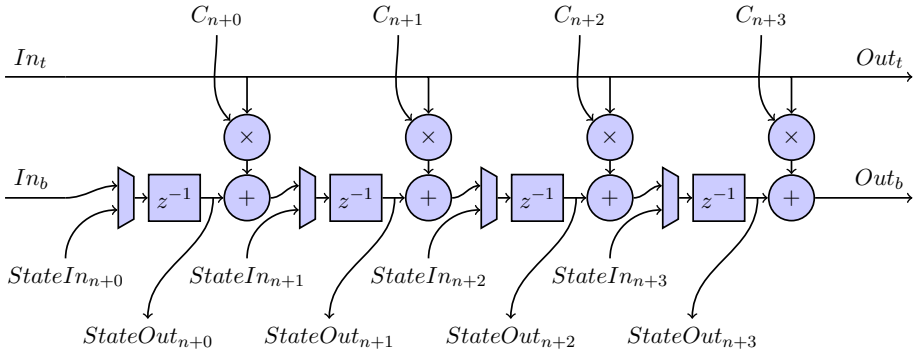


Figure 4.7: FIR slice with state loading

As shown in figure 4.7 the state of a FIR slice is loaded using the $StateIn_n$ inputs. During loading of the state, the input of the registers is connected to the $StateIn_n$ input. During normal execution the multiplexers are in the upward position such that the slice of figure 4.7 resembles the transposed FIR structure of figure 4.4. When the next slice should be executed, the changed state of the current slice has to be stored. For this, the $StateOut_n$ outputs are used. Again, the state of the slices is stored outside of the ALU.

4.4 Merge of algorithms

The analysis of the three beamforming algorithms has shown that they all require four multiplications and a number of additions and subtractions. In terms of operations, the butterfly is the most complex graph (4 multiplications, 3 additions and 3 subtractions). The complex multiplication is part of the butterfly, but the 4 taps FIR cannot be found in the butterfly because of a missing addition. By modifying a subtractor from the butterfly operation such that it can execute both subtraction and addition, the FIR slice also matches with the butterfly in terms of operations. The three algorithms can now be merged into a single graph which exploits the overlap in operations. Figure 4.8 shows the resulting graph with additional multiplexers for switching between different functionality.

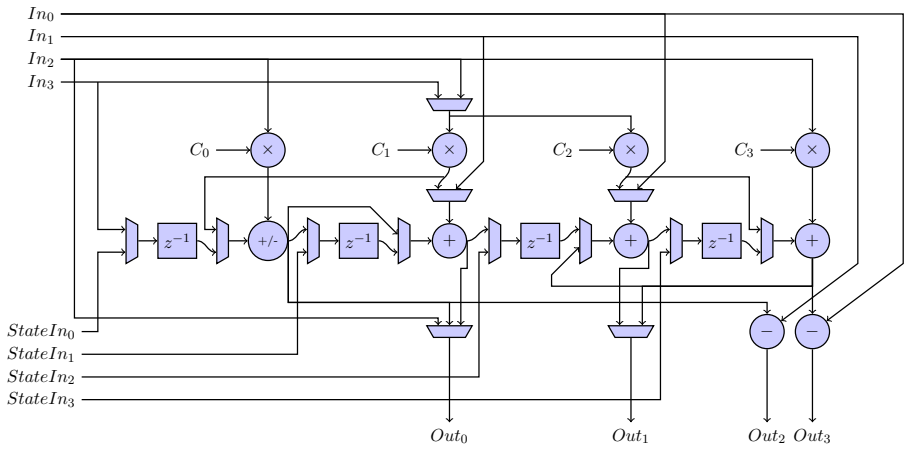


Figure 4.8: Merged DFGs for ALU design

Figure 4.9 shows the graph configured as a FIR slice. Note again that executing a filter slice requires two phases, first the slice-state is loaded using the $State_n$ signals while execution happens during the second phase. The state can be loaded by setting every multiplexer in front of a register (z^{-1}), in the down position. During normal execution the multiplexers are in the upward position as shown in figure 4.9. The inputs $In_{t,b}$, coefficients $C_{0,1,2,3}$ and outputs $Out_{t,b}$ match with the slice shown in figure 4.7.

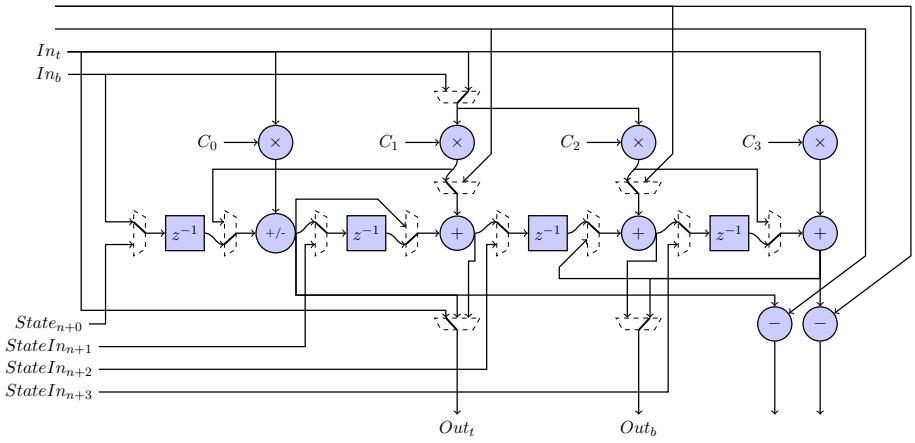


Figure 4.9: Configuration for FIR slice

By selecting the multiplexers as depicted in figure 4.10, the graph is configured as a complex multiplier. Z_1 is presented on the inputs while Z_2 is presented as constants for the multipliers. Note that both the real and imaginary part of Z_2 are duplicated as input constants. A complex multiplication contains no state so the registers and the multiplexers in front of them are

not used. The result Z is presented on the output using two multiplexers. The remaining outputs from the subtractors are only used in the butterfly operation.

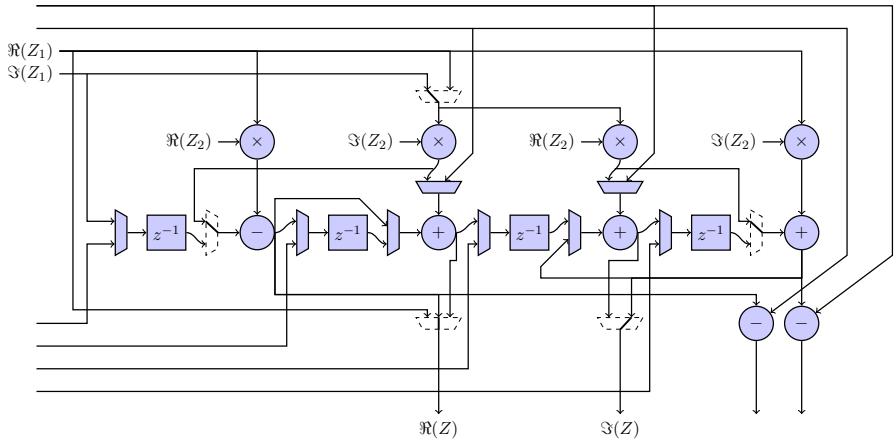


Figure 4.10: Configuration for complex multiplication $Z = Z_1 \times Z_2$

Figure 4.10 shows the configuration when the butterfly operation is selected. The twiddle factor of the butterfly operation is implemented using a complex multiplication so the constants are duplicated again over the multipliers. Also the butterfly operation is state-less so the registers with the corresponding multiplexers are not used. This selected mode shows the execution of the graph of figure 4.2b.

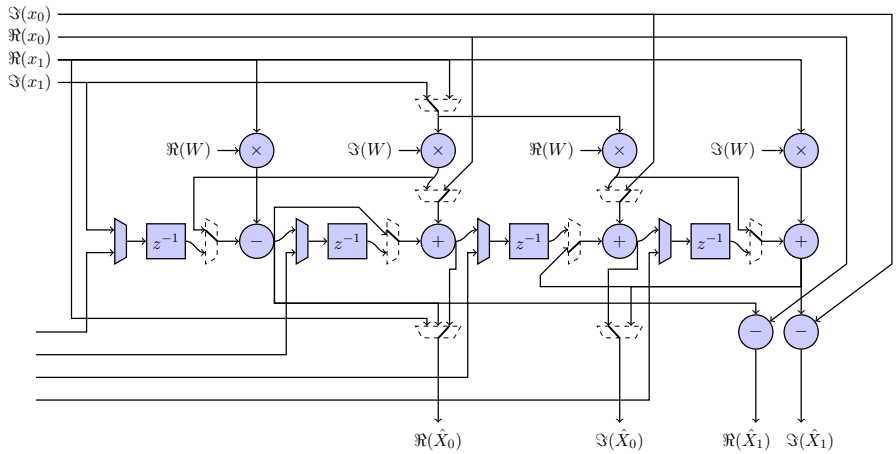


Figure 4.11: Configuration for butterfly

5

FlexCore architecture

This chapter presents the design of the FlexCore. The FlexCore has been designed in two phases: first a standard dataflow architecture including back-pressure has been designed to act as reference design. This is presented in section 5.1 and section 5.2. During the second phase elaborated in section 5.3, the architecture is extended such that the merged dataflow graph presented in chapter 4 can be used as ALU for efficient execution of beamforming operations.

5.1 Overview of architecture

Figure 5.1 shows an overview of the FlexCore with its modules. The FlexCore has a similar circular structure as the early dataflow architectures presented in [12],[17] and [19]. It consists of three main modules, the Router which routes internal tokens of the FlexCore and tokens from the NoC, the Dispatcher which implements the firing of nodes based on the availability of tokens and the ALU which performs the actual calculations.

The router manages the flow of tokens from the NoC and for the processor itself. Every token that should be processed by the Router has a value and a destination-address. Based on this address the router can determine whether to send an incoming token to the dispatcher or to the NoC. Tokens flowing trough the processor have the following format:

$$token = \{\underbrace{globalAddress, localAddress, inputAddress}_{destination}, value\} \tag{5.1}$$

As shown in the expression above, the address consists of three parts. The $globalAddress$ is used by the router to determine whether a token should be sent to the dispatcher or to the NoC. A global address is assigned to the FlexCore, this means that the router forwards an incoming token to the dispatcher when $globalAddress$ of that token is equal to the address assigned to the FlexCore. When this is not the case the router sends it to the NoC. The $localAddress$ field is used to address nodes of the DFG according to the

ETS principle explained in figure 2.3. *inputAddress* indicates to which input of the node in the DFG, the token should be sent. The *value* field of the token contains the actual data. All data values in the FlexCore are 16 bit signed numbers which is a common wordlength for DSP applications[33],[34]. The general architecture of the FlexCore is shown in figure 5.1.

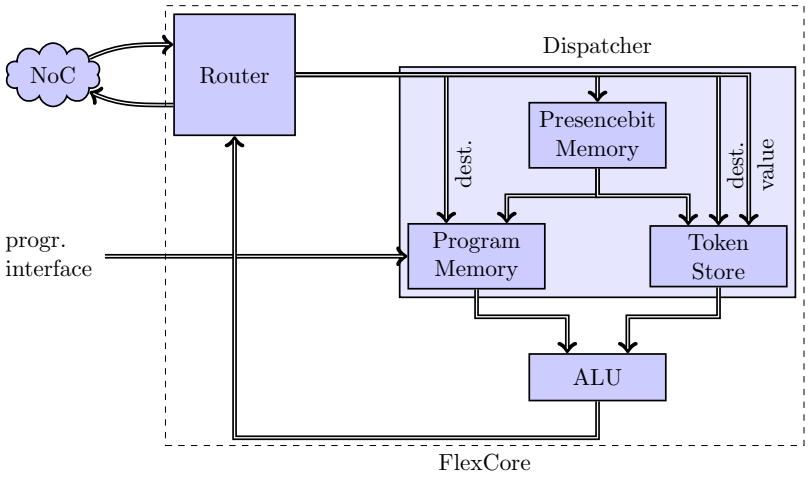


Figure 5.1: FlexCore architecture overview

The dispatcher is responsible for matching tokens and sending them with an instruction to the ALU. The router sends tokens to the dispatcher which implements the ETS principle. The dispatcher itself is constructed using three sub modules: the presence bit memory, program memory and token store.

The presence bit memory uses only the local address from the incoming token. Based on this local address a presence bit is fetched from the memory. This bit is sent to both the program memory and the token store. During every access of a presence bit, the value is inverted. When the presence bit being sent out contains *true*, a match is found and an instruction is dispatched on the ALU.

The program memory accepts both the local address from the incoming token as well as the presence bit produced by the presence bit memory. When the presence bit contains the value *true*, the instruction addressed by the local address should be scheduled. This instruction is stored in a memory where every location contains the instruction and several destination addresses. These destination addresses are the nodes to which the result of the operation should be sent as explained in figure 2.3. When the presence bit is *false*, the instruction should not be scheduled. Therefore both the address and the presence bit are removed from the arc and no tokens are produced by the program memory.

The token store is triggered in the same way as the program memory, it produces only a token on the output when the presence bit token contains a

value *true*. The token store accepts three inputs, a presence bit, the value of the incoming token and the local destination address of that token. When the presence bit is *false* no match has occurred so the incoming token is stored in the memory of the token store addressed by the local destination address from the incoming token. When both operands are available i.e. a match occurs, a presence bit containing *true* is received by the token store. The token store sends both the incoming token and the previously stored token to the ALU. When two tokens are sent to the same input of a node, the dispatcher interprets this as a match which is not the case. The input data should be restricted such that this situation will never occur.

The ALU performs all the calculations and accepts the instruction and destination(s) from the program memory. The operands are received from the token store. When the operands and the instruction are available in the FIFOs, the firing rule is satisfied and the ALU starts the execution. The resulting value is sent sequentially to all destinations (max. 4), the operations may take 4 clockcycles to complete from which only one cycle is needed for the calculation.

5.2 Implementation

ALU

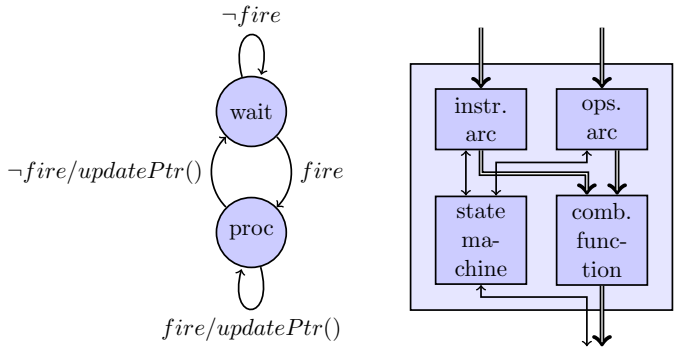
The first design of the FlexCore contains a small ALU which is only able to perform additions, subtractions and multiplications. The ALU has two inputs, one for receiving instructions and one for the operands. As with every input of a dataflow node, backpressure and buffering is supported by using arcs. The firing rule and execution of this module is implemented by a statemachine. The execution may only start when there is an instruction on the instruction arc $instr.\ arc$, when a set of operands[1] is available in the operands arc $ops.\ arc$ and if there is still space available on the arc to which the result will be sent (Router). The firing condition then becomes:

$$fire = \neg empty(instr.\ arc) \wedge \neg empty(ops.\ arc) \wedge \neg full(Router) \qquad (5.2)$$

The general structure of the ALU and the firing rule implemented by the state machine shown in figure 5.2.

The ALU can execute nodes from a dataflow graph with up to four destinations. The result, produced by the operation, is therefore sent to all the nodes in the dataflow graph to which an output of the producing node is connected. The output of the ALU however only allows one token per clock cycle to be sent to the router. The tokens are therefore sent sequentially to all destinations.

[1]The two operands for the instruction are sent in a single token.



(a) Statemachine (b) Internal structure

Figure 5.2: Implementation of ALU

The number of destinations $destCnt$ is encoded in the instruction which has the following format: $(instr, destCnt, dest_0, dest_1, dest_2, dest_3)$ where $instr$ can be $mult$, add or sub and $destCnt$ must be in $\{1, 2, 3, 4\}$. The statemachine iterates over $destCnt$ destinations to which the result will be sent. Every time the statemachine leaves the $proc$ state, a pointer pointing to the current destination is updated. Updating the pointer is performed by $updatePtr()$ shown in algorithm 1.

Algorithm 1: Implementation of $updatePtr()$

```
1 if curDest = destCnt then
2 |   curDest ← dest_0
3 else
4 |   curDest ← curDest + 1
```

After a reset of the FlexCore, $curDest$ is initialized to $dest_0$, i.e. during the first cycle the first destination is always selected. After the first cycle a transition is made in the statemachine where $updatePtr()$ is executed. Depending on the number of destinations, $destCnt$, the next destination is selected or the first($dest_0$) for the next execution of the ALU. When the result is sent to the last destination, both input arcs will be read such that storage is freed for the next instruction and operands.

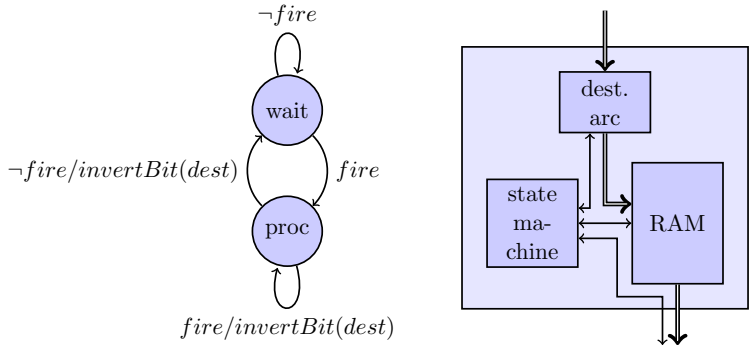
Dispatcher

The dispatcher is not a complete dataflow node like the ALU but is a wrapper around the presence bit memory, program memory and the token store. It is however constructed in such a way that the behavior from outside is the same as any other dataflow node. The design therefore remains consistent with HSDF. As can be seen in figure 5.1 the dispatcher has only one input

which is used by all three modules inside of the dispatcher. The $full$ signal from dispatcher to router must therefore be dependent on full signals from all modules in the dispatcher. The input of the dispatcher is full when any of the $full$ signals of the internal modules are set. The following approach is applied to the signal coming from the presence bit memory: the $full$ signal going into the presence bit memory should be set when either the $full$ signal from program memory or token store is set. The outputs of the dispatcher are simply forwarded signals from the program memory and token store. This requires no additional logic.

Presence bit memory

Detecting a match is performed by the presence bit memory. For binary (only two inputs per node) dataflow graphs only a single bit is needed per node in the DFG[19]. Initially, all presence bits in the memory are reset when the whole FlexCore is reset. This means that for all nodes in the DFG, no token has yet been received. When a token arrives at one of the inputs, the bit is set. Based on the presence bit and a new incoming token addressed to the same node, a match is found and the presence bits memory will inform the program memory and token store.



(a) Statemachine (b) Internal structure

Figure 5.3: State machine and internal structure of the presence bit memory

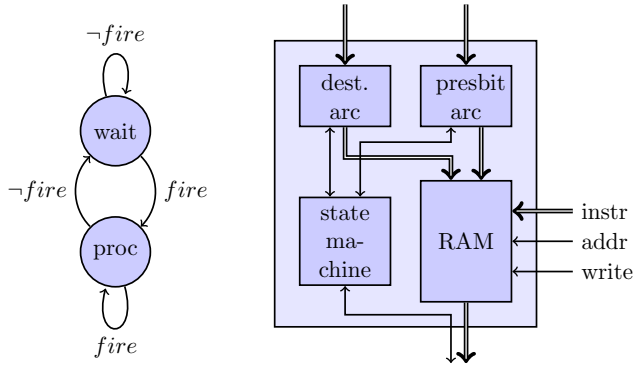
The presence bit memory has only a single input, the destination address of the token sent from router to dispatcher is used. This address is, conform to the dataflow model, stored in an arc (see structure of the module in figure 5.3b). The firing rule and reading from the arcs is implemented by the statemachine shown in figure 5.3a. The firing rule states:

$$fire = \neg empty(dest.\ arc) \wedge \neg full(Program\ memory/Token\ store) \quad (5.3)$$

For every incoming destination address, the bit pointed by that address is inverted. Although only a single bit for every node in the DFG is required, the number of tokens that can be stored on an arc of the DFG is restricted to one. A match is found when an address of the presence bit memory is addressed twice. The input to which the token should be sent is ignored to save hardware. The inputs to which the tokens forming a match should be sent, is determined in the token store.

Program memory

The program memory selects an instruction when a match is found in the presence bit memory. Which node should be executed is determined by the destination address from the token sent from router to dispatcher. When the presence bit memory reports that no match is found, the program memory simply does not send an instruction to the ALU. Additional to the dataflow in- and outputs, the program memory has an external programming interface. This is used by the testbench to load the dataflow graph (the program that should be executed) into the FlexCore. The internal structure of the program memory can be seen in figure 5.4b.



(a) Statemachine (b) Internal structure

Figure 5.4: State machine and internal structure of the program memory

The state machine shown in figure 5.4a is only used to implement the firing rule. The actual selecting of the memory is performed combinatorially during the *proc* state. When the presence bit memory sends a token containing *true* i.e. a match is found, the instruction addressed by the incoming destination address is sent to the ALU. When *false* is received, the address is ignored and nothing is sent to the ALU. An instruction in the program memory has the following format: $(instr, destCnt, dest_0, dest_1, dest_2, dest_3)$, which is exactly the same as the instruction input of the ALU. Again, the firing rule depends on the in- and outputs. The firing rule for the program memory therefore is:

$$fire = \neg empty(dest.\ arc) \wedge \neg empty(presbit\ arc) \wedge \neg full(ALU) \tag{5.4}$$

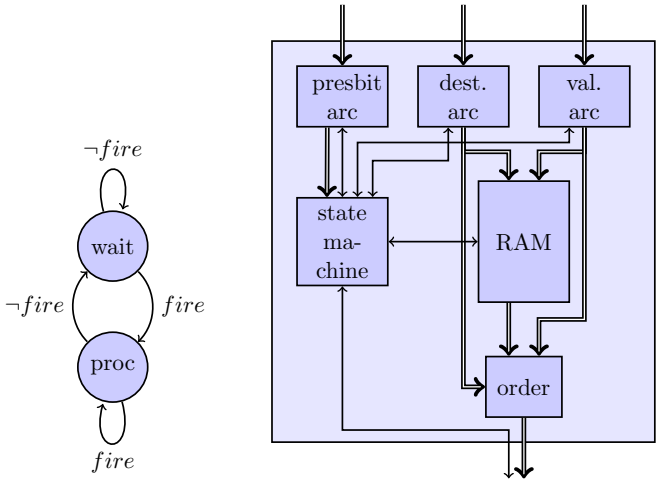
Token store

The token store is responsible for storing tokens on the arcs of a DFG until all tokens for a node are available. The nodes of the DFG always have two inputs, the token store therefore only has to be able to store one token per node. When the second token of a node arrives, a match is found and the incoming token combined with the token with the same address (destination in the DFG) form the set of operands. Tokens triggering a match therefore never have to be stored but are directly forwarded to the ALU. The token store always stores the first arriving token for a node, independent to which input the token is sent. Only the value contained within the token is stored in the token store because the input can be determined when the second token arrives. Because of this, the standard FlexCore only supports two input nodes in the DFG, the input to which the first token should be sent is the inverse of the input to which the second is sent: If the first token is sent to the left input of the destined node, the second token is sent to the right input and vice versa. This imposes a restriction on the arrival of tokens: only one token can be stored on an arc. This is caused by the fact that the token store can only store one token per node and the presence bit memory reports a match when two tokens with the same address arrive (independent of whether the are sent to the same input). The order in which two operands for a node arrive is not restricted, a function $order()$ makes sure the the operands are sent to the correct input.

The statemachine for the token store is like all others: processing starts only when the firing rule is satisfied. The token store requires three incoming tokens: a presence bit, the destination of the incoming token and the value. The firing rule then becomes:

$$\begin{aligned} fire = \neg empty(presbit\ arc) \quad &\wedge \quad \neg empty(dest.\ arc) \wedge \\ \neg empty(val.\ arc) \quad &\wedge \quad \neg full(ALU) \end{aligned} \tag{5.5}$$

The general architecture of the token store is shown in figure 5.5b. Based on the presence bit coming from the presence bit memory, the incoming value is stored at the address read from *dest. arc.* when the presence bit memory reports that no match occurred, the incoming token is stored. When a match does occur for the same node in the DFG the previously stored token combined with the incoming one are sent to the ALU as operands. As explained in the beginning of this section, the order of arrival for tokens is not known until runtime. An *order* module is introduced to guarantee the correct order of arrival of the operands. Algorithm 2 shows the functionality of the token store.



(a) Statemachine (b) Internal structure

Figure 5.5: State machine and internal structure of the token store

Algorithm 2: Algorithm for the token store

```
1  while true do
2      while ¬fire do
3          wait;
4      if presbit = true then
5          write ← true;
6          operandsOut ← order(dest, value, loadFromRAM(dest));
7      else
8          write ← false;
9          operandsOut ← ∅;
10         storeInRAM(value, dest);
```

Router

The router is the only module in the FlexCore without a statemachine. This is because the firing rule depends on the destination to which an incoming token should be sent. If for example a token should be sent to the dispatcher, the firing rules require that there is space at the input of the dispatcher. When a token should be sent out of the processor, firing does not depend on the dispatcher but on the router output. In order to avoid a complex statemachine, the routing is performed combinatorially.

The local input of the router has a higher priority than the input from outside the core. This means that local communications caused by tokens flowing between nodes of the DFG are less influenced by tokens from outside

the core. Tokens from outside the core can only enter if there is space in the input of the router. Because the local input of the router has priority over the external input, external tokens are accepted only if temporarily no tokens are in the local input. This means that the dispatcher is able to accept data from outside of the core again. Figure 5.6 shows the structure of the router.

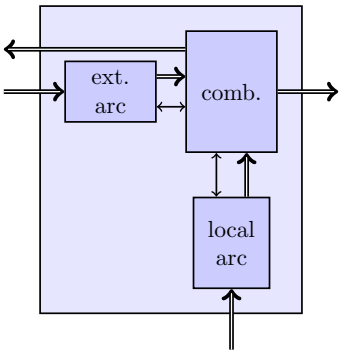


Figure 5.6: Combinatorial implementation of firing rule in router

For the availability of tokens in the arcs, three basic cases can be distinguished: there is data available in the local arc, there is data in the external arc or there is no data at all. The algorithm of the router in shown in algorithm 3.

Algorithm 3: Algorithm for routing tokens

```
1 if ¬empty(localArc) then
2     if localToken.dest = local ∧ ¬full(Dispatcher) then
3         dataToDispathcer ← localToken;
4     else if localToken.dest = external ∧ ¬full(External) then
5         dataRouterOut ← localToken;
6 else if ¬empty(extArc) ∧ ¬full(Dispatcher) then
7     dataToDispathcer ← extToken;
8 else
9     wait for data;
```

5.3 Extended architecture for beamforming ALU

In order to include the DFG of the merged algorithm shown in figure 4.8 (the DFG that combines a complex multiplication, butterfly and a FIR slice into a single graph), the architecture requires some changes. First a match cannot be found based on a single presence bit. The number of required input tokens is now dependant on the number of inputs of a node in the DFG. For example, a

FIR slice and a complex multiplication require two inputs while the butterfly operation requires four inputs. Note that the complex multiplication requires two inputs because the second complex operand is supplied as a constant from the program memory. Secondly a FIR slice has state which has to be supplied to the ALU every time a slice is executed. After execution the FIR slice has a new state and it should be stored for subsequent execution of the same slice.

First the required changes to the dispatcher are elaborated in order to support the matching for different numbers of inputs. This is followed by the handling of state from FIR slices. After that the implementation of the modules is explained. The extended architecture is shown in figure 5.7.

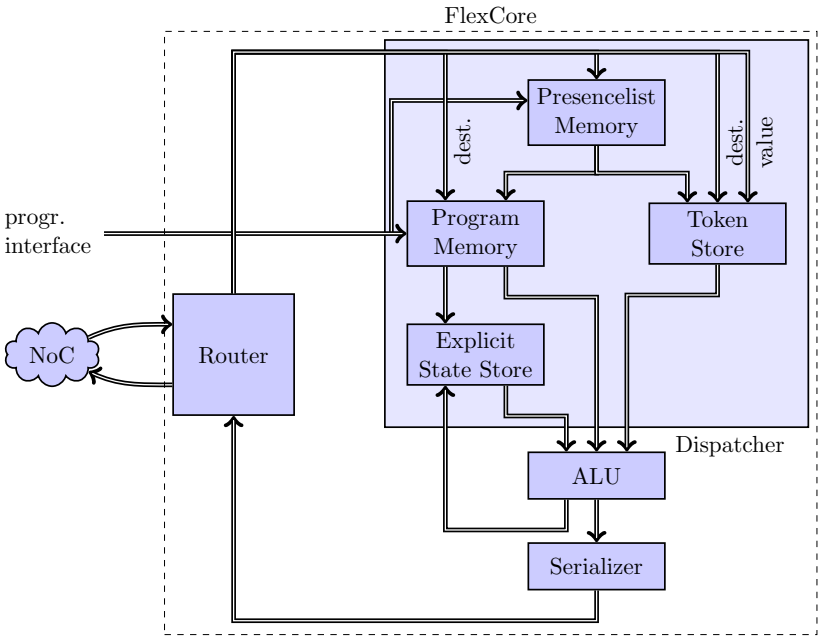


Figure 5.7: Extended FlexCore

Overview

The general structure of the extended FlexCore is very similar to the architecture shown in figure 5.1, except the naming and additional modules. Tokens from outside of the FlexCore are sent to the router which forwards these to the dispatcher. The matching process takes place in the dispatcher by checking whether all operands are available. When this is the case, the *Presencelist memory* reports a match which is sent to both the *Program memory* and the *Token store*. The program memory may now schedule a node from the DFG onto the ALU. Depending on the instruction, a corresponding state is required (currently only a *FIR-slice* has state) and is fetched from the Explicit State Store (ESS). The state for an instruction is therefore always available before

the instruction is scheduled. The operands for the instructions are stored in the *Token Store* until a match is found. All operands, the instruction and an optional state are then sent to the ALU which performs the instruction. Several tokens at once are produced by the ALU which are sent sequentially to the router by the *Serializer*. The router of the standard FlexCore is therefore reused.

Changes to Dispatcher

Because the number of inputs for nodes of a DFG for the extended FlexCore may differ, the matching cannot be based on a single presence bit anymore. When a node has four inputs, four operands must be available while a two input node only requires two operands. The presencelist memory therefore must know if the addressed node has four or two inputs. This is solved by adding an additional field (*type*) to the elements of the presencelist memory. This type field represents how many inputs that particular node has and is set during programming of the FlexCore. Every element now consist of a type field and a list of bits where every bit represents the availability of an operand. The extended FlexCore uses the same addressing scheme from ETS as the standard FlexCore.

The Explicit State Store (ESS) is added to the dispatcher to hold the state information for certain nodes in the DFG. When the program memory encounters a FIR slice, the state address for that instruction is sent to the ESS. The ESS then selects the state for the slice and sends it to the ALU. After execution, the state of the slice is changed and the ALU sends it back to the ESS for the next execution of the same node.

Another difference with the standard FlexCore is that instructions now also contain coefficients for the FIR slices, and complex numbers for complex multiplications and twiddle factors. This is done to restrict the number of inputs for nodes in the DFG and therefore simplifying the matching (a maximum of four tokens now have to be matched instead of six). All elements in the program memory now have a set of four coefficients.

Implementations

Presencelist memory

Instead of single bits, the presencelist memory contains a list of bits for matching (one bit for every input). Based on a type field the presencelist memory can determine whether all operands are available. A single element has the following structure: $(type, (bit_0, bit_1, bit_2, bit_3))$ where *type* can be $twoInputs$ or $fourInputs$. When $type = twoInputs$, the incoming token and the first two bits form the condition for a match. A match is found when the incoming token is destined to the first input and second bit is set or the other way around. When $type = fourInputs$, the whole list of bits is used. A match

occurs when all bits, except the one destined by the incoming token, are set. The presencelist memory sends a token containing *true* to the program memory and token store when a match is found or *false* otherwise. The interface of the presencelist memory to other modules is not changed. The algorithm of the presencelist memory is shown in algorithm 4.

Algorithm 4: Algorithm for the presencelist memory

```
1  while true do
2      while ¬fire do
3          wait;
4      addr ← dest.address;
5      inp ← dest.input;
6      if mem[addr].type = twoInputs then
7          if inp ∪ inputsAvailable = {input_0, input_1} then
8              presbitOut ← true;
9              inputsAvailable ← ∅;
10         else
11             presbitOut ← false;
12             inputsAvailable ← inputsAvailable ∪ inp;
13     else
14         if inp ∪ inputsAvailable = {input_0, input_1, input_2, input_3}
           then
15             presbitOut ← true;
16             inputsAvailable ← ∅;
17         else
18             presbitOut ← false;
19             inputsAvailable ← inputsAvailable ∪ inp;
```

First the presence list memory waits for an incoming destination from the router. When this is available, the local address and the input are extracted from the the destination token (*addr* and *inp* on line 4 and 5). The address is used to select the presence list corresponding to the addressed node. Depending on the *type* field, there should either be two or four tokens available to form a match. Handling the match is performed in the same way as the token store of the standard FlexCore. A match is found when the input destination of the incoming token combined with the set of already received tokens (*inputsAvailable*) is the complete set for the given type. When a match is found, a presence bit with value *true* is sent to the program memory and the token store (line 8 and 15). After sending this presence bit, the set of available tokens on inputs is reset (line 9 and 16) such that new tokens can be received to form a new match for that particular node. When no match is found, the

presence bit sent out, contains the value $false$ and the input of the incoming token is added to the set $inputsAvailable$.

Program Memory

The main difference between the program of from the standard FlexCore and the Extended FlexCore is the support for state. The states of nodes are however located in an other module: the Explicit State Store (ESS). A state is selected using a state address which is sent to the ESS using an additional output of the program memory. A second difference is that the coefficients and twiddle factors are stored in the program memory. This is done to simplify the matching, no matching is needed for constants, and to keep the coefficients close to the ALU, where they are needed. Although the program memory elements are much larger due to the additional set of coefficients, it should be more efficient from a locality of reference point of view. The coefficients stay close to where they will be used instead of being sent from outside of the FlexCore. The new instruction format now has the following form:

$$instr = (operation, dest_0, dest_1, dest_2, dest_3, stateAddr, \\ coef_0, coef_1, coef_2, coef_3) \tag{5.6}$$

The Extended FlexCore only supports complex multiplications, butterfly operations and FIR slices. The *operation* field of the above format can therefore only be *cmult*, $bfly$ or fir. Following the operation are four possible destinations like in the instruction of the standard FlexCore, however the number of destinations is fixed per operation(both fir and *cmult* have two outputs while $bfly$ has four). The *stateAddr* field is the address of the state which corresponds to the given instruction. Currently only FIR slice nodes have state but this could be extended for other stateful operations like an integrator. The last four fields are the coefficients which are used for all operations as constant operands. The algorithm performed by the program memory is shown in algorithm 5.

As can be seen on line 7 of algorithm 5, the fir instruction is the only instruction having a corresponding state. The address of the state is encoded in the instruction and is sent to the Explicit State Store when a fir node is requested.

Token store

Instead of only two inputs, the Extended FlexCore also supports nodes with four inputs (the butterfly operation). After a match is found by the presencelist memory, the token store selects four operands at the same time and sends these to the ALU. The token store contains a memory with an address space four times as large as required to address all possible nodes of a DFG.

Algorithm 5: Algorithm for the program memory of the Extended FlexCore

```
1 while true do
2     while ¬fire do
3         wait;
4     if presbit = true then
5         instrWord ← mem[dest.localAddress];
6         instrOut ← instrWord;
7         if instrWord.instr = fir then
8             stateAddrOut ← instrWord.stateAddr;
```

Every input of every dataflow node therefore has a unique address. Ordering of tokens is not needed anymore because they are stored at the right addresss in the memory. This address can be found by taking the *localAddress*, shifting it two bits to the left and adding the *inputAddress* to it. Every time a token needs to be stored the address of the location where the token will be stored is found this way.

The order of arrival is, the same as with the standard FlexCore, not known until runtime. When the token causing a match, enters the token store, it has to be combined in the right order with the already stored tokens. The token store always sends four tokens containing the operands to the ALU at the same the time. Only those tokens required for calculations are used by the ALU, the rest is ignored.

Algorithm 6 shows how the incoming operands are ordered before they are sent to the ALU. First the local address and the input are extracted from the destination part of the incoming token. The addresses are used to select operands from the memory of the token store. Depending on the value of the presence bit, the incoming token has to be stored or combined with the already stored tokens to form the set of operands. Line 7 shows that the value of the incoming token is stored in a memory addressed by both the local address and the input. By using both the local and the input address all possible inputs of all nodes can be addressed and used to store tokens. When a match is found (the presence bit has value *true*), the incoming token is combined with the stored ones to form the complete set of operands. Based on the input of the token forming the match, the ordering of operands is determined as can be seen from line 10 and further.

Explicit State Store

The Explicit State Store (ESS) holds the states for all the FIR slices. When a slice should be executed, the program memory sends the address of the state

Algorithm 6: Algorithm showing the ordered sending of operands to the ALU

```
1  while true do
2      while ¬fire do
3          wait;
4      addr ← dest.localAddress;
5      inp ← dest.input;
6      if presbit = false then
7          mem[addr, inp] ← val;
8          operandsOut ← ∅;
9      else
10         switch inp do
11             case input_0
12                 operandsOut ←
                   {val, mem[addr, input_1], mem[addr, input_2], mem[addr, input_3]};
13             case input_1
14                 operandsOut ←
                   {mem[addr, input_0], val, mem[addr, input_2], mem[addr, input_3]};
15             case input_2
16                 operandsOut ←
                   {mem[addr, input_0], mem[addr, input_1], val, mem[addr, input_3]};
17             case input_3
18                 operandsOut ←
                   {mem[addr, input_0], mem[addr, input_1], mem[addr, input_2], val};
```

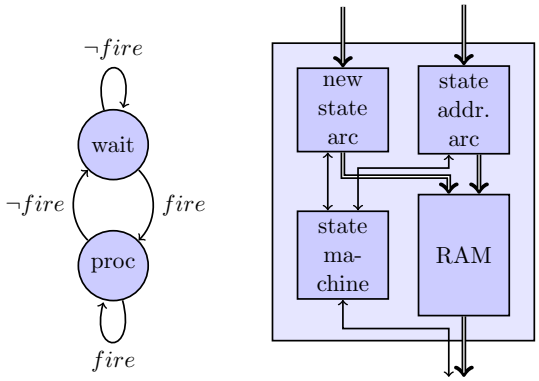
belonging to that particular slice to the ESS. The ESS then selects the state with that address and sends this to the ALU. After execution of a FIR slice the state is changed and has to be stored again in the ESS. The ESS has an additional input where the new state of a slice is received. This new state is stored in the ESS such that the next execution of the same FIR slice uses the new state.

The firing rule of the ESS is a little bit different compared to the rest of the modules in dispatcher. Although the ESS has two inputs, only one of them needs to have a token available to trigger execution. Execution is triggered when either a state address is available or a new state from the ALU . The firing rule for the ESS then becomes:

$$fire = (\neg empty(state\ addr.\ arc) \wedge \neg full(ALU)) \vee \neg empty(new\ state\ arc) \tag{5.7}$$

The state machine of the Explicit State Store is shown in figure 5.8a, which

is the same as most other modules in the FlexCore. The internal structure of the ESS is shown in figure 5.8b.



(a) Statemachine (b) Internal structure

Figure 5.8: Implementation of the Explicit State Store

As shown in figure 5.8b, the ESS accepts both addresses from the program memory and new states from the ALU. An address from the program memory results in a fetch of a state from the memory (line 7 of algorithm 7). This state is then simply forwarded to the ALU. When, after execution of a FIR slice, the state is changed, the changed state is sent to the ESS. When the changed state comes available on the *new state arc*, the state will be stored (line 5 of algorithm 7). The ALU also includes the address where this changed state should be stored.

Algorithm 7: Algorithm of Explicit State Store showing storing and fetching of states

```
1 while true do
2     while ¬fire do
3         wait;
4     if ¬empty(newState) then
5         storeState(newState.stateAddr, newState.stateValues) ;
6     if ¬empty(stateAddr) ∧ ¬full(stateOut) then
7         stateOut ← fetchState(stateAddr) ;
```

The ESS enforces a restriction on the arrival of tokens to the same FIR slice of the dataflow graph. The changed state of a FIR slice must be stored in the ESS before that same slice is enabled again. If the same slice is enabled too soon after the previous execution, the new state is not yet stored in the ESS. The old state is therefore used for the new execution and incorrect results

follow. By analysing the loop formed by the ESS and ALU, a delay of three cycles is required in between to accesses of the same slice.

These three clock cycles are found by taking the following path: program memory → ESS, ESS → ALU and ALU → ESS. This path crosses three modules and therefore passes three times an arc. Every arc requires a clock cycle before the input single has passed through. After three cycles the state of a slice is stored in the ESS and the state of the same slice may be fetched again.

ALU

The biggest differences between the standard FlexCore and the Extended FlexCore can be found in the ALU. Instead of a single operation like add, multiply or subtract, the ALU of the Extended FlexCore can execute complete butterfly operations, complex multiplication or FIR slices. The internal structure has already been determined in section 4.4 where the algorithms for beamforming are merged into a single graph with multiplexers. This merged graph forms the basis for the ALU. Additional logic is added to include the arcs and firing rule. The general structure of the ALU of the Extended FlexCore is shown in figure 5.9.

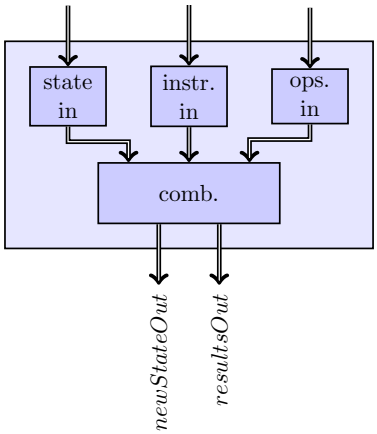


Figure 5.9: Structure of ALU in Extended FlexCore

When the ALU performs a butterfly or complex multiplication only the *instr. in* and *ops. in* inputs are used and the results are produced on output *resultsOut*. However, when a FIR slice is received as instruction, both the *state in* input and the *newStateOut* output are also required. The firing rule is therefore dependent on the instruction i.e. data dependant like the router. The ALU therefore has no state machine and the firing rules are implemented combinatorially. The firing rule is split into two parts: one for

the FIR instruction and one for the others instructions :

$$firFire = \neg empty(state\ in) \quad \wedge \quad \neg empty(instr.\ in) \wedge \neg empty(ops.\ in) \\ \wedge \quad \neg full(new\ state) \wedge \neg full(results) \quad (5.8)$$

$$otherFire = \neg empty(instr.\ in) \wedge \neg empty(ops.\ in) \wedge \neg full(results) \quad (5.9)$$

Based on the instruction received in the *instr. in* arc, the ALU determines combinatorially which firing rule should be used. When that firing rule is satisfied, the instruction can be executed and the results are put on the *resultsOut* output. If a FIR slice is executed, the ALU also produces a new state which should be stored in the ESS. Algorithm 8 shows the algorithm with the basic functionality of the ALU.

Algorithm 8: Algorithm of Arithmetic Logic Unit showing the two firing rules

```
1  if instr. in = fir then
2      if firFire = true then
3          (resultsOut, newStateOut) ←
           mergedDFG(instr. in, state in, ops. in);
4      else
5          (resultsOut, newStateOut) ← ∅;
6  else
7      newStateOut ← ∅;
8      if otherFire = true then
9          resultsOut ← mergedDFG(instr. in, ∅, ops. in);
10     else
11         resultsOut ← ∅;
```

As can be seen on line 2 and 8 of algorithm 8, the ALU uses two firing rules. When a firing rule is not satisfied, no data is sent (line 5 and 11) to any output like any other module.

The function $mergedDFG()$ of algorithm 8 is the merged dataflow graph introduced in section 4.4. This dataflow graph is however statefull i.e. there are registers in the dataflow graph, which allows streaming data in addition to single sampled tokens. The Extended FlexCore has no support for tokens with more than a single value per token. The state of a FIR slice is therefore not placed in registers of the merged DFG but is placed on the *state in* input by the ESS. The same holds for the changed state after executing a FIR slice: the changed state is now put on an output of the ALU instead stored in registers. The changed state is received by the ESS such that it can be used for the next execution. The resulting DFG without registers, being used in the *comb* block of figure 5.9, is shown in figure 5.10.

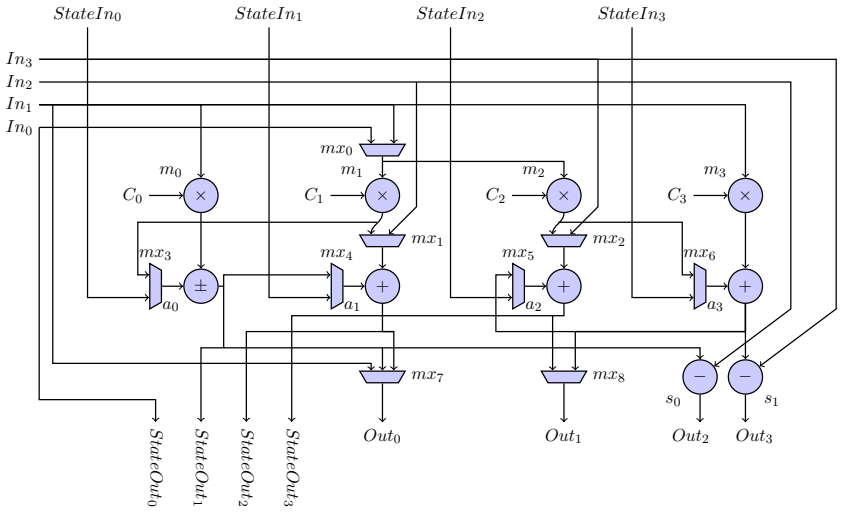


Figure 5.10: ALU for extended FlexCore

The main advantage of removing the registers is that no additional clock cycle is required to load the state. As can be seen in figure 5.10, all processing is now combinatorial and is performed in a single clock cycle. Depending on the instruction, the multiplexers are selected such that the graph performs a fir, $bfly$ or $cmult$ operation. Also the functionality of a_0 depends on the instruction, only for a fir instruction it should behave like an adder. a_0 is therefore an adder-subtractor[35].

The operands from the token store arrive at the inputs in_{0-3} of which some may be unused because the operation requires only two inputs (fir and $cmult$). When a FIR slice is executed, the $StateIn_{0-3}$ supply the current state of the slice. The $StateOut_{0-3}$ forward the changed state to the ESS for the next execution of that slice. The outputs on which the results are placed (Out_{0-3}), are grouped into a single parallel bus connected to the Serializer. The Serializer then sends every value on such an output sequentially to the router.

Serializer

The serializer is introduced to send the 4 results from the ALU sequentially to router. The router therefore only needs a single token input. The router of the standard FlexCore is therefore reused in the Extended FlexCore. The serialization process works the same as in the ALU of the standard FlexCore, the results are sent sequentially to the router. In the Extended FlexCore the serialization process is in a separate module (the serializer) to simplify the implementation and testing. The behavior seen from the router is however still the same.

The number of tokens that have to be sent by the serializer depends on the instruction that produces these tokens. A FIR slice for example has two outputs and therefore produces two tokens. The butterfly operation however has four outputs and will therefore also produce four tokens. The serializer therefore not only receives the resulting tokens but also the instruction that produced those tokens. Based on this instruction, the serializer sends either two or four results to the router.

The input of the serializer has the following format:

$$tokenIn = \{instr, token_3, token_2, token_1, token_0\} \tag{5.10}$$

A token $token_n$ is a standard token containing a destination address and a value. The format of a standard token can be found in equation (5.1) at the beginning of this chapter. The process of sending the result tokens sequentially is shown in algorithm 9.

Algorithm 9: Iterating over result tokens from the ALU

```
1 if curToken = token_3 ∨ (instr ∈ {fir, cmult} ∧ curToken = token_1)
  then
2 |  curToken ← token_0
3 else
4 |  curToken ← curToken + 1
```

The algorithm uses a pointer called $curToken$ to iterate over all the results tokens from the ALU. When the last possible token ($curToken$), is sent, the pointer is reset as shown on line 2. The last token to be sent of a $bfly$ instruction is $token_3$ while for the $cmult$ and fir instruction, $token_1$ is the last one. When not all tokens have been sent, the pointer forwards to the next token (line 4).

6

Results

This chapter gives the results derived during synthesis and simulation of both FlexCore designs. First the synthesis results are given in section 6.1. In section 6.2, power numbers are derived using several test-applications. The test-applications are: a FIR filter, an FFT and a set of complex multiplications. Section 6.3 elaborates on the performance of the test-applications on both architectures. The last section, section 6.4, gives a summary of the results.

6.1 Synthesis

Both the standard FlexCore and the Extended FlexCore have been synthesized for ASIC using the 90 nm TSMC low power[1] libraries. Both designs also have been synthesized for a Xilinx FPGA. First the synthesis results for the ASIC implementations of the FlexCore are described followed by the FPGA synthesis.

ASIC synthesis

Initially, both designs were synthesized with a clock frequency of 150 MHz. Although this was no problem for the standard FlexCore, timing violations occurred in the Extended FlexCore. The clock frequency was therefore lowered to 100 MHz in order to make a fair comparison between both designs, which should be valid assuming that the energy consumption scales linearly with the clock frequency. For both designs the longest combinatorial path is in the ALU. 100 MHz is a rather low clock frequency but this is caused by the lack of pipelining, this will be further elaborated in the discussion (chapter 8). The area numbers derived from the synthesis results are shown in table 6.1.

As can be seen in table 6.1 the dispatcher requires most of the area (over 90%) due to all the memories. The ALU requires about 6% of the total area. The increase of area of the dispatcher is caused by the larger instructions and the bigger token store (three tokens can be waiting for a node instead of one).

[1] General purpose libraries are not used as they are less energy efficient

Part	Standard FlexCore		Extended FlexCore	
ALU	$11 * 10^{-3}\ mm^2$	5.8%	$36 * 10^{-3}\ mm^2$	6.4%
Dispatcher	$172 * 10^{-3}\ mm^2$	91.7%	$507 * 10^{-3}\ mm^2$	91%
Presence memory	$4 * 10^{-3}\ mm^2$	2.2%	$18 * 10^{-3}\ mm^2$	3.2%
Program memory	$118 * 10^{-3}\ mm^2$	63.2%	$297 * 10^{-3}\ mm^2$	53.4%
Tokenstore	$49 * 10^{-3}\ mm^2$	26.3%	$180 * 10^{-3}\ mm^2$	32.4%
Expl. State store	-	-	$12 * 10^{-3}\ mm^2$	2.1%
Serializer	-	-	$9 * 10^{-3}\ mm^2$	1.7%
Rest (router,clock)	$4 * 10^{-3}\ mm^2$	2.5%	$5 * 10^{-3}\ mm^2$	0.9%
Total	$187 * 10^{-3}\ mm^2$	100%	$557 * 10^{-3}\ mm^2$	100%

Table 6.1: Area results from ASIC synthesis

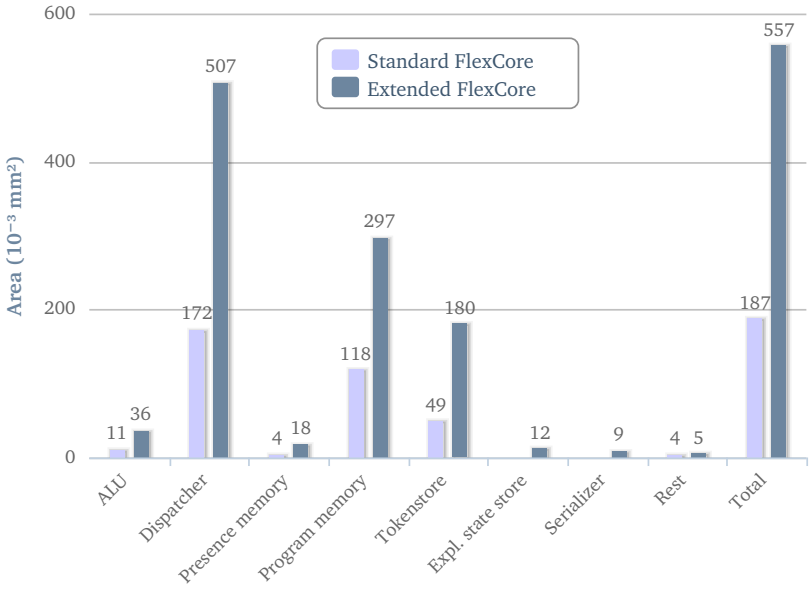


Figure 6.1: Area of different modules in both designs of the FlexCore

The area of the router is included in *Rest* because it was synthesized into a lot of small cells. The router could therefore not be recognized as a single module anymore. Also the area of the clock network is included in the rest part. The numbers of table 6.1 are shown graphically in the plot of figure 6.1.

Both implementations of the FlexCore are synthesized without real ASIC memories. The reason for this is that the available memories only have synchronous read and write interfaces. Both implementations rely however on asynchronous read to select the element in the memories within one cock cycle. In order to support ASIC memories, the read process should be split in to phases: first read in the address using the standard dataflow process, fol-

lowed by sending this address to the memory such that the memory element is available after a rising edge of the clock.

Clock gating was initially enabled during synthesis of both designs. However this turned out to be a problem for the Extended FlexCore. After synthesis of the Extended FlexCore the design did not work anymore. By disabling clock gating this issue was resolved. In order to make a fair comparison, clock gating is disabled for both the standard and the Extended FlexCore.

FPGA synthesis

Both the standard and the Extended FlexCore have also been synthesised for an FPGA. The designs are synthesised for a Xilinx 6VLX240TLFF1156 FPGA using Mentor Graphics Precision[36]. As with the ASIC synthesis, the longest combinatorial path is located in the ALU for both the standard and the Extended FlexCore. Table 6.2 shows the synthesis results.

	Standard FlexCore	Extended FlexCore
Function generators	937	4025
CLB slices	235	1007
DFFs/Latches	175	713
Block RAMs	0	0
DSP blocks	1	4
max. clock frequency	178 MHz	102 MHz

Table 6.2: Area results from synthesis for FPGA

As can be seen in table 6.2, the Extended FlexCore requires about four times as much logic resources as the standard FlexCore. The maximum clock frequency is determined by the longest combinatorial path in the ALU but still reaches over 100 MHz. The tooling was however not able to infer block RAMs for the memories in the dispatcher. This has the same reason as for the ASIC synthesis: the memories require both a synchronous read and write interface while the design relies on asynchronous read. Block RAMs have registers on the output which makes them completely synchronous. The tooling therefore uses the Function Generators (look up tables) as small memories but this requires more logic resources.

6.2 Power consumption

The power consumption is determined by executing several test applications on both FlexCores and comparing those results. The test applications are executed during the simulation after place and route such that power consumption by wires is also taken into account. Only the power consumption

during calculations is presented, the power consumption during programming is ignored. During the simulation of the test applications, all changes of signals are stored in a so called Value Change Dump (VCD) file. Finally, power consumption is determined using Synopsis Primetime which uses the VCD file and combines this with the design from place and route and the library files containing the power information.

FIR filter

Both implementations of the FlexCore have been tested with a 16 taps FIR filter. For the standard FlexCore, a much bigger dataflow graphs is needed than for the Extended FlexCore. A 16 taps FIR filter requires 16 multipliers and 15 adders. The registers are implemented by placing initial tokens on the edges between the adders. The graph for the standard FlexCore requires $15+16=31$ nodes while the Extended FlexCore requires only $\lceil \frac{16}{4} \rceil = 4$ nodes to implement the filter(recall that the FIR slice of the Extended FlexCore executes four taps at once).

Both FlexCores are programmed with the FIR dataflow graph before measuring the energy consumption. The filter is executed by determining the step response with one additional sample. The filter is therefore executed seventeen times and startup effects of the FlexCore should thus be averaged away. Executing the FIR DFGs on the processors running at 100 MHz results in power numbers shown in table 6.3.

Part	Standard FlexCore		Extended FlexCore	
ALU	835 μW	4.3%	2.8 mW	4.9%
Dispatcher	14.4 mW	73.6%	42 mW	73.5%
Presence memory	412 μW	2.1%	1.5 mW	2.6%
Program memory	9.2 mW	47%	23.2 mW	40.6%
Tokenstore	4.8 mW	24.6%	16.1 mW	28%
Expl. State store	-	-	1.3 mW	2.2%
Serializer	-	-	1.1 mW	1.8%
Rest (clock)	4.77 mW	22.1%	12.1 mW	19.8%
Total	20 mW	100%	57 mW	100%

Table 6.3: Power consumption for executing FIR filters

Table 6.3 shows that the standard FlexCore consumes an average power of $20mW$ while the Extended FlexCore consumes $57mW$. In both designs the dispatcher consumes most power while the ALU in both designs consumes less then 5% of the total power. The power results are shown graphically in figure 6.2.

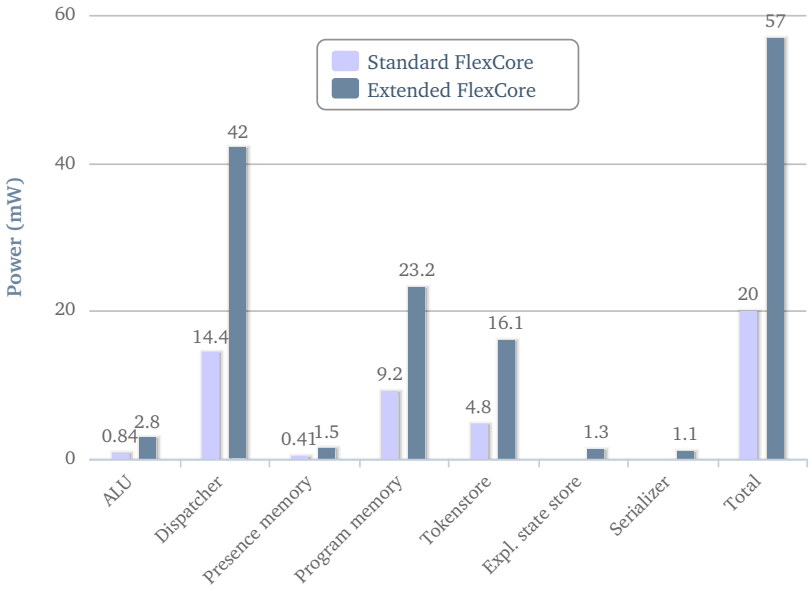


Figure 6.2: Power consumption during execution of FIR filter

The standard FlexCore requires 1050 cycles to complete the step response of the whole filter. By using the fact that the 100 MHz clock has a period of 10 ns, the average time to execute a single filter tap therefore takes $\frac{1050*10ns}{16taps*17samples} = 39ns$. By multiplying this with the power consumption, the energy required for a single filter tap is found. A single filter tap executed on the standard FlexCore consumes $39ns * 20mW = 0.78nJ$ of energy.

The Extended FlexCore on the other hand, requires only 193 cycles to complete the step response with one additional sample. The average execution time for a single filter tap is therefore $\frac{193*10ns}{16taps*17samples} = 7ns$ per sample. A single filter tap executed on the Extended FlexCore requires $7ns * 57mW = 0.40nJ$ of energy.

Although the Extended FlexCore uses more power($57mW$ versus $20mW$) executing a filter tap on it is more energy efficient. When the energy for a single filter tap executed by the standard FlexCore is defined as 100%, the percentage of energy saved by using the Extended FlexCore becomes:
$\frac{0.78-0.4}{0.78} * 100\% = 49\%$

FFT

Both designs have been tested with a four point FFT. For the standard FlexCore, a much bigger dataflow graphs is needed. For a four point FFT, $\frac{N}{2} * log_2(N) = 4$ butterfly operations are required. A butterfly operation consists of ten basic dataflow nodes(four multiplications, three additions and three subtractions). A butterfly operation for the standard FlexCore requires

$4 * 10 = 40$ nodes in the DFG while the Extended FlexCore requires only four.

Both FlexCores are programmed with the FFT dataflow graph before measuring the energy consumption. The FFT is executed ten times in order to average away startup effects. Executing the FFT DFGs on the processors running at 100 MHz results in power numbers shown in table 6.4.

Part	Standard FlexCore		Extended FlexCore	
ALU	976 μW	4.9%	2.1 mW	3.7%
Dispatcher	14.7 mW	73.2%	42 mW	74.4%
Presence memory	408 μW	2%	1.4 mW	2.6%
Program memory	9.25 mW	46.2%	23.1 mW	41%
Tokenstore	5 mW	25%	16.5 mW	29.2%
Expl. State store	-	-	971 μW	1.7%
Serializer	-	-	965 μW	1.7%
Rest (clock)	4.32 mW	21.9%	11.9 mW	20.2%
Total	20 mW	100%	57 mW	100%

Table 6.4: Power consumption for executing FFTs

Table 6.4 shows that the standard FlexCore consumes an average power of again $20mW$ while the Extended FlexCore consumes $57mW$. Again, the dispatcher consumes most power while the ALUs in both designs consume less then 5% of the total power. The power results are shown graphically in figure 6.3.

The standard FlexCore requires 882 cycles to execute the four point FFT ten times. Executing a single butterfly operation requires $\frac{882*10ns}{40butterflies} = 221ns$, by using a clock period of 10 ns. By multiplying this with the power consumption, the energy required for a single butterfly is found. The energy required for a single butterfly operation is thus $221ns * 20mW = 4.4nJ$.

Only 241 cycles are required for the Extended FlexCore to complete the ten FFTs. This results in an average execution time for a single butterfly operation of $\frac{241*10ns}{40butterflies} = 60.3ns$. The amount of energy required to execute a single butterfly operation on the Extended FlexCore is thus $60.3ns*57mW = 3.4nJ$.

Although the Extended FlexCore uses more power($56mW$ versus $20mW$) executing butterfly operations on it is more energy efficient. By defining the energy for a single butterfly executed by the standard FlexCore as 100%, the percentage of energy saved by using the Extended FlexCore is
$\frac{4.4-3.4}{4.4} * 100\% = 23\%$.

Complex multiplications

The last test application is a chain of four complex multiplications. This chain is executed on both architectures and consist of $4 * 6 = 24$ dataflow nodes

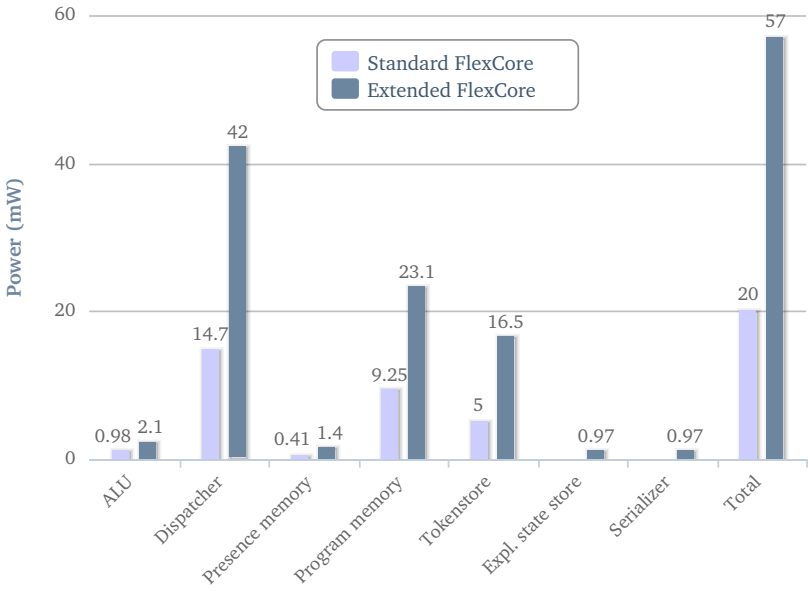


Figure 6.3: Power consumption during execution of FFT

for the standard FlexCore and 4 dataflow nodes for the Extended FlexCore. Every complex multiplication consist of four multiplications one addition and one subtraction.

Both FlexCores are programmed with the chain dataflow graph before measuring the energy consumption. The chain is executed ten times in order to average away startup effects. Executing the chain on the processors running at 100 MHz results in power numbers shown in table 6.5.

Part	Standard FlexCore		Extended FlexCore	
ALU	1.1 mW	5.4%	3.2 mW	5.5%
Dispatcher	14.7 mW	72.8%	42.6 mW	72.9%
Presence memory	399 μW	2%	1.51 mW	2.6%
Program memory	9.24 mW	45.7%	23.1 mW	39.7%
Tokenstore	5.1 mW	26%	16.9 mW	29%
Expl. State store	-	-	1 mW	1.8%
Serializer	-	-	1.2 mW	2%
Rest (clock)	4.2 mW	21.8%	11 mW	19.6%
Total	20 mW	100%	58 mW	100%

Table 6.5: Power consumption for executing complex multiplications

Table 6.5 shows that the standard FlexCore consumes an average power of $20mW$ while the Extended FlexCore consumes $58mW$. Also for the complex

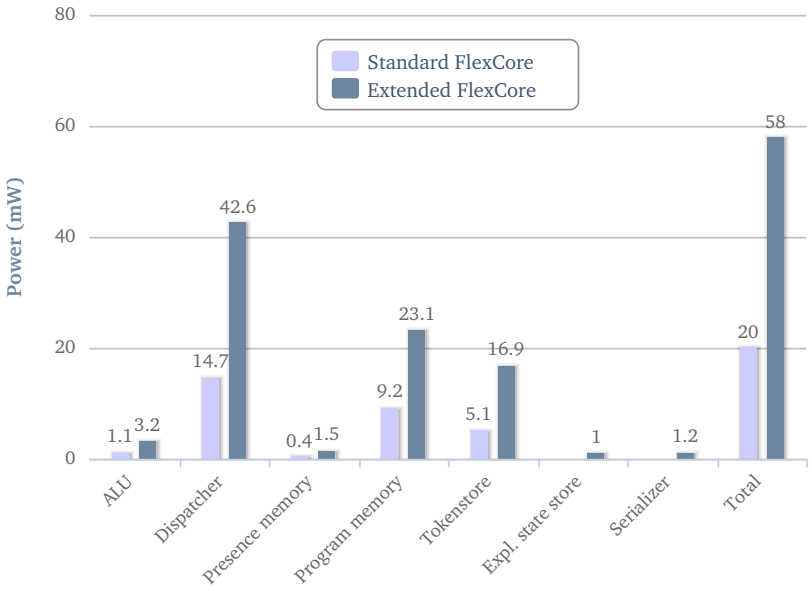


Figure 6.4: Power consumption during execution of complex multiplication chain

multiplications, the dispatcher of both designs consumes most power. The ALU in both designs consume a little bit more than 5% of the total power, again in agreement with the other test applications. The power results are shown graphically in figure 6.4.

The standard FlexCore requires 494 cycles to complete the complex multiplication chain ten times. The average time to execute a single complex multiplication is $\frac{494*10ns}{40compl.mults} = 124ns$ per operation. The energy required for a single butterfly operation can be found by multiplying the average time to execute a butterfly operation with the average power consumption of the standard FlexCore: $124ns * 20mW = 2.5nJ$.

116 cycles are required by the Extended FlexCore to complete the ten complex multiplication chains. The average execution time for a single complex multiplication is therefore $\frac{116*10ns}{40compl.mults} = 29ns$. The energy to execute a single butterfly operation on the Extended FlexCore is $29ns*58mW = 1.7nJ$.

Also for complex multiplications the Extended FlexCore turns out to be more energy efficient. By defining a single complex multiplication executed by the standard FlexCore as 100%, the amount of energy saved by using the Extended FlexCore is $\frac{2.5-1.7}{2.5} * 100\% = 32\%$.

6.3 Performance evaluation

This section elaborates on the utilization of the ALU of both the standard and the Extended FlexCore. The utilization is determined for the same ap-

plications which were used to find the energy consumption: FIR filter, FFT, and butterfly. Also performance numbers are given in terms of operations per clock cycle.

FIR filter

A 16 taps FIR filter is used as dataflow graph to evaluate the performance of both FlexCores. The step response of this FIR filter is determined followed by feeding it one additional sample (to check whether the state remain correct after a full step response). The DFG for the standard FlexCore consists of $15*2+1 = 31$ nodes (recall that the first filter tap only consist of a multiplier, see figure 4.4). The DFG for the Extended FlexCore consist only of $16/4 = 4$ nodes.

The standard FlexCore is fed with data such that the step response with an additional sample is determined. The step response is found by feeding the FIR filter 16 tokens all containing the value 1. Every node in the DFG of the standard FlexCore should therefore be executed 17 times. This means that the ALU should be executing an instruction for $31 * 17 = 527$ cycles. 1050 cycles were required to execute this FIR application. The utilization of the ALU of the standard FlexCore is therefore $\frac{527}{1050} * 100\% = 50\%$.

The DFG for the Extended FlexCore is fed the same data so it should also be executed 17 times. The DFG of the Extended FlexCore consist of 4 nodes (4 filter slices). The ALU should therefore be active for $17*4 = 68$ cycles. 193 were required to execute the DFG on the Extended FlexCore. The utilization of the ALU of the Extended FlexCore is therefore $\frac{68}{193} * 100\% = 35\%$.

The standard FlexCore requires 1050 cycles to execute the FIR application consisting of $16 * 17 = 272$ filter taps. This results in a performance of $\frac{272}{1050} = 0.26$ taps per clock cycle. The Extended FlexCore on the other hand requires only 193 cycles to execute 272 filter taps. This results in a performance of $\frac{272}{193} = 1.4$ taps per clock cycle.

FFT

Also the FFT is used to evaluate the performance of both FlexCore implementations. A dataflow graph of a four point FFT is made and executed on both the standard and the Extended FlexCore. A four point FFT consist of four butterfly operations. This is implemented with 40 nodes for the standard FlexCore and 4 for the Extended FlexCore.

Both FlexCores are fed with data such that the FFT is executed 10 times. A butterfly operation on the standard FlexCore consist of 10 nodes. The ALU should therefore perform $4*10*10 = 400$ instructions. 882 cycles were required to execute the FFT ten times. This results in a utilization of $\frac{400}{882}*100\% = 45\%$.

The same data is also fed to the FFT implementation on the Extended FlexCore but now the dataflow graph consist only of four butterfly operations.

The number of butterfly operations is therefore 40. As the Extended FlexCore required only 241 to execute the FFT ten times, the resulting utilization of the ALU will be $\frac{40}{241} * 100\% = 17\%$.

The standard FlexCore requires 882 cycles to execute the FFT application consisting of 40 filter butterfly operations. This results in a performance of $\frac{40}{882} = 0.05$ butterfly operations per clock cycle. The Extended FlexCore on the other hand requires only 241 cycles to execute 40 butterfly operations. This results in a performance of $\frac{40}{241} = 0.17$ butterfly operations per clock cycle.

Complex multiplications

The last test performed on both the standard and the Extended FlexCore is the chain of multiplications. The chain consists of four cascaded complex multiplications which is implemented using $4 * 6 = 24$ nodes on the standard FlexCore and 4 nodes on the Extended FlexCore.

Again, both FlexCores are fed with data such that the chain is executed ten times. The ALU of the standard FlexCore should therefore be executing instructions for $24 * 10 = 240$ clock cycles. 494 clock cycles were required to execute the chain ten times on the standard FlexCore. The utilization of the ALU is therefore $\frac{240}{494} * 100\% = 49\%$.

The DFG for the Extended FlexCore is also fed the same data and is therefore also executed 10 times. The DFG for the Extended FlexCore however consists only of 4 nodes. The ALU is therefore occupied for $4 * 10 = 40$ clock cycles. 116 cycles were required to execute the complex multiplication chain 10 times, the resulting utilization of the ALU is therefore $\frac{40}{116} * 100\% = 34\%$.

The standard FlexCore requires 494 cycles to execute the FFT application consisting of 40 complex multiplications. This results in a performance of $\frac{40}{494} = 0.08$ complex multiplications per clock cycle. The Extended FlexCore on the other hand requires only 116 cycles to execute these 40 complex multiplications. This results in a performance of $\frac{40}{116} = 0.34$ complex multiplications per clock cycle.

6.4 Summary

The synthesis results show that for both implementations of the FlexCore, the dispatcher requires more than 90% of the total area mainly consisting of memories. The area required by the ALUs is about 6% while the rest is used by the router and clock network.

Also energy numbers have been determined for the FIR filter, FFT and complex multiplications. Again, the dispatcher requires most energy, about 73% of the total energy consumption is used in the dispatcher. The amount of energy required for basic operations (filter tap, butterfly and complex multiplication) is shown in table 6.6.

Application	Standard FlexCore	Extended FlexCore	Gain
FIR	$0.78nJ$	$0.40nJ$	49%
FFT	$4.5nJ$	$3.4nJ$	23%
compl. mult.	$2.5nJ$	$1.7nJ$	32%

Table 6.6: Energy savings for test applications on both architectures

For both designs of the FlexCore, the utilization of the ALU has been determined for all test applications. The results are shown in table 6.7. It clearly shows that the utilization of the ALU is much lower compared to the standard FlexCore. Recall that the path from ALU to router to dispatcher only allows a single token to be sent per clock cycle. The dispatcher can only process a single operand per clock cycle. This means that an operation requiring four operands, is scheduled at most once per four clock cycles. All operations on the standard FlexCore require only two operands which is why the utilization of the ALU is close to 50% for all test applications. The utilization on the Extended FlexCore is lower because there are nodes having more inputs which all have to be processes sequentially by the dispatcher. The performance can be increased by making a dispatcher which can process several tokens in a single clock cycle. Also the path from ALU to router to dispatcher needs then support for several tokens per clock cycle.

Application	Standard FlexCore	Extended FlexCore
FIR	50%	35%
FFT	45%	17%
compl. mult.	49%	34%

Table 6.7: Utilization of ALUs

All performance numbers in terms of basic operations (filter tap, butterfly or complex multiplication) per clock cycle are extracted from both implementations and shown in table 6.8. Also the speedup by using the Extended FlexCore instead of the standard FlexCore is given.

Application	Standard FlexCore	Extended FlexCore	speedup
FIR	0.26	1.4	$1.4/0.26 = 5.4$
FFT	0.05	0.17	$0.17/0.05 = 3.4$
compl. mult.	0.08	0.34	$0.34/0.08 = 4.3$

Table 6.8: Performance numbers for test applications

7

Comparison with other architectures

Here, the FlexCore is put in perspective by comparing it with other architectures in terms of energy per operation. Comparing architectures is however a tricky business, the following explanation should therefore be seen as an indication. The following calculations are based on the results shown in [37]. For each architecture the amount of energy per butterfly operation is determined such that the architectures can be ordered by efficiency. Only the absolute power numbers are shown i.e. no technology scaling is applied.

The main architecture described in [37] is the Montium Tile Processor. This is a coarse grain reconfigurable architecture especially developed for streaming applications[38]. The energy consumption for the Montium is determined by executing a single 64 points FFT on it. By including the energy for communication, the amount of energy E_{total} required to execute a 64 point FFT on the Montium is $E_{total}125nJ$ as found in [38].

The whole 64 point FFT is executed once by feeding it a block of 64 samples. The FFT consists of $\frac{64}{2}log_2(64) = 192$ butterfly operations. The amount of energy for a single butterfly operation E_{BF} then becomes:

$$E_{BF} = \frac{E_{total}}{number\ of\ butterflies} = \frac{125nJ}{192BFs} = 0.65nJ/BF$$

The next architecture investigated in [37] is the FASRA(FFT algorithm specific reference architecture) ASIC. The FASRA performs a 1024 point FFT on a block of 1024 input samples. This means that $\frac{1024}{2}log_2(1024) = 5120$ butterfly operations are performed. For the whole FFT, $1938nJ$ of energy is required. The amount of energy required for a single butterfly operation E_{BF} then becomes:

$$E_{BF} = \frac{E_{total}}{number\ of\ butterflies} = \frac{1938nJ}{5120BFs} = 0.379nJ/BF$$

The FASRA architecture has also been implemented on a Xilinx XC2VP2 FPGA. A 64 point FFT is executed consisting of 192 butterfly operations. For the FASRA FPGA implementation, the numbers in [37] are given in terms of $\mu W/MHz$ where a butterfly operations is executed every clock cycle. At a clock frequency of 1 MHz, a single clock cycle lasts $1\mu s$. The total power

consumption is 12155 $\mu W/MHz$. The amount of energy E_{BF} for a single butterfly operation can now be found by multiplying the time for a single butterfly T_{BF} with the total power P_{total}:

$$E_{BF} = P_{total} * T_{BF} = 12155\mu W/MHz * 1MHz * 1\mu s = 12.2nJ/BF$$

The last architecture investigated in [37] is a general purpose processor, the ARM920T. The ARM920T requires $0.25mW/MHz$ when executing a single instruction per clock cycle. A single butterfly operation however requires 21 clock cycles. The total power number is 21 times higher: $21 \times 0.25mW/MHz = 5.25mW/MHz$. At a butterfly execution rate of 1 MHz, a single butterfly operation lasts $1\mu s$. The energy of a single butterfly operation E_{BF} is:

$$E_{BF} = P_{total} * T_{BF} = 5.25mW/MHz * 1MHz * 1\mu s = 5.25nJ/BF$$

All the numbers for the architectures, including the Extended FlexCore, are shown in table 7.1 and figure 7.1.

	ASIC	Montium	Extended FlexCore	ARM920T	FPGA
nJ/BF	0.379	0.65	3.4	5.25	12.2

Table 7.1: Energy per butterfly operation on different architectures

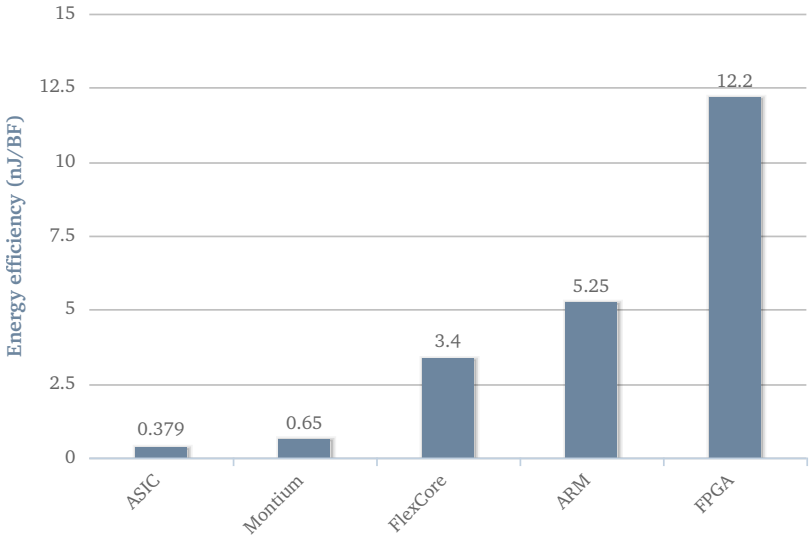


Figure 7.1: Energy required for executing a butterfly operation on a particular architecture

8

Discussion & Future work

Although both the standard and the Extended FlexCore have been implemented, there are still a few issues. The first issue is the scalability of certain programs which should run on the architectures (a 1024 point FFT of LOFAR for example requires a lot of nodes in the dataflow graph and therefore a lot of memory). The second issue is that the ALU is not yet pipelined. Currently all modules of both designs should produce a result within a single clock cycle. The resulting clock frequency is therefore much lower because of long combinatorial paths in the ALU. Another issue is that both designs do not use memories for ASIC but let the tooling create memories using only flip flops. This results in more area and higher power consumption. Clock gating had to be turned off during synthesis because it broke the design of the Extended FlexCore.

8.1 Scalability

Both the standard FlexCore and the Extended FlexCore have to be programmed in such a way that the program memory contains a complete static dataflow graph. This size of the dataflow graph depends highly on the scalability of the algorithms.

Consider an N point FFT. An FFT can be constructed using $\frac{N}{2}log_2(N)$ butterfly operations. To implement a 1024 point FFT, $\frac{1024}{2}log_2(1024) = 5120$ butterfly operations are required. The Extended FlexCore therefore requires 5120 memory locations in the presence list memory, program memory and tokenstore to support a dataflow graph containing this many nodes. The standard FlexCore requires even more nodes. The standard FlexCore has no support for a complete butterfly operation, it is constructed using just multiplications, additions and subtractions. To execute a 1024 point FFT on the standard FlexCore, $\frac{1024}{2}log_2(1024) * 10 = 51200$ nodes are required (recall that a butterfly operation can be constructed using 10 basic operations).

Both the standard FlexCore and the Extended FlexCore use an address width of 7 bits to address nodes in the memories of the dispatcher. This allows a maximum size of $2^7 = 128$ nodes for the dataflow graph. The area used by the dispatcher is already 90% of the total area and dissipates 70% of the total

power. To support a 1024 point FFT, almost all area and memory will be spent on memory and not on calculations. The architecture therefore requires something to support repetition of parts of the dataflow graph. The FFT does not scale very well on the current implementation of the FlexCore.

Instead of changing the architecture to a complete dynamic dataflow architecture, the FFT could be implemented as a pipelined FFT[39]. A pipelined FFT combines the butterfly operations of every stage to a single butterfly. All the butterfly operations in a stage are then executed sequentially. In between are so called commutators with memories which implement the addressing of data in the FFT. By adding support for these commutators and memories, a pipelined FFT is supported which reduces the number of required dataflow nodes to $log_2(N)$. The scalability is not really a problem for the FIR filter and complex multiplications because they scale linearly with the number of coefficients which are usually all unique.

8.2 Pipelining

Although the complete architecture can be considered to be pipelined because the modules are separated by FIFOs, the modules themselves are currently not pipelined. The longest combinatorial path is for both implementations of the FlexCore located in the ALU. The resulting clock frequency was therefore limited to 150 MHz and 100 MHz respectively. All other modules contain mostly memory hence pipelining the ALUs should increase the maximum clock frequency significantly. Pipelining should be implemented on the lowest level of the dataflow implementation such that it can easily be used on a higher level. Because pipelined modules are not supported, memories for ASIC cannot be used. This is because they are completely synchronous as explained in the synthesis results (section 6.1).

8.3 ASIC memories

Currently the memories are implemented by the tooling which implements them using flipflops. Most memories (including those available during the design of the FlexCore) for ASICs only support synchronous read. Both implementations of the FlexCore require however an asynchronous read interface. Reading from these ASIC memories therefore requires an additional cycle. This additional cycle should be implemented by pipelining the modules containing memories. The standard FlexCore has also been synthesised for an FPGA where the problem with the asynchronous read also occurred. Several FPGA tools (Precision, Xilinx ICE and Altera Quartus) were all unable to implement the memories using blockRAM and implemented them as registers from the logic blocks of the FPGA[40].

8.4 Clock gating

For the standard FlexCore the synthesis tooling was able to detect parts (especially the memories) of the design that could be clock gated and applied it successfully. During synthesis of the Extended FlexCore however, the tooling was able to apply clock gating but it broke the design. During post synthesis simulation the design did not work anymore (debugging showed that signal remained invalid even if the data was sent correctly to the FIFOs). In order to make a fair comparison between both designs, clock gating had to be disabled for both the standard FlexCore and the Extended FlexCore which results in higher power dissipation of mainly the memories. After disabling clock gating both designs worked flawlessly.

8.5 Streaming

Currently the dispatcher processes a single token per firing. The architecture can however be extended such that streams of several samples are supported. Matching is then based on complete streams instead of single tokens. The dispatcher is only active when a new stream arrives that has to be matched. As the dispatcher requires most power, applying matching on streams should reduce the complete power consumption significantly.

8.6 Programming

Although the FlexCore is now programmed by hand using the architecture's instruction set, changing a compiler to support the FlexCore should not be hard. The compiler should be able to create a dataflow graph consisting of FIR slices, butterfly operations and complex multiplications. There already exist several programming languages that are used for dataflow architectures: Id[13], LUCID[41] and parallel Haskell (pH)[42].

9

Conclusions

Multiple algorithms of the LOFAR telescope have been analyzed and regularities have been revealed. These regularities are exploited in the Extended FlexCore, a dataflow architecture designed especially for executing the beamforming operations of LOFAR. The Extended FlexCore is compared with the standard FlexCore and a significant (23%, 32% and 49%) energy efficiency gain is found. Both implementations of the FlexCore rely heavily on dataflow principles for internal communication. These dataflow principles were first implemented using VHDL and later used as basis for building both FlexCore implementations.

Dataflow graphs

Dataflow graphs have been directly implemented in VHDL. Every input of a node in such a dataflow graph is implemented using a FIFO with space for at least two elements. It has been shown that two elements is the lower bound to achieve a full throughput of one token per clock cycle. Making designs using these dataflow principles should also be scalable because the longest combinatorial path covers only two dataflow nodes. Using dataflow graphs as a design method for both FlexCore implementations has shown to be very useful as synchronization is performed by dataflow principles while functionality of the modules can be considered local.

Analysis of algorithms

The beamforming algorithm for the LOFAR telescope has been analyzed and have revealed regularities and overlap among the operations. Beamforming is performed mainly by FIR filters, FFTs and complex multiplications. An ALU is designed which combines these algorithms into a single piece of hardware. The ALU is able to execute a FIR slice of four filter taps, a complex multiplication or a butterfly operation in a single clock cycle. It has also been shown how a long FIR filter can be executed by splitting it into slices and executing these sequentially.

Implementation of both FlexCores

Two implementations of the FlexCore have been made. The standard FlexCore is a dataflow architecture based on the Explicit Token Store (ETS) principle to implement the matching procedure. This implementation is used as reference design for comparison with the second implementation, the Extended FlexCore. The Extended FlexCore includes the ALU designed during the analysis of the beamforming algorithms and supports FIR slices, complex multiplications and butterfly operations. To handle the state of the FIR slices an additional module, the Explicit State Store (ESS), is introduced.

Both designs of the FlexCore have been implemented using the TSMC 90 nm low power library. Place and route has been performed for both designs after which the power consumption is determined by executing several applications on both architectures. The simulations have shown that the Extended FlexCore consumes significantly less energy per operation compared to the standard FlexCore. The Extended FlexCore consumes 23%, 49% and 32% less energy for executing the FFT, FIR filter and complex multiplication application respectively.

The research question as how the granularity of dataflow execution can be increased to increase efficiency, can be answered as follows: By combining common sub graphs in the dataflow graphs of the LOFAR beamforming application into a larger ALU, the power consumption is reduced significantly. By using a bigger ALU the amount of tokens that should be processed by the dispatcher is reduced and the performance is increased. Even though the Extended FlexCore required almost three times as much power, the amount of energy required per operation has shown to be significantly less than the reference implementation, on the standard FlexCore.

9.1 Acknowledgements

By writing this part I am finally able to remove the last todo from the list. I want to use this part to express my gratitude to all who made this work possible, especially to my committee: André Kokkeler, Jan Kuper, Kenneth Rovers, Anja Niedermeier, André Gunst and Albert-Jan Boonstra. I would like to thank you all for the fruitful discussions, help regarding tooling and mathematics and the reviews of my work.

I would also like to thank Bert Molenkamp for the quick answers on my VHDL questions. Last but not least I would like to thank everybody else from the CAES group for making CAES such an accessible group with a great atmosphere.

List of Acronyms

ALU Arithmetic Logic Unit

ASIC Application-specific Integrated Circuit

DFT Discrete Fourier Transform

DFG Data Flow Graph

DSP Digital Signal Processing

EP Empty Predictor

ESS Explicit State Store

ETS Explicit Token Store

FB Fractional Bandwidth

FFT Fast Fourier Transformation

FIFO First In First Out

FIR Finite Impulse Response

FP Full Predictor

FPGA Field Programmable Gate Array

HSDF Homogeneous Synchronous Data Flow

LOFAR LOw Frequency ARray

NoC Network on Chip

RAM Random Access Memory

SDF Synchronous Data Flow

SKA Square Kilometer Array

SoC System-on-Chip

VCD Value Change Dump

VHDL (Very High Speed Integrated Circuit) Hardware Description Language

Bibliography

[1] LOFAR low frequency array. `http://www.lofar.org/`.

[2] M. de Vos, A.W. Gunst, and R. Nijboer. The lofar telescope: System architecture and signal processing. *Proceedings of the IEEE*, 97(8):1431–1437, August 2009.

[3] SKA square kilometre array. `http://www.skatelescope.org/`.

[4] P. Hall. The square kilometre array: An international engineering perspective. In Peter Hall, editor, *The Square Kilometre Array: An Engineering Perspective*, pages 5–16. Springer Netherlands, 2005.

[5] K. C. Rovers, M. D. van de Burgwal, A. B. J. Kokkeler, and G. J. M. Smit. Rationale for and design of a generic tiled hierarchical phased array beamforming architecture. In *18th Annual Workshop on Circuits Systems and Signal Processing (ProRISC), Veldhoven, the Netherlands*, pages 160–168, Utrecht, November 2007. Technology Foundation STW.

[6] C. Chang, J. Wawrzynek, and R.W. Brodersen. Bee2: a high-end reconfigurable computing system. *Design Test of Computers, IEEE*, 22(2):114 – 125, 2005.

[7] M. Goris, A. Joseph, G. Hampson, and F. Smits. Adaptive beamforming system for radio-frequency interference rejection. *Radar, Sonar and Navigation, IEE Proceedings* -, 146(2):73 –77, April 1999.

[8] Kenneth C. Rovers, Marcel D. Burgwal van de, Jan Kuper, Andre B.J. Kokkeler, and Gerard J.M. Smit. On reconfigurable tiled multi-core programming. In *Proceedings of the 20th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2009*, pages 507–514. Technology Foundation, November 2009.

[9] M. Wiggers, M. Bekooij, P. Jansen, and G. Smit. Efficient computation of buffer capacities for multi-rate real-time systems with back-pressure. In *Hardware/Software Codesign and System Synthesis, 2006. CODES+ISSS '06. Proceedings of the 4th International Conference*, pages 10 –15, 2006.

[10] Paul G. Whiting and Robert S.V. Pascoe. A history of data-flow languages. *IEEE Annals of the History of Computing*, 16:38–59, 1994.

[11] Paul J. M. Havinga and Gerard J. M. Smit. Design techniques for low-power systems. *Journal of Systems Architecture*, 46(1):1 – 21, 2000.

[12] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.

[13] Wesley M. Johnston, J. R. Paul Hanna, Richard, and J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv*, 36:1–34, 2004.

[14] E.A. Lee and D.G. Messerschmitt. Synchronous data flow. *Proceedings of the IEEE*, 75(9):1235 – 1245, 1987.

[15] A.H. Ghamarian. *Timing analysis of synchronous data flow graphs*. PhD thesis, Technische Universiteit Eindhoven, Eindhoven, July 2008.

[16] Maarten Hendrik Wiggers. *Aperiodic Multiprocessor Scheduling for Real-Time Stream Processing Applications*. PhD thesis, University of Twente, Enschede, June 2009.

[17] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, New York, NY, USA, 1975. ACM.

[18] J. R Gurd, C. C Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[19] Gregory M. Papadopoulos. Monsoon: an explicit token-store architecture. In *In Proc. of the 17th Annual Int. Symp. on Comp. Arch*, pages 82–91, 1990.

[20] B.D. Van Veen and K.M. Buckley. Beamforming: a versatile approach to spatial filtering. *ASSP Magazine, IEEE*, 5(2):4–24, April 1988.

[21] John G. Proakis and Dimitris G. Manolakis. *Digital Signal Processing*. Pearson Education, Inc., fourth edition, 2007.

[22] Ronald E. Crochiere and Lawrence R. Rabiner. *Multirate Digital Signal Processing*. Prentace-Hall Inc., 1983.

[23] B. Allen and M. Ghavami. *Adaptive array systems: fundamentals and applications*. John Wiley & Sons Inc, 2005.

[24] G. Bos. Radio astronomy signal processing on a tiled reconfigurable architecture. Master's thesis, Univ. of Twente, July 2010.

[25] Modelsim. `http://model.com/`.

[26] Synopsis design compiler. `http://www.synopsys.com/`.

[27] Cadence encounter. `http://www.cadence.com/`.

[28] Synopsis primetime. `http://www.synopsys.com/`.

[29] Qing Wu, M. Pedram, and Xunwei Wu. Clock-gating and its application to low power design of sequential circuits. *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on*, 47(3):415 –420, March 2000.

[30] Douglas L. Perry. *VHDL: Programming by Example*. McGraw-Hill, Inc., New York, NY, USA, 2002.

[31] Pong P. Chu. *RTL Hardware Design Using VHDL: Coding for Efficiency, Portability, and Scalability*. Wiley-IEEE Press, 2006.

[32] Keshab Parhi. High-level algorithm and architecture transformations for dsp synthesis. *The Journal of VLSI Signal Processing*, 9:121–143, 1995. 10.1007/BF02406474.

[33] V. Baumgarte, G. Ehlers, A. Nückel F. May, M. Vorbach, and M. Weinhardt. PACT XPP A self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2), 2003.

[34] Recore Systems BV. Montium®reconfigureable digital signal processing tile processor (tp) datasheet. Technical report, Recore Systems BV., September 2007.

[35] M. Morris Mano and Charles R. Kime. *Logic and Computer Design Fundamentals and Xilinx Student Edition 4.2 Package (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.

[36] Mentor graphics precision. `http://www.mentor.com/products/fpga/synthesis/precision_rtl/`.

[37] P. M. Heysters, G. J. M. Smit, and E. Molenkamp. Energy-efficiency of the montium reconfigurable tile processor. In *Proceedings of the International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA'04), Las Vegas, USA*, pages 38–44. CSREA Press, June 2004.

[38] P. M. Heysters. *Coarse-Grained Reconfigurable Processors - Flexibility meets Efficiency*. PhD thesis, Univ. of Twente, Enschede, September 2004.

[39] Shousheng He and Mats Torkelson. A new approach to pipeline fft processor. In *Proceedings of the 10th International Parallel Processing Symposium*, IPPS '96, pages 766–770, Washington, DC, USA, 1996. IEEE Computer Society.

[40] Anja Niedermeier, Rinse Wester, Kenneth Rovers, Christiaan Baaij, Jan Kuper, and Gerard Smit. Designing a dataflow processor using cλash. In *Proceedins of Norchip 2010*, 2010.

[41] William W. Wadge and Edward A. Ashcroft. *LUCID, the dataflow programming language*. Academic Press Professional, Inc., San Diego, CA, USA, 1985.

[42] Shail Aditya, Arvind, Jan-Willem Maessen, Lennart Augustsson, and Rishiyur S. Nikhil. Semantics of ph: A parallel dialect of haskell. In *IN PROCEEDINGS FROM THE HASKELL WORKSHOP (AT FPCA 95*, pages 35–49, 1995.

A

VHDL example code

The following listing shows an example on how dataflow nodes are implemented in VHDL. Here the ALU of the standard FlexCore is given because it includes the buffering of input data using arcs, a functional module with performs the calculations and a statefull part (the current destination pointer iterating over all destination of the dataflow node).

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3  use ieee.numeric_std.all;
4  library work;
5  use work.FlexCore_pkg.all;
6
7  entity ALU is
8    generic (
9      data_width : natural := 16;       --16 bit fixed point operations
10     valid_dests_width : natural := 2; --two bits for number of destinations
11     opcode_width : natural := 2;      --number of bits for an instruction
12     dest_addr_width : natural := 4    --width of the destination address
13   );
14   port (
15     clk, rst : in std_logic;
16
17     --the dataflow input for operands
18     operands_in_data : in std_logic_vector( 2*data_width-1 downto 0 );
19     operands_in_write : in std_logic;
20     operands_in_full : out std_logic;
21
22     --the dataflow input for instruction
23     destinstr_in_data : in std_logic_vector( opcode_width +
           valid_dests_width + 2**valid_dests_width*dest_addr_width-1 downto 0
           );
24     destinstr_in_write : in std_logic;
25     destinstr_in_full : out std_logic;
26
27     --output for destination and the result
28     destres_out_data : out std_logic_vector( dest_addr_width+data_width-1
           downto 0 );
29     destres_out_write : out std_logic;
30     destres_out_full : in std_logic
31   );
```

```
32 end entity;
33
34 architecture ALU_arch of ALU is
35
36   component InternalArc
37     generic (
38       data_width : natural
39     );
40     port (
41       rst, clk, wrt, rd : IN STD_LOGIC;
42       datain : IN STD_LOGIC_VECTOR( data_width-1 downto 0 );
43       full : OUT STD_LOGIC;
44       dataout : OUT STD_LOGIC_VECTOR( data_width-1 downto 0 );
45       empty : OUT STD_LOGIC
46     );
47   end component;
48
49   component CBPStateMachine
50     port
51     (
52       clk, rst, proc_cond : in std_logic;
53       cur_state : out cnstate
54     );
55   end component;
56
57   signal ops_fifo_out : std_logic_vector( 2*data_width-1 downto 0 );
58   signal destinstr_fifo_out : std_logic_vector( opcode_width +
       valid_dests_width + 2**valid_dests_width*dest_addr_width-1 downto 0 );
59   signal ops_fifo_rd, destinstr_fifo_rd, ops_fifo_empty,
       destinstr_fifo_empty : std_logic;
60   signal destPtr : unsigned( valid_dests_width-1 downto 0 );
61   signal instr : instruction;
62   signal valid_dests : std_logic_vector( valid_dests_width-1 downto 0 );
63   signal destination0, destination1, destination2, destination3,
       curDestination : std_logic_vector( dest_addr_width-1 downto 0 );
64   signal operand1, operand2 : std_logic_vector( data_width-1 downto 0 );
65   signal aluState : cnstate;
66
67 begin
68   Aops : InternalArc
69     generic map( data_width => 2*data_width )
70     port map(
71       rst => rst,
72       clk => clk,
73       wrt => operands_in_write,
74       rd => ops_fifo_rd,
75       datain => operands_in_data,
76       full => operands_in_full,
77       dataout => ops_fifo_out,
78       empty => ops_fifo_empty
79     );
80
81   Adestinstr : InternalArc
82     generic map( data_width => opcode_width + valid_dests_width + 2**
```

```
        valid_dests_width*dest_addr_width )
83    port map(
84      rst => rst,
85      clk => clk,
86      wrt => destinstr_in_write,
87      rd => destinstr_fifo_rd,
88      datain => destinstr_in_data,
89      full => destinstr_in_full,
90      dataout => destinstr_fifo_out,
91      empty => destinstr_fifo_empty
92    );
93
94  --split the input buses to operands instruction and destination
95  instr <= SLVToInstruction( destinstr_fifo_out( destinstr_fifo_out'high
        downto destinstr_fifo_out'high-opcode_width+1 ) );
96  valid_dests <= destinstr_fifo_out( 4*dest_addr_width + valid_dests_width-1
         downto 4*dest_addr_width );
97  destination0 <= destinstr_fifo_out( 4*dest_addr_width-1 downto 3*
        dest_addr_width );
98  destination1 <= destinstr_fifo_out( 3*dest_addr_width-1 downto 2*
        dest_addr_width );
99  destination2 <= destinstr_fifo_out( 2*dest_addr_width-1 downto
        dest_addr_width );
100 destination3 <= destinstr_fifo_out( dest_addr_width-1 downto 0 );
101 operand1 <= ops_fifo_out( data_width-1 downto 0 );
102 operand2 <= ops_fifo_out( ops_fifo_out'high downto data_width );
103
104 --process that handles the current destination pointer
105 process( clk, rst, valid_dests, destinstr_fifo_empty, ops_fifo_empty,
        destres_out_full, aluState, destPtr )
106 begin
107   if( rst = '0' ) then
108     aluState <= waiting;
109     destPtr <= "00";
110   else
111     if( rising_edge( clk ) ) then
112       --check wether the firing rule is satisfied -> state transitions
113       if( ( destinstr_fifo_empty = '0' ) and ( ops_fifo_empty = '0' ) and
              ( destres_out_full = '0' ) ) then
114         --firing rule satiesfied
115         aluState <= processing;
116       else
117         --firing rule NOT satiesfied
118         aluState <= waiting;
119       end if;
120       --on exit of of processing state:modify pointer
121       if( aluState = processing ) then
122         if( destPtr = unsigned( valid_dests ) ) then
123           destPtr <= "00";
124         else
125           destPtr <= destPtr + 1;
126         end if;
127       end if;
128     end if;
```

```
129     end if;
130   end process;
131
132   --comb. process that handles the  read and write signals
133   process( aluState, destPtr, valid_dests )
134   begin
135     case aluState is
136       when waiting =>
137         --waitong so no read and no write
138         destinstr_fifo_rd <= '0';
139         ops_fifo_rd <= '0';
140         destres_out_write <= '0';
141       when processing =>
142         if( destPtr = unsigned( valid_dests ) ) then
143           --last destination of list
144           destinstr_fifo_rd <= '1';
145           ops_fifo_rd <= '1';
146         else
147           --NOT last destination of list: do not clear fifo yet
148           destinstr_fifo_rd <= '0';
149           ops_fifo_rd <= '0';
150         end if;
151         destres_out_write <= '1';
152     end case;
153   end process;
154
155   curDestination <=
156     destination0 when destPtr = "00" else
157     destination1 when destPtr = "01" else
158     destination2 when destPtr = "10" else
159     destination3;
160
161   process( aluState, instr, operand1, operand2, curDestination )
162   begin
163     if( aluState = processing ) then
164       case instr is
165         when add =>
166           destres_out_data <= curDestination & std_logic_vector( signed(
                operand2 ) + signed( operand1 ) );
167         when sub =>
168           destres_out_data <= curDestination & std_logic_vector( signed(
                operand2 ) - signed( operand1 ) );
169         when mult =>
170           destres_out_data <= curDestination & std_logic_vector( resize(
                signed( operand2 ) * signed( operand1 ), data_width ) );
171         when others =>
172           destres_out_data <= ( others => '0');
173       end case;
174     else
175       destres_out_data <= ( others => '0');
176     end if;
177   end process;
178 end ALU_arch;
```

B

Literature report

This appendix includes the literature report. The literature report was written to become familiar with the concepts of dataflow architectures.

Dataflow architectures

Individual Assignment (121174)

by

Rinse Wester

Supervisors:
dr. ir. André B.J. Kokkeler
ir. Kenneth Rovers
Albert-Jan Boonstra
André W. Gunst

Computer Architecture for Embedded Systems
Faculty of EEMCS
University of Twente

August 17, 2010

Contents

List of Acronyms

ALU Arithmetic Logic Unit

CBR Code Block Register

CM Configuration Manager

DBR Data Base Register

DLP Data Level Parallelism

EDGE Explicit Data-Graph Execution

ETS Explicit Token Store

FFT Fast Fourier Transform

FIFO First In First Out

ILP Instruction Level Parallelism

ISA Instruction Set Architecture

NoC Network on Chip

NML Native Mapping Language

PAC Processing Array Cluster

PAE Processing Array Element

PE Processing Element

RISC Reduced Instruction Set Computer

SCM Supervising Configuration Manager

SKA Square Kilometre Array

SoC System on Chip

TLP Thread Level Parallelism

VHDL (Very High Speed Integrated Circuit) Hardware Description Language

XPP eXtreme Processing Platform

1

Introduction

Current and future radio telescopes rely for a great part on digital signal processing. The large dishes are replaced by a large number of small antennas from which the signals are digitized. This allows large arrays and electronic beamsteering such that pointing the telescope to specific areas of the sky is performed using software instead of mechanically steering dishes. All these techniques require a lot of processing power and with it a large amount of energy. Future telescopes like the Square Kilometre Array (SKA) require a huge amount of processing power which makes conventional processor architecture infeasible.

In the seventies a different approach to processing was developed called Dataflow-architecture. Instead of looking at the control of a program the dataflow approach is looking at the data-dependencies. All independent operations can be executed in parallel, dataflow processors can exploit this and can therefore achieve high performance. Because the operations performed in radio-astronomy are relatively simple but have a high level of parallelism, dataflow-like architectures seem a natural path to take. An open research question therefore is whether dataflow architectures can meet the demands in processing power and energy efficiency.

First a wide knowledge of dataflow architectures has to be gained. This literature report gives an overview of the developments in dataflow architectures. The aim is to use the developments in dataflow architectures to design a System-on-Chip core which very efficiently performs beamforming operations in, for example, radio astronomy and Radar.

This report starts with an introduction to the dataflow architecture concept followed by an exploration of static and dynamic dataflow architectures. Afterwards an overview of the recent developments of dataflow architectures is given. Chapter 6 shows the problems occurring in dataflow architectures and some solutions addressing these problems. The report concludes with the problem statement for the master-thesis in chapter 7.

2

Introduction to dataflow graphs

While conventional von Neumann machines are programmed using control-statements and load/store-instructions, dataflow architectures are data-driven. This means that dependencies in the dataflow graph determine the order of execution. The architectures directly execute these dataflow graphs containing all the operations and their dependencies. These graphs expose much more parallelism than the sequential style of a von Neumann architecture because only mathematical dependencies are present in dataflow graphs.

2.1 Dataflow graphs

Dataflow programs are expressed in graphs where the operators are called *nodes* and edges *arcs*(the dependencies among operations). The data-packets which are being transferred over these arcs are called *tokens*. Nodes are *enabled* if all the required input-tokens, according to the enabling-rule, are available. A node *fires* when it executes its operation by consuming its input-tokens and producing one or more output-tokens.

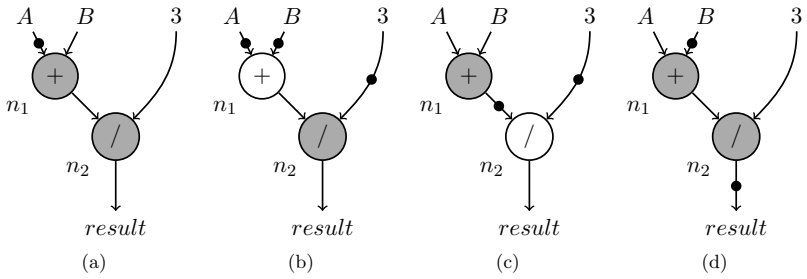


Figure 2.1: Firing rules in dataflow graphs

Figure 2.1 shows four asynchronous events in a dataflow graph created from the formula $result = \frac{A+B}{3}$. Node n_1 (the adder) is the first to receive a token but is not yet enabled because it requires two input-tokens. At a later time (Figure 2.1b) the right operand has arrived as well and node n_1 becomes enabled. Node n_2 (the divider) also receives a token but remains disabled

until n_1 has produced its result. Figure 2.1c shows the point in time where n_1 has produced its result and therefore enables n_2. When n_2 fires it consumes the tokens on its inputs and produces an output-token containing the result. This is depicted in Figure 2.1d.

These graphs are completely asynchronous meaning that the time between two events can be arbitrary. A pure implementation of such graphs therefore does not require a global clock because all the synchronization is enforced by the firing rules.

2.2 Conditional dataflow graphs

Conditional execution or loop structures require special types of nodes to handle boolean tokens coming from comparator nodes. There are two types of nodes required to support conditional constructs; *branch* and *merge* nodes. Branch nodes direct tokens to a certain path in the graph based on a boolean condition-token. A merge combines two paths together by simply forwarding any token that is on any of the inputs.

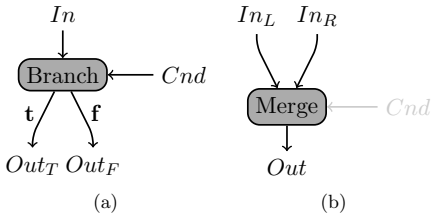
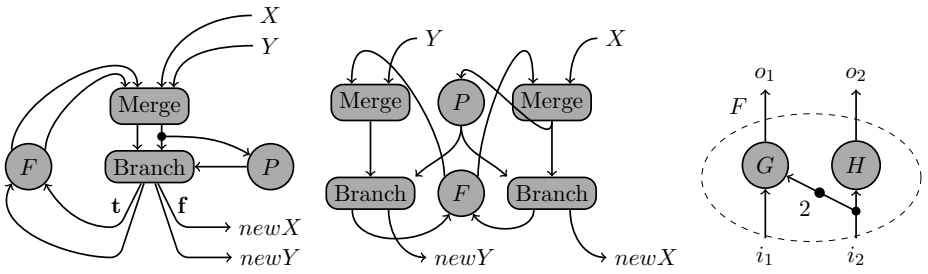


Figure 2.2: Conditional nodes: Branch and Merge.

Figure 2.2 shows the branch and the merge nodes as they are used in dataflow graphs. The branch node in figure 2.2a forwards the input token to the left when a control token with value *true* is present at the Cnd-input, the value is forwarded to the right output otherwise. A merge node is shown in Figure 2.2b which can either be deterministic or non-deterministic(deterministic requires a control token). When the conditional input is available the firing rule states that an output is only produced when both a control-token and a corresponding input-token are available.

Using these merge and branch nodes, also loops can be constructed. Great care has to be taken designing these loops in order to avoid race conditions. Several techniques have been proposed to ensure safety of the graphs [Vee86]. One technique is the *locking*-method which merges two loops together as depicted in Figure 2.3a.

Figure 2.3a shows a graph containing the safe implementation of the following loop: **while**$(P(x))$ $\quad (x, y) := F(x, y)$. The x and y tokens loop trough the graph in pairs and can therefore not outrun each other. Firing rules of F,



(a) Safe loop (b) Unsafe loop (c) Internal of F

Figure 2.3: Safe and unsafe loop constructs

merge and branch ensure that every x and y token is accepted and produced at the same time.

Figure 2.3b on the other hand shows a graph with two loops outrunning each other. The node F is a subgraph, the implementation is given in Figure 2.3c. If node G is slow compared to H, H will fire before G. The resulting token created by H leaves the subgraph and enters the loop for x-tokens again (right side of graph). If the function $P(x)$ still returns true the x-token enters the subgraph from the bottom. G is still processing its tokens so the loop for the y-tokens is still stalled. The token that entered the subgraph is sent to both G and H. Node H is now enabled so it accepts the new token while the token going to G has to wait.

When node H is several times faster than G, several x-tokens have run trough the x-loop before G finally produces a token. Because G is also dependant on x-tokens they will accumulate at the right input of G. This is depicted in Figure 2.3c which shows the number of waiting tokens on the input. The graph is unsafe because a new x-token will arrive before G accepts the token on the arc. Due to this accumulation an infinite amount of storage is needed if the complete graph would receive an infinite amount of x and y tokens. Two methods exist to overcome this problem, these are elaborated in the following sections.

2.3 Acknowledge method

An alternative to locked loops is using ack tokens which require an additional arc from the destination back to the source node. This ack ensures that there can be only one token on an arc and guarantees therefore safety of the graph. When a source sends a token to the destination node and the arc is free, the destination node sends back an acknowledge. When there is no space left on the arc the destination node does not send back an acknowledge, so the source buffers output-token until the arc is free again.

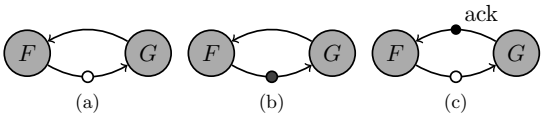


Figure 2.4: Successful acknowledge

During the first phase of execution in Figure 2.4, all arcs are free while node F produces a token. In the second phase the token is successfully placed on the arc. In phase 3 this is acknowledged by node G which sends an acknowledge back to the source.

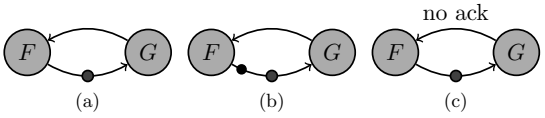


Figure 2.5: Failing acknowledge

When an arc is already occupied and the source tries to send a token, it is lost. The sender therefore doesn't receive an acknowledge and is has to try again at a later time. The result-token is saved in the sender until an acknowledge is received. This is depicted in Figure 2.5.

2.4 Tagging method

Distinguishing different iterations of loops is often realized using *tags*. These tags are used to make sure that only tokens with the same tag are accepted by receiving nodes. A tag is attached to every token to make sure that tokens of different loops do not interfere. The enabling rule then states that a node is enabled only if each input arc contains a token with the same tag. Again, such a machine is safe when no arc will ever contain more than one token with the same tag.

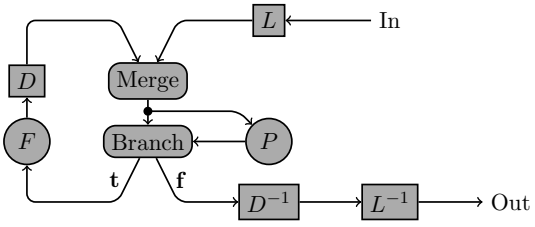


Figure 2.6: Tagging to implement loops

Figure 2.6 shows the dataflow graph of a simple loop. It contains four nodes that handle the different iteration-contexts using tags. The first node L creates a new loop-context by attaching a tag to tokens entering the loop. These tags ensure that tokens inside the loop are unique in the graph. The tokens in the loop can then be distinguished from tokens of the graph outside of the loop. Loop-in-loop constructs can also be made using these loop-tags(on every start of a loop a new loop-context is allocated). When a loop is completed the previous context is restored by a special node L^{-1}. Again different iterations of the loop have to be distinguished in order to prevent race conditions, the nodes D and D^{-1} are responsible for that. In every iteration of the loop tokens have a unique tag attached to them by the D-node which creates a unique context for each iteration. This allows different iterations of the loop to run in parallel because all tokens of different loops also have a different tag. When the loop is completed the iteration-context is restored, this usually corresponds to resetting a loop-counter.

2.5 Procedure calls

Tagging can also be used to implement procedure calls. When a call is made, a new tag-space(context) is allocated such that the nodes and arcs of the callee body are separated from the rest of the graph. This also enables recursive-procedure-calls. Machines using this tagged-token principle are called tagged-token-machines.

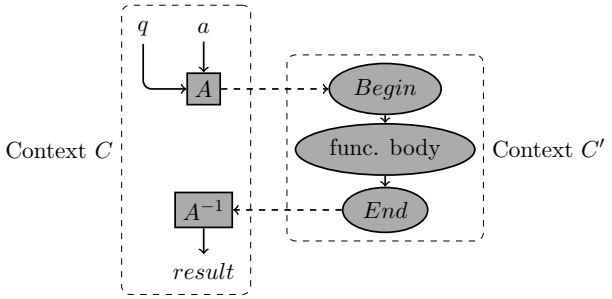


Figure 2.7: Using contexts to implement procedure calls.

Figure 2.7 shows a dataflow graph of a procedure call. The procedure is invoked from context C by the node A which creates a token with value a (the argument) with destination q (address of the function). Node A then creates a new context C' for the tokens in the called procedure. The argument a with the new tag arrives at the first node of the called procedure denoted by *begin*. The nodes in the procedure only accept tokens with a tag corresponding to the new context. Often a special node at the end of the procedure is used to transfer the return-value, the *End*-node. This return-value eventually arrives

at the A^{-1}-node which restores the context C completing the procedure call.

The context-nodes can also be used for implementing recursive procedure calls. Every call results in the creation of a new context in which the graph of the called function executes. Every call instance of the graph is therefore unique. The following example code shows an implementation of recursion, it calculates 3^n.

Listing 2.1: Recursion in C: 3^n

```
int pow3( int n )
{
  if( n>0 )
    return 3*pow3( n-1 );
  else
    return 1;
}
```

Figure 2.8 shows the three stages for calculating 3^2 which is an unfolded translation of the code in Listing 2.1. When executing the above program, three nested calls will occur. These calls can be translated to a dataflow graph where the arguments enter from above and results leave at the bottom. Every call from within a function-call is implemented using an outgoing arc connected to the top-node of the next call.

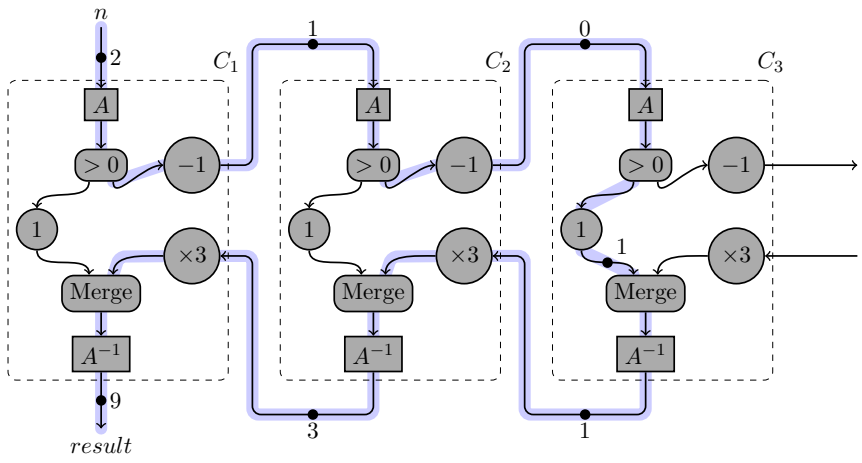


Figure 2.8: Unfolded dataflow graph for calculating 3^2.

As can be seen in figure 2.8, the calculation of 3^2 starts by sending the argument $n = 2$ to the first instance of the procedure-body. When the parameter-token enters the procedure-body, a new context C_1 is created. All the nodes in context C_1 now only accept tokens with a tag corresponding to context C_1. After the context creation, the node > 0 (similar to a branch node) sends the token to node -1 if the accepted value is bigger than 1, and to the constant node 1 otherwise. The token has a value 2 and is directed to subtraction node

-1. A new token is produced containing the value 1 which enters the next instance of the graph.

The tokens entering the second instance follow the same path because the value contained in the tokens is not yet zero. The token leaving this second instance has passed trough the -1 node making its value equal to zero. When it enters the third instance of the graph the > 0-node triggers constant-node 1 to create a token. This token is then accepted by the A^{-1}-node trough the merge-node. The A^{-1}-node restores the previous context. The token leaving A^{-1} therefore belongs to the context C_2.

The token with value 1 leaving the instance with context C_3 is now accepted by the $\times 3$-node of the second (C_2) instance. This triggers the creation of a token with value 3 which passes trough the merge-node of C_2 before the first context(C_1) is restored. The token leaving the second instance of the procedure body now has a tag corresponding to the context of the first instance(C_1).

In the first instance the token also passes trough the $\times 3$-node which completes the calculations. The context before the recursive-procedure-call is restored so the result-token has a tag corresponding to context of the caller-graph. This result-token can now be used in the caller graph.

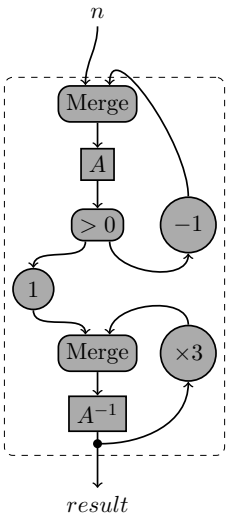


Figure 2.9: Folded implementation of recursive procedure-call.

Because all the different instances of the recursive procedures have a distinct context and the same structure, the instances can be merged. Tokens produced by the -1-node have to be fed back to the input and tokens produced on the output have to be transferred into the $\times 3$-node. Figure 2.9 shows the resulting merged graph.

The argument entering the graph causes a new context to be allocated,

before its value is decreased by one. This token re-enters the graph from the top which causes again a new context-creation. This sequence of operations is repeated until the token leaving the -1-node has a value 0. The > 0-node then triggers the creation of a constant token trough node 1 which is accepted by the merge and sent to the node A^{-1}.

For every token entering the graph trough the top merge-node a new context is created while the previous one is still active. This results in a stack of contexts. Only the context on top of that stack is active so all the nodes in the graph then only accept tokens with a tag corresponding to that context. The same stack of contexts enforces several invocations of the lower part of the graph.

Tokens on the bottom of the graph are fed back trough the $\times 3$-node if they have a tag corresponding to the current context. Via the merge-node these tokens cause the previous context to be restored(the context on top of the stack is popped off). After the previous context has been restored the resulting token is again sent to the A^{-1}-node via the $\times 3$ and the merge. The sequence of context-restores continues until the context, which was active before the procedure-call, is restored. Every time the loop on top of the graph is passed, the same amount of contexts are created. Therefore the same amount of loop-iterations on the bottom of the graph have to be executed such that the stack of contexts is reduced to the size it had before the recursive call was made.

3

Static dataflow architectures

The first machines to implement direct dataflow execution were static dataflow machines. These machines use either the lock-method or acknowledges as described in section 2.2 and 2.3.

First the structure of a basic static dataflow architecture is described followed by an implementation. The implementation described in section 3.2 is the MIT static dataflow machine. This should give a basis on static dataflow architectures which will be extended in chapter 4 where tagged token machines are described. These machines allow more advanced dataflow graphs like loops and procedure-calls.

3.1 Basic static dataflow

Static dataflow processors only support dataflow graphs without dynamical constructs like procedure-calls and recursion. A basic static dataflow machine has a structure like the one depicted in Figure 3.1.

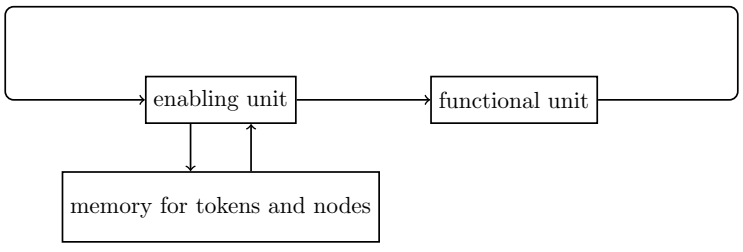


Figure 3.1: Basic structure of a static dataflow machine.

Nodes of a dataflow graph are stored in memory in the form of instructions. Instructions are described with an operand-code and a list of destination nodes. When a node has several destination nodes it produces a set of tokens that can be processed in parallel. Nodes accepting several input-tokens require additional control to make sure that when the node is executed all the required data is available. The unit performing this control is called the *enabling-unit*.

The actual firing of nodes is done in the *function-unit* which contains modules like multipliers, adders, etcetera.

The basic execution-flow as depicted Figure 3.1 starts with some initial tokens from the token-memory or from outside the machine. Tokens enter the enabling-unit and are stored in the token-memory. When all required tokens for a certain node are available, the execution may start. The enabling-unit is responsible for determining whether enabling-rules are satisfied. When the enabling-rule is satisfied the input-tokens and the node description are extracted from memory and packed into an *execution-packet*. This packet consists of the input-tokens, the operand-code and a list of destinations. This packet is sent to the functional unit which executes the operation determined by the operand code. The resulting token(s) leave the functional unit and enter the enabling unit again which completes the dataflow cycle.

3.2 MIT static dataflow processor

The first static dataflow machine being developed was the MIT static dataflow machine [DM75]. The structure of the MIT static dataflow machine depicted in figure 3.2 has a very similar circular structure compared to the scheme of Figure 3.1.

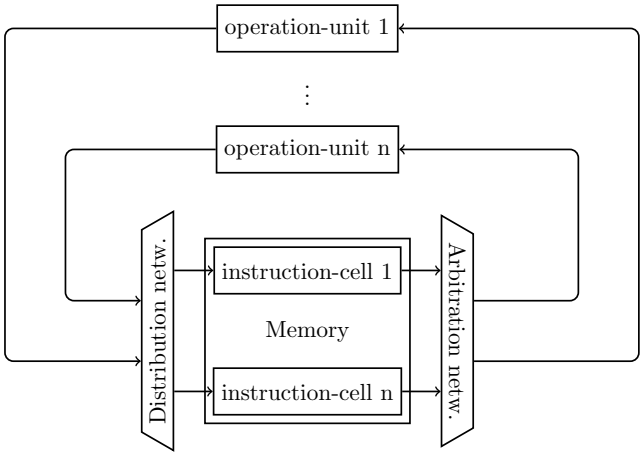


Figure 3.2: Structure of MIT static dataflow processor.

Dataflow programs written for the MIT static dataflow machine reside in the memory in so called *instruction-cells*. These cells contain three registers; one for the opcode and two registers for operands. According to the firing rule a register-cell becomes enabled when all the operands are available. This means that every instruction-cell has effectively its own matcher. The cell then informs the arbitration-network that there is data to be transmitted to the *operation-units* (modules at the top of Figure 3.2). An operation-

packet containing operands and instruction is sent to the correct operation-units by the arbitration-network. The arbitration-network can determine the destination (operation-unit) of the packet based on the opcode.

When an operation-unit fires, the resulting value is put in a packet together with one or more destination addresses. The *distribution-network* then makes sure the value is transferred to the correct instruction-cells. Operands of instruction-cells are addressed using the address contained in the packets produced by the operation-units. The result-values filling the operand-registers may enable new instructions which are then transferred to the operation-units again. This completes the cycle in which the data flows.

Several instruction-cells may become enabled at the same time and the arbitration-network is responsible to efficiently transfer all the packets to the correct operation-units. Several arbiters in the arbitration-network decide which instructions are extracted from the memory in a round-robin fashion. These packets then pass trough switches which direct the packets to the operation-units based on the opcode.

Conditional structures

In order to support loops and other conditional constructs, the MIT static dataflow processor uses *gating* which allows nodes to accept tokens based on a conditional token. Merge nodes are also available, although these are not physically implemented but integrated in the distribution-network. A branch-node can be composed of two gates as depicted in Figure 3.3.

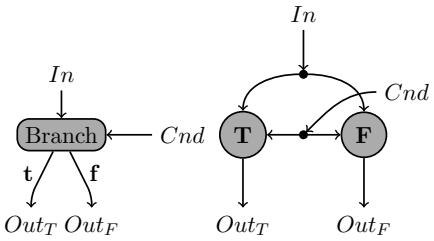


Figure 3.3: Branch made with gates.

The branch-equivalent in the above figure accepts a token and duplicates it, so an incoming token is sent to both gates. The *T-gate* only passes the incoming token to the output when it has received a control-token with value true. *F-gates* do the opposite: the input token is forwarded when a control-token contains false as value.

While in conventional dataflow graphs the branch and merge nodes are available as operation nodes, the MIT static dataflow machine has no direct implementation of these nodes. The gates are embedded in the operand reg-

isters of the instruction-cells. Also nodes performing decisions like comparing reside at a special location.

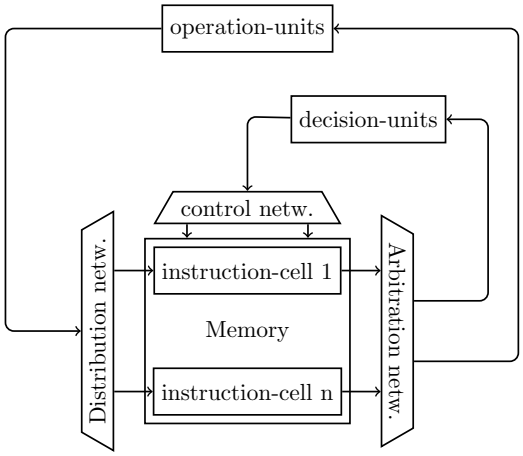


Figure 3.4: Processor with gating by decision-units.

Instructions that have to be processed by decision-units are selected and transferred by the arbitration-network. These decision-units produce only binary results which bypass the operation-units and go directly to the instruction-cells and act as control-packets. The gates are integrated in the instruction-cells i.e. depending on the mode of the operand register, an incoming operand is only accepted if also a control-packet is received.

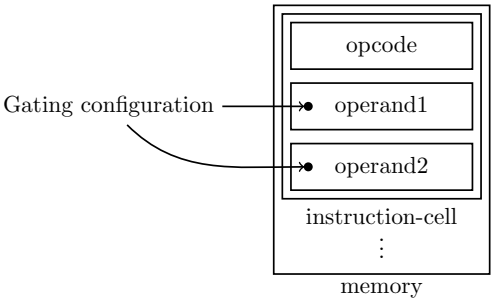


Figure 3.5: Structure of MIT static dataflow processor.

The operand-registers can be configured in four modes which are shown in Table 3.1. These modes control whether the incoming tokens are gated or not.

When an operand register is not gated tokens are accepted by the default firing rule. In *true*-mode an additional requirement must be satisfied. Not just all the input-tokens must be available but also a control-token must be available on the control port of the operand before it can fire. Depending on

no	The associated operand is not gated.
true	An arriving operand is accepted when a control-token containing true is accepted, the operand is discarded otherwise.
false	An arriving token is accepted in combination with a false-valued control-token.
cons	The operand is a constant value.

Table 3.1: Possible gating configurations.

the mode(true of false) the operand register only accepts tokens if the required control-packet has the correct value. When an operand register is configured in the *cons*-mode it doesn't accept tokens at all. The constant value residing in the operand-register is fixed and can only change during reprogramming of the processor.

Enhanced memory management

Generic dataflow graphs are often unbalanced which means that a lot of instructions need to be available in the memory while only a few are actively used. The MIT static dataflow processor has an additional memory in which the non-active instructions are stored while the instruction-cell act as cache. When an incoming token should be processed by an instruction-cell that is not yet available it is fetched from the instruction memory.

4

Tagged token machines

In order to support more advanced programming concepts like procedure-calls and recursion, dynamic dataflow machines were developed. These machines are called tagged token machines and use tags to distinguish different dataflow paths as described in section 2.4.

First the basic elements of a tagged token machine are explained followed by three implementations. Although around a dozen implementations have been made, the processors give a sufficient overview until the beginning of the nineties.

4.1 Basic tagged token machine

Figure 4.1 shows the basic components of a tagged token machine. A comparison of this structure with the basic structure of a static dataflow machine of Figure 3.1 shows that the enabling unit is now split into two stages. Because recursive constructs are allowed which require multiple active contexts as described in section 2.4, the space required on the arcs can get arbitrary large. This is because multiple tokens with different tags can be present on a single arc. Storing these tokens in the nodes is therefore not practical which is why a *token-memory* is introduced. The same concept also applies to the nodes: some graphs can get very large which is why the nodes of the dataflow graph are dynamically assigned to functional-units by the *fetching-unit*.

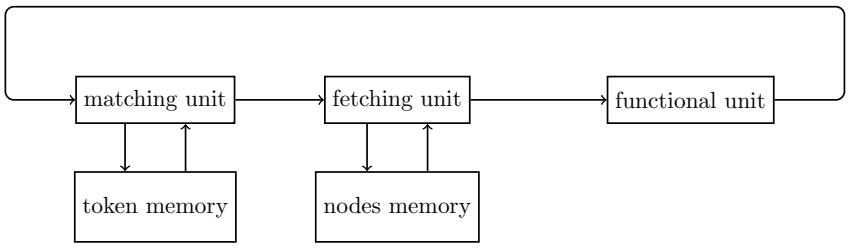


Figure 4.1: Basic structure of a tagged token machine.

The matching unit accepts tokens and checks whether the node that should

process this new token is enabled. Most machines limit the maximum number of inputs for a node to two which simplifies the matching unit. Tokens addressed to nodes with only a single input are ignored by the matcher and forwarded directly to the fetching-unit. Only tokens that are addressed to nodes with two inputs have to be matched. The matching-unit combines the destination and tag into an address and checks whether there is a token at that address of the token-memory. If that is the case the incoming token and the one in memory are passed on to the fetching-unit. The fetching-unit combines the incoming tokens into a packet with the node-description which is sent to the functional unit. The functional unit produces new tokens which are fed into the matcher completing the dataflow cycle. The combination of node and tag into an address forms a huge address-range called the *matching-space*. Managing this space has a direct effect on the performance of the matching unit which determines for a great part the overall performance of the architecture.

4.2 Manchester tagged token machine

Around 1976 the development of the Manchester tagged token machine began [GKW85] at the university of Manchester. The overall architecture of this machine is shown in the following figure.

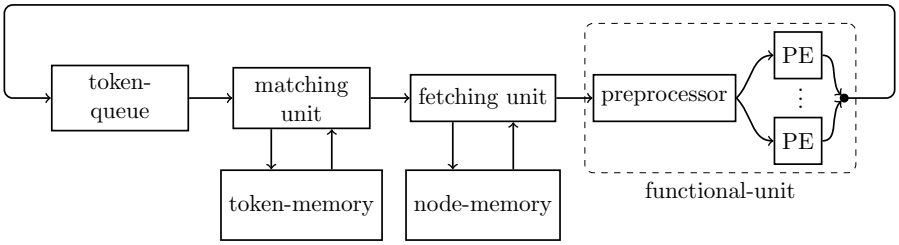


Figure 4.2: Basic structure of the Manchester tagged token machine .

Figure 4.2 shows the circular pipeline as it is implemented in the Manchester tagged token machine. It consists of four stages: token-queue, matching unit, fetching-unit and the functional unit. The communication between the units is asynchronous while internally they are synchronous. The token-queue is a FIFO buffer for smoothing the irregular arrival of tokens from the functional-unit. Tokens leaving the queue enter the matching-unit which only sends complete combinations of tokens to the fetching-unit. The nodes in the Manchester tagged token machine have either one or two inputs which simplifies matching.

The fetching-unit accepts the complete sets of tokens and combines these into an executable packet with a description of the destination-node. The maximum number of destinations of a node is also limited to two. If an

argument of a node is constant, the fetching-unit alters the packet such that the constant is placed on one of the inputs of the node.

The functional-unit consists of a preprocessor followed by several processing-elements. Execution-packets are sent to the Processing Element (PE)s by a distribution-network and result-tokens are sent to the token-queue by an arbiter located before the token queue. The preprocessor executes instructions that require access to counter memory. Examples of these counters are loop-iteration-counters and activation-frame-counter. The activation-frame-counter is used to create unique tags in order to support unique contexts.

Matching and tag-space

Matching is only required for tokens that have to be accepted by nodes with two inputs. During matching, the machine searches its memory for a token with the same destination and tag. The matching-unit verifies whether the destined node of the token has one or two inputs. Tokens for nodes with one input are simply forwarded but tokens destined for nodes with two inputs have to be matched. Matching is done based on an address created from the destination and tag. This address is used to determine whether that location in the token-memory is occupied.

Consider a single node n_i with 2 inputs that has to process a token-pair t_1 and t_2 which behaves according to the firing rule as explained in section 2.1. Initially no tokens are on the arcs of the node which corresponds to an empty token-memory-location. When either t_1 or t_2 arrives at the matcher it is stored in the token-memory at the address calculated from the destination and tag. After a while the second token also arrives which has the same destination and tag which results in the same address of the token-memory. The matcher then has two tokens with the same tag and destination which corresponds to an enabled node in the dataflow graph. These two tokens are then put in a packet and sent to the fetch-unit.

The address-space generated by combining destination and tag is too large to be physically implemented. Even if it could be implemented, the occupation would be very sparse. The Manchester tagged token machine uses a hashing algorithm implemented in hardware to generate addresses for the token-memory. The cell destined by this address has room for one token including the destination and tag. An extra bit is used to indicate whether the location is occupied. The biggest disadvantage of hashing in this case is that different incoming tokens can yield the same address after hashing. If a match fails, the token still has to be stored. If the hash-function yields the address of an already occupied location, another location with that address has to be found. In the Manchester tagged token machine this is implemented by using 16 parallel memory banks. When all these 16 locations are occupied and the hash-function yields again the same address, the token is stored in the overflow-unit. The overflow-unit is a memory containing a linked list of the

overflow-tokens. Searching for a match in this linked list is very slow. Another one-bit memory with the same address range as the token-memory is used to indicate whether a memory-location has overflown.

The tag attached to tokens consists of two parts: the activation-name and an index. The activation-name is similar to a context described in section 2.4. Indices in loops are also used in the same way as described in section 2.4. The sizes of these fields are determined at runtime while the total size is fixed. This distinction only exists inside the processing-elements. This split in tags allows programs with a lot of recursion and programs with a lot of different iterations to run efficiently on the architecture.

4.3 MIT tagged token machine

Around the same time as the start of the Manchester tagged token machine the MIT also started a project based on the tagged token principle. The resulting machine is the MIT tagged token machine [AN90]. Figure 4.3 shows the structure of a single processing element.

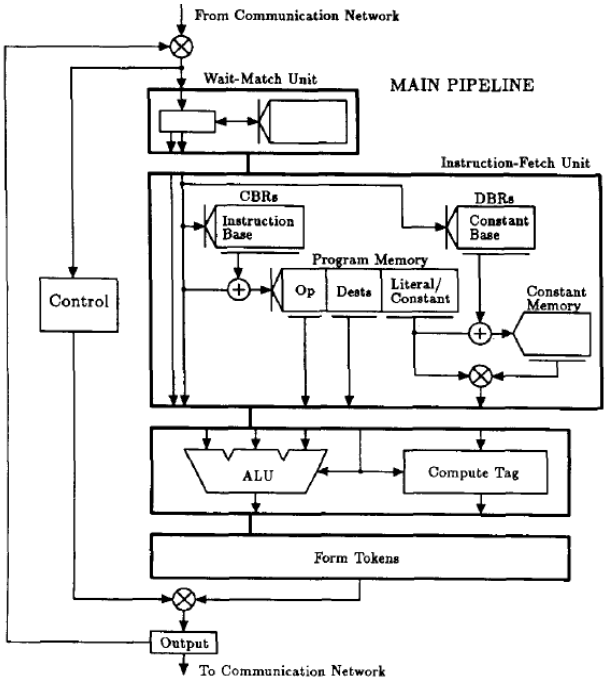


Figure 4.3: One processing element of the MIT tagged token architecture (reprinted from [AN90]).

As can be seen in Figure 4.3 the overall structure is the same as the general structure depicted in Figure 4.1. Tokens coming from the communication network outside of the processing element enter the *Wait-Match unit* from

above. Programs being executed on the MIT tagged token machine are translated into several dataflow graphs called *code-blocks*. The code-blocks reside in memory as an arbitrary linear sequence except for call and linkage addresses. The nodes of the graphs in the MIT tagged token machine are also limited to two inputs just like the Manchester tagged token machine .

Instructions are formed by several fields: the opcode, literals or constant and several destinations. All addressing is relative inside a code-block such that relocating the code to a new processing element is relatively simple. All destination addresses for tokens are therefore also relative to the address of the node producing this token. Because the whole program resides in memory, a way to determine the current code-block is required. The MIT tagged token machine uses a special register for this: the Code Block Register (CBR). It holds the address of the first node of the graph described by the current code-block. The full address can then be found by adding the relative address to the address contained in the CBR.

Token-matching in the MIT tagged token machine is done in the same way as with the Manchester tagged token machine. An incoming token is simply forwarded to the *instruction-fetch unit* when it is destined to a single-input-node and is matched if the destined node has two inputs. Based on the destination and tag an address is generated which points to a location that may contain the other required token of the pair. When that location is empty, the incoming token is stored in memory while a match occurs if the location is occupied. Both tokens are then sent to the instruction-fetch unit.

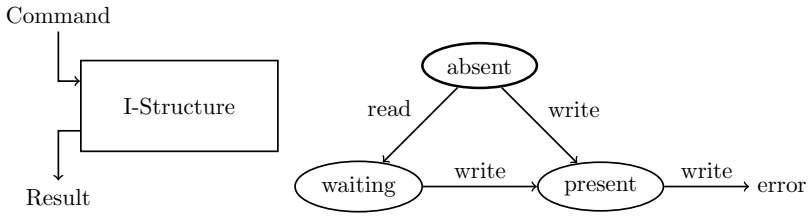
Once inside the instruction-fetch unit, a packet is created based on the destination and tag of the token. Literals and constants are also inserted into the packet by the same unit. Figure 4.3 also shows a register called Data Base Register (DBR). This register contains an absolute address of the constant-memory and is used in the program in the same way as the CBR. Constants are addressed relative to the address contained in the DBR.

The actual operations are performed using two Arithmetic Logic Unit (ALU)s. The left ALU of Figure 4.3 performs all the standard operations of the dataflow graph while the right ALU (contained in the Compute tag unit) creates a new tag for the result-token. The *form-token unit* combines the result of the calculation with the tag to form the result-token. This token may be sent to another PE or back into the current PE which completes the cycle of the architecture.

I-structures

In [AN90] a description of the I-structure is given which adds support for memory semantics(load/store operations). An I-structure can be seen as a memory controller connected to conventional memory. This means that not only tokens may contain data but also this global memory can be used in the same way as in von Neumann architectures. One important difference is

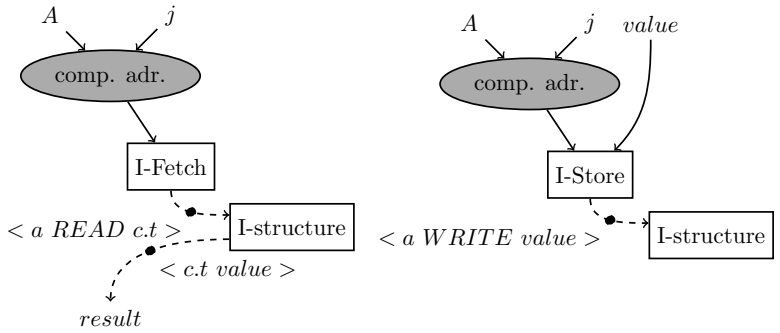
that locations in I-structures can be written only once(per code-block). These single-assignment semantics support a higher level of determinacy compared to conventional memory operations. Figure 4.4 shows an I-structure and the states in which an element of such a structure can be.



(a) Basic I-structure (b) Possible states of element

Figure 4.4: I-structure with single-assignment semantics.

An I-structure is controlled by read or write commands. The consumer of the result-tokens has to wait until the requested element becomes available. The initial state of the elements is set using an initialization command. Figure 4.4b shows the states of an element in the I-structure. During initialization, the state is set to *absent* and remains that way until a read or write command arrives. When a read-command arrives, the state changes to *waiting* which indicates that the result has to be sent to the destination as soon as the write-command arrives. A read before the element is written is therefore deferred until the element becomes *present*. A write-command directly after the initialization does obviously not produce a result-token. Because I-structures have single-assignment semantics a write after write is not allowed and results in a runtime-error.



(a) Read element (b) Write element

Figure 4.5: Read and write of I-structure.

Figure 4.5 shows the dataflow graphs to implement read and writes using an I-structure. The examples show how the element with index j of array A is

accessed. A is the address of the first element of the array and j is the offset so the address of $A[j]$ can be found by adding j to the base address of A. This address then triggers the creation of a read or write command which is sent to the I-structure.

Reading an element from the array as depicted in Figure 4.5a, starts with calculating the address and triggering a fetch using the *I-Fetch*-unit. This unit sends the actual command to the I-structure as a token $< a, READ, c.t >$ where a is the address and $c.t$ is the context with target t to which results have to be sent. The result-token $< c.t\ value >$ then leaves the I-structure and can then be consumed by target t.

Writing to an element of the array is done the same way except this doesn't produce any results. The write-procedure depicted in Figure 4.5b starts again with the address calculation which triggers the I-Store unit to send its command. A command is created by the *I-Store*-unit which accepts an address and the value.

I-structures do not reduce latencies which occur with communication to large memories. Instead dataflow architectures mask these delays by performing tasks of an other part of the dataflow graph. Because context-switches are relatively cheap in dataflow architectures, waiting nodes are temporarily replaced by nodes that can process data. When the requested data from an I-Structure finally becomes available, the requesting node is scheduled again so the data can be processed by the node. The time a node has to wait for data from memory is therefore not wasted but used by other dynamically scheduled nodes.

4.4 Monsoon

The monsoon processor is a general purpose dataflow architecture [Pap90] developed at the MIT as the successor of the MIT tagged token machine. The Monsoon supports loops and recursion using tags. It consists of several PEs and I-structures connected by a packet-switched-network. The network routes tokens based on the tag which uniquely defines to which PE the token has to be sent. Dataflow graphs to be run on the Monsoon are partitioned in code-blocks bound to PEs. These code-blocks are executed on the highly pipelined PEs as shown in Figure 4.6.

On each cycle the PE accepts a token which enters the Instruction-Fetch unit. Based on the tag and destination of the incoming token, an instruction can be directly addressed and fetched. The tokens are stored and fetched from the *Frame-Store* based on bitvalues in the *Presence-Bits-Memory* indicating whether a token is available or not. When all the required tokens are available, the actual operations of the nodes can be performed by the ALU. Parallel to this, a new tag is computed such that the *Form-Token*-unit can create a result-token. This token is multiplexed based on the destination. When the

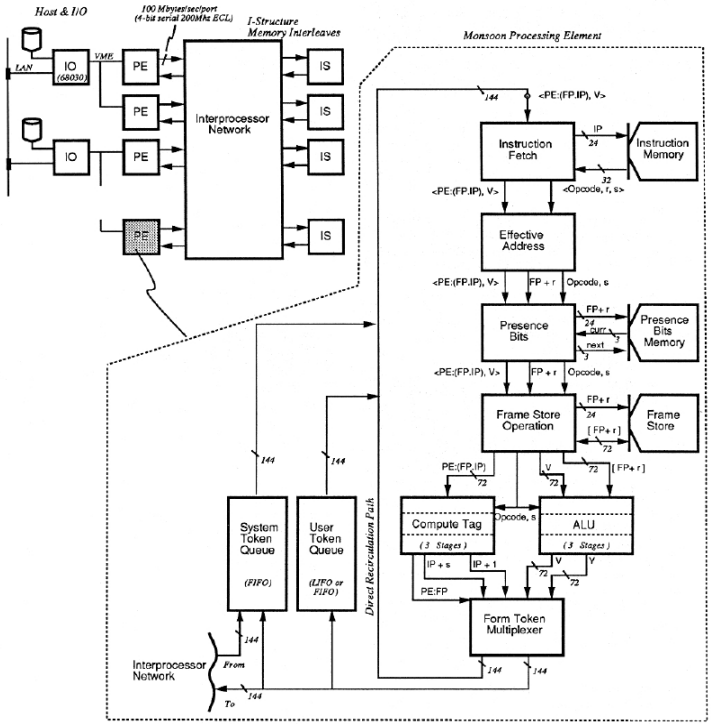


Figure 4.6: Overview of Monsoon architecture (reprinted from [Pap90]).

destination is local the token is sent by the recursive path directly to the fetch-unit. Non-local tokens are sent to the inter-processor-network. Efficient matching is done using an Explicit Token Store (ETS) which will be explained in the following section.

Explicit token store

Tokens entering the matcher implicitly claim storage when no match occurs. When there is not enough storage available in the token-store, deadlocks might occur. The Monsoon uses an ETS to solve this problem. The central principle of ETS is to dynamically allocate large blocks while the internal details about storage are determined at compile-time. When a code-block is activated an *activation-frame* is allocated to provide storage for all the tokens that are produced during the execution of the whole block. Arcs in the dataflow graph are therefore statically mapped to slots in the activation frames similar to register assignment in von Neumann architectures.

As can be seen in Figure 4.7, the frame-memory is the token-store of the ETS-principle so incoming tokens that do not match are stored in it. After storing the token the *presence-bit* is set from $absent(a)$ to $present(p)$. All the locations in the frame-memory are determined during compile-time so

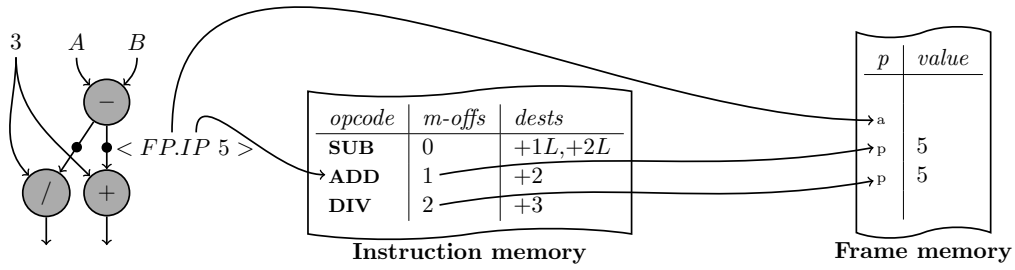


Figure 4.7: Example of ETS-principle.

during execution all the addresses of the tokens are constants which simplifies matching.

Figure 4.7 also shows an example-graph based on the ETS-principle. The subtracter in the graph has just produced two tokens while the token coming from the constant 3 still has to be received. The figure also shows instruction-memory and the frame-memory. When a token enters the instruction-fetch-unit the destination is read from the tag(depicted in the figure as $FP.IP$). The first part of the tag FP is the *Frame-Pointer* which is the address of the first memory location that is allocated for frame-memory. FP also determines the total context of the graph running on one PE. The IP part of the tag is the *Instruction-Pointer* which is the address of the instruction residing in the instruction-memory that should accept the incoming token.

Every instructions contains three fields: the opcode, a memory-offset and at most two destinations. The opcode determines the operation of the node that either requires zero,one or two operands. Matching is only needed when a node requires two operands. The memory-offset determines where matching should take place inside the frame-memory. This offset is relative to the frame-pointer FP which remains constant during the execution of a code-block. The last field of an instruction denoted with *dests* contains destination addresses. All the addresses are relative to the current instruction address. For example the subtracter in the dataflow graph has two destinations, the divider and the adder. The add-instruction comes directly after the sub-instruction so the relative offset is +1 while the div-instruction comes after that. Relative to the add-instruction it has an offset of +2. When a token is destined to a two-input-node an extra label is required in order the determine whether the token has to be sent to either the left or the right input. In the *dests*-field $+1L$ means send to the next instruction on the left input while $+1R$ means to the right input.

When the graph of Figure 4.7 is executed on the Monsoon, the token on the arc between − and + enters the instruction-fetch-unit. Here the *ADD* is fetched together with the matching-offset and the destination. The matching offset is then used to check whether the presence-bit is set. In this example example it is not yet set, but if it was, the corresponding value would be fetched from frame-memory. The incoming token combined with the one fetched from memory satisfy the firing-rule so the instruction is scheduled. The result-tokens from the scheduled operations are fed back into the fetch-unit which completes the dataflow cycle.

5

Recent developments

A lot of research on dataflow architectures has been done from the beginning of the seventies to the beginning of the nineties. But after that activities decreased and was directed towards Reduced Instruction Set Computer (RISC) architectures, because of new production-technologies. These technologies made it possible to easily increase the performance by increasing the clock-frequency. Increasing the clock-frequency any further seems to be stopped [AHKB00] and is known as the "frequency wall". Dataflow architectures are therefore becoming more interesting. This chapter gives an overview about the current developments on dataflow architectures.

First the WaveScalar is described which is the only real dataflow architecture described in this chapter. The second architecture is the XPP which actually is not a dataflow architecture but a coarse-grain-reconfigurable architecture with dataflow concepts. This chapter ends with the TRIPS architecture which is a mixture between dataflow and von Neumann based architectures.

5.1 WaveScalar

The WaveScalar Instruction Set Architecture (ISA) is a scalable and high performance architecture with traditional memory semantics [SMSO03] developed at the university of Washington. All the instructions on the WaveScalar are executed inside of the memory system and communicate with its dependant instructions in a dataflow fashion. Because modern superscalar processors do not scale well due to slow communication [AHKB00], complexity and its implicit serial execution, the WaveScalar takes a different approach. Instead of focusing on maximum utilization, the WaveScalar's goal is to minimize communication costs such that communication latencies are reduced. The WaveScalar has completely abandoned the program counter while keeping the memory ordering of a conventional von Neumann architecture. By removing the Program Counter, a central bottleneck is removed because all the PEs do not have to communicate with a single sequential component anymore.

The WaveScalar is a cache-only computer architecture consisting of a collection of processing elements. Conceptually the instructions execute directly in the memory and execute all in parallel in a dataflow manner. In practice

the instructions are cached and executed by an intelligent cache system called the *WaveCache*. The WaveCache loads these instructions from a memory into the processing units where they can remain for several clock cycles.

Waves and memory ordering

The compiler for the WaveScalar partitions the dataflow-graph into several blocks called *Waves*. These waves are similar to blocks used in the Monsoon architecture described in section 4.4. The WaveScalar processor executes one such wave at a time. Each instruction in a wave is executed only once per wave-activation. Waves can therefore have no internal loops. Loops and other conditional construct are implemented by conditionally executing whole waves. Control can therefore only enter the wave at a single point. These properties of waves makes it easier for the compiler to manage the memory ordering. The compiler partitions the applications in maximal(the largest possible amount of parallel instructions without loops) waves to reduce overhead.

Tagging is implemented in the WaveScalar by using *wave-numbers* and these numbers are attached to every data-value flowing trough the processor. Wave-numbers enable a single-functional unit to handle tokens from different loop-iterations for example. A special instruction called *Wave-Advance* accepts a token, increments its wave-number by one, and then outputs the tokens with the same value but with the incremented wave-number. On top of each wave is a wave-advance node for every input that accepts tokens from other waves. Tokens entering a new wave are assigned a new wave-number. All the tokens in the new wave have the same wave-number because they can only come from a single preceding wave. This is due to the rule that waves have a single point of entry. Loops are implemented by feeding tokens leaving the wave back in to the same wave. This does not introduce hazard because all the token pass trough a wave-advance node first. These wave-advance nodes therefore allow complete distributed wave-management and are controlled by software.

Traditional imperative programming languages provide total load-store ordering. The WaveScalar supports this by using *wave-ordering*. Wave ordered memory keeps track of memory operations by using annotations. Every memory-instruction is annotated with a unique number(location in the wave) and its relation with other memory instructions(predecessor and successor). When these instructions are executed, the annotations are used to enforce the correct order (to prevent data hazards).

The compiler locates the load/store instructions and assigns a unique number in breadth-first order. Next the compiler extracts the dependencies among the operations and constructs a graph. Every node in this graph represent a load or store operation and every edge represents a dependency among the connected operations. After constructing the graph, the operations are labeled. This label contains the unique number of the node, the predecessor and the

successor. When there is no unique successor or predecessor, a wildcard "?" is assigned.

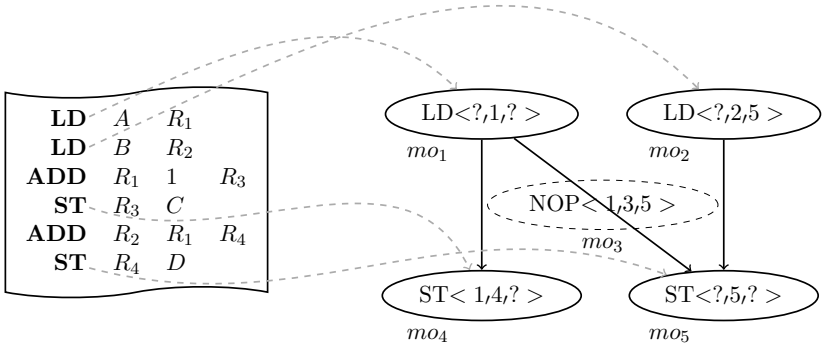


Figure 5.1: Memory ordering using annotations.

The memory controller may not execute any operation, before the operation it depends on is executed. When the memory controller cannot decide whether to execute a memory-operation based on the current and previous accepted labels, a gap is found. The memory controller can be sure that there exist no gap, when for each memory operation m in the graph:

- either m's successor number is them same as the the unique number of the operation arriving directly after m,
- or when m has multiple successors (wildcard "?"), the operation arriving after m has predecessor m.

Graphs violate the second rule when there exist an edge connecting an operation m_1 and m_2 where m_1 has several dependent operations and m_2 depends on several operations. An example of this is shown in Figure 5.1. This violation can be avoided by inserting an NOP operation that has no effect on the memory but solves the undecidability of the memory controller.

Figure 5.1 shows an example of wave ordered memory operations. The graph is a representation of the dependencies among the memory operations that can be found in the program on the left. The value fetched from the memory location A is used by two ADD instructions. The ADD instructions store the results in C and D which makes the operations mo_4 and mo_5 dependent on operation mo_1. The second ADD instruction is also dependent on operation mo_2 which is the reason why mo_5 has no unique predecessor. Because mo_1 has no unique successor, it uses a wildcard as successor. The successor of mo_1 and the predecessor of mo_5 both contain a wildcard which is

a violation of the second rule about gaps. The memory controller can therefore not know if mo_1 is executed and might decide to execute mo_5 before mo_1. This gap is filled by the NOP operation mo_3 (the dashed node). The graph is functional the same but both rules about the gaps are now satisfied.

The WaveCache

The eventual implementation of the WaveCache should contain about 2000 PEs grouped in clusters of 16. A physical physical prototype in silicon has not been build but an array of FPGA boards is used. Each PE has an instruction-cache of 8 instructions. The input size of the queues for receiving tokens is 2 tokens per instruction. The input-queues are indexed relative to the current wave-number and a small multiported RAM holds presence-bits for each possible element in the input-queues. The matching-unit uses these bits to determine whether enough tokens have arrived for executing an instruction. Each cluster of PEs has an L1-cache which can access DRAM trough an L2 cache. Figure 5.2 shows the architecture.

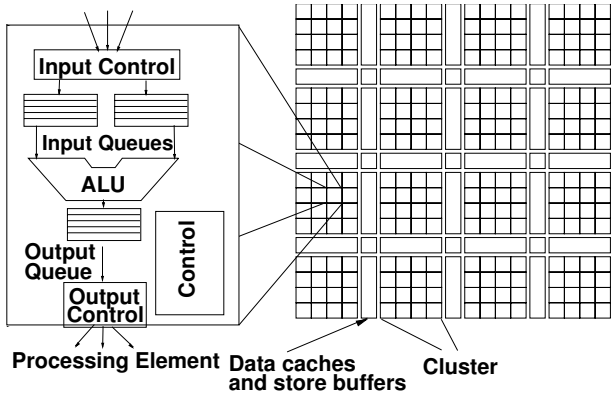


Figure 5.2: WaveCache overview (reprinted from [SMSO03]).

Within a cluster, the PEs communicate via a set of shared buses and PEs in the same cluster receive results at the end of a clock-cycle. The size of the cluster has its influence on performance: a large cluster needs more wiring within the cluster and introduces a higher wire-delay while smaller clusters result in more intra-cluster wiring. Changing the cluster size has only a marginal effect on the total performance: even with a cluster size of one element the performance (instructions per second) is reduced by only 51% [SMSO03].

The WaveScalar has distributed storebuffers(memory units supporting wave-ordered memory) and each set of four clusters is assigned to such a buffer. Each wave is bound to a storebuffer during execution, by means of a special instruction (Mem-Coordinate) that acquires the buffer. When acquired, the name of

this buffer is transferred to every node performing a memory-operation in the current wave.

In the paper describing the WaveScalar [SMSO03] a performance analysis is done by comparing it with an aggressively configured superscalar-processor (Alpha processor). The WaveScalar outperforms the von Neumann architecture with a factor of 2-7 in terms of instructions per second. This performance is achieved without speculative execution of conditional constructs. Branch prediction allows an additional increase in performance of a factor of 1.5. The evaluation was performed using benchmarks for the superscalar processor. The parts of these benchmark that are responsible for 95% of the execution time are translated using a binary translator for the WaveScalar. Programs are loaded in the main instruction memory where the WaveCache automatically fetches the instructions for execution trough the instruction cache.

5.2 PACT XPP

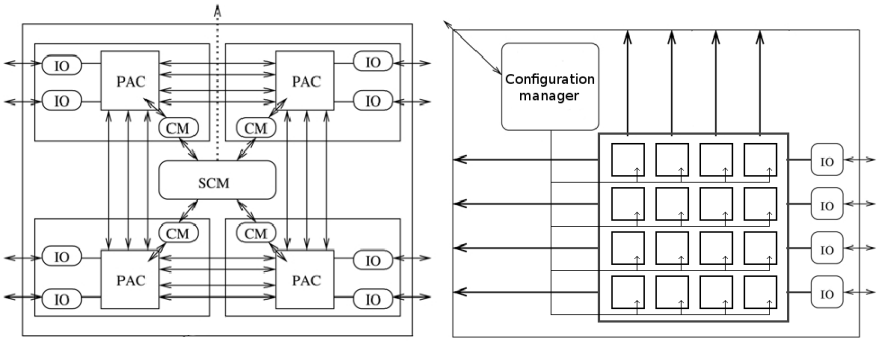
The eXtreme Processing Platform (XPP) is a runtime-configurable data processing architecture. While this architecture is not a real dataflow machine it does contain a lot of principles of dataflow architectures. The processor consists of several grids of processing elements which are connected via a packet-switched-network. The XPP is designed for digital signal processing applications. The chip is running at a clock-frequency of 150 MHz and should be able to perform 57.6 GigaOps/s [BEFM$^+$03], where an operation is an add, multiply or shift for example.

The XPP combines arrays of processing elements with configurations. These configurations are direct mappings of nodes from a dataflow graph onto physical processing elements. The configuration selects the functionality of the processing elements and configures the paths that connects these processing elements together. The processing elements in the XPP are ALUs supporting arithmetic and logical operations. During execution, the configuration remains unchanged. When the execution is completed the following configuration can be be applied. Since a configuration should be active for a large number (hundreds/thousands) of cycles, a reconfiguration does not introduce a lot of overhead. The results produced during execution are stored inside First In First Out (FIFO) buffers or in distributed memories. Subsequent configurations can then use these results for further processing.

Architecture

The processing elements in the XPP are called Processing Array Element (PAE)s. They consists of ALUs and buffering for data and are grouped in Processing Array Cluster (PAC)s. A complete XPP device contains four PACs which are connected to a Supervising Configuration Manager (SCM). This central configuration-manager is connected to local Configuration Manager

(CM)s attached to the PACs. This enables simultaneous execution of different applications on different part of the architecture.



(a) XPP device (b) PAC with CM

Figure 5.3: Different levels of the XPP (reprinted from [BEFM+03]).

Figure 5.3a shows an overview of the XPP-architecture. The central configuration-manager shown in the middle is the SCM which is the overall configuration-unit and is connected to an external configuration memory. A complete reconfiguration is always initiated from the SCM. The local CMs can also perform reconfigurations without interrupting the applications running on other PACs.

Figure 5.3b shows the internal structure of a PAC with its CM. The CM consist of a statemachine and internal RAM used for caching configurations. The CM is connected to the PAEs by a dedicated configuration-bus. All PAEs are connected by horizontal and vertical buses such that they can communicate with PAEs of other PACs.

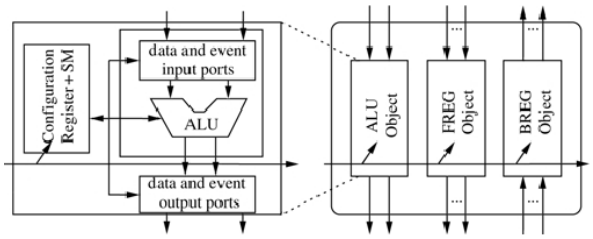


Figure 5.4: Structure of PAE (reprinted from [BEFM+03]).

Figure 5.4 shows the internal structure of a PAE. It consists of an ALU with buffering and two memory-blocks for storing data from other PAEs. Every ALU has a small configuration register with a statemachine that controls which operation the ALU has to perform. PAEs are self-synchronizing in the dataflow sense. Execution starts once all the inputs are available. The results

are forwarded by the packet-network that can transmit one packet per cycle. Operations in the ALU can also trigger reconfigurations.

Configuration

Applications running on the XPP usually consist of several phases that should be executing sequentially. The whole flow of configurations is handled by the SCM while local configurations are performed by the CM local to a PAC. The CMs local to PACs cache configurations so using these configurations several times yields higher configuration-performance.

The XPP also supports differential-configurations which are changes to a complete configuration. Typical examples are changes of coefficients and changes in the dataflow paths. Often not all PAEs in a PAC are used. This allows parts of an other configuration to be loaded in parallel. When the running configuration terminates the partial configuration is completed after which it starts to run.

The XPP can be programmed using the PACT proprietary Native Mapping Language (NML). This is a structural language that directly describes the functionality of the PAEs. The instantiation and the connections between PAEs is implemented similar to the use of components in (Very High Speed Integrated Circuit) Hardware Description Language (VHDL). Reconfigurations of the processors need to be explicitly defined in the program. This is implemented using special modules that start a reconfiguration on acceptance of a token. The application written in NML is translated to an XPP-binary by the compiler called *xmap*. This is a direct mapping of modules from the application to PAEs of the chip.

There is also a Vectorizing C compiler available which translates functions to NML modules. Only a subset of the C language can be translated to NML (a special I/O library is needed for streaming data). Loops in the input program are unrolled and pipelined such that they can be directly translated to NML. The resulting NML can then be compiled using xmap and executed on the device.

5.3 TRIPS architecture

The TRIPS architecture is a scalable and high performance architecture developed at the University of Texas Austin [BKM+04]. TRIPS is a highly parallel architecture with additional configuration options to address Instruction Level Parallelism (ILP), Thread Level Parallelism (TLP) and Data Level Parallelism (DLP) [SNL+03]. The current prototype has two processors each containing sixteen PEs.

EDGE

The TRIPS architecture uses Explicit Data-Graph Execution (EDGE) which is a compilation technique to exploit parallelism for highly parallel architectures like the TRIPS [SGM+06]. EDGE is based on direct instruction communication which means that instructions send their results directly to dependant instructions. The required storage is determined a compile time in the same way as with an ETS. Single shared units like register-files are therefore not needed and it therefore removes an important bottleneck. The instructions execute in dataflow order meaning instructions only have a reference to dependent instructions instead of register addresses.

EDGE is based on block-based execution where the blocks are constructed by the compiler. These blocks are issued and executed dynamically while the locations of the instructions inside those blocks are statically assigned to the PEs. These blocks are called *Hyperblocks* and contain the dataflow graph of a selected part of the input-program. Hyperblocks are made as large as possible by the compiler to express more parallelism. Loops are not available in hyperblocks but are formed around them such that only complete hyperblocks can be executed in one iteration.

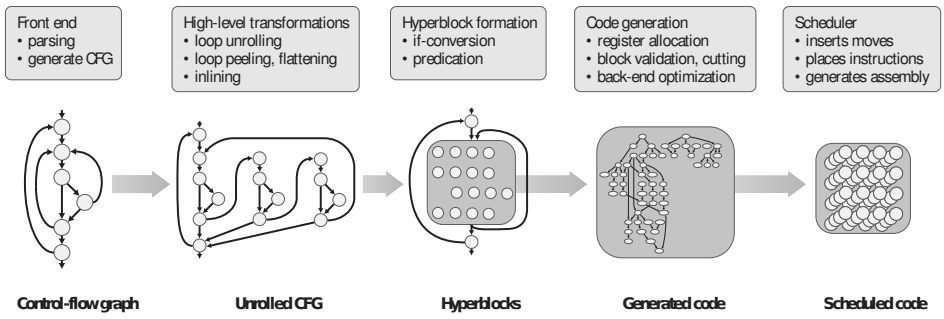


Figure 5.5: Compilation for EDGE (reprinted from [BKM+04]).

Figure 5.5 shows the different phases of compilation. First the program is translated into a control-flow-graph to determine dependencies. After this phase, the compiler can distinguish different threads in the program. The compiler then decides on which core these should be executed to exploit TLP. In the second phase the compiler optimizes away overhead by inlining and unrolls loops to expresses more parallelism. In this phase DLP and ILP are exposed as independent nodes in the dataflow graph. During the third phase, the nodes in the dataflow graphs are grouped in large hyperblocks. The hardware dependent part starts in phase four where the code for the TRIPS is generated. The required amount of buffering on the arcs of the dataflow graph is calculated and assigned statically to registers. The last phase schedules the execution

of the hyperblocks and maps the instructions directly to PEs. The compiler also tries to exploit both DLP and ILP by placing independent instructions on different PEs. Dependent instructions are placed as much as possible on the same PE such that communication delays between PE can be avoided.

Architecture

The TRIPS architecture is a set of blocks consisting of an array of processing elements designed for achieving good performance out of applications containing a lot of ILP. The hyperblocks formed by compilation of a program are directly executed by the group of processing elements of the TRIPS. These processing elements are grouped in cores forming, with memory, the complete architecture.

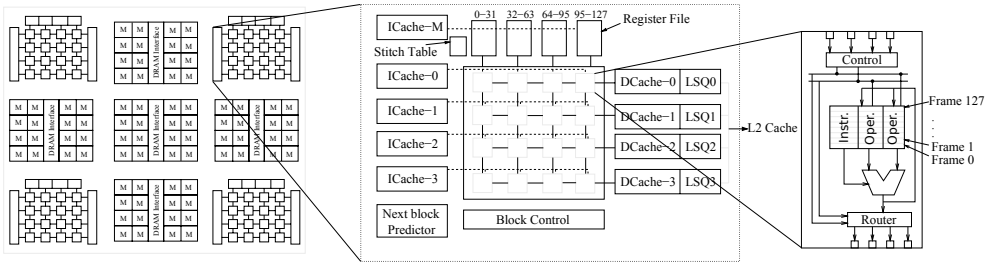


Figure 5.6: Overview of the TRIPS architecture (reprinted from [SNL+03]).

Figure 5.6 shows the TRIPS chip as it was planned to be implemented. The chip consists of four processing-cores combined with five memory-cores. These cores are interconnected by a routing network. The memory-cores implement L2-cache functionality combined with a memory-controller which connects to external DRAM-memory.

The second figure zooms in on a single TRIPS-core and shows sixteen processing elements aligned in a 4×4 grid. This grid executes one hyperblock at a time. All instructions and data are initially fetched from external memory but reside in the *ICache* and *DCache* units during execution of a hyperblock. Intermediate values produced by the processing elements are stored in register-files located at the top of the core. Switching to an other hyperblock when the current one is completed is controlled by the *Block-control* module. This module receives an event when a hyperblock is completed and decides which hyperblock should follow. The TRIPS architecture also supports speculative execution. This is implemented by the *Next-Block-Predictor* which is similar to a branch-predictor in von Neumann architectures. The Block-Control-module fetches new hyperblocks based on predictions from the Next-Block-Predictor.

The rightmost scheme shown in Figure 5.6 shows a processing element consisting of an integer-ALU, a floating-point-unit and a reservation-station. The reservation station behaves as matcher, it schedules instructions when all

required operands are available. The instructions are initially placed inside the PE before starting the execution of a hyperblock so they can be scheduled very quickly. After execution, the PE forwards the result to the local reservation-station or a station of an other PE in the grid. Results can be sent to any PE in the grid using the data-network that connects all nodes together.

6

Problems with dataflow architectures

Although the dataflow-principle seemed simple to implement, the first architectures showed that keeping the utilization high was difficult. The main bottleneck was matching and managing the token storage which had a large impact on the total performance [Vee86]. Because associative memories were very expensive finding the correct token in token-memory was done by hashing which turned out to be relatively slow and introduced problems with overflows: real applications often required a lot of token storage such that memories filled up quickly. As a result the utilization of the PEs did not exceed 25% of the available processing power [Vee86]. Hashing required an additional memory, the overflow memory, which stores the overflow-tokens in a linked list structure. For every overflow the whole linked list had to be searched which is not only slow but also dependent on the number of tokens that are already stored in it.

The complexity of matching was partly solved with ETS. Now the compiler determined the amount of token-storage and the location where the matching of tokens takes place in the token-store. Due to direct addressing, the slow hashing and the overflow-unit could be removed due to analysis at compile-time.

An other problem with early dataflow architectures is that they ignore storage hierarchy which causes a lot of outstanding memory-requests . These cannot all be masked by scheduling of an other part of the dataflow-graph [HHLL93]. Scheduling based only on the firing-rule of dataflow graphs ignores knowledge available at a higher level of the program. For example, instructions of a second thread are scheduled before the first thread completes. Completing the first thread before starting the next may reduce the amount of token storage required.

It has been argued [CSE93] that dataflow machines could not hold enough data near the processor to achieve substantial parallelism. That was based on the assumption that creating a lot of processing elements on a single chip was hard and getting fast memory near the PE was expensive. With current technology, these problems do not apply anymore as has been shown with the WaveScalar and TRIPS architecture.

High utilization of the PEs depends on how much parallelism can be ex-

pressed in the programming language. Imperative language like C, C++ and Java are by their nature sequential and restrict therefore the amount of parallelism. This led to the development of new programming languages like Val and Id which yield more parallelism [JHM04].

6.1 Common approaches to problems

Current dataflow architectures use EDGE or a similar technique to exploit the parallelism available in the programs. As described in section 5.3 EDGE first takes the program and expands it(unrolling loops and inlining procedures) to expose as much parallelism as possible. These expanded graphs are partitioned in blocks without conditional constructs in order to achieve higher utilization. The conditional constructs are implemented using conditional execution of whole blocks. Although different names are used for the grouping of instructions (Blocks [Pap90], Hyperblocks [BKM+04], Waves [SMSO03]), the basic principle is the same for the recent architectures(WaveScalar and TRIPS).

Another common solution among current dataflow (like) machines is to determine the token-storage at compile time. The locations of tokens that have to be stored are fixed at runtime which means that hashing and overflow memories are not required anymore. Matching is reduced to checking a single bit in a presence memory. This explicit token store ETS is usually implemented using one or several registers. ETS is one of the steps performed in EDGE.

7

Problem statement

The purpose of this literature report was to get insight in dataflow architectures. An overview of the developments in dataflow architectures has been given starting with the first static dataflow machine (MIT static dataflow machine described in 3.2) until the recent developments on the TRIPS architecture(section 5.3).

The eventual goal within the CAES research group of the University of Twente is to develop a System on Chip (SoC) that can efficiently perform beamforming operations used in radio astronomy and radar applications. This chip is a tiled reconfigurable architecture with processing cores connected by a Network on Chip (NoC). The type of processing core used in this SoC is the FlexCore proposed in [RvdK+09]. How this processing element should be implement is still an open question.

In my master thesis I am going to implement the FlexCore in VHDL such that the architecture can be synthesized and evaluated. The question of how this core should be implemented can be split in two parts.

The first part is to determine which principles should be used to implement the matching and routing of data trough the core. The resulting core is then evaluated such that more can be said about the performance in terms of processing-power, area and energy consumption.

The second sub-question is how the granularity of the processing elements should be altered such that the overhead, introduced in the first part, is acceptable? If for example the adders and multipliers are replaced by larger blocks such as butterfly operations or complete Fast Fourier Transform (FFT)s, what effect does it have on the performance?

The problems encountered during the development of dataflow architectures described in chapter 6 are important guidelines. Important aspects are:

- Matching (should there be one specific module or should these be distributed)
- Utilization (how well are the ALUs occupied)
- Datastructures (streams from streaming applications)
- Memory-hierarchy (storage of tokens and (partial) streams)

Bibliography

[AHKB00] Vikas Agarwal, M. S. Hrishikesh, Stephen W. Keckler, and Doug Burger. Clock rate versus IPC: the end of the road for conventional microarchitectures. *SIGARCH Comput. Archit. News*, 28(2):248–259, 2000.

[AN90] K. Arvind and Rishiyur S. Nikhil. Executing a program on the MIT Tagged-Token Dataflow Architecture. *IEEE Trans. Comput.*, 39(3):300–318, 1990.

[BEFM+03] V. Baumgarte, G. Ehlers, A. Nückel F. May, M. Vorbach, and M. Weinhardt. PACT XPP A self-reconfigurable data processing architecture. *The Journal of Supercomputing*, 26(2), 2003.

[BKM+04] Doug Burger, Stephen W. Keckler, Kathryn S. McKinley, Mike Dahlin, Lizy K. John, Calvin Lin, Charles R. Moore, James Burrill, Robert G. McDonald, William Yoder, and the TRIPS Team. Scaling to the end of silicon with EDGE architectures. *Computer*, 37(7):44–55, 2004.

[CSE93] David E. Culler, Klaus E. Schauser, and Thorsten von Eicken. Two fundamental limits on dataflow multiprocessing. In *PACT '93: Proceedings of the IFIP WG10.3. Working Conference on Architectures and Compilation Techniques for Fine and Medium Grain Parallelism*, pages 153–164, Amsterdam, The Netherlands, The Netherlands, 1993. North-Holland Publishing Co.

[DM75] Jack B. Dennis and David P. Misunas. A preliminary architecture for a basic data-flow processor. In *ISCA '75: Proceedings of the 2nd annual symposium on Computer architecture*, pages 126–132, New York, NY, USA, 1975. ACM.

[GKW85] J. R Gurd, C. C Kirkham, and I. Watson. The manchester prototype dataflow computer. *Commun. ACM*, 28(1):34–52, 1985.

[HHLL93] A. R. Hurson, A. R. Hurson, Ben Lee, and Ben Lee. Issues in dataflow computing. *Adv. in Comput*, 37:285–333, 1993.

[JHM04] Wesley M. Johnston, J. R. Paul Hanna, and Richard J. Millar. Advances in dataflow programming languages. *ACM Comput. Surv.*, 36(1):1–34, 2004.

[Pap90] Gregory M. Papadopoulos. Monsoon: an explicit token-store architecture. In *In Proc. of the 17th Annual Int. Symp. on Comp. Arch*, pages 82–91, 1990.

[RvdK+09] Kenneth C. Rovers, Marcel D. Burgwal van de, Jan Kuper, Andre B.J. Kokkeler, and Gerard J.M. Smit. On reconfigurable tiled multi-core programming. In *Proceedings of the 20th Annual Workshop on Circuits, Systems and Signal Processing, ProRISC 2009*, pages 507–514. Technology Foundation, November 2009.

[SGM+06] Aaron Smith, Jon Gibson, Bertrand Maher, Nick Nethercote, Bill Yoder, Doug Burger, Kathryn S. McKinle, and Jim Burrill. Compiling for EDGE architectures. *Code Generation and Optimization, IEEE/ACM International Symposium on*, 0:185–195, 2006.

[SMSO03] Steven Swanson, Ken Michelson, Andrew Schwerin, and Mark Oskin. Wavescalar. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 291, Washington, DC, USA, 2003. IEEE Computer Society.

[SNL+03] Karthikeyan Sankaralingam, Ramadass Nagarajan, Haiming Liu, Changkyu Kim, Jaehyuk Huh, Doug Burger, Stephen W. Keckler, and Charles R. Moore. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture. In *ISCA '03: Proceedings of the 30th annual international symposium on Computer architecture*, pages 422–433, New York, NY, USA, 2003. ACM.

[Vee86] Arthur H. Veen. Dataflow machine architecture. *ACM Comput. Surv.*, 18(4):365–396, 1986.