The Dissertation Committee for Mark William McDermott certifies that this is the approved version of the following dissertation:

Dataflow-Processing Element for a Cognitive Sensor Platform

Committee:

Jacob Abraham, Supervisor

Andreas Gerstlauer

Glenn Lightsey

Haris Vikalo

Steven Smith

Arjang Hassibi

Dataflow-Processing Element for a Cognitive Sensor Platform

By

Mark William McDermott, BSEE; MSE

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2014

Dedication

This is dedicated to my family for the support they have provided in my crazy dream to complete this degree.

Acknowledgements

I would like to acknowledge and thank Dr. Margarida Jacome for planting the seed about the power of Synchronous Dataflow many years ago. It was the genesis of this dissertation and has taken me on a six-year journey of researching, designing and implementing this platform.

I also need to thank Dr. Tony Ambler for introducing me to a great book about the future of engineering, specifically “The Engineer of 2020”. This book and its companion book “Educating the Engineer of 2020” inspired me to work on this Ph.D. so that I can become more directly involved in the education of future engineers.

Many thanks to the members of my committee. It is a real honor to work with them on this endeavor, as they are all involved in the sensor ecosystem from different reference points. I am especially honored to be able to work with Dr. Abraham on both the academic teaching side and on this research topic. It has been fourteen years since he formed the “Circuit Board” and spawned a world-class Circuits and Systems program at the University of Texas at Austin. I have learned so much from him about how to put together excellent teaching programs. As my research advisor he has been instrumental in helping me to focus on what really matters. He has this wonderfully subtle way of putting ideas in my head about what to do next.

And, eternal thanks to everyone who has helped me fill my toolbox of life. You know who you are and I am paying it forward.

Dataflow-Processing Element for a Cognitive Sensor Platform

Mark William McDermott, Ph.D.

The University of Texas at Austin, 2014

Supervisor: Jacob Abraham

Cognitive sensor platforms are the next step in the evolution of intelligent sensor platforms. These platforms have the capability to reason about both their external environment and internal conditions and to modify their processing behavior and configuration in a continuing effort to optimize their operational life and functional utility. The addition of cognitive capabilities is necessary for unattended sensor systems as it is generally not feasible to routinely replace the battery or the sensor(s). This platform provides a chassis that can be used to compose embedded sensor systems from composable elements. The composable elements adhere to a synchronous data flow (SDF) protocol to communicate between the elements using channels. The SDF protocol provides the capability to easily compose heterogeneous systems of multiple processing elements, sensor elements, debug elements and communications elements. The processing engine for this platform is a Dataflow-Processing Element (DPE) that receives, processes and dispatches SDF data tokens. The DPE is specifically designed to support the processing of SDF tokens using microcoded actors where programs are assembled by instantiating actors in a graphical modeling tool and verifying that the SDF protocol is adhered to.

Table of Contents

List of Figures

List of Tables

Chapter 1. Introduction to Reactive Sensor Systems

1.1 Transducers, sensors, actuators and systems

A transducer is a device converts energy from one form to another. There are two types of transducers: 1) sensors that detect a change in a physical stimulus and turn it into a signal that can be measured and 2) actuators that output action into the physical world. Figure 1.1 below shows an example of typical sensor/actuator system. The sensor in this system can be thermistor that measures temperature and the actuator is a relay that controls a cooling or heating system.

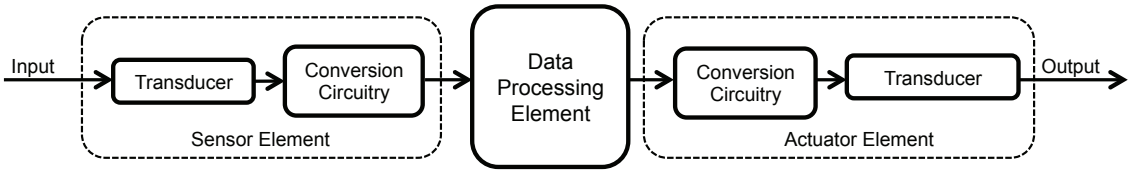


Figure 1.1: Typical sensor/actuator system

Typical input transducers would include strain gauges, piezoelectric devices, phototransistors, hall-effect devices, magnetometers, thermo-couples, ion-sensitive transistors, capacitive displacement devices, thermistors and bio-sensing devices. Table 1.1 below shows examples of various types of stimulus sources.

As can be seen in Figure 1.1, the input data from the sensor element is processed before it outputted to the actuator. There are three levels of processing capability associated with sensor systems and are described in more detail in the next section.

Table 1.1: Examples of stimulus sources for sensor systems

STIMULUS TYPE	EXAMPLES
Acoustic	Wave amplitude, spectrum, polarization, velocity, phase
Electrical	Voltage, charge, current, electric field, conductivity, permittivity
Magnetic	Magnetic field, amplitude, phase, polarization, flux, permeability
Optical	Wave amplitude, spectrum, polarization, velocity phase, emissivity, reflectivity, absorption
Thermal	Temperature, flux, thermal conductivity, specific heat
Mechanical	Position, mass, shape, density, acceleration, force, stress, strain, pressure, torque, stiffness, orientation

1.2 Hierarchy of sensor systems

The hierarchy of sensor systems is generally described as having three distinct levels of capabilities. The first level is a smart sensor system where the system is able to identify its purpose and is able to communicate information to and from other devices. The second level is generally classified as intelligent and this is achieved by adding the ability to recognize, interpret and understand sensor stimuli. The third level of capability is the cognitive sensor system that adds reasoning and cognition to the intelligent sensor system, allowing it to make decisions based on the sensor stimuli and to be totally aware of the environment. This classification is graphically illustrated in Figure 1.2 below [1].

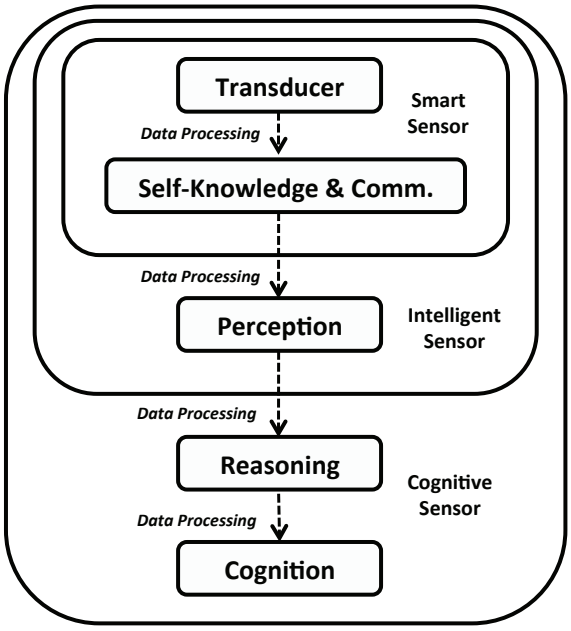


Figure 1.2: Computational hierarchy of advanced sensor systems

Each level of hierarchical capability requires a corresponding improvement in computational and energy performance. While transistor scaling has provided some of the additional computational efficiency, the energy supply for embedded sensors is staying virtually constant since Moore's Law does not apply to battery technology. That said, Koomey, et al, observe that computational efficiency (measured in computations/Joule) is improving at a similar rate to Moore's Law [2]. This may be adequate for workloads whose computational efficiency requirements remain constant from one generation to the next. However as will be shown below in Section 1.5 there needs to be a huge increase in computational efficiency in the next 20 years and relying on transistor scaling will not be an option.

1.3 Reactive systems

Sensor systems are generally considered to be event-driven (reactive) systems. The classical definition of a reactive system is "A system that changes its inherent operations, outputs and system state in response to stimuli both from within or externally generated" [3]. Reactive systems follow a pace dictated by the environment and they only need to be as fast as required to handle the stimuli. Additionally, reactive systems are concurrent, discrete valued and time varying. The concurrency of a reactive system is an essential feature that can be implemented using a number of different models including Dynamic Dataflow (DDF), Synchronous Dataflow (SDF), Discrete Events (DE), Petri nets, Khan Process Networks (KPN), and the synchronous/reactive model. For this work Synchronous Dataflow (SDF) is used to model how the processing elements process tokens as they propagate through the network. SDF is a special case of dataflow where the flow of control is predictable at compilation time [4].

1.4 Synchronous Dataflow network

A Synchronous Dataflow network is a collection of functional nodes, which are connected and communicate via unbounded First-In-First-Out (FIFO) queues [5] [6]. Each node is called an "Actor" and performs computations on the data that is communicated via the queues. Each datum is called a "Token" and in a Synchronous Dataflow model there are a fixed number of tokens consumed each time an actor performs a computation. The actor "Fires" when all of the tokens have been received by the FIFO queue. When fired, the actors consume input tokens and produce output tokens. Figure 1.3 below shows an example of a Finite-Impulse-Response (FIR) actor and an addition (ADD) actor. The FIR actor consumes a stream of one input token and produces

one output token. The ADD actor consumes two input tokens and produces one output token.

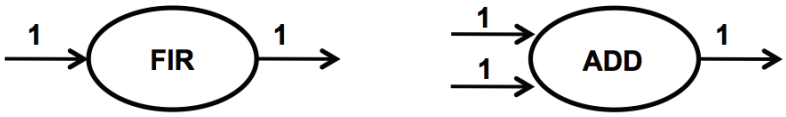


Figure 1.3: Synchronous Dataflow Actors

A key property of a dataflow model is that the output token sequences do not depend on the firing order of the actors. Figure 1.4 shows an example of a system composed of three actors. Actor ONE consumes five tokens and produces one token. Actor TWO consumes three tokens and produces two tokens. Actor THREE consumes the tokens from actors ONE and TWO and produces five output tokens.

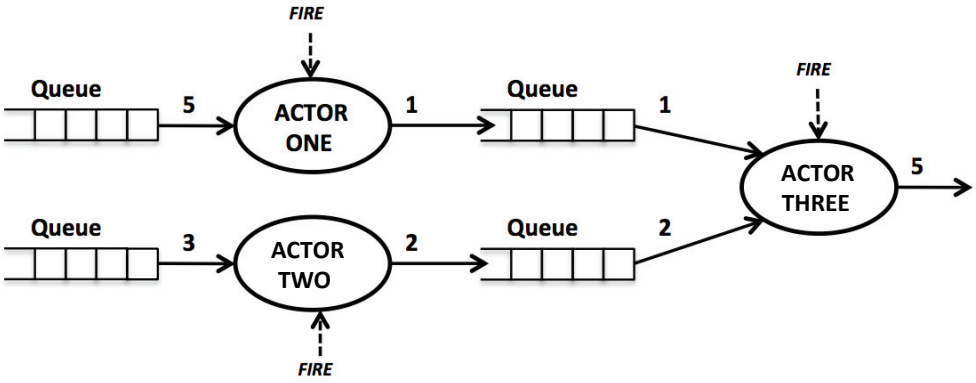


Figure 1.4: SDF system composed of three actors

The firing order of actors ONE and TWO does not matter since actor THREE will not fire until all of the tokens are received in its two input FIFO queues. For every actor

there is a set of rules that determine when an actor will fire. An actor with more than one input stream can have N firing rules. This is expressed as:

$$\mathfrak{R} = \{\boldsymbol{R}_1, \boldsymbol{R}_2, \ldots, \boldsymbol{R}_N\} \quad (1.1)$$

The actor can fire if and only if one or more of the firing rules is satisfied, where each firing rule constitutes a set of *patterns*, one for each of the *p* inputs:

$$\boldsymbol{R}_i = \{R_{i,1}, R_{i,2}, \ldots, R_{i,p}\} \quad (1.2)$$

The pattern $R_{i,j}$ is a (finite) sequence. For firing rule *i* to be satisfied, each pattern $R_{i,j}$ must form a prefix of the sequence of unconsumed tokens at input *j*. The input queues for this platform will be designed to support a maximum of two inputs and $i + x$ tokens for each input, where *x* is the number of entries in the FIFO needed for functional node operation.

Every input to an actor has a queue and the queues in an SDF network are unbounded which means that they will not block tokens from entering the queue [4]. Practically speaking it is not possible to build an unbounded queue due to size and power limitations. The system designer will need to perform a complete system simulation to determine if the depth of the input queues will support all possible scenarios. This design uses a bounded queue that has blocking signals to prevent queue overrun. Figure 1.5 below illustrates the direct mapping of the Actors shown above in Figure 1.4 to three processing elements, each composed of a control block, datapath, and register file. The processing elements are connected via channel nodes that contain the queues.

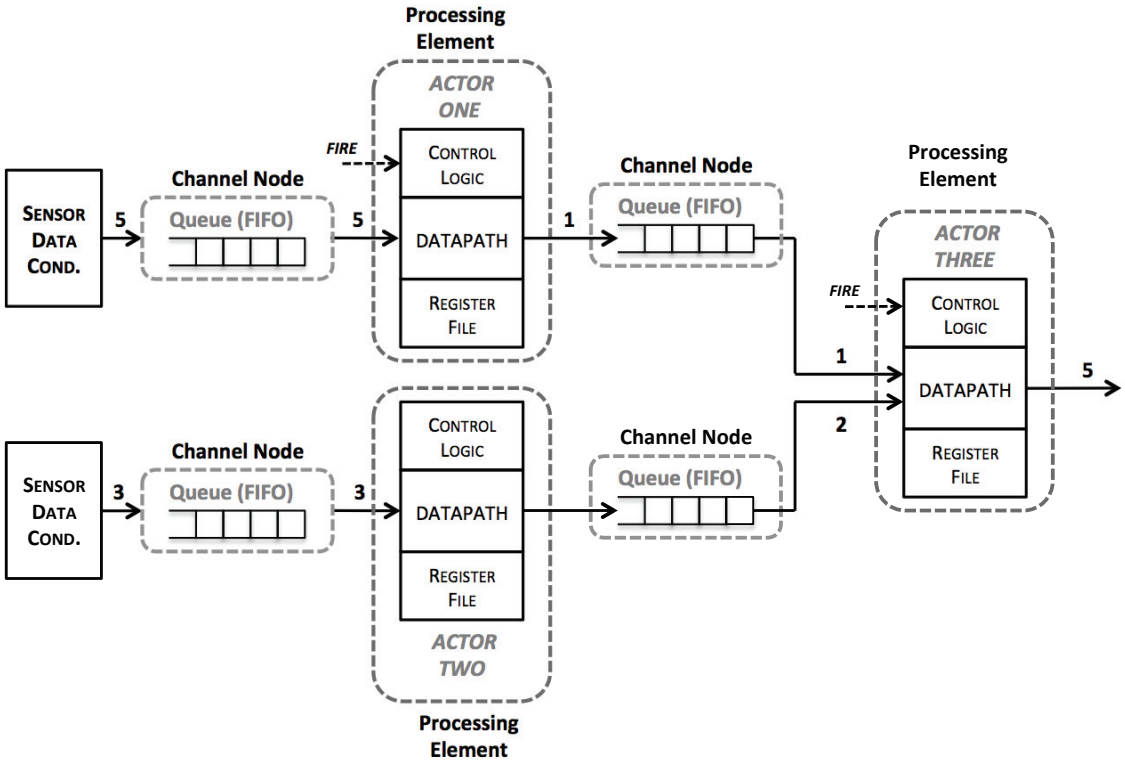


Figure 1.5: Mapping of actors to processing elements

The Dataflow-Processing Element (DPE) that is used in this platform is implemented using a stack-based microarchitecture. A key feature of this micro-architecture is the merger of the SDF input queue with a stack removing the dependency to fetch token data from the queue and moving it to the stack. The merger of the SDF input queue and the stack based register file is shown below in Figure 1.6. The merged element is referred to as a Queued-Stack (QS) and is described in detail in Chapter 5. Note that the channel node has been decomposed into an output FIFO and an input queue so that the stack register file could be integrated into the input queue.

In this example the three Actors are individually mapped to single Dataflow Processing Elements. It is possible to "compose" a system where the Actors are mapped to different DPE configurations. This will be described in the next section.

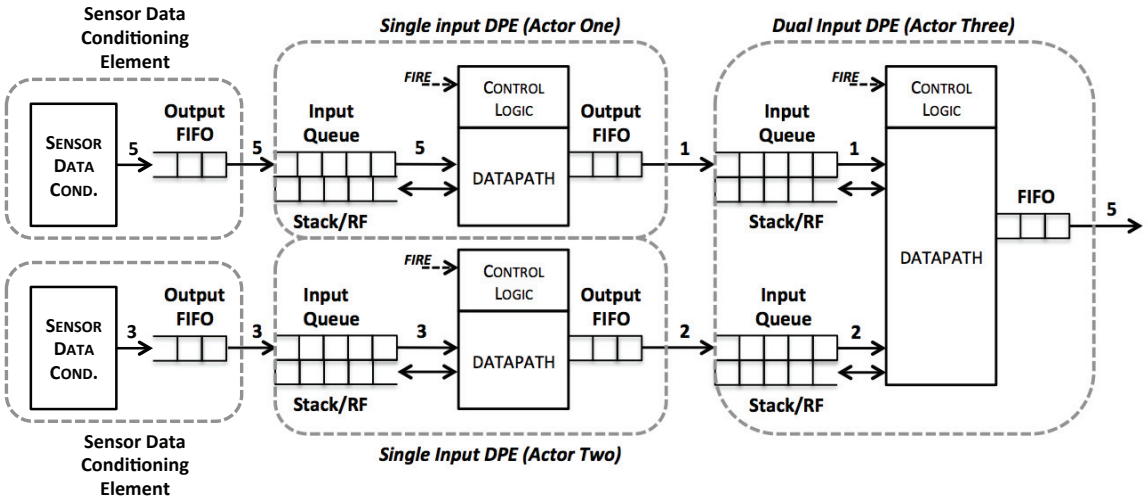


Figure 1.6: Merged input-queue and stack based register file

1.5 System composability

Composability is a key requirement of the sensor platform design presented in this dissertation. It provides the ability to select composable elements and assemble them into various topologies as needed for a specific algorithm. For a component to be composable it must be modular (self-contained) and can be deployed independently. It must also be stateless which means that it treats each request (or firing) as an independent transaction, unrelated to any previous request [5] [6]. The composition rules for this platform are listed below.

1. All inputs to a component will have FIFO queues. Outputs can have FIFO queues to satisfy the need of Rule #2 below.
2. All data propagates through the dataflow network via channels. Note: channel nodes convert data streams as they pass through the network, e.g. serial-parallel, parallel-serial, stream-FLIT, FLIT-stream, etc.
3. For Push-Mode operation, Reads to the FIFO will block, however, Writes will not. For Pull-Mode operation the inverse is true. Note: this platform is designed to support both Push-Mode and Pull-Mode operation.
4. The composed system will be determinate, which requires that each actor is functional and that the set of firing rules are sequential. Functional means that an actor firing lacks side effects and that the output tokens are purely a *function* of the input tokens consumed in that firing.
5. Components can be software routines. Rule #4 states that these routines can be moved to alternate computational engines and execute without modification.

The implementation details of channel nodes and functional nodes as well as Push/Pull modes are described in detail in Chapter 4. Figure 1.7 below shows an example of a composed system where Actors One and Two are in a single DPE. Figure 1.8 below shows a single DPE system where all three Actors are in a single DPE.

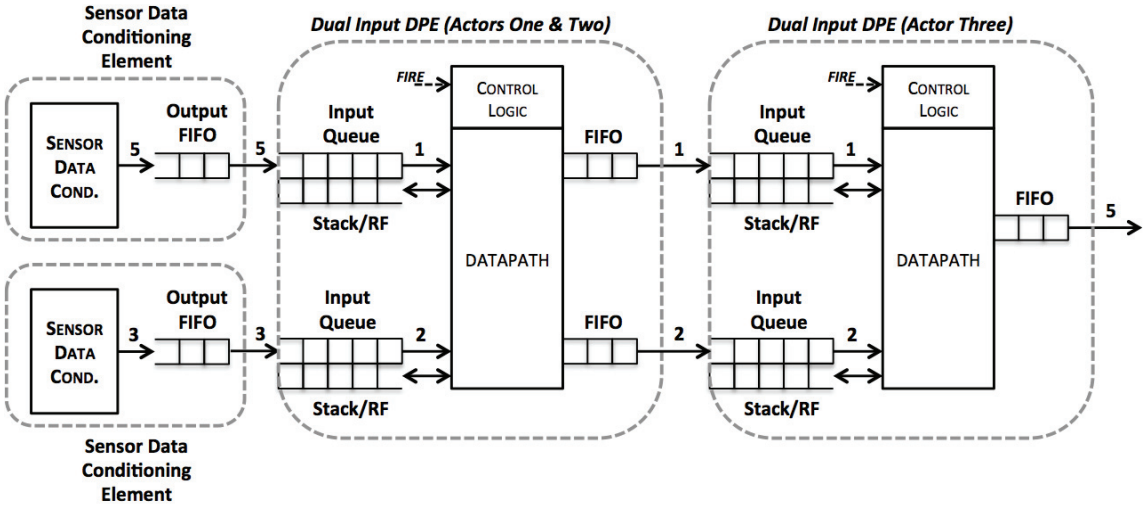


Figure 1.7: Composed system using two dual-input DPEs

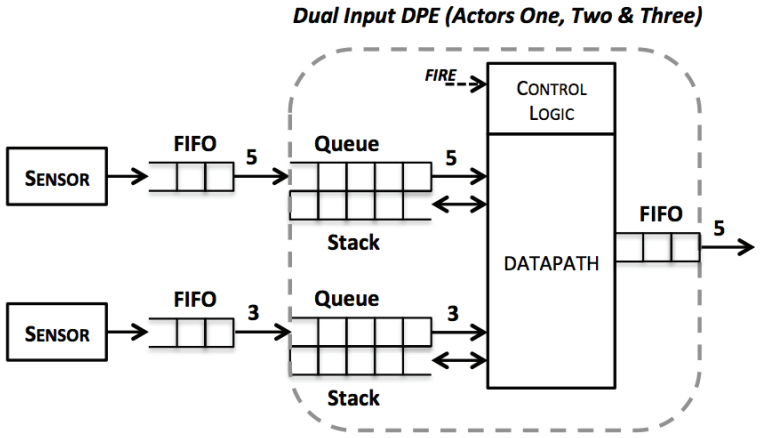


Figure 1.8: Composed system using a single DPE

Figure 1.9 below shows a sensor system topology where the channel nodes are configured as routers. The channel routing node routes tokens through the network in a predefined pattern. The patterns are loaded during system initialization and are generally static. These types of network topologies provide flexibility in building a wide range of

sensor platforms at the expense of increased energy requirements. The design of the channel routing node will be described in more detail in Chapter 4.

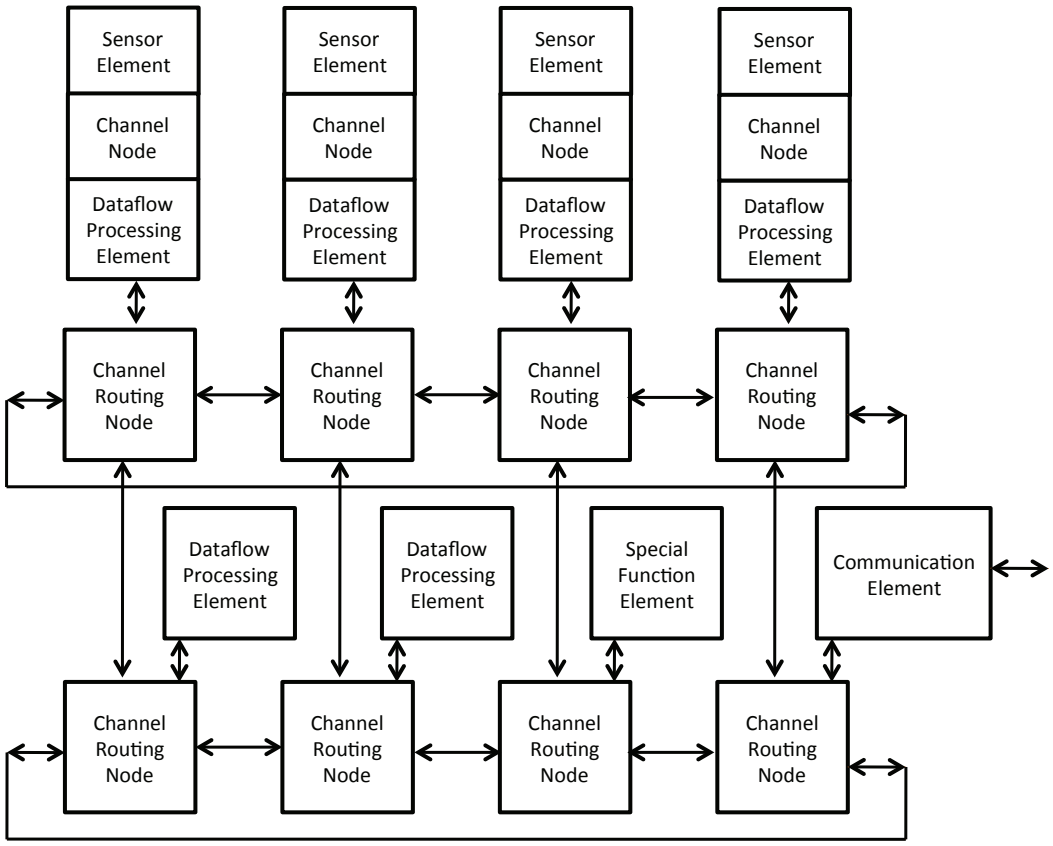


Figure 1.9: Composed system showing channel nodes configured as routers

1.6 Research motivation and contribution

The world is becoming increasing connected via the ever-expanding Internet. The data that populates the Internet is for the most part generated by humans. Most of the data relates to ideas and very little to the things that make up our environment. This has led to the notion that perhaps the Internet should comprehend “things”. This idea is called the

Internet of Things (IoT) and was first introduced by Kevin Ashton in 2009. With the advent of the next generation Internet protocol (IPv6), it is now conceivable to assign a unique address to every "thing" on the planet. In fact the IoT is precisely about doing so. It is easy to envision where there will be between 100-200 addressable sensors per person by 2050. These would be in the form of such things as RFID tags, home automation sensors, territorial/security monitoring, resource monitoring, and health monitoring, etc. This would indicate that there could be in excess of 1.5 trillion sensors connected to the IoT. It is interesting to do an energy analysis of what 1.5 trillion sensors would require:

Assume:

- 1% duty cycle, which is ~ 315 K-Sec per year
- 25μW power per sensor

The Energy/Sensor would be:

$$Energy/Sensor \; = \; 25\mu W \; * \; 315K - Sec \; = \; 7.8\, Joules/Sensor \qquad (1.3)$$

The energy requirements per year for the 1.5 trillion sensors would be 11.8 T-Joules

Assume that 1% of the sensors are powered by single-use batteries and that battery technology in 2050 would provide 7.0 MJ/Kg of specific energy, the result is a massive pile of batteries that must be disposed of every year.

$$(11.8TJ \; / \; 7000KJ \; / \; Kg) \; * \; 1\% \; \approx \; 16{,}875\, Kg\; of\; batteries/year \qquad (1.4)$$

What is not obvious from the analysis is that 25μW of power per sensor system is a very aggressive assumption if transistor technology is the only source of energy improvement. Figure 1.10 below shows the estimated power usage for the wireless sensor nodes that will be presented in the next chapter [8] [9] [10] [11] [12]. Most of the power

usage is in the wireless subsystem, primarily due to the high data transmission rates (10kbps – 100kbps) and the sensor-networking overhead. The idle power is also driven by the wireless subsystem where the nodes are listening for relevant transmission data from other sensors and data aggregators in the network.

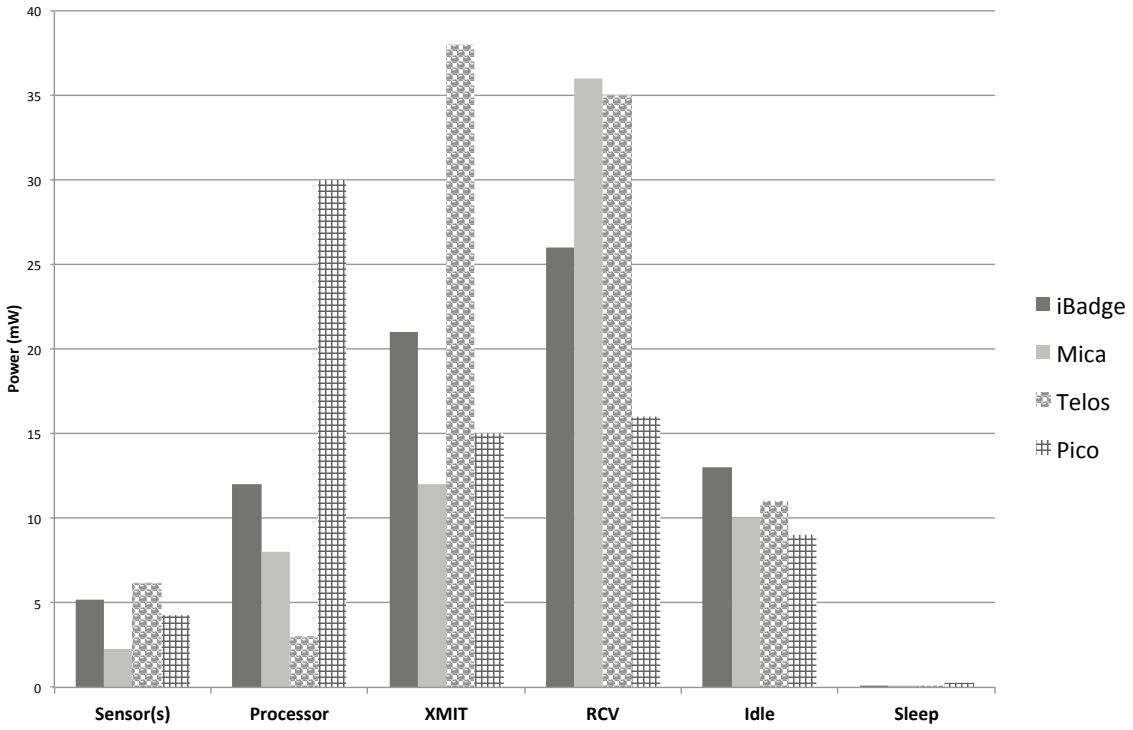


Figure 1.10: Subsystem power usage for various wireless sensor nodes

To save power, the duty cycle for these platforms is usually between 0.1% and 5%. Figure 1.11 shows the power usage for 1% duty cycle operation. The total power is the weighted sum of the power usage for each of the sensor node subsystems. The weighted sum power usage for these platforms is between 550μW and 850μW, which is considerably larger than the 25μW that was assumed above. The wireless subsystem consumes too much power and the optimal solution is to limit the amount of data that

must be transmitted and received. Additionally the idle time power must be limited by reducing the percentage of time that the sensor node listens for commands. The transducer/sensor(s) power will require advances in material science to substantially reduce their power requirements. The processor power and the sleep power are addressed by the processing element described in this dissertation and will come from aggressive optimization of computational efficiency at the system level.

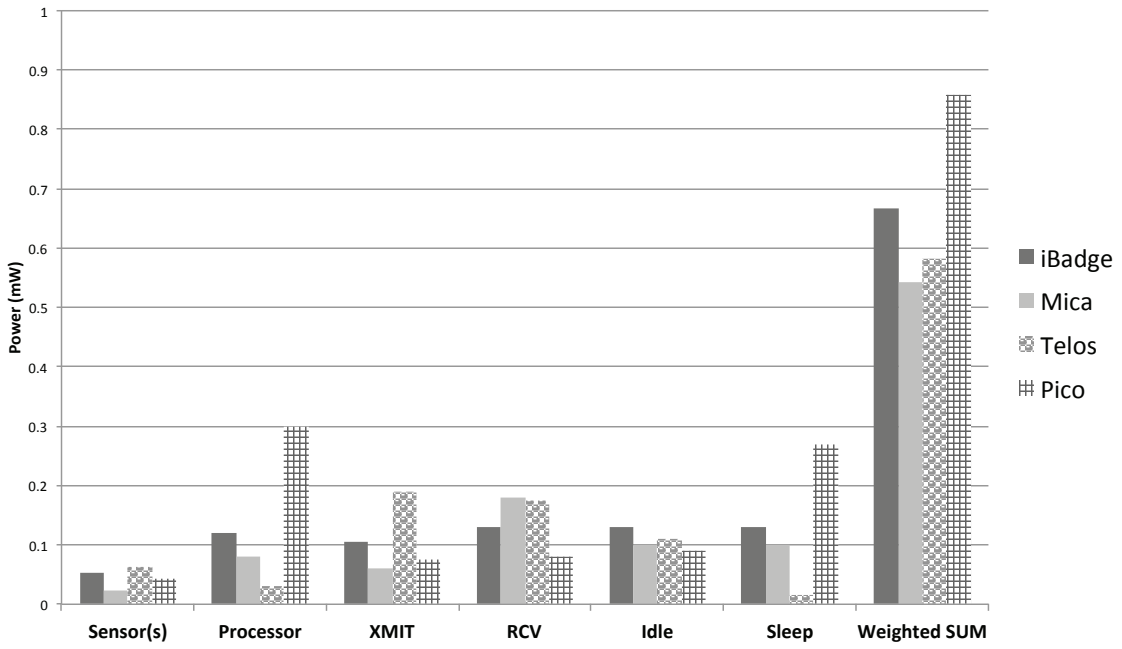


Figure 1.11: Weighted-Sum power usage for various wireless sensor nodes

From a system-level design perspective optimal computational efficiency is achieved by "impedance matching" the four domains that comprise a sensor platform design. The four domains include the algorithmic domain, the software program domain,

the hardware architecture and micro-architecture, and lastly the silicon technology. This is described by the following pseudo-equation:

$$Computational\ Eff\ \propto \frac{\#inst}{Task} * \frac{TP}{\#inst} * \frac{PP}{TP} * \frac{LL}{PP} * \frac{ns}{LL} * Joules \qquad (1.5)$$

where: #inst = number of executed instructions
TP = Trace parallelism
PP = Processor parallelism
LL = Levels of Logic
ns = nanoseconds or 1/frequency

The number of instructions per task is the mapping of the application/algorithmic domain by the software compiler into single or multiple software threads. Ideally the number of software threads is matched to the number of processing elements or hardware threads. The processing element(s) would then be designed to provide the optimal energy-performance in order to accomplish the task as determined by levels of logic needed per clock cycle. It can be seen that solving the equation "as is" results in a value of one Joules-ns/Task, which indicates an ideal impedance match between all components of the equation.

For very high volume applications excellent computational efficiency can be accomplished using an Application Specific Processor (ASP) or an Application Specific Integrated Circuit (ASIC). For low volume applications however, commercial-off-the-shelf (COTS) microprocessors or microcontroller are generally used. These COTS computing elements provide excellent programmability and debug capabilities at the expense of non-optimal energy/computation efficiency. The computational element designed for this platform is considered to be an ASP and is targeted for low-energy

embedded sensor applications specifically those that can be implemented using an event-driven computational model.

The research presented in this dissertation is primarily focused on two areas: 1) the implementation of a computationally efficient processing element that 2) addresses the micro-architectural requirements needed to support an event-driven Cognitive Sensor Platform (CSP). This research has resulted in a computationally efficient processing element that can handle the workloads of deeply embedded sensor platforms that require some level of cognitive capabilities. All domains described above in Equation 1.5 were optimized in the implementation of the processing element. These include the following.

1. *Algorithmic level:* algorithms are modeled in Matlab and verified in Simulink/SimEvents, which can simultaneously model continuous time and discrete time systems. This level of modeling provides the best opportunity to optimize system behavior and energy usage. System parameters such as data precision, token throughput and redundancy can be tuned for a specific sensor application.
2. *Application software and operating system level:* the discrete time components are instantiated using a library of actors which can be implemented either as hard coded logic blocks or as is the case in this implementation using microcoded routines. The requirements for an operating system and corresponding middleware are removed using this mechanism of specifying operations.
3. *Hardware architecture level*: optimal mapping of the software domain to the hardware domain is accomplished by implementing a machine that directly executes actors and follows a Synchronous Dataflow (SDF) protocol. The parallelism specified in the SW domain can be precisely matched in the HW domain using low overhead system composability. The HW implementation is

novel in that it is not a Von-Neumann or Harvard style machine. The merger of the SDF input FIFO's and the stack-based register file (Figure 1.5 above) provides an extremely energy efficient mechanism of managing and consuming tokens. The same structure is used to manage and fetch actors while allowing asynchronous events to be queued, executed and deprecated.

4. *Hardware microarchitecture level*: the ratio of control logic to datapath logic is optimized for maximum power efficiency. Features such as nested looping, microinstruction repeat functionality and conditional execution are implemented with less than a 4% increase in area of the control logic block. One-hot encoding is used throughout the design including the Queued-Stack element, eliminating glitching power caused by decoding logic. Aggressive use of clock gating is possible due the event driven system architecture.
5. *Transistor implementation:* the levels of logic required to implement the HW microarchitecture is matched to algorithmic performance requirements specified at the system level design.

It should be noted that the actual design and implementation of a cognitive sensor platform is not part of this dissertation. During the research into existing sensor platform implementations it became obvious that the classical stored program mechanism (Von-Neumann) was not optimal for deeply embedded low-energy sensors. This drove the subsequent research and development of the computationally efficient processing element presented in this dissertation.

1.7 Dissertation flow

This dissertation focuses on the implementation of a Dataflow-Processing Element (DPE) for a Cognitive Sensor Platform (CSP). This platform is used in embedded applications that are extremely energy sensitive and require a high level of

autonomy. Chapter 2 will survey some key sensor platforms and operating systems that have been developed over the last 15 years. In Chapter 3 the key requirements of the CSP are presented and the system architecture of the CSP is presented in Chapter 4. The microarchitecture of the DPE is described in detail in Chapter 5 and the DPE microprogramming environment will be discussed in Chapter 6. Chapter 7 presents a high-level modeling environment for the CSP using Matlab/Simulink and SimEvents. Chapter 8 presents a performance analysis of the DPE for the FIR and IIR algorithms and the corresponding energy analysis for the two workloads. Future work and conclusions will be presented in Chapters 9. In Appendix C the results of synthesizing, placing, routing and extraction are presented for a 180nm implementation. An identical implementation of the DPE integrated circuit design was prototyped in an FPGA and the prototyping results are presented in Appendix D.

Chapter 2. Survey of Sensor Platform Architectures

2.1 Commercial microprocessors for sensor platforms

There are a number of low power sensor platform architectures that have been developed over the last fifteen years. The very early platforms used commercial-off-the-shelf (COTS) microcontrollers from Motorola and Intel, specifically the 8-bit families such as the 68HC05, 68HC08, and 8051. The more recent platforms used either custom microprocessors or COTS microprocessors from Atmel, Microchip, Texas Instruments and ARM. Table 2.1 below shows a comparison of the various COTS microcontrollers used in the sensor platforms described in this chapter.

Table 2.1: COTS Microcontrollers used in sensor platforms

Manufacturer	Device	SRAM (KB)	FLASH (KB)	Active (mA)	Standby (µA)	Release Date
Atmel	AT90LS8535	0.5	8	5	15	1998
	ATMega 128	4	128	8	20	2001
	ATMega 165/325/645	4	64	2.5	2	2004
	AT91 (ARM-THUMB)	256	1024	38	160	2004
Motorola	HC05	0.5	32	6.6	90	1988
	HC08	2	32	8	100	1993
	HCS08	4	60	6.5	1	2003
Intel	8051 (8-bit)	0.5	32	30	5	1995
	8051 (16-bit)	1	16	45	10	1996
	XSCALE PXA27	256	-	39	574	2004
Phillips	80C51 (16-bit)	2	60	15	3	2000
Microchip	PIC	4	128	2.2	1	2002
Texas Instruments	MSP430F14	2	60	1.5	1	2000
	MSP430F16	10	48	2	1	2004
	MSP430F26	64-128	128-256	.25	.1	2010

The sensor platforms that were built using these COTS processors have additional components such as wireless/infrared transmitters and a wide variety of sensors including: acoustic, seismic, magnetometers, temperature, pressure, light, accelerometers, ultra-sound and location sensing (GPS). Table 2.2 below is a partial list of sensor platforms developed by various university research programs since 1998 [8] [9] [10] [11] [12]. The key aspects of each of these platforms are discussed below. Some of these programs have been completed and are now part of commercialization efforts.

Table 2.2: Survey of first generation sensor platform configurations

Node	Years Active	CPU	Memory	I/O & Sensors	Research Group
iBadge	2000 to 2007	ATMEGA-103L TI TMS320VC5416		Temperature, pressure, humidity, magnetometer, accelerometer, acoustic	UCLA
Medusa MK-II	1999 to 2005	ATMEGA-128L AT91FR4081-ARM THUMB	1MB Flash, 136KB RAM	Ultrasound transceivers to perform high accuracy distance measurements	UCLA
Smart Dust/Motes	1999 to 2008	ATMEGA-128L	4K RAM 128K Flash	See Table 2.3	UC Berkeley & Crossbow
PicoNode	1998 to 2004	STRONG-ARM 1100	4Mb DRAM, 4mB FLASH	Temperature, humidity, light, sound, acceleration, magnetic fields and provisions for GPS.	UC Berkeley Wireless Research Center
µAMPS	1999 to 2004	STRONG-ARM 1100	16Mb RAM, 512KB ROM	Seismic and acoustic	MIT

2.1.1 UCLA iBadge

The iBadge sensor platform was designed for the NSF supported Smart Kindergarten project [8]. The sensor platform was designed to be worn by children to help create a smart problem-solving environment for early childhood education. It was equipped with a microphone and loudspeaker and was capable of capturing and playing back speech and possessed enough capacity to handle complex speech processing algorithms. Figure 2.1 below shows the platform block diagram. There are two processing elements, one for general purpose control processing (Atmel ATMEGA-103L) and one for speech processing (Texas Instruments TMS320VC5416). The platform has a large number of sensors including: acoustic in/out, temperature, pressure, humidity, magnetometer, accelerometer, ultrasound localization, magnetometer and accelerometer. It contains a Bluetooth radio for transmission of sensor data to a central processing system for analysis.

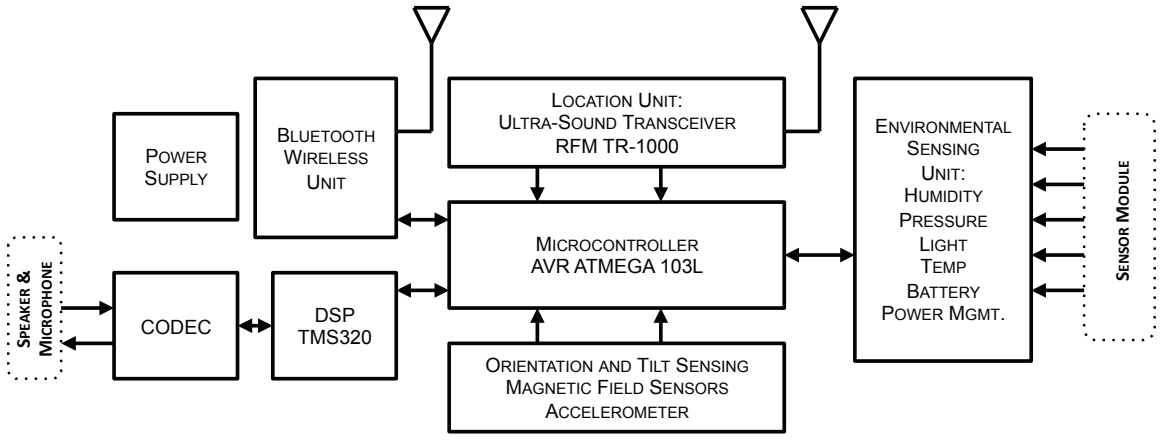


Figure 2.1: Block diagram of iBadge sensor platform

2.1.2 UCLA: Medusa MK-2

The Medusa-2 platform was designed to support research into different sensing technologies and was employed as a ceiling mounted beacon node for the Smart Kindergarten program [9]. The platform was also used for the development of network protocols for low energy embedded sensing environments. Figure 2.2 below shows a block diagram of the MK-2 platform [9]. There are two processing elements. The first one is an Atmel 8-bit 4MHz ATMega128L microcontroller with 32KB of FLASH memory and 4KB of RAM that is used as an interface to the sensors and for radio baseband processing. The second processing element is a 16/32-bit AT91FR4081 ARM THUMB processor also from Atmel and has 136KB of RAM and 1MB of on-chip FLASH memory. The sensing subsystem contains a MEMs accelerometer (ADXL202E from Analog Devices) and a temperature sensor. The platform has a generous amount of I/O channels including: eight 10-bit ADC inputs, serial ports (I^2C, RS-232, RS-485, SPI) and standard general-purpose ports.

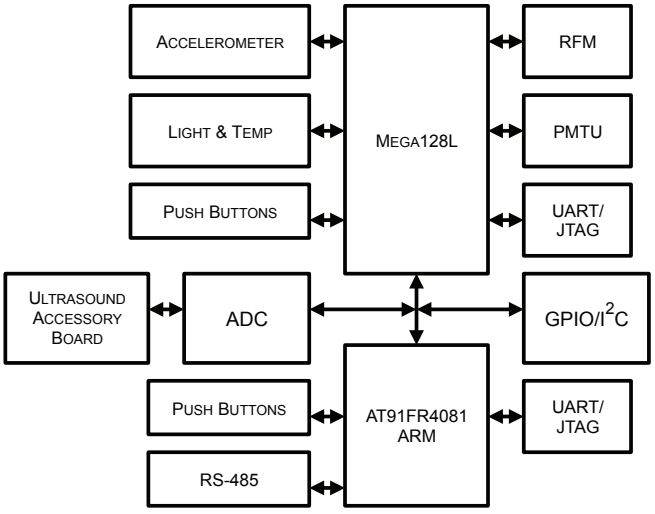


Figure 2.2: Block diagram of the Medusa MK-II platform

2.1.3 UC Berkeley Smart Dust Motes

The first platform of UC Berkeley Smart Dust Motes program was introduced in 1998 with the WeC platform. The Smart Dust program goal was to make a cubic millimeter autonomous sensing device [10]. Early platforms were built with COTS components and referred to as COTS Dust. Table 2.3 below shows the family of Mote platforms [10].

Table 2.3: Family of Berkeley Motes

Mote Type **Years Active**	**WeC** **1998 - 2001**	**René** **1999 - 2002/**	**René2** **2000 - 2004**	**Dot** **2000 - 2004**	**Mica** **2001 - 2005**	**M2Dot** **2002 - 2005**	**Mica 2** **2002 - 2006**	**Telos** **2004 - 2010**
Microcontroller								
Part Number	AT90LS8535		ATmega163		ATmega128			MSP430
Program Memory (KB)	8		16		128			48
RAM (KB)	0.5		1		4			10
Active Power (mW)	15		15		8		33	3
Sleep Power (µW) **Wakeup Time (µS)**	45 1000		45 36		75 180		75 180	15 6
Communications								
Radio part number	TR1000				TR1000	CC1000		CC2420
Data Rate (kbps)	10				40	38.4		250
Modulation Type	OOK				ASK	FSK		O-QPSK
Receive Power (mW)	9				12	29		38
Transmit Power at 0dBm (mW)	36				36	42		35
Power Consumption								
Minimum Operation (Volts) **Total Active Power (mW)**	2.7 24	2.7 24	2.7 24	2.7 24	2.7 27	2.7 44	2.7 89	1.8 41
Expansion and Sensor Interface								
Expansion bus	None	51-pin	51-pin	None	51-pin	19-pin	51.pin	16-pin
Integrated Sensors	No	No	No	Yes	No	No	No	Yes

Figure 2.3 below shows the basic microarchitecture of a Smart Dust Mote. The timers that are driven by the real time clock (RTC) generate events that are registered by the operating system and tasks are queued to handle them.

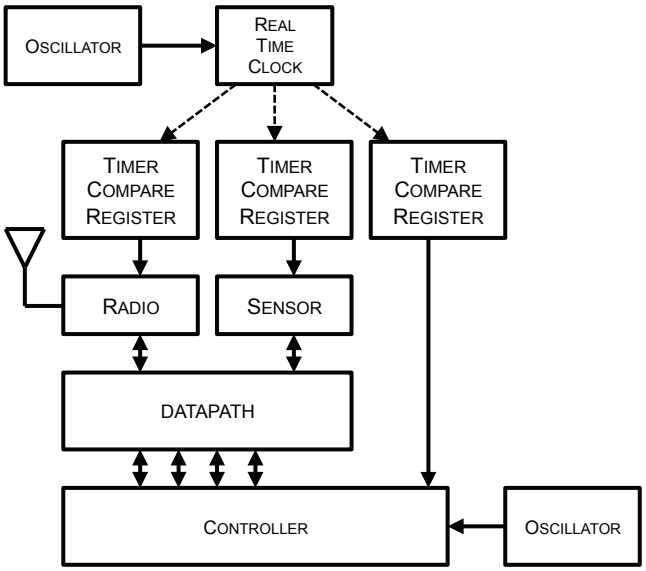


Figure 2.3: Block diagram of a basic Smart Dust Mote System

The WeC platform was built with an 8-bit Atmel AVR-AT90LS8535 microprocessor with about 4 MIPS (million instructions per second) of throughput capability. The platform was used for early testing of wireless communications utilizing an integrated printed circuit board (PCB) antenna. The communications capability and reprogrammable memory provided the ability to update the platform software remotely. The board was approximately the size of a silver dollar and contained temperature and light sensors that were connected via a 51-pin connector.

The René platforms were produced by Crossbow Technologies and were similar in design to the WeC platform. The modular sensor board had temperature and light

sensors and built using stackable boards connected via a 51-pin connector; providing the capability to design application specific sensor boards. The René-2 platform replaced the AT90LS8535 processor with an ATmega128L

The Dot platform was similar to the René platform but removed the 51-pin connector limiting its sensing capability to temperature and light. It was a proof of concept design to demonstrate wireless re-programmability, ad hoc network discovery, routing, and aggregation.

The MICA mote platform was developed in collaboration with Intel Research. It includes an 8-bit Atmel ATMEGA-128L microcontroller, 132K of memory and 512K of nonvolatile FLASH memory. The platform had a 40 Kbps radio operating at 433 MHz or 916 MHz with software programmable frequency hopping for better noise immunity and increased range.

The Telos platform [7] was the first platform to be designed using the TI MSP430 microcontroller. At that time the MSP430 had the lowest power consumption in both sleep and active modes and operated down to 1.8V. This allowed the designers to use two batteries in series and operate down to the 0.9V cut-off voltage for each battery. The MSP430 had the fastest wakeup time from standby to active mode in less than 6μs. The wireless interface used the IEEE 802.15.4 wireless standard operating at 2.4GHz. The antenna was built into the PCB and was tuned to match the radio interface. TinyOS was redesigned for the platform using a three-tier architecture that was independent of the processor or wireless radio used.

2.1.4 UC Berkeley PicoNodes

The PicoNode was designed to provide maximum system flexibility and low energy consumption [11]. It consisted of four main functional modules: computing, sensing, communications and power as shown below in Figure 2.4.

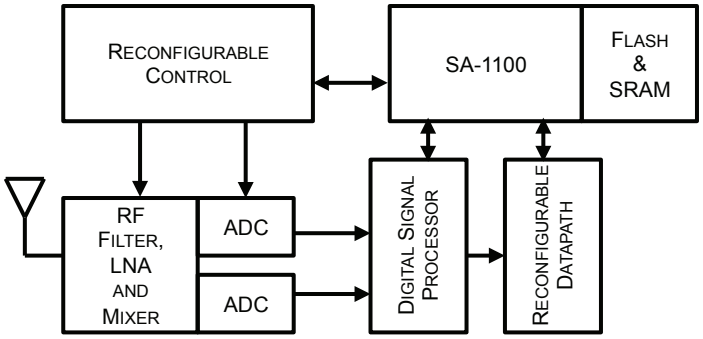


Figure 2.4: Block diagram of the PicoNode platform

The first module contains two computational units including a StrongARM SA-1100 processor with 4MB RAM and 3MB FLASH and a configurable logic unit using a Xilinx XC4020XLA FPGA. The SA-1100 is used for both general purpose computing and for DSP functions. It provides a CPU core and a variety of integrated controllers for services such as standard I/O control and timers. The FPGA is used to emulate tasks that are assigned to configurable or custom logic on the PicoNode. The communication module includes a configurable digital physical layer and a simple direct-down conversion RF front end. The sensor module is customized for each application. These modules are interconnected by a low-energy interconnect scheme. The driving force behind the design of the PicoNode platform was to provide a balance between flexibility (FPGA) and programmability (ARM processor). The designers provided a development infrastructure to support the mapping of algorithms and tasks to the ARM processor, the

FPGA or both. A kernel was developed that provided access to the various resources in the SA-1100 as well as a port abstraction to the FPGA.

2.1.5 MIT's µAMPs Platform

The MIT µAMPS platform utilized the StrongARM SA-1100 processor coupled to a seismic sensor and an acoustic sensor as shown below in Figure 2.5 [12].

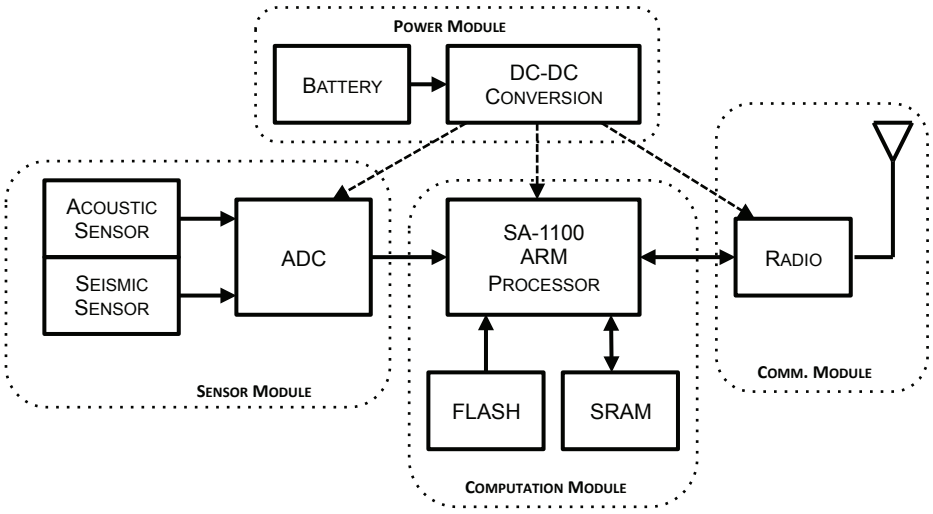


Figure 2.5: Block diagram of the COTS version of the MIT µAMPS platform

Energy efficiency was the primary goal of this platform design. A number of lower power design techniques were used to save energy including: sub-threshold circuits, dynamic voltage and frequency scaling, energy harvesting and adaptive routing of cluster sensor data. The team developed two custom ICs for the platform: 1) an ASIC for the RF module that included an ultra-low power FFT and 2) a low-power A/D converter for the sensor module. The platform was used for acoustic acquisition

applications. Acoustic data was filtered and fused using a beam-forming algorithm to reduce the amount of data that needed to be transmitted to the central collection node.

A variation of the uAMPS was proposed which coupled a DSP with specialized hardware accelerators as shown below in Figure 2.6 [12]. The hardware accelerators provided optimal energy-performance-mm^2 for the application domain that the platform was designed for.

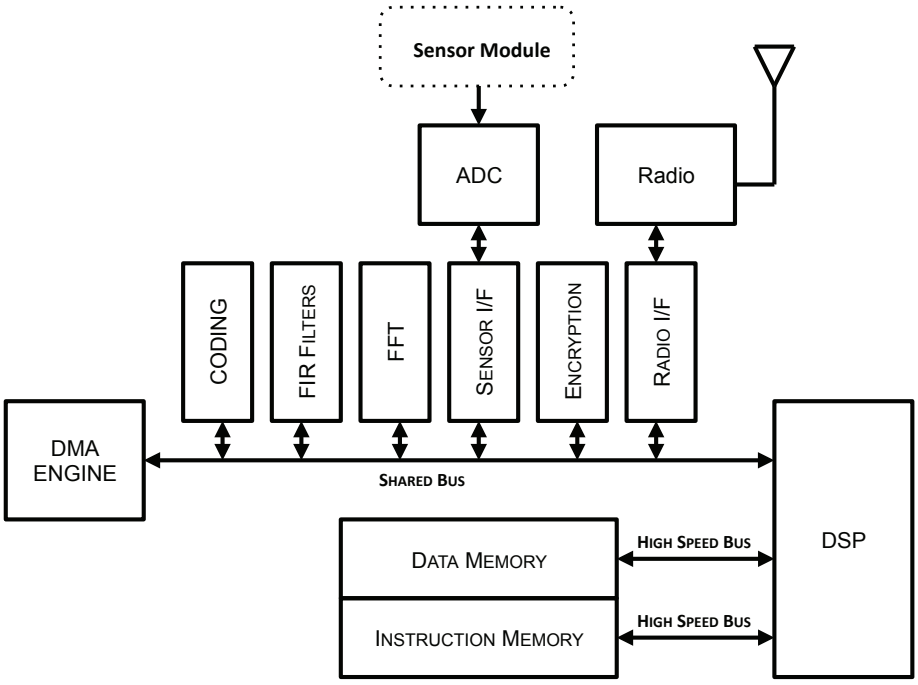


Figure 2.6: Custom implementation of the MIT μAMPS platform

2.2 Custom Microprocessors for Sensor Platforms

A number of custom microprocessor architectures have been developed for use in sensor platforms as shown below in Table 2.4 below [14] [15] [16]. The Pleiades and Subliminal processor development efforts are no longer active, but the SNAP processor from Cornell is still being used for low-power sensor platform designs.

Table 2.4: Custom processors for sensor platforms (2004-2011)

Processor	Arch Style	Data Path Width	Event Driven	Circuit Family	Memory (KB)	Process (nm)	Voltage	MIPS	Energy (pJ/Inst)
Pleiades	DSP	16	N	STD	0.512	600	1.5	14	205
SNAP	RISC+ Accel	16	Y	ASYNC	8	180	1.8 0.6	200 23	218 24
BitSNAP	RISC Bit Serial	16	Y	ASYNC	8	180	1.8 0.6	54 6	152 17
Subliminal	GP	8	Y	Sub-Threshold	0.256	130	~0.360	0.8	2.6
SmartDust	RISC	8	N	STD	3.125	250	1.0	.5	12

These custom processors were designed to improve computational efficiency by utilizing advance circuit techniques such as asynchronous logic and sub-threshold logic, the use of hardware acceleration for specialized tasks, and reconfigurable control/data logic for algorithmic flexibility. The Pleiades, SNAP and Subliminal processor will be discussed in more detail below.

2.2.1 Pleiades Platform

The Pleiades platform from UC Berkeley was designed for voice processing in wireless applications. Figure 2.7 below shows the high-level block diagram of the platform [14]. The processing unit combines an ARM core with 21 satellite processors: two MACs, two ALUs, eight address generators, eight embedded memories, and an embedded low-energy programmable array.

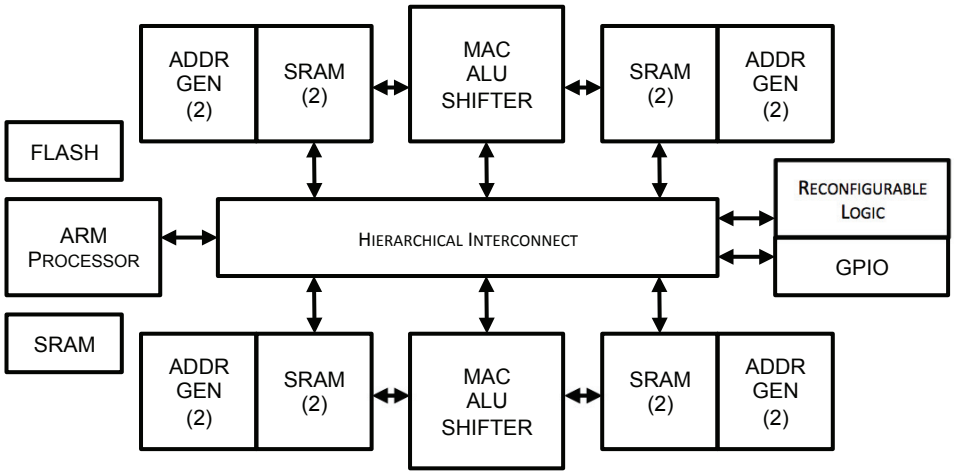


Figure 2.7: Block diagram of the Pleiades Platform

The ARM processor configures the memory-mapped satellites via a hierarchical interconnect block that contains three levels of interconnect hierarchy that superimposed nearest neighbor, mesh and tree architectures for optimal energy usage. The dual-stage pipelined MAC (Multiply-Accumulate including shift, round and saturate functions) and the ALU (Arithmetic Logic Unit) can be configured to handle a wide range of operations. Synchronization between the satellite processors is accomplished by a data-driven communication protocol in accordance with the dataflow nature of the computations performed in the kernel. An interface wrapper is placed around each of the satellite processors to comply with the inter-processor communication protocol. The address generators and embedded memories are distributed to supply multiple parallel data streams to the computational elements. The address generator has a small local instruction memory that can be programmed to support various types of addressing patterns and nested loops using loop counters and stride counters (similar to the BlackFin® DSP from Analog Devices). The programmable logic block is optimized for arithmetic and data-flow control functions.

2.2.2 SNAP

The SNAP (Sensor Network Asynchronous Processor) from Cornell University is based on an asynchronous data-driven 16-bit RISC core [15] as shown below in Figure 2.8. The processor instruction set is optimized for sensor-network applications, with support for event scheduling, pseudo-random number generation, bit-field operations, and radio/sensor interfaces. In addition, the platform has a hardware event queue and event coprocessors, which allow the processor to avoid OS overhead such as task schedulers and external interrupt servicing.

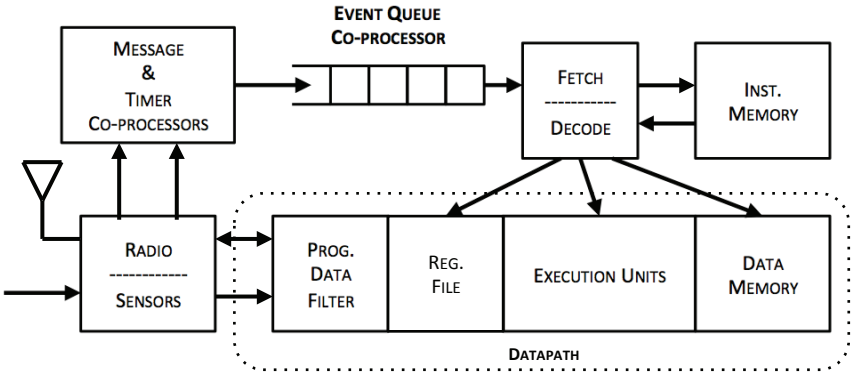


Figure 2.8: Block diagram of the SNAP

The designers of the SNAP processor were concerned with the following system requirements: low-power sleep mode, low overhead wakeup mechanism, low power consumption while awake and a simple programming model. The use of quasi-delay-insensitive (QDI) asynchronous circuits in SNAP resulted in automatic, fine-grained power management, because the circuits that are not required to perform a particular operation do not have any switching activity. Using QDI circuits also eliminates glitches or switching hazards in the processor, removing a key source of energy usage. By default

asynchronous circuits have minimal clocking requirements, which facilitates low-power clock-gated operation. The SNAP processor does not perform continuous program execution, but instead responds to events from the message/timer coprocessor. The event queue coprocessor generates tokens to designate which event handler the processor will execute; resulting in a very rapid wakeup mechanism and very rapid response to system events. The processor executes the sequence of instructions until a done instruction is executed. If the event queue is empty, the processor stalls (in a low power sleep mode) waiting for a new event. The timer coprocessor consists of three self-decrementing timer registers that post a timer event token when the registers reach zero. The message coprocessor serves as the interface between the processor and the sensors or the radio. Each time a byte arrives in the radio, the message coprocessor posts an event to the event queue. The event triggers the processor to execute the appropriate communication instructions to read or write the data byte from the radio unit. The same is true when sensor data arrives. A bit-serial implementation of the SNAP processor was designed to investigate the power savings that can be obtained from minimizing energy needed to switch the wide busses. As can be seen from Table 2.4, the gains are not as much as would be expected. Note that the performance degraded by 75% while the energy/instruction only decreased by 30%.

2.2.3 Subliminal processor

The subliminal processor was designed to run in sub-threshold mode at a supply voltage of 0.36V. It was one of the first processors designed specifically for wireless sensor systems [16]. Figure 2.9 below shows a block diagram of the processor. It is a simple 3-stage pipeline with an 8-bit data path and 12-bit instruction width. The designers optimized the microarchitecture and instruction set for the specific workloads they would

be running. These workloads included an ad-hoc router control algorithm, run length encoded compressor, encryption algorithm, CRC check, FIR filter, binary search and maximum value search.

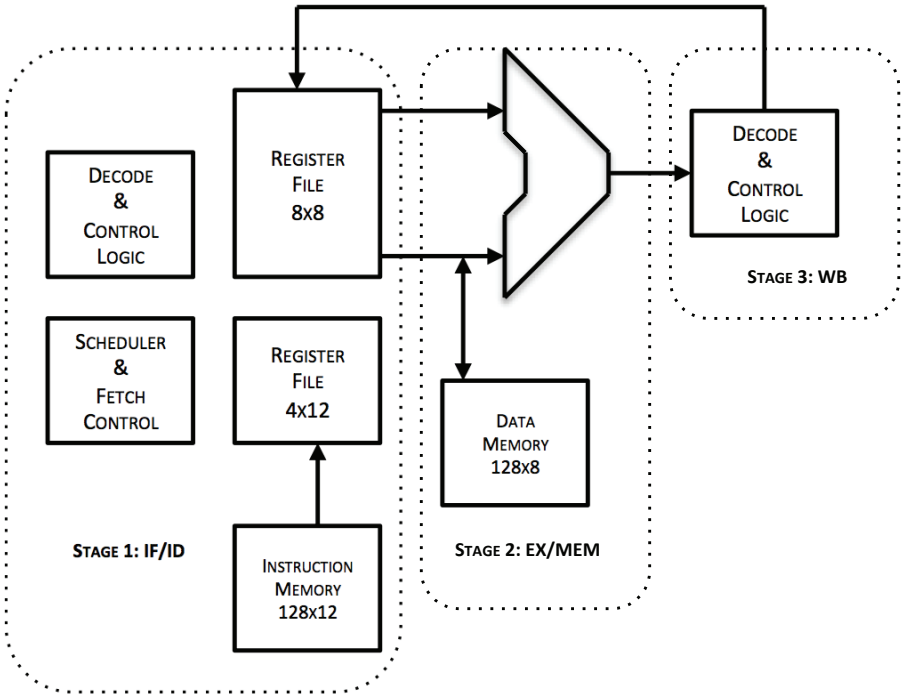


Figure 2.9: Microarchitecture block diagram of the Subliminal Processor

The designers investigated a number of variable-width and fixed-width instruction sets and decided to use a two-operand RISC-like Instruction Set Architecture (ISA) that supported merged micro-operations. In addition, application specific instructions like event scheduling, timer control and pointer manipulation were added to the ISA. Due to energy limitations a simplified Branch-Taken mechanism was implemented. A limited out-of-order execution feature was implemented, which monitored one instruction ahead in the instruction prefetch buffer to determine if it could be fed into the pipeline before the dependent ALU operation.

Three key lessons came out of the implementation of the Subliminal Processor with respect to energy optimization in the subthreshold design domain [16]:

1. Area must be minimized, as it is a critical energy factor due to the increased leakage energy at subthreshold voltages.
2. Transistor utilization must be maximized because effective transistor computation offsets static leakage power, which permits a lower operating voltage and lower overall energy consumption for the design.
3. The clocks per instruction (CPI) must be minimized at the same time; otherwise, gains through small area and high transistor utility are squandered on inefficient computation.

The first key lesson is generally applicable to all energy sensitive designs. Leakage is a strong function of the average width of all devices in the design, which ultimately drives die area. Increasing die area results in added wire capacitance while increasing dynamic power.

The second lesson is related to the first; switching transistors are more effective than leaking transistors. Optimize the use of transistors by eliminating marginally used logic blocks. The ratio of datapath logic to control logic should also be maximized.

The last lesson is key to all chip design; a one percent increase in area should ideally result in a one percent increase in performance. This is described in more detail in Section 8.8.

2.3 Software systems for sensor platforms

Computational efficiency is a function of the application software and the operating system (OS) as described in equation (1). The early platforms used cyclic executives and commercial real time operating systems (RTOS) to control platform operations. These were not well suited to the low energy requirements and limited resources of sensor platforms. To address this obvious mismatch, the researchers at UC Berkeley developed the TinyOS operating system [20] [21], which has become the OS of choice for most COTS based sensor platforms. The researchers at UCLA developed SOS (Sensor Operating System) [18] to fix some shortfalls of TinyOS, specifically the inability to dynamically reconfigure a sensor node once it was deployed. In parallel, the scientists at the Swedish Institute of Computer Science developed Contiki to address the need to dynamically load and unload individual programs and services [19]. Nano-RK was developed at CMU to provide a full preemptive RTOS for multi-hop wireless networks [17].

The key elements of these operating systems include some or all of the following:

- Time management system
  - Measurement resolution
  - Event timer resolution
  - Synchronization in distributed systems
- Networking and Communications
  - Safe data sharing
  - Hazard prevention and deadlock avoidance
  - Bounded channel access and message transmission delays
  - Networking stack support

- Task scheduling
    - Static or dynamic
    - Deterministic
    - Preemptive or non-preemptive using round robin or priority policies.

2.3.1 TinyOS

The Tiny Micro-threading Operating System (TinyOS) was developed at UC Berkeley for the Smart Dust Motes described above [20] [21]. The designers of the operating system were trying to address the needs of a networked sensor system, which included small physical size, lower power consumption, diversity in design and usage, limited computational capability and concurrent intensive operations. The TinyOS concurrency model does not support blocking or spin loops, which is ideal for reactive processing and interfacing with hardware. It also does not define a system/user boundary or a set of system services that need to be part of each compilation. Instead, it provides a framework for defining such boundaries and allows applications to select services needed for a particular user application. Additionally, there is a large set of common services available including timers, data acquisition, power management, and networking.

TinyOS applications are written in nesC, which is a variant of C. The basic units of nesC are components that connect via standard interfaces called ‘wires’ and use configuration tables to specify the connectivity. Modules are components that have variables and executable code. Figure 2.11 below shows basic configuration of a component. A component has four interrelated parts: a set of command handlers, a set of event handlers, an encapsulated fixed-size frame, and a bundle of simple tasks.

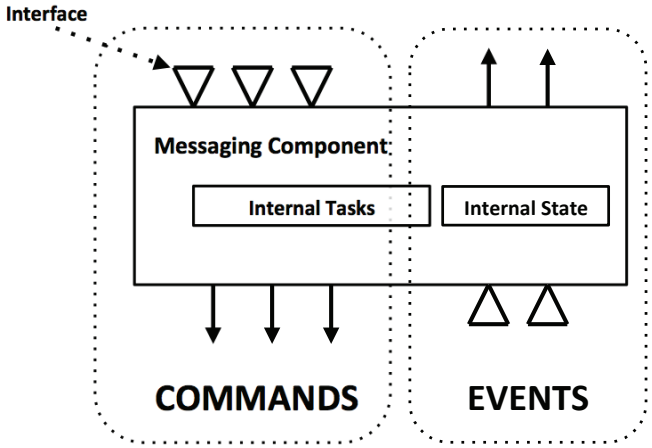


Figure 2.11: TinyOS component configuration.

Commands are non-blocking requests made to lower level components. A command must provide feedback to its caller by returning a status indicating whether it was successful or not. Event handlers are invoked to deal with hardware events, either directly or indirectly. The low level components have handlers directly connected to the hardware interrupts, which may be external interrupts, timer events, or counter events.

Tasks perform the primary work in TinyOS and are atomic with respect to other tasks. The tasks run to completion but can be preempted by events. Tasks can call lower level commands, signal higher-level events and schedule other tasks within a component. The run-to-completion semantics of tasks make it possible to allocate a single stack that is assigned to the currently executing task.

The task scheduler is a simple FIFO scheduler, utilizing a bounded size scheduling data structure. Depending on the requirements of the application, a more sophisticated priority-based or deadline-based structure can be used.

2.3.2 SOS – Sensor Operating System

The Sensor Operating System (SOS) was developed at UCLA in the Network and Embedded Systems Lab (NESL) for use with Atmel based COTS microprocessors [18]. The OS was designed to provide re-configurability of sensor nodes after the network had been deployed. Key to this was the ability to incrementally add, remove or update software components while the sensor is functioning. A secondary goal was to provide a set of application programming interface (API) primitives that aided in the development of wireless sensor networks. The list of features in SOS include:

- Ability to reconfigure individual components of a deployed system, enabling heterogeneous system deployments.
- Easy program development:
  - Programs written using standard C code and compilers.
  - Kernel support for common services such as dynamic memory allocation, simple garbage collection, and priority scheduling.
- Truly modular system development. The modules that are used to create an application remain modular when deployed in the network.
- Debugging support via standard C code debuggers such as GDB.

As with many university developed sensor platforms, SOS is no longer being supported. The designers suggest using Contiki or TinyOS, which have been successfully moved into the public domain and have a large user base of sensor system developers.

2.3.3 Contiki

The Contiki operating system was developed at the Swedish Institute of Computer Science [19]. It is a portable multitasking operating system designed for memory

constrained networked sensor systems. Typical system configurations require a few hundred bytes of SRAM and a few kilobytes of code space. Contiki supports a per-process preemptive multi-threading, inter-process communication mechanism using message passing through events. It also supports IP communication for both IPv4 and IPv6. The IPv6 stack combined with a special purpose MAC (Media Access Controller) provides the ability for battery-operated devices to use IPv6 networking. Contiki supports header compression, IETF RPL IPv6 routing, and the IETF CoAP application layer protocol, among many other protocols and mechanisms.

The kernel uses a lightweight event based scheduler to dispatch events to active processes. All processes are triggered by events or in some instances by polling which implies that the kernel does not preempt the event handler once it has been scheduled. That said the event handlers could use built-in functions to provide preemption capability. The kernel handles both asynchronous and synchronous events. Contiki supports dynamic linking of services at run-time. Services appear as a shared library to the programmer. Typical examples of services include sensor device drivers, protocol stacks and high-level functions such as filter or data fusing algorithms.

2.3.4 Nano-RK

Nano-RK was developed at CMU to provide a fully preemptive RTOS [17]. The design goals for Nano-RK included:

- Small footprint
- Support for multitasking
- Networking stack support
- Support for priority based preemption

- Low-energy operation utilizing resource usage limits

Nano-RK has support for multi-hop networks, which is required for some wireless sensor networks. It currently runs on the Fire-Fly Sensor Networking Platform and on Mica-Z Motes. The lightweight kernel requires approximately 2KB of SRAM and 18KB of FLASH/ROM and provides excellent functionality and timing support. The kernel supports fixed priority preemptive multitasking to ensure task deadlines are met and supports a reservation mechanism for access to sensors, actuators, network and CPU resources. The tasks specify resource demands and the kernel provides timely, guaranteed and controlled access. Figure 2.10 below shows the architecture of the Nano-RK OS [17].

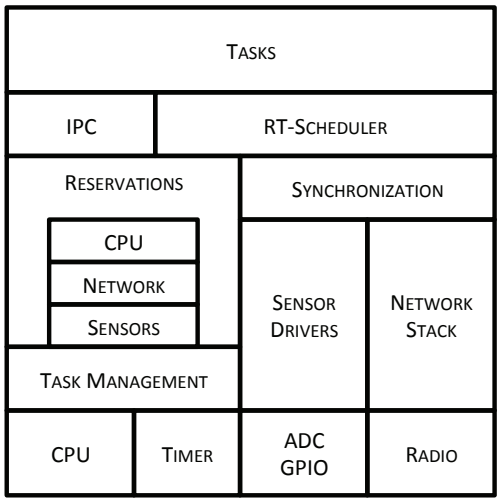


Figure 2.10: Architecture of the Nano-RK operating system

2.3.5 Sensor OS middleware

The primary purpose of OS middleware is to support development, maintenance, deployment, and execution of applications or services. In some cases support for re-programmability and repair-ability is provided by the middleware. There are numerous

implementations of middleware for sensor networks. Many are simply application programming interfaces (API) on top of the sensor OS while others are implemented as virtual machines with a set of domain specific interpreted instructions or support for generic scripting languages. Typical application or services supported by middleware for sensor systems include [22]:

- Abstraction support for multiple platforms, computational elements, communication element, algorithmic accelerators.
- Resource management of computational energy, memory usage, battery power.
- Dynamic reconfiguration of program operation, network configurations, sensor re-calibration, and node/sensor repair.
- Data fusion and filtering of sensor data. Generation of metadata from fused data.
- Domain specific support.
- Programming environment.
- Adaptive processing.
- Scalability — static or dynamic.
- Security.
- Quality and robustness of response time, availability, bandwidth allocation, etc.

A number of middleware packages have been developed for TinyOS including Maté [23], Tiny Lime [24], Tiny Cubus [25], Agilla [26] and TinyDB [27]. Maté is a virtual machine that is implemented on top of TinyOS and is designed to abstract the asynchronous behavior and race conditions of the operating system. It uses a stack-based

architecture and supports multiple contexts. Tiny Lime provides data aggregation and data/feature extraction of collected sensor data. Tiny Cubus provides a cross-layer framework, a configuration engine and small data management framework. Agilla is a mobile agent that allows agents to move from one sensor node to another. It uses a stack-based architecture similar to Maté to reduce code size. TinyDB is a database that is used for collecting data from sensor nodes for message aggregation using a built-in query manager. TinyDDS provides a pluggable framework that allows Wireless Sensor Node (WSN) applications to have fine-grained control over application-level and middleware-level non-functional properties. Figure 2.12 illustrates where middleware fits into the OSI layer model paradigm.

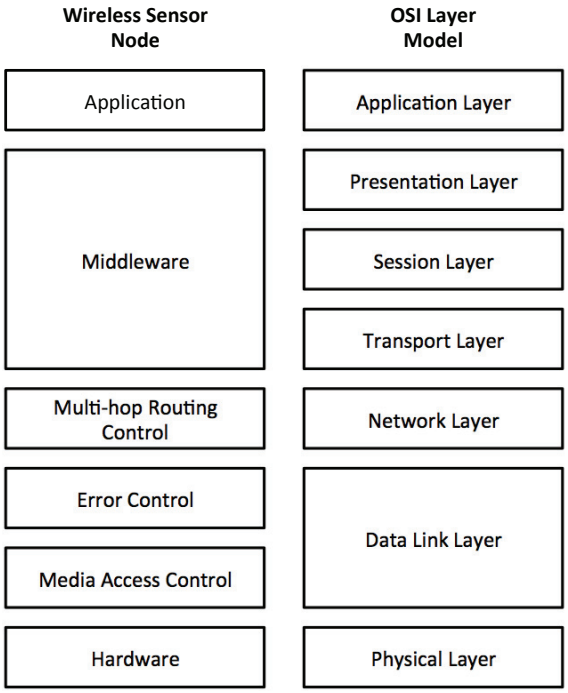


Figure 2.12: Wireless Sensor Middleware vs. OSI Model

2.4 Summary

The sensor platform research described above has spawned the next generation of platforms that are just now appearing in research journals. Many of these new platforms are focused on biological sensing and analysis including electronic noses, electronic tongues, genomics and proteomics. The hyper-integration of mechanical and electronic sensing elements into bulk CMOS processes is producing low-energy, low-cost Labs-on-Chip (LoC). However these systems continue to use classic stored program (Von-Neumann) class of microcontrollers. The platform described in this dissertation is designed to support these new biological sensor platforms as well as low energy embedded sensor platforms using a novel event driven microarchitecture. The concept of an operating system and supporting middleware is not applicable to this platform. Programming is accomplished by instantiating Actors that consume and produce tokens. The microarchitecture is custom designed to directly execute this new "Software" paradigm. Additionally this platform is intended to support the system requirements of a sensor platform that has basic cognitive capabilities. These requirements are described in the next chapter.

Chapter 3. Cognitive Sensor Platform (CSP) Requirements

3.1 Overview

This platform is designed be used in low energy embedded sensor applications including medical implants, structural implants and remote sensing. The key figure of merit (FOM) for this class of embedded sensors is energy-performance/volume where battery volume is the limiting factor as it determines the number of joules available for system operation. The addition of cognitive capabilities is necessary for these types of unattended applications as it is generally not feasible to routinely replace the battery or sensor(s) in these applications. Cognition in the context of a sensor platform is defined as the "process of knowing, including aspects of awareness, perception, reasoning, and judgment" [1]. Figure 3.1 shows conceptually how the process of knowing applies to fault detection and repair in an embedded sensor system that has basic cognitive capabilities.

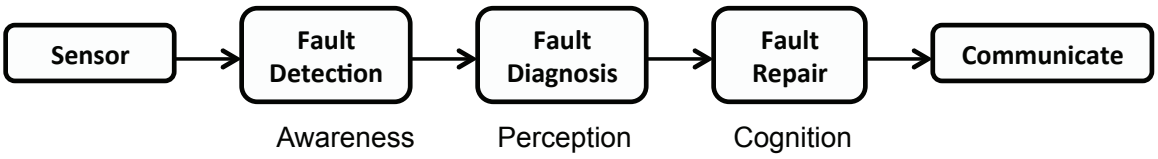


Figure 3.1: Example of fault detection/repair in a cognitive sensor system

This process of knowing drives the following baseline functional requirements of the CSP. These include the ability to:

- Perform self-diagnostics and self-calibration
- Reason about the state of the system and perform needed services to maintain optimal system performance

- Detect and repair corrupted data
- Compensate for systematic errors, system drift and random errors produced by system parametric changes such as sensor aging, battery aging, etc.
- Anticipate potential systematic changes and modify operational behavior
- Transmit/receive information to/from other devices via a standard network protocol

The functional requirements described above drive a number of key architectural features that are described in the following sections. These include the following capabilities:

- Self-diagnosis and self-calibration
- Time Stamping
- Adaptive capabilities including:
  - Configurable data lookup capability
  - Reconfigurable and event-driven programming capability
  - Dynamic sampling and frequency scaling
  - Dynamic data precision
- Fuzzy Logic capability
- Data fusion capability
- Communications capability

3.2 Self-diagnostics and self-calibration

The CSP provides a computational (digital) diagnostic mode that utilizes auxiliary channels to confirm that the primary channels are performing as expected [28]. Injecting calibration tokens into the SDF network and analyzing the response to confirm

computational accuracy accomplishes the diagnostic. The source of the calibration tokens is generally the Debug Unit; however, the Channel Nodes may be designed to inject tokens in response to a control signal from the Debug Unit or scan test unit. A wide range of diagnostics is accomplished using digital tokens including discrete value validation and the temporal response of the SDF network.

In addition to the digital diagnostics, the Debug Unit can inject analog values into the sensor readout circuitry. The values are generally limited to those that are easily implemented using voltage references that are insensitive to aging, power supply variations and temperature changes. In most cases these will be built using some form of a band-gap reference circuit. If the system has an analog actuator such as a Digital-to-Analog Converter (DAC) then the Debug Unit can inject a wide range of analog values into the SDF network.

Calibration of the sensor readout circuitry is accomplished using a programmable content-addressable lookup table (CLT) in the sensor data-conditioning element. The CLT is initialized at reset with the stored calibration data. During system operation, the CLT values can be updated to adapt to environmental changes in the sensor transducer. If the baseline calibration data is located in some form of reprogrammable memory such as FLASH memory, then the updated lookup values can be transferred from the CLT to the memory so that subsequent reset operations load new calibration data into the CLT.

3.3 Time stamping

Time stamping capability is required to synchronize tokens as they propagate through the network. It used by a number different algorithms including token fusion, token quality analysis and repair. The time stamp reference can be generated internally or provided from an external source. In most deeply embedded applications it will be

generated internally which requires that system level synchronization be done externally [45] [46] [47].

A time stamp is a unique value that is tagged to the token data produced by A/D converter and is propagated as sideband tokens. It can also be used externally if it is transmitted along with the data tokens. The time stamping algorithm is described below in Section 4.3.2.1.

3.4 Adaptive capabilities

The CSP detects environmental changes by tracking factors such as rate of change of sensor data, computational data errors, battery voltage, temperature, etc. The CSP can adapt to these changes using pre-defined rules. There are four adaptive capabilities that are required for this platform. These include having a dynamically configurable lookup table, reconfigurable and event-driven programming capability, dynamic sampling, dynamic voltage/frequency scaling and dynamic data precision. Many of these adaptation techniques utilize Fuzzy Logic decision-making [29], which is described below in Section 3.5.

3.4.1 Dynamically reconfigurable data lookup capability

The CSP contains a CLT that can be dynamically reconfigured under program control. Additionally, the Debug Unit can load and unload the CLT contents from external FLASH memory using the scan test unit. The CLT is used to linearize sensor data, hold route tables for a Network-on-Chip (NOC) topology (Figure 1.8) or provide complex logic functions as is typically done in an FPGA [37] [38]. Additionally it is used for Fuzzy Logic operations specifically in the defuzzify operation in which the

antecedents are mapped to deterministic output values. Implementation details for the CLT are described later in Section 5.4.2.2.

3.4.2 Reconfigurable event-driven programming

This feature is required for cognitive systems that need to dynamically modify their algorithms based on operational conditions [39] [40]. This capability in the CSP is achieved by using an Actor/Event queue. The queue is actually a variation of the Queued-Stack introduced in Section 1.3. The relative order of how the commands are processed can be dynamically changed by reentering or reordering the actors in the queue. The actors are entered into the queue in either FIFO or LIFO order. The actors are executed in a first-in first-out order (FIFO). Each actor is mapped to a microcode routine that terminates using a Wait-for-Event instruction. Interrupts and other asynchronous events can also enter commands to the queue by inserting them into the program stream in a similar manner as is done in the SNAP processor [15]. These events are squashed from the Actor/Event queue after they are executed. The Actor/Event queue is circular which allows it to execute continuously until a break condition is encountered. Typically a break condition occurs when new token data is needed.

The Debug Unit can preload the actor-queue with a prescribed sequence of actors via the scan test unit. As the CSP becomes operationally/conditionally aware of its environment the operation-queue will be periodically saved to external FLASH memory so that the new state can be reloaded during the next reset cycle. This is also accomplished via the same scan test unit that is used to preload the operation-queue.

3.4.3 Dynamic sampling and frequency scaling

The CSP can modify the sampling rate of the sensor data if the rate of change of the incoming data is low or high. Additionally, the operating frequency of the CSP can be modified, as needed using the same mechanism. The frequency scaling is accomplished by changing the divider value in the system PLL.

3.4.4 Dynamic data precision

The datapath in the Dataflow-Processing Element (DPE) is designed to use signed saturating arithmetic. The data precision can be dynamically modified to save power by changing the saturation limits and scaling the data tokens as needed. The base configuration is to saturate at +/- 24 bits; however, the DPE datapath can be reconfigured for +/- 8 bits and +/- 16 bits. The impact of this re-configurability is that the power requirements are reduced when the datapath precision is reduced.

3.5 Fuzzy Logic capabilities

The CSP contains a Fuzzy Logic engine to analyze sensor data and make systematic adjustments to the operation of the platform plus provide specialized functions like data fusing (described in Section 3.6). This engine is implemented using a combination of specialized hardware functions and microcode routines. The specialized hardware consists of logic to perform minimum, maximum and table lookup functions. The microcode engine performs the membership, rule evaluation and weighted average functions.

As mentioned above, the CSP can control energy usage by controlling the sampling rate of the sensor data and/or controlling the clock frequency. A Fuzzy Logic

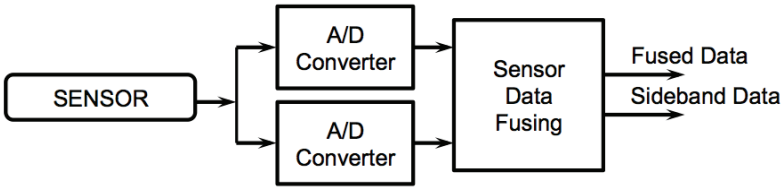
based algorithm is used to determine the sampling rate by analyzing the change and the rate of change of the input data.

3.6 Data fusion

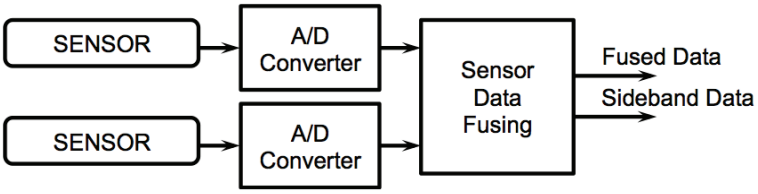
Data fusion is important for data reliability and robustness, data compression and composing complimentary or spectral data [31]. The CSP supports low-energy data fusion using a combination of microcode routines and the Fuzzy Logic engine to perform the fusion operations [32] [33]. This low-energy approach is preferred over mathematically intensive algorithms using least square-based estimation methods such as Kalman Filtering [34] or probabilistic methods such as Bayesian analysis [35]. For those sensor applications that require a more accurate data fusion algorithm, a hybrid of Kalman filters and Fuzzy Logic can be implemented with minimal additional logic [36]. The limitation of this hybrid approach is the limited data precision provided by the DPE and the increased energy usage.

Data from either single or multiple sensors can be fused into a composite data stream. The fused data contains more information than the original inputs and is used either locally in the CSP and/or transmitted to a receiving node for further processing. Figure 3.2 shows single-sensor multi-converter configuration and two multi-sensor configurations. In (a), the same sensor is connected to multiple analog-digital (A/D) converters. The A/D converters can be symmetrically or asymmetrically configured. Asymmetrical operation provides the ability to analyze different characteristics of the same sensor. Symmetrical operation provides redundant conversion data. In (b), there are multiple transducers and A/D converters. The sensors will generally be measuring different physical effects and the fused data is a composite of the two, however, the two sensors can be identical providing redundancy at the sensor and conversion level.

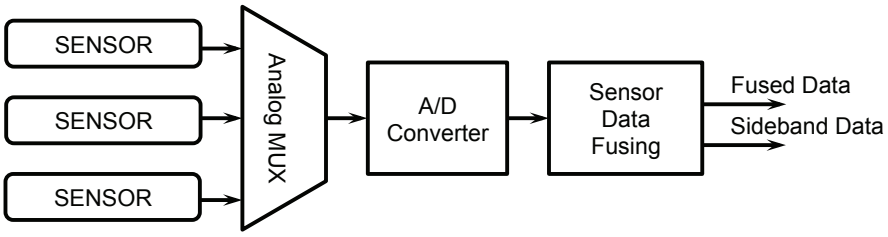
Alternatively in (c), multiple transducers can be time multiplexed into a single A/D converter to save area and energy while providing redundancy at the sensor level.



(a) Single transducer, multiple A/D converters



(b) Multiple transducers and A/D converters



(c) Multiple transducers and a single A/D converter

Figure 3.2: Multi-sensor configurations

3.6.1 Fuzzy Logic data fusion algorithm

The Fuzzy Logic data fusion algorithm involves aggregating data from the input sensors and utilizing predictive data from past aggregation to generate fused data and optional sideband data as shown below in Figure 3.3 [32].

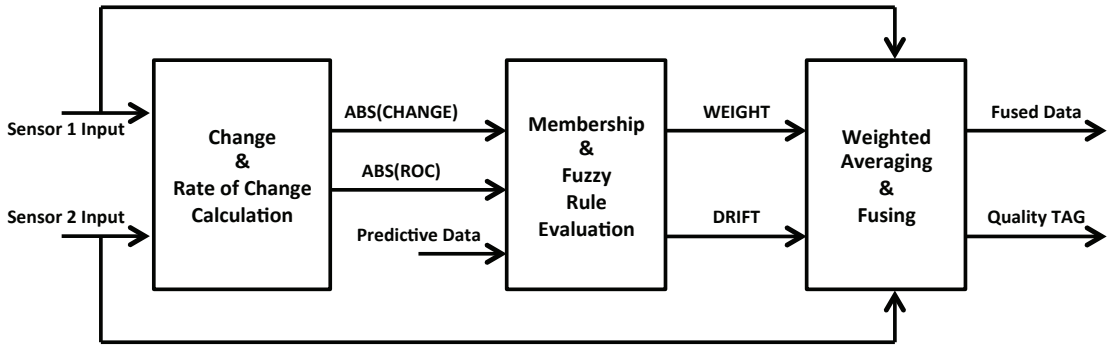


Figure 3.3: Flow diagram of Fuzzy Logic data fusion

In this particular example, the data from two asymmetrical sensors are fused together to produce a composite signal that has the key characteristics from each sensor. The first operation involves determining the absolute value of the change and rate of change (ROC) for the two sensors. The second operation performs a membership evaluation and series of fuzzy rule evaluations to produce a weighting factor and a sensor drift value. The third operation uses the weighting and sensor drift information to produce the fused data and a quality tag. The quality tag is sideband data that can be used by the CSP to adapt to sensor drift, data sampling issues, etc.

The fuzzy rule evaluation uses a series of antecedents and consequences to infer what the weighted value should be. A typical set of rules (where S1 and S2 are sensor inputs) would be expressed as:

```
IF ABSOLUTE (S1-S2) IS SMALL AND ABSOLUTE (DS2/DT) IS SMALL
THEN: WEIGHT SHOULD BE SMALL

IF ABSOLUTE (S1-S2) IS SMALL AND ABSOLUTE (DS2/DT) IS LARGE
THEN: WEIGHT SHOULD BE LARGE

IF ABSOLUTE (S1-S2) IS LARGE AND ABSOLUTE (DS2/DT) IS SMALL
THEN: WEIGHT SHOULD BE VERY SMALL

IF ABSOLUTE (S1-S2) IS LARGE AND ABSOLUTE (DS2/DT) IS LARGE
THEN: WEIGHT SHOULD BE VERY LARGE
```

Additionally, the drift value would be calculated using the following rule:

```
IF ABSOLUTE (S1-S2) IS SMALL AND ABSOLUTE (DS2/DT) IS LARGE
THEN: DRIFT SHOULD BE LARGE
```

The weighted averaging and fusing operation utilizes a combination of algorithmic calculations and table lookup to generate the fused data. The CLT is used for linearization, compensation and interpolation of sensor input data when performing the fusing operation. Additionally, the CLT can be used to defuzzify the results from the fuzzy rule evaluation. Appendix A describes the Fuzzy Logic techniques used by the CSP.

3.7 Communications capability

The basic communication protocol for the CSP is based on the IEEE-1451 standard. It describes a set of open, common, network-independent communication interfaces for connecting transducers (sensors or actuators) to receiving devices. This protocol is useful for both wired and wireless systems. It is not practical for low energy embedded applications where battery volume is minimal. Low energy communication protocols such as ANT™ [41] should be investigated as an alternative for future implementations of the CSP. The ANT™ documentation states:

ANT™ is a practical wireless sensor network protocol running in the 2.4 GHz ISM band. Designed for ultra-low power, ease of use, efficiency and scalability, ANT

easily handles peer-to-peer, star, connected star, tree and fixed mesh topologies. ANT provides reliable data communications; flexible and adaptive network operation and cross-talk immunity. ANT protocol stack is extremely compact, requiring minimal microcontroller resources and considerably reduces system costs.

3.8 Summary

The requirements specified above pertain to a class of sensor platforms that are generally battery powered low-energy autonomous systems. The addition of basic cognitive capabilities extends the operational life and functional utility of these sensor platforms. Adding additional features will impact the energy-performance characteristics of the CSP and must be considered carefully.

Chapter 4. Cognitive Sensor Platform (CSP) Architecture

4.1 Overview

The CSP is an event driven Synchronous Dataflow architecture. The system is composed by instantiating functional elements that are connected via channels. The functional elements provide key operational services commonly called actors in dataflow systems. In the current implementation the channels are modeled as bounded FIFOs (described above in Section 1.3). The datum that is communicated via the channel interface is referred to as a token. The current implementation of the CSP uses a Pull-Mode channel interface protocol to compose the functional elements. The Pull-Mode protocol adheres to the rules outlined in Section 1.4. The channel signaling protocol is described below in Section 4.3.3.

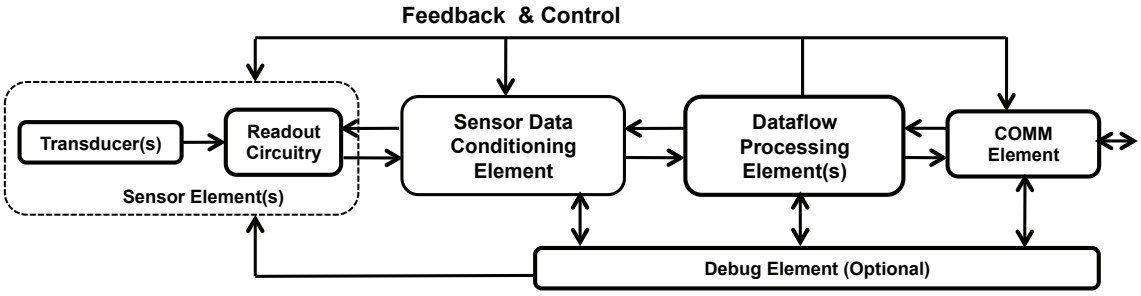


Figure 4.1: CSP high level block diagram

Figure 4.1 above shows the five basic functional elements that are used to compose a CSP:

- Sensor Element – Transducer and Readout Circuitry (Section 4.2)
- Sensor Data Conditioning Element (Section 4.3)
- Dataflow-Processing Element (Section 4.4)

- Communications Element (Section 4.5)
- Debug Element (Section 4.6)

The output from the readout circuitry in the Sensor Element will generally be an analog signal that will require some additional analog processing such as filtering, amplification and conversion to a digital representation using an analog-to-digital converter (ADC). This additional processing is done in a preprocessing unit in the SDC element. The CSP will have one or more DPEs to process the data from the SDC and communicate the output data via the COM element to a receiving device. In addition to these four elements, an optional debug element can be used to debug functional failures and reconfigure the CSP during normal operations.

4.2 Sensor element

This platform is designed to support a reasonably wide variety of sensing techniques including, voltage, resistive, capacitive, inductive, optical, magnetic, force and acceleration. Typical transducers would include strain gauges, piezoelectric devices, phototransistors, hall-effect devices, thermo-couples, ion-sensitive transistors, capacitive displacement devices, and bio-sensing devices. Figures 4.2, 4.3 and 4.4 are examples of capacitive displacement devices [42], ion-sensitive field effect (FET) transistor [43], and electrochemical/photovoltaic bio-sensing devices [44].

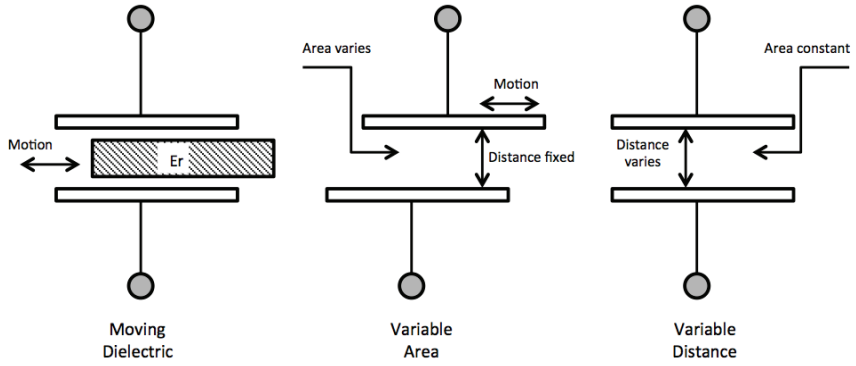


Figure 4.2 Various capacitive displacement transducers

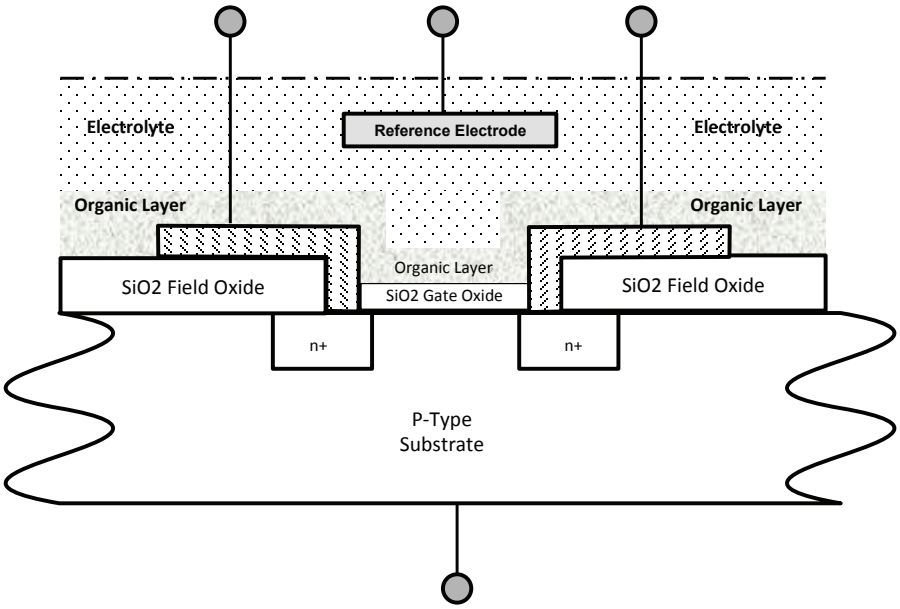


Figure 4.3: Ion-Sensitive FET transducer

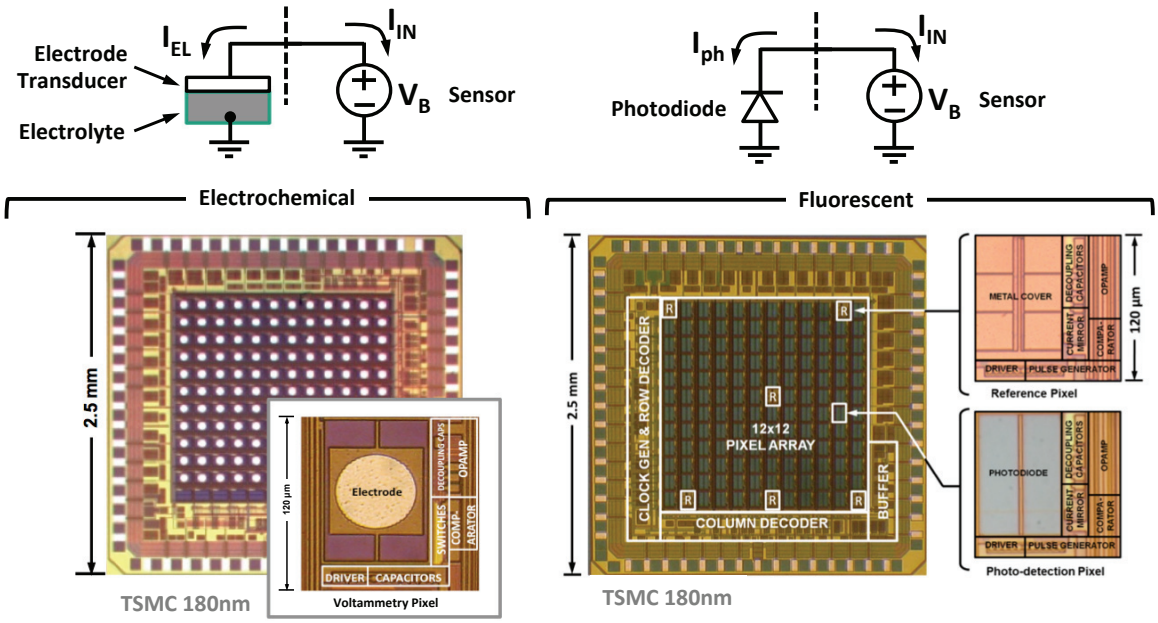


Figure 4.4: Bio-sensing transducers

4.3 Sensor Data Conditioning (SDC) Element

The sensor data-conditioning element provides a broad range of data conditioning and transformation services. These services include:

- Signal conditioning
- Signal conversion
- Detection functions
- Data reduction
- Data fusion

The high-level block diagram for the SDC element is shown below in Figure 4.5 and contains three basic units:

1. Preprocessing unit (PPU) - includes filters, A/D converters, etc.

2. Functional services unit (FSU) - performs data conditioning services.

3. Channel node

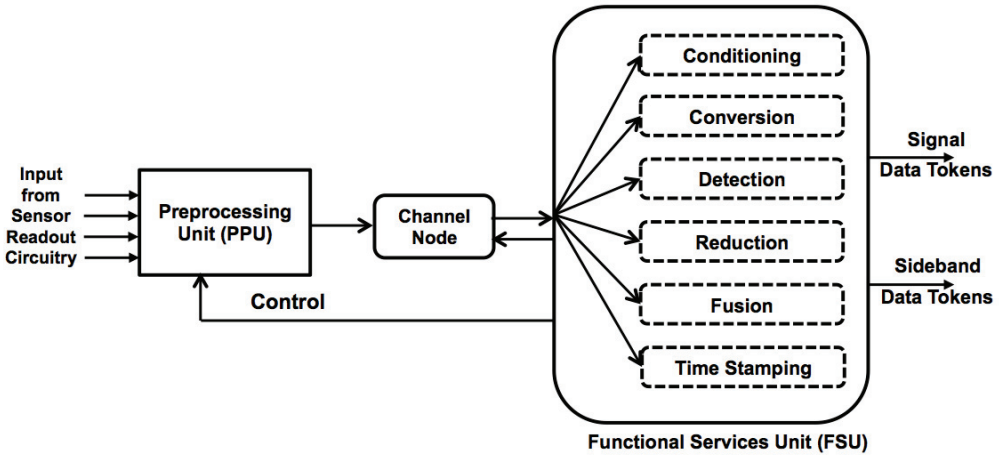


Figure 4.5: Sensor Data Conditioning (SDC) Element block diagram

The basic architecture of each unit is described below in sub-sections 4.3.1, 4.3.2 and 4.3.3.

4.3.1 Preprocessing Unit (PPU)

A typical preprocessing unit contains some combination of the following components: filters, amplifiers, analog-to-digital converters (ADC), sample-hold circuits and analog multiplexors as shown in Figure 4.6.

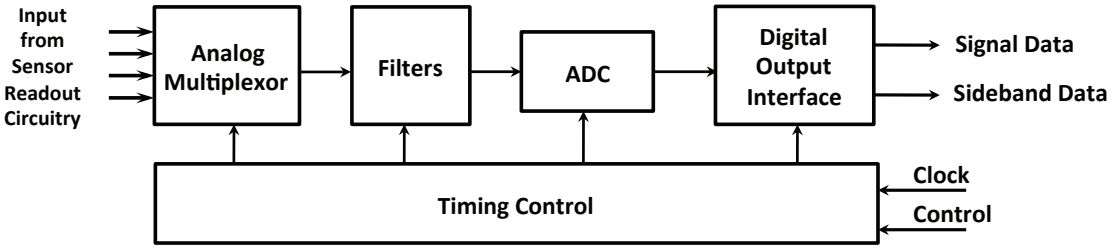


Figure 4.6: Typical configuration of a PPU

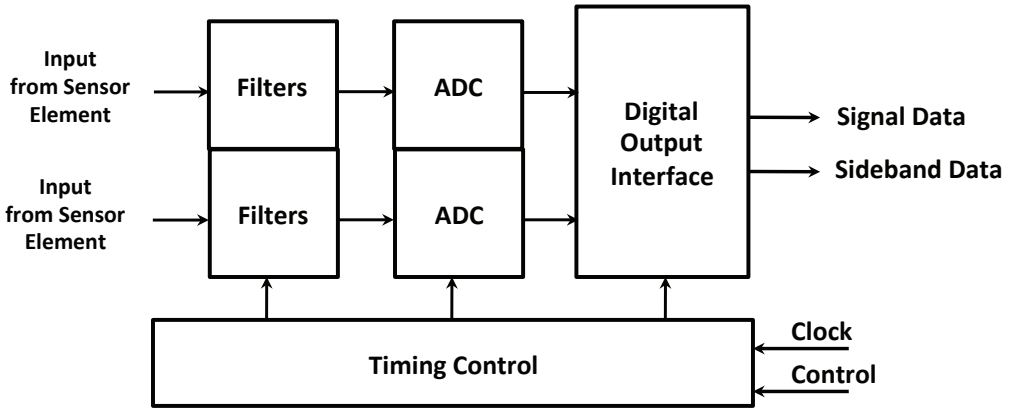


Figure 4.7: High performance configuration of a PPU

In a low cost, low power system configuration the outputs from multiple sensors are multiplexed through one ADC, however, in time-critical applications multiple ADCs can be used as shown above in Figure 4.7. The digital and analog outputs of the PPU may be time division multiplexed or output in parallel from the PPU.

The timing control block will generally get input from the Functional Services Unit (FSU) or from the Dataflow-Processing Element (DPE). The control block in the PPU performs three basic functions:

1. Provide clocks to the multiplexor and ADC
2. Latch valid data in the digital interface logic
3. Perform reset and calibration functions

The FSU has the capability to modulate the frequency of the PPU based on the analysis of the data-flow characteristics of the system. For example, slow-changing signals to FSU may result in a lower sampling frequency in the PPU in order to conserve

power. The FSU can compensate for the lower frequency of operation by modulating the width and period of the time stamp window.

4.3.2 Functional Services Unit (FSU)

The FSU can be implemented using a synthesized hard-coded logic unit or with a microcoded engine such as the DPE. The synthesized implementation is preferred for basic services including:

- Averaging
- Data compression
- Transition counting
- Event triggering
- Threshold detection

The microcoded engine is best suited for the complex services listed below, as they typically require detailed processing of temporal data:

- Linearization and smoothing
- Edge detection
- Data suppression
- Data fusion
- Filtering
- Signal reproduction

Figure 4.8 below shows the microcoded implementation of the FSU. The architecture of the FSU is similar to the Dataflow-Processing Element (DPE), which will

be described in more detail in Chapter 5. The primary difference between the two is addition of the timing reference and two output channels – one for data tokens and the other for sideband data tokens.

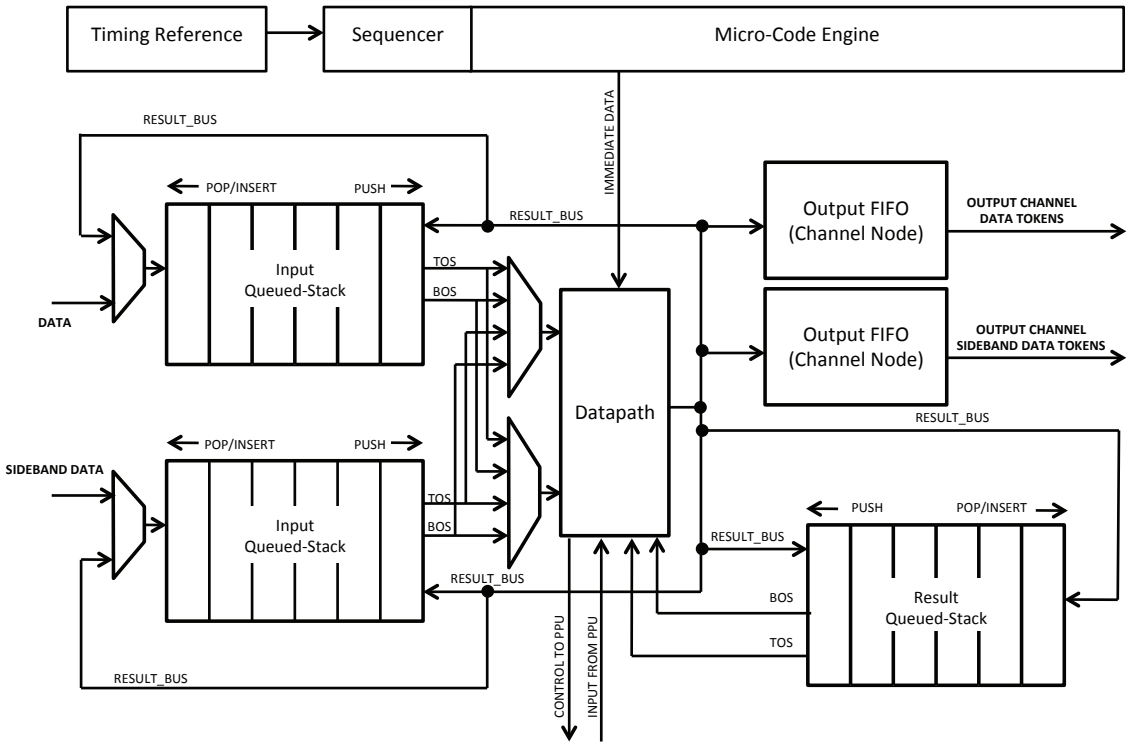


Figure 4.8: Block diagram of a microcoded FSU

In addition to the data conditioning services listed above, the FSU is used for the following special functions:

- Time stamping
- Energy management
- Signal quality analysis

These special functions are described below in the following sections.

4.3.2.1 Time stamping

The time stamp function uses a timing reference to generate a unique number that is tagged to the data from the PPU as it enters the input queue in the FSU. This timing reference may originate internally in the CSP or be derived from an external network synchronization signal [45]. If the timing reference is derived internally the external receiver must synchronize to the internal timing reference by algorithmic means [46] [47]. Depending on the mode of operation, the FSU will check to see if multiple samples are to be acquired before the time stamp value is incremented. The time stamping flow is described in the flow chart shown below in Figure 4.9.

The FSU waits for valid data from the pre-processing unit (PPU) and then tags the data with the current time stamp value based on the number of samples that will be used. The number of samples is determined by the operation that the CSP is intended to perform. For example, if five samples are being averaged to a single datum then the timing window is valid for five samples and a single time stamp value is issued. Note that the time stamp value is incremented for every sample and one of the five time stamp values is used to tag the data depending on the algorithm being performed. In this example, the third time stamp value could be used when averaging five samples. This technique provides a built in time reference for all data that is being processed by the FSU and the DPE.

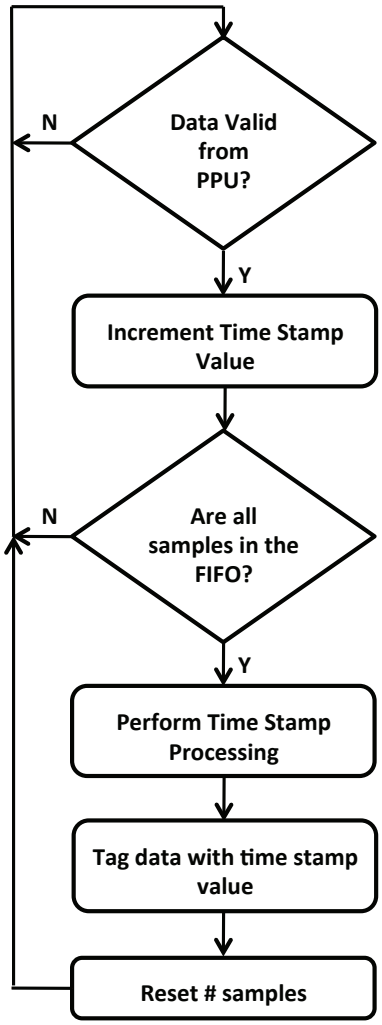


Figure 4.9: Time stamping flow chart

The number of unique time stamp values needed is determined during system composition. It is a function of the depth of the computational latency (CL) through the CSP and is measured from the input to the PPU to the output of the communication element (COM). It is deterministic if the CSP does not dynamically modify its operating conditions. Non-deterministic operation requires that the CSP re-characterize the CL value.

The CL value is determined by the following equation:

$$\mathrm{CL} = \sum_{0}^{\mathrm{N}} (\#\mathrm{clks}(\mathrm{actor}(\mathrm{N})) + \#\mathrm{clks}(\mathrm{COM}) + \#\mathrm{clks}(\mathrm{PPU}) \quad (4.1)$$

Where:

N == Total number of actors needed to process a token from the output of the PPU to the input of the communications element.

#clks(COM) == Total number of clocks needed to communicate the token from the PPU

#clks(PPU) == Total number of clocks to process the data from the output of the sensor readout circuitry.

The CL is the total number of system clocks needed to process one token, assuming that the token propagates through the FSU and DPE as a single token and is not fused, average, filtered, etc. This number is used to seed the maximum time stamp value. This ensures that there is a unique time stamp value for every token in the CSP. The time stamp value is transmitted with an output token and is used to externally synchronize data from multiple CSPs.

4.3.2.2 Energy management

The CSP controls energy usage by modifying the sampling rate of the sensor data and/or the clocks to the DPE. A Fuzzy Logic based algorithm is used to determine the sampling rate by analyzing the change and the rate of change of the input data. Figure 4.10 below shows a flow diagram for the Fuzzy Logic engine [30]. The first step

performs a membership evaluation on the inputs. There are two inputs; one is the absolute value of the sensor data change and the second input being the rate of change of data change. The second step performs the evaluation of the rules that determine the energy levels. The third step converts the energy levels into sampling rates for the Sensor Element and SDC element.

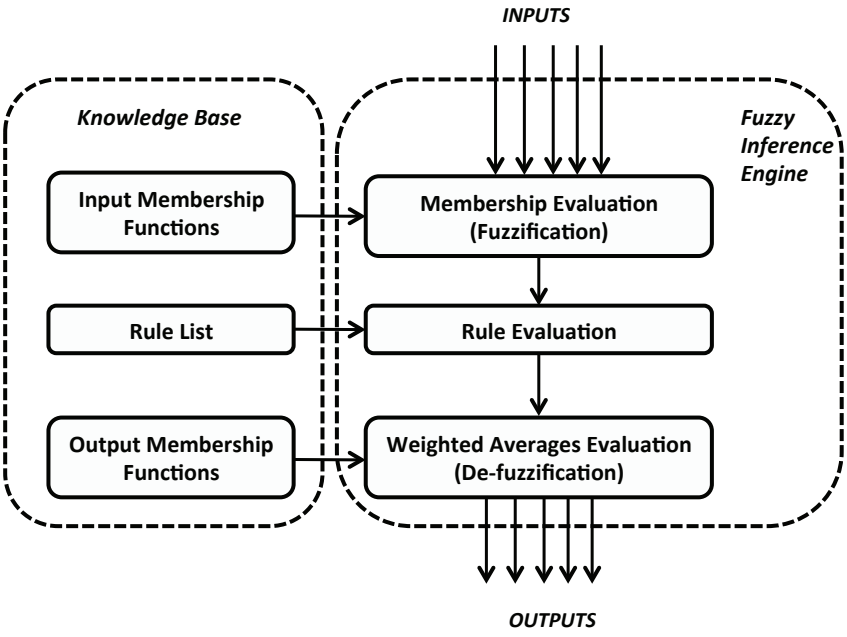


Figure 4.10: Fuzzy Logic flow diagram

Figure 4.11 below shows a table of the potential outputs from the rule evaluation of the energy function. The lower energy operations occur when the input data is not changing very rapidly and the 'rate of change of the change' is also not varying rapidly. Conversely if the data is changing rapidly a higher energy is required to process the data.

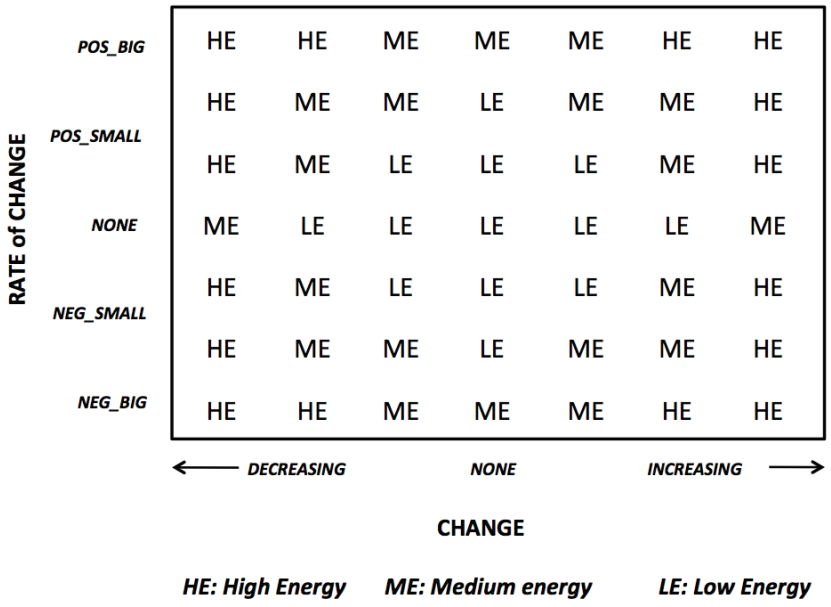


Figure 4.11: Energy usage rule evaluation table

Figure 4.12 below shows an example of the change and rate of change calculations for a sine wave. The slow rate of change for a clean sine wave indicates the CSP can function in the low energy mode.

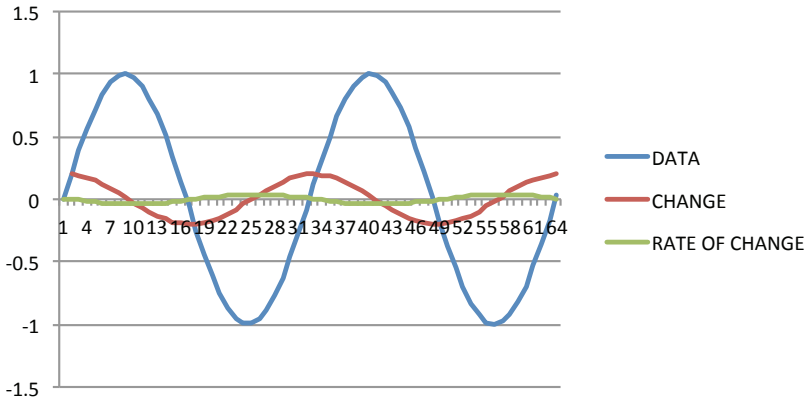


Figure 4.12: Rate of change calculations for a clean sine wave.

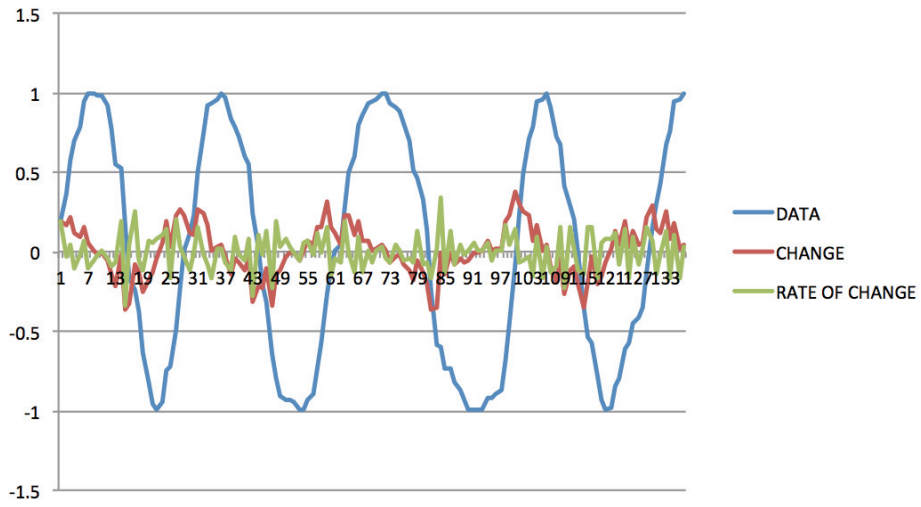


Figure 4.13: Rate of change calculations for a noisy sine wave.

Figure 4.13 above shows the effects that random noise on the sine wave signal has on the rate of change calculations. In this case the CSP will most likely be in a medium-energy mode. Figure 4.14 below shows a square wave where the CSP would be in a high-energy mode to handle the rapid rate of change of the data.

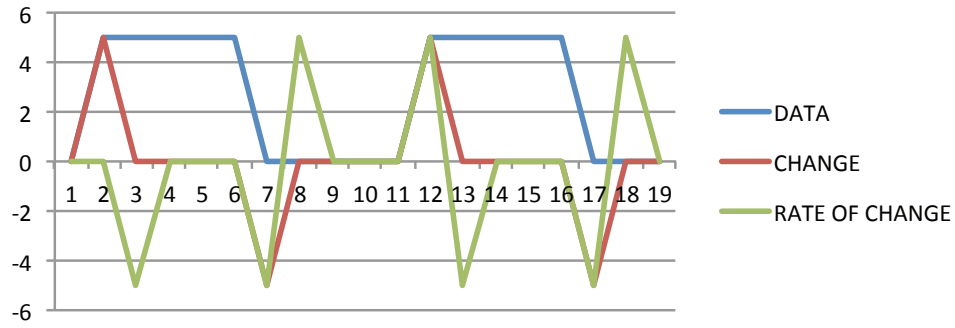


Figure 4.14: Rate of change calculations for a square wave.

4.3.2.3 Signal quality analysis

The Fuzzy Logic analysis of the sensor data is used to derive a signal quality value that is provided as sideband data tokens to the DPE along with signal data tokens. The quality value is determined by the following two functions:

1. Rate of change of the sampled data – if the data is changing too rapidly then it is possible that there is noise being injected into the sensor element and the data values should be analyzed accordingly.
2. Value of the data – if the data is not changing and/or is at the extreme limits of the data range then it is possible the sensor is broken and that the channel needs to be shut down or recalibrated.

The data encoding for the quality tag is fuzzy-compatible. This encoding indicates the quality error value and the rate of change of the quality error value under faulty and normal operating conditions as shown below in Tables 4.1 and 4.2. The signal quality data tokens are used to further qualify the sensor data tokens.

Table 4.1: Quality tag encoding – faulty operation

[Inc]_[Error]_[Dec]	Error	Error
1__00__1	NONE	NOT CHANGING - FAULT
1__01__1	SMALL	NOT CHANGING - FAULT
1__10__1	MID	NOT CHANGING - FAULT
1__11__1	LARGE	NOT CHANGING - FAULT

Table 4.2: Quality tag encoding – normal operation

[Inc]_[Error]_[Dec]	Error	Error Rate
0__00__1	NONE	DECREASING
0__00__0	NONE	NOT CHANGING
1__00__0	NONE	INCREASING
0__01__1	SMALL	DECREASING
0__01__0	SMALL	NOT CHANGING
1__01__0	SMALL	INCREASING
0__10__1	MED	DECREASING
0__10__0	MED	NOT CHANGING
1__10__0	MED	INCREASING
0__11__1	LARGE	DECREASING
0__11__0	LARGE	NOT CHANGING
1__11__0	LARGE	INCREASING

4.3.3 Channel nodes and channel routing nodes

Channel nodes handle all transmission and buffering of data between the Functional Service Unit (FSU) and the PPU. The FSU is fired when the channel node has buffered all of the token data from the PPU and is ready to transmit it. The channel node interface protocol is described below in Section 4.3.3.1. There is a variation of the channel node that buffers and routes data between multiple DPEs. These Channel Routing Nodes can be used to implement a network-on-chip (NOC) interconnection between Dataflow-Processing Elements. Figure 4.15 below shows a modified torus connection between eight heterogeneous processing elements.

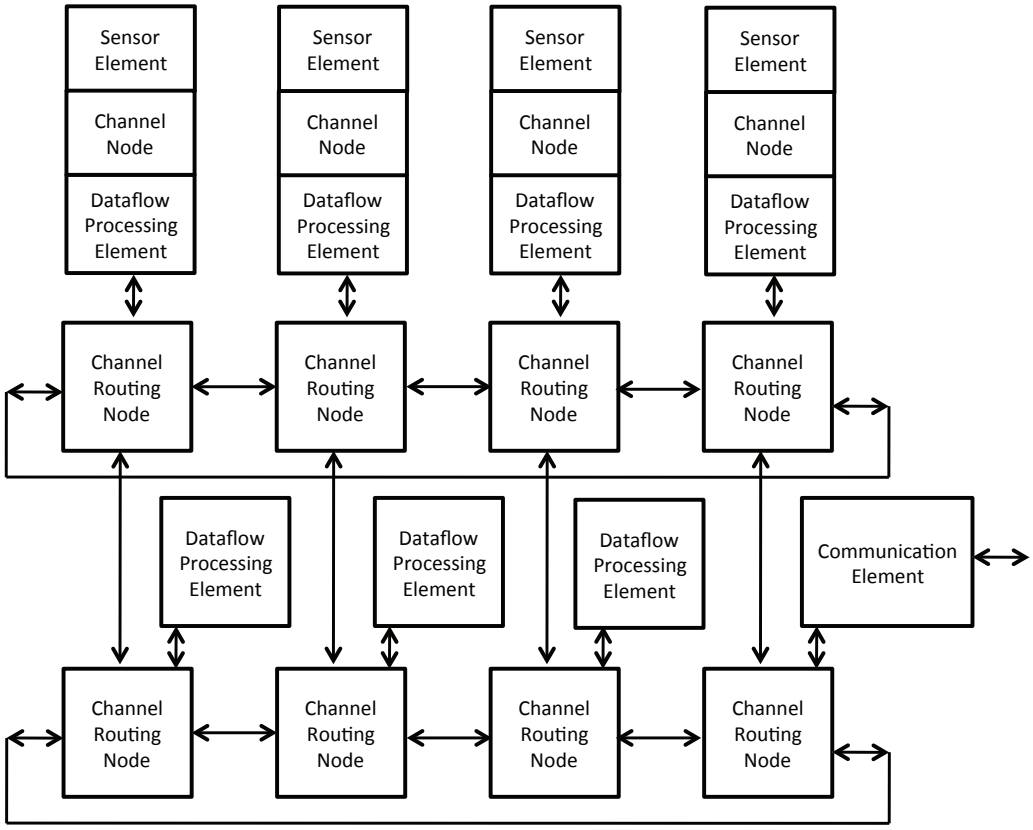


Figure 4.15: Channel routing nodes connecting multiple DPEs

A flit (flow-control unit) is the basic unit of communication between NOC Channel routing Nodes. There are two types of flits: event and token. Each flit utilizes two bits to encode its type. An event flit is either a control flit or a tail flit. The control flit contains channel node destination and debug information while the tail flit indicates the end of the transmission. The token flit is either a data word or a tag word. Figure 4.16 below shows the encoding for the four different types of flits. The flits are converted to the channel node interface protocol as shown below in the block diagram of a Channel Routing Node (Figure 4.17).

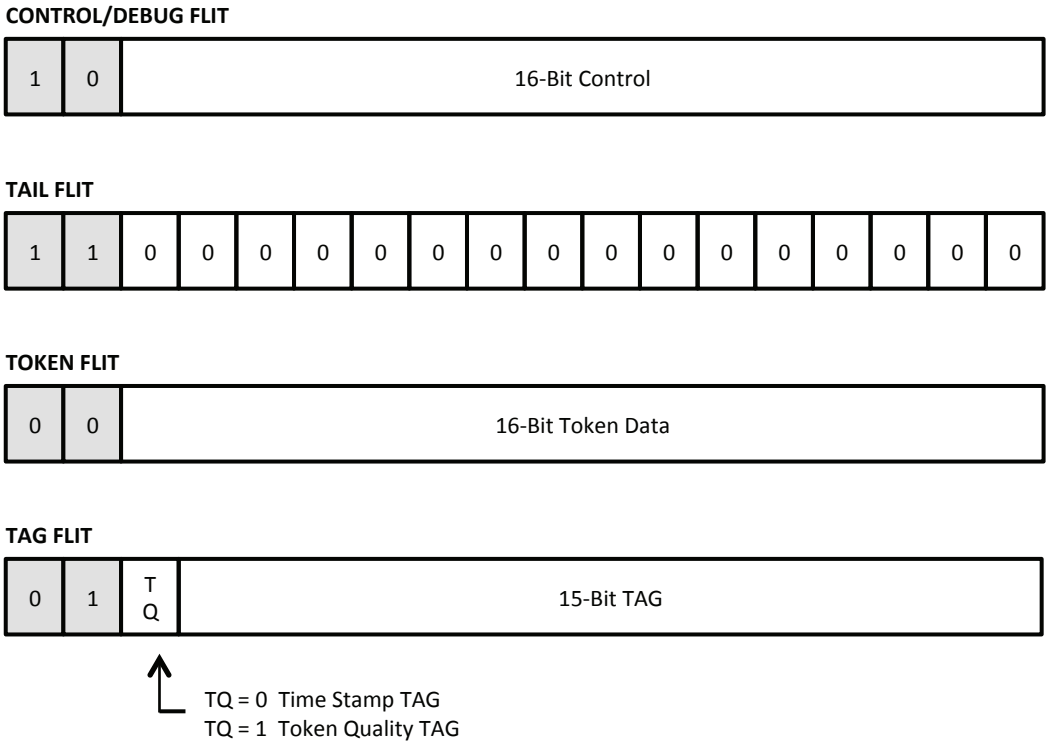


Figure 4.16 Flit encoding

The channel routing node consists of a standard channel interface unit and FIFO buffer to receive data from the DPE/FSU/COM elements. The channel routing node also unpacks data from the unit routers (via the cross-bar) and transmits the data to the input channel on the processing elements.

A full channel routing node consists of two X routing ports and two Y routing ports and a single channel node. The routing ports can be parallel or serial depending on the system implementation.

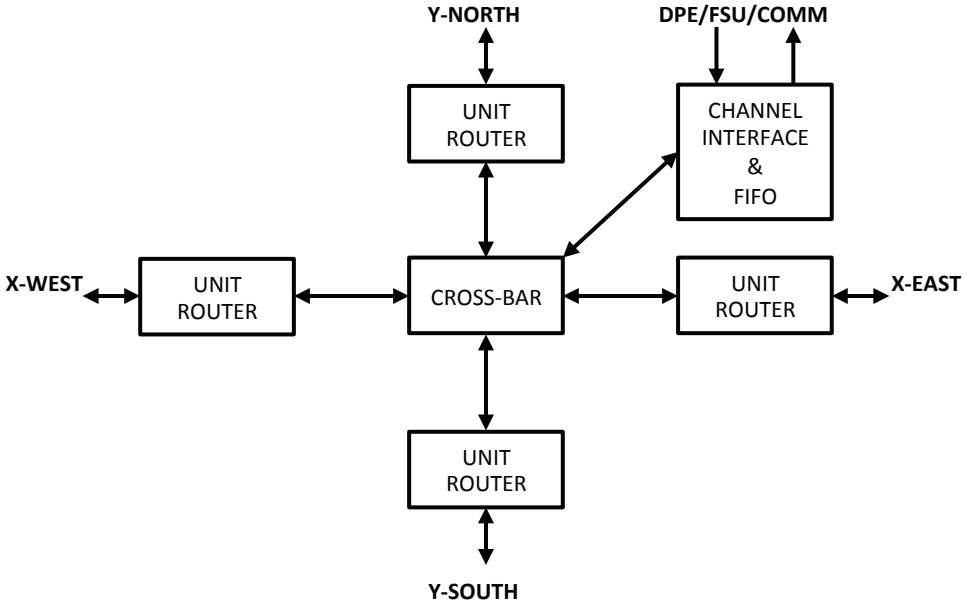


Figure 4.17: Block diagram of full Channel Routing Node

4.3.3.1 Channel node interface protocol

The I/O signals for the channel interface are described below in Table 4.3. There are four signal groupings: event, data, control and debug. The event signals are used to trigger the FSU/DPE to start execution. The control signals are used to reset the FSU/DPE and force a bypass condition for special testing operations. The debug signals are connected to the scan test unit and are used to put the channel node into either a debug or test mode configuration.

Table 4.3: Summary of I/O signal descriptions for Channel Interface

Signal	I/O	Group	Description
READY_IN	I	EVENT	INDICATES RECEIVING NODE CAN START PROCESSING DATA
READY_OUT	O	EVENT	INDICATES SENDING NODE HAS VALID DATA
HOLD_IN	I	EVENT	INDICATES THAT SENDING NODE SHOULD HOLD DATA
HOLD_OUT	O	EVENT	INDICATES RECEIVING NODE IS NOT READY TO PROCESS DATA
TOKEN_SZ_IN	I	EVENT	INDICATES # OF TOKENS TO CONSUME
TOKEN_SZ_OUT	O	EVENT	INDICATES # TOKENS TO BE SENT TO RECEIVING NODE
TOKEN_CLK_IN	I	EVENT	CLOCK TO SEND TOKENS TO RECEIVING NODE
TOKEN_CLK_OUT	O	EVENT	CLOCK TO SENDING NODE TO TRANSFER TOKENS
DATA_IN	I	DATA	DATA INPUT TO THE SERVICE NODE. CAN BE MULTIPLE TOKENS
DATA_OUT	O	DATA	DATA OUTPUT FROM THE SERVICE NODE
BYPASS	I	CONTROL	FORCES ELEMENTS TO BYPASS PROCESSING STEP(S)
CLOCK/CLOCK_90	I	CONTROL	REFERENCE CLOCKING SIGNALS
RESET	I	CONTROL	MASTER RESET
SE_DEBUG	I	DEBUG	SCAN ENABLE VS. DEBUG MODE
SCAN_IN	I	DEBUG	SCAN DATA CHAIN INPUT
SCAN_OUT	0	DEBUG	SCAN DATA CHAIN OUTPUT
SCK_IN	I	DEBUG	SCAN CLOCK INPUT
SCK_OUT	O	DEBUG	SCAN CLOCK OUTPUT

As described above in Section 4.1, the FSU and DPE in the CSP are modeled as Synchronous Data Flow (SDF) actors. This implies that each element adheres to an event-driven processing paradigm. As indicated above in Table 4.3, there are four event control signals that provide the handshaking between channel and service nodes. These are READY, HOLD, TOKEN_SZ and TOKEN_CLK. The READY signal is an indication that the sending node has valid data to be processed by the receiving node. The

HOLD signal is used to keep the READY signal asserted until all of the tokens have been received. The TOKEN_SZ signal indicates how many tokens to process. The TOKEN_CK is the clock signal that transfers the tokens from the sending node to the receiving node. Figure 4.18 below shows the timing diagram for the four event signals during a receive operation.

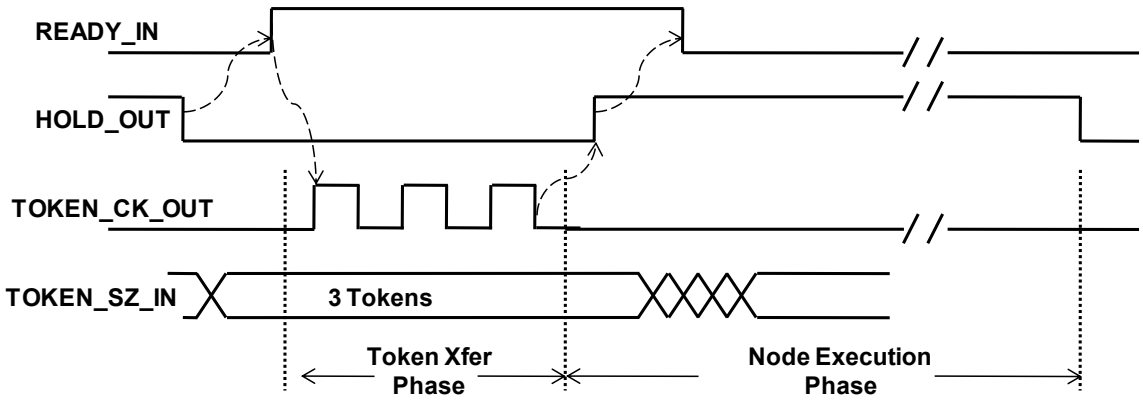


Figure 4.18: Timing diagram for event signals during the receive cycle

The READY_IN signal generates an edge-triggered event that starts the sequencing of operations in the receiving node. The token clock transfers the requisite number of tokens and then asserts the HOLD_OUT signal, which indicates to the sending node that the READY_IN signal can be negated. The receiving node processes the token data while the HOLD_OUT signal is asserted. The negation of the HOLD_OUT signal occurs when the processed token data has been successfully transferred to the next node. Figure 4.19 below illustrates the timing sequence during the send cycle. Once HOLD_IN is negated the sending node asserts the READY_OUT signal and the TOKEN_SIZE signals to indicate another tranche of tokens are available to be consumed.

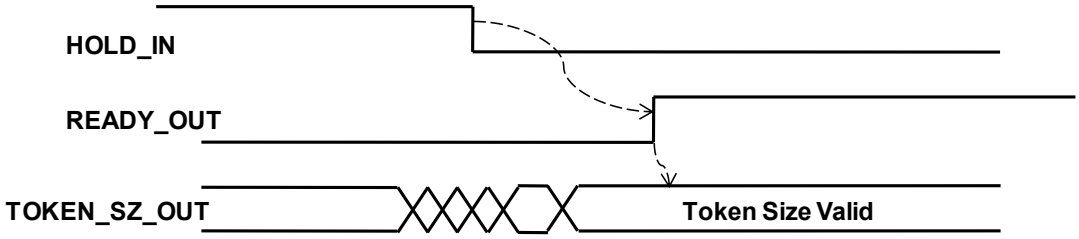


Figure 4.19: Timing diagram for event control signals during the send cycle

The sequencing of the channel or service node is controlled either by an external synchronous clock or via an internal self-timed clocking mechanism. Figure 4.20 below shows the timing diagram of a self-timed clocking circuit that drives the SPU and DPE.

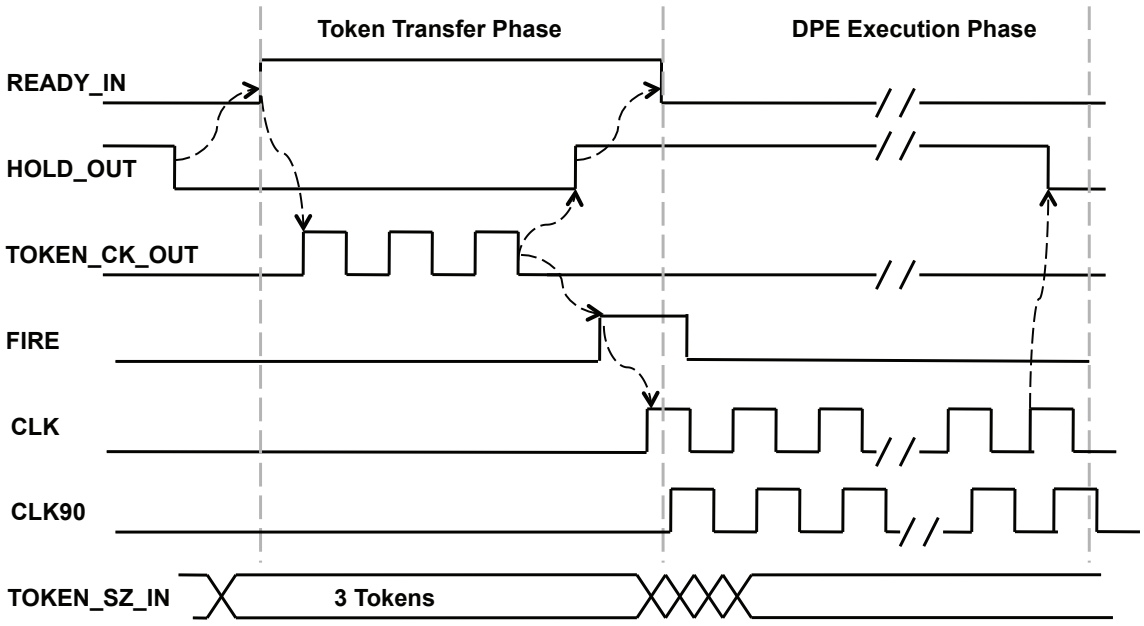


Figure 4.20: Self-timed clocking timing diagram

The CLK/CLK90 clock signals are enabled by the edge detection of the FIRE signal and disabled when the actor completes processing the token data. The clocks run until an idle condition is detected or when a Wait-for-Event instruction is executed. All

operations in the FSU/DPE execute in a single cycle. Figure 4.21 shows the phasing of the CLK and CLK90 clock signals.

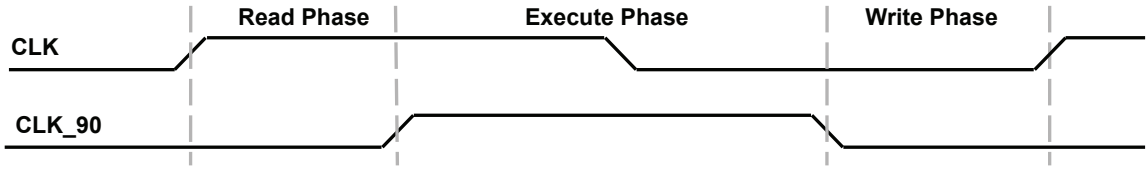


Figure 4.21: CLK/CLK90 timing diagram

4.4 Dataflow-Processing Element (DPE)

The DPE is an event-driven microcoded Queued-Stack based machine. Stack based processors were originally designed to support stack oriented languages such as Forth [49] and Java [50]. Specialized stack based register files have also been used for VLIW machines to provide multiple operands for parallel operations [51]. As mentioned earlier the DPE merges an input queue with a stack thus removing the dependency to fetch token data from the queue and moving it to the stack. The stack is controlled via a unique one-hot addressing method that significantly reduces power compared to a classical register file.

The DPE processes the token data and forwards the results as tokens to other DPEs or to the communications element (COM). Typical functions performed by the DPE include:

- Signal Processing
- System Diagnostics and Calibration
- Algorithmic optimization

- Compensation for systematic errors, system drift and random errors produced by system parametric changes such as sensor aging, battery aging.
- Reasoning about the state of the system and performing needed services to maintain optimal system performance.
- Anticipating potential systematic changes and modifying operational behavior.

The DPE is described in more detail in Chapter 5. The microprogramming for the DPE is described in Chapter 6 and the high-level modeling environment in Chapter 7.

4.5 Communications element (COM)

The communication element for the current implementation of the CSP is an IEEE-1451 compliant module that provides Level-0 communications capability [48]. The IEEE-1451 is a family of Smart Transducer Interface Standards. It describes a set of open, common, network-independent communication interfaces for connecting transducers (sensors or actuators) to microprocessors, instrumentation systems, and control/field networks. The key feature of these standards is the definition of Transducer Electronic Data Sheets (TEDS). The TEDS is a memory device attached to the transducer that stores transducer identification, calibration, correction data, measurement range, and manufacture-related information, etc. The goal of IEEE-1451 is to allow the access of transducer data through a common set of interfaces whether the transducers are connected to systems or networks via a wired or wireless means.

Compliance with this standard provides a number of benefits for intelligent sensor systems. These are:

- Develop network-independent and vendor-independent transducer interfaces

- Define standardized Transducer Electronic Data Sheets (TEDS) that contain manufacture-related data
- Support a general model for transducer data, control, timing, configuration, and calibration
- Eliminate error prone, manual entering of data and system configuration steps, ultimately achieving Plug and Play
- Allow transducers (sensors or actuators) to be installed, upgraded, replaced or moved with minimum effort
- Able to get wired or wireless sensor data and information seamlessly from a host system or network anywhere in the world.

4.6 Debug element

The debug element is designed to provide observability and controllability of the CSP functional elements. It uses an IEEE 1149.7 Compact Joint Test Action Group (cJTAG) controller to provide the following features:

- Single stepping
- Run-to-WFE
- Run-to-Halt
- Queued-stack loading and unloading
- Loading and modifying microcode
- Loading the CLT state
- Injecting calibration tokens into the CSP

The IEEE 1149.7 standard improves upon the IEEE 1149.1 standard, commonly referred to as JTAG (Joint Test Action Group) that is used for testing and debugging

integrated circuits. The purpose of 1149.7 is to meet an expanding set of challenges facing debug and test systems while preserving the hardware and software investments of the community currently using the 1149.1 standard. The cJTAG implementation builds upon 1149.1 and is a true superset of JTAG. The benefits of cJTAG are:

- Reduced pin count
- Core level bypass for multi-core or NOC systems
- Individual device addressing
- Star topology
- Additional power management features

Figure 4.22 below shows examples of core level bypass and individual device addressing.

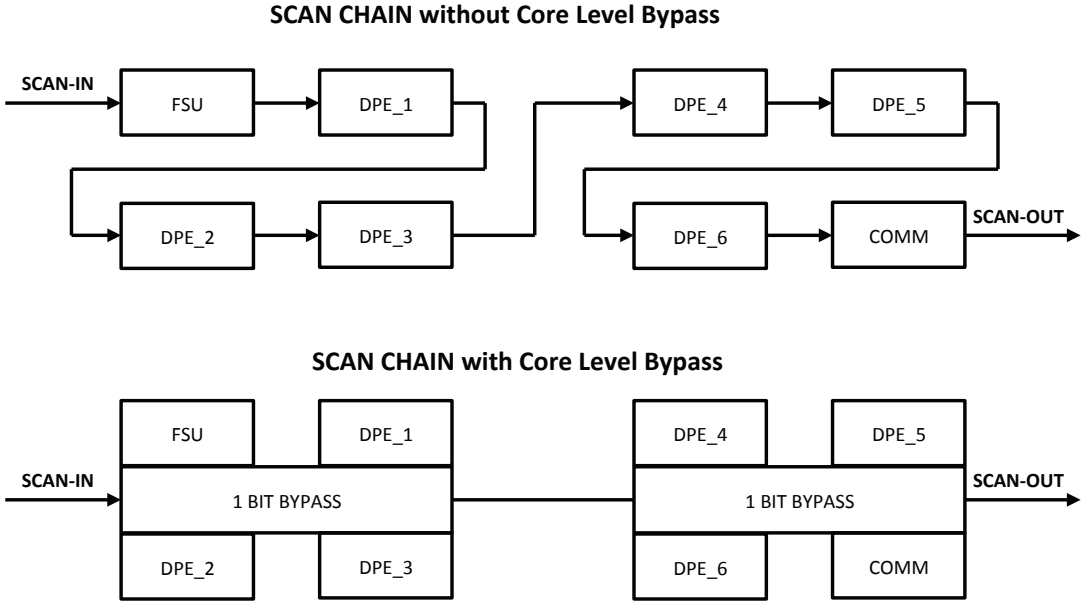


Figure 4.22: Example of Core Level Bypass in scan chains

4.7 Summary

The architecture presented above is a high level view of what a typical CSP would look like. There is considerable more detail that could be presented but is beyond the scope of this dissertation. A list of follow-on research topics related to this dissertation is presented in Chapter 9.

Chapter 5. Dataflow-Processing Element

5.1 Overview

The Dataflow-Processing Element (DPE) in the CSP is implemented using a stack-based microcoded engine with advanced features such as nested looping, conditional execution, repeat execution, Actor/Event queuing, Fuzzy Logic acceleration and a programmable content-addressable lookup table. Figure 5.1 below shows a block diagram of a typical DPE implementation using two Input Queued-Stack (IQS) units, one Result Queued-Stack (RQS) unit and one output FIFO.

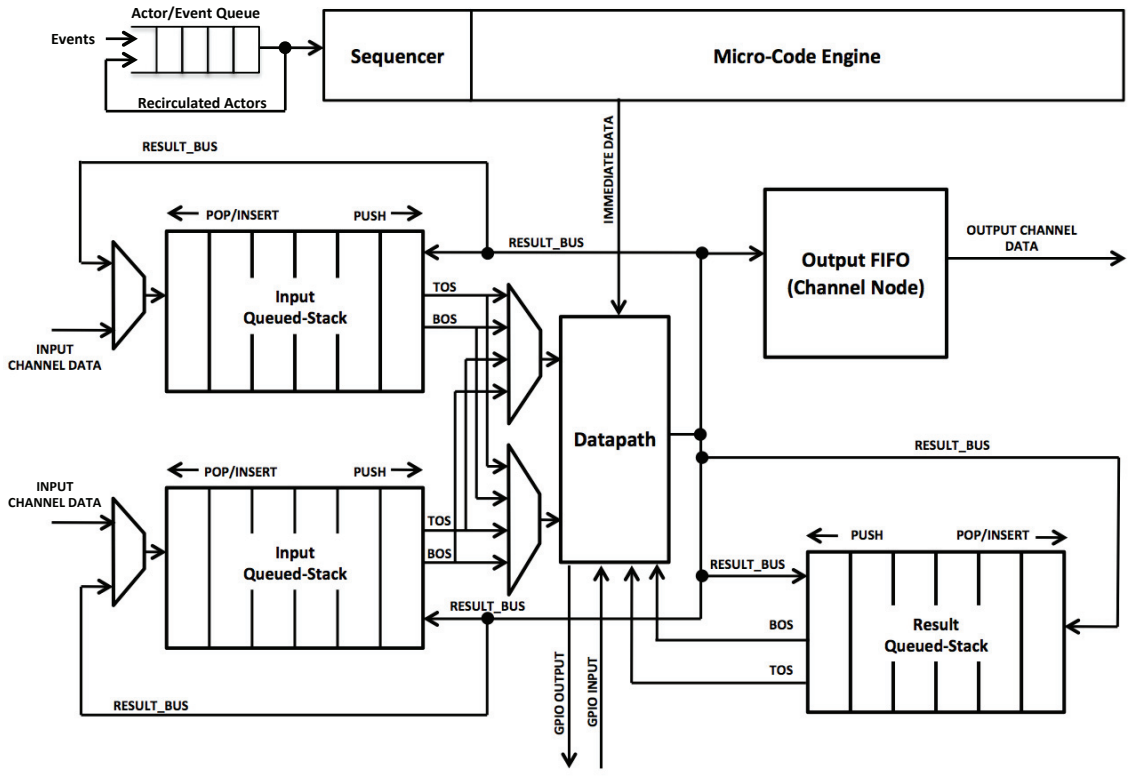


Figure 5.1: Dataflow-Processing Element block diagram

The two IQS units are used to receive input channel data. Alternatively they can be configured such that one IQS unit is used for receiving channel data while the other is used to store operands or result data. The outputs of the two IQS units are symmetrically multiplexed onto the datapath. The RQS is used to store the higher precision results of the datapath operations. The datapath supports single cycle shift-multiply-accumulate operations, Boolean logic operations and Special-Function operations.

The microcode engine controls the DPE using a 96-bit wide control word. Key control signals are 1-hot which eliminates the need for complex decoding and prevents logic glitching. The code memory used in the microcode engine is implemented using a standard single port ROM or a Writable Control Store (WCS) memory compiler. The WCS configuration is useful for systems where the microcode needs to be updated from an external source such as FLASH memory.

The DPE is implemented using a parameterized synthesizable model where the width and depth of the stacks, functional units, and datapaths are determined during algorithmic development time. For systems that are composed of multiple DPEs, it is feasible for each DPE to be configured for a particular task or group of tasks during the synthesis process by selecting the optimal parameters.

5.2 Input Queued-Stack (IQS) Unit

There are two basic modes of operation for the Input Queued-Stack unit: Pull-Mode and Push-Mode. In Push-Mode the input channel data is asynchronously inserted into the IQS. This can result in an overrun condition if the DPE cannot process the channel data from the previous transaction. In Pull-Mode the IQS will fetch data from the input channel once the microcode engine is in an idle condition thus preventing overrun. In Pull-Mode, the output data from the sending DPE must be stored in an output FIFO to

prevent stalling its processing element. Pull-Mode provides an additional benefit in that it allows the IQS to be used as a circular buffer to store filter coefficients used in many filtering algorithms. In both modes the microcode engine will wait for the fire signal before it starts to process the data.

The IQS is implemented as a circular buffer that has two circular pointers, one to track the queue (FIFO) data and the other to track the stack (LIFO) data. The pointer registers are implemented using 1-hot shift registers. This eliminates decoding logic and reduces power. A three-entry IQS is shown below in Figure 5.2.

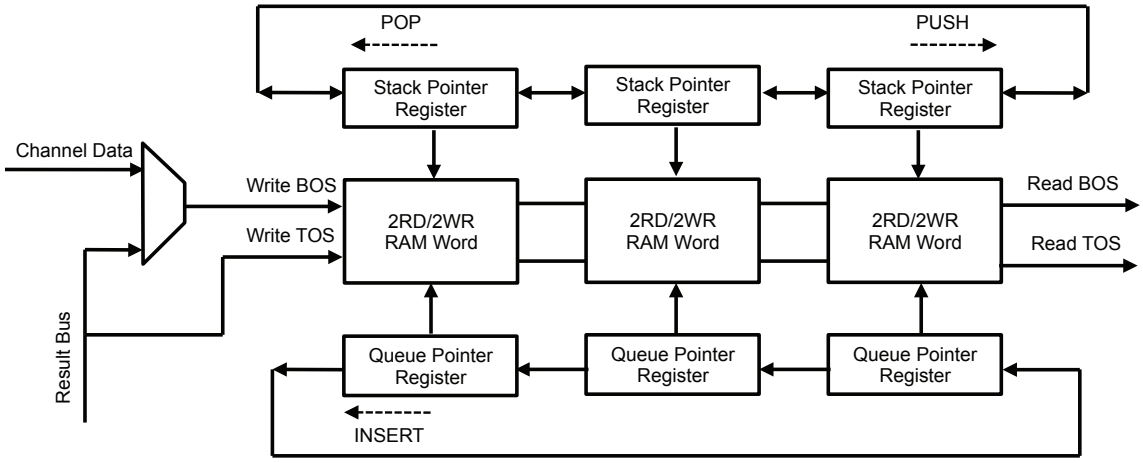


Figure 5.2: Three-entry Queued-Stack unit block diagram

The FIFO-queue pointer tracks data inserted into the queue from either the channel data or the result data from the datapath operations. The LIFO-stack pointer is used to track pushes and pops from the stack. The IQS can insert data into the queue while simultaneously pushing or popping data on/off the stack. This allows channel data to be asynchronously inserted into the queue element while the computational engine is processing data from a previous transaction. The IQS uses a 2-read-port, 2-write-port

memory cell. This provides the capability to simultaneously access both the top-of-stack (TOS) data and the bottom-of-stack (BOS) data. The microcode engine can manipulate the pointer registers to select data anywhere in the circular buffer. For example, the stack pointer can be rotated two positions to the right and the queue pointer can be rotated two positions to the left in a single instruction by executing a POP command and an INS_NW command with a REPEAT of two. Similar operations can be performed on the Result Queued-Stack in the same micro-operation.

The pointer registers and read/write operations for both IQS units are under microcode control, which allows a number of simultaneous operations to be performed each clock cycle. These operations are specified in Table 5.1 below.

Table 5.1: Input Queued-Stack Operations

Operation	Description
PUSH	ROTATE TOS POINTER RIGHT AND WRITE RESULT-BUS VALUE TO NEW TOS
POP	ROTATE TOS POINTER LEFT (W/O WRITE)
POP_WR	ROTATE TOS POINTER LEFT AND WRITE RESULT-BUS VALUE TO NEW TOS
INS	ROTATE BOS POINTER LEFT AND WRITE RESULT-BUS VALUE TO BOS
INS_NW	ROTATE BOS POINTER LEFT (W/O WRITE)
PUSH_NW	ROTATE TOS POINTER RIGHT (W/O WRITE)
TOP	WRITE RESULT BUS VALUE TO TOS W/O ROTATING POINTER
BOT	WRITE RESULT BUS VALUE TO BOS W/O ROTATING POINTER
TOP_BOT	WRITE RESULT BUS VALUE TO TOS/BOS W/O ROTATING POINTERS
PUSH_INS	ROTATE BOTH POINTERS AND WRITE RESULT-BUS VALUE TO TOS/BOS
POP_BOT	ROTATE TOS POINTER LEFT AND WRITE RESULT-BUS TO BOS
POP_INS	ROTATE TOS POINTER LEFT, ROTATE BOS LEFT AND WRITE RESULT-BUS TO NEW BOS
POP_WR_BOT	ROTATE TOS POINTER LEFT AND WRITE RESULT-BUS TO BOS AND TO NEW TOS
PUSH_NW_BOT	ROTATE TOS POINTER RIGHT AND WRITE RESULT-BUS TO BOS
TOP_INS	ROTATE BOS POINTER LEFT AND WRITE RESULT-BUS TO TOS AND TO NEW BOS
NOP	NO OPERATION

The IQS operations specified above are performed on the second half of the clock cycle as shown below in Figure 5.3. The Queue and Stack pointers are modified on the falling edge of the clock, i.e., shifted left or right. The data on the result-bus from the datapath is written at this time. Both pointer addresses remain pointing to valid data for

the next operation on the rising edge of the CLOCK signal. The read-muxes shown above in Figure 5.1 drive the TOS and BOS data from the two IQS units to the datapath.

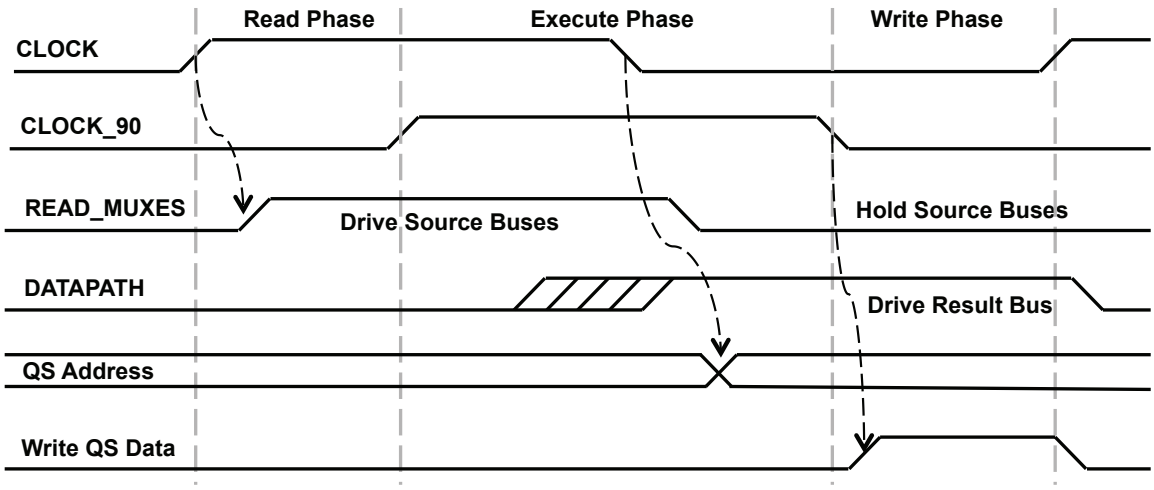


Figure 5.3: Queued-Stack timing diagram

5.3 Result Queued-Stack (RQS) Unit

The RQS uses the same parameterized building blocks as the IQS. The width of RQS is generally the width of the adder in the datapath unit. The class of algorithms determines the depth of the RQS that the DPE is being synthesized to perform. The RQS performs the same Queued-Stack operations as the IQS as shown above in Table 5.1

Figure 5.4 below shows the block diagram for a three entry RQS. The primary difference between IQS and the RQS is that the write-data is only sourced from the result bus.

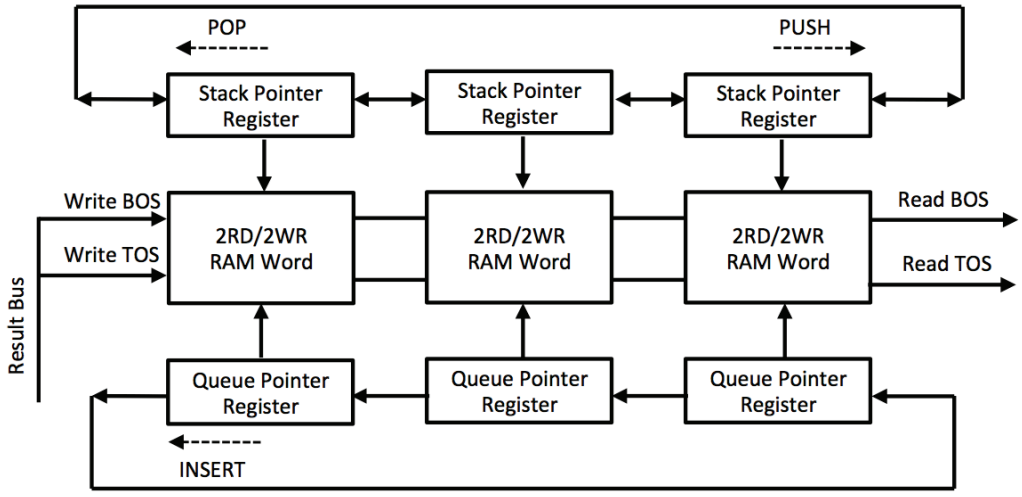


Figure 5.4: Three-entry Result Queued-Stack unit block diagram

5.4 Datapath Unit

The DPE datapath is composed of five major units: shifter, multiplier, adder, logical unit (LU) and a special function unit (SFU) as shown below in Figure 5.5.

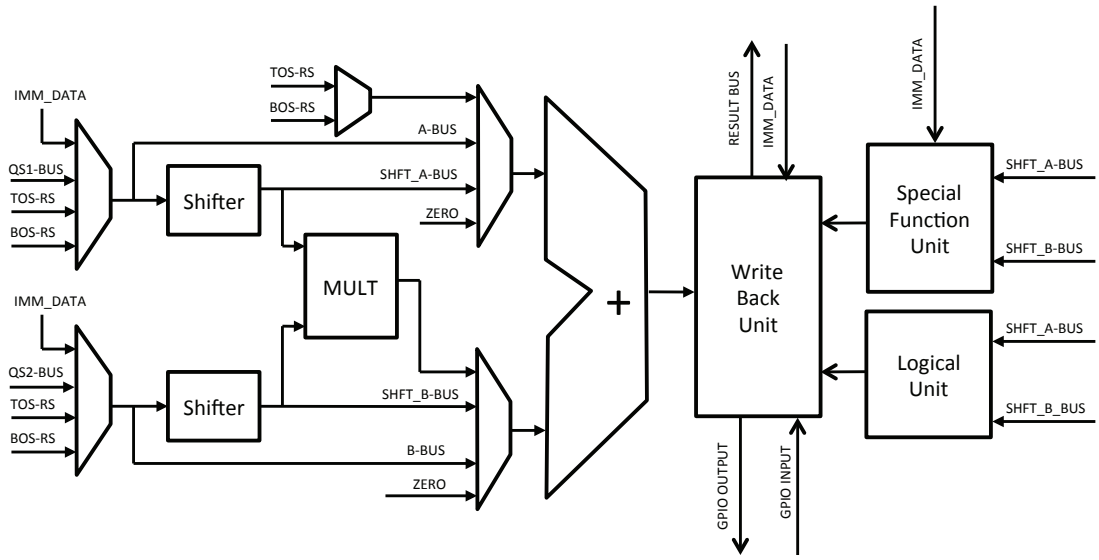


Figure 5.5: Block Diagram of the DPE Datapath

There are two levels of multiplexors that control the source data to the datapath units. Table 5.2 below shows the various source data configurations.

Table 5.2: Input data source for the datapath units

Shifter Input	Multiplier/LU/SFU Input	Adder Input
TOS/BOS IQS1	SHIFTER_A/SHIFTER_B	SHIFTER_A/SHIFTER_B
TOS/BOS IQS2	TOS/BOS IQS1	TOS/BOS IQS1
TOS/BOS RQS	TOS/BOS IQS2	TOS/BOS IQS2
IMMEDIATE DATA	TOS/BOS RQS	TOS/BOS RQS
	IMMEDIATE DATA	IMMEDIATE DATA
		MULTIPLIER

The 4-input multiplexors that feed the datapath have latching outputs that are used to prevent spurious transactions propagating through the shifter-multiplier-adder paths. This reduces power and can also be used to store intermediate data from previous transactions. The adder input multiplexors either zero-fill or sign-extend the input data to match the output width of the adder. For example, in the current implementation the IQS is 16-bits wide, the shifter is 32-bits wide, the adder is 48-bits wide and the RQS is 48-bits wide.

The datapath is capable of executing the following combination of instructions in a single cycle:

- Shift-Multiply-ADD/SUB or Shift-Multiply-Saturating ADD/SUB
- Multiply- ADD/SUB or Multiply-Saturating ADD/SUB
- Shift- ADD/SUB or Shift-Saturating ADD/SUB
- ADD/SUB or Saturating ADD/SUB

- Arithmetic & Logical Shift
- Boolean operations: AND, NAND, OR, NOR, INVERT, XOR, XNOR
- Table lookup

The Write-Back-Unit multiplexes data from the adder, special function unit, logical unit and the general purpose input port on to the result bus. The result bus is connected to the two IQS units, the RQS unit and the output FIFO as shown above in Figure 5.1. The RQS is used to store the results of the datapath transactions and is synthesized to be the width of the output of the adder. Note: an IQS unit can also be used to store results, however, it is limited to storing data that is the width of the incoming channel data.

The functions that the SFU performs are determined during the algorithmic design phase. Typical functions include: table-lookup for sensor recalibration, interpolation, linearization, averaging, Fuzzy Logic acceleration, data compression, data fusion, time stamping, edge detection, threshold detection, period measurements, etc.

5.4.1 Condition code generation

There are two sets of condition codes generated within the DPE. The first set is generated by the arithmetic operations in the datapath. The second set is generated in the Special Function Unit. The condition codes are used by the microcode engine for conditional branch instructions and the conditional execution of microinstructions.

The write-back data from each arithmetic datapath operation is compared to a reference value each cycle to determine if it is greater-than, equal to, less-than a specified reference value. The reference value is stored in the Write-Back Unit by specifying the WB_LAT control bit in the microinstruction. Additionally, there is a reference value stored in the SFU that is used to compare the results of SFU operations that are written

back via the Write-Back Unit. The SFU reference value is stored in the Write-Back Unit by specifying the SFU_LAT control bit in the microinstruction.

Figure 5.6 below shows the timing diagram for a typical datapath operation where the condition codes are determined during the high phase of the CLOCK_90 signal. The condition codes a registered on the following edge of the CLOCK_90 signal and used during the next cycle for conditional micro-opcode execution or branching operations.

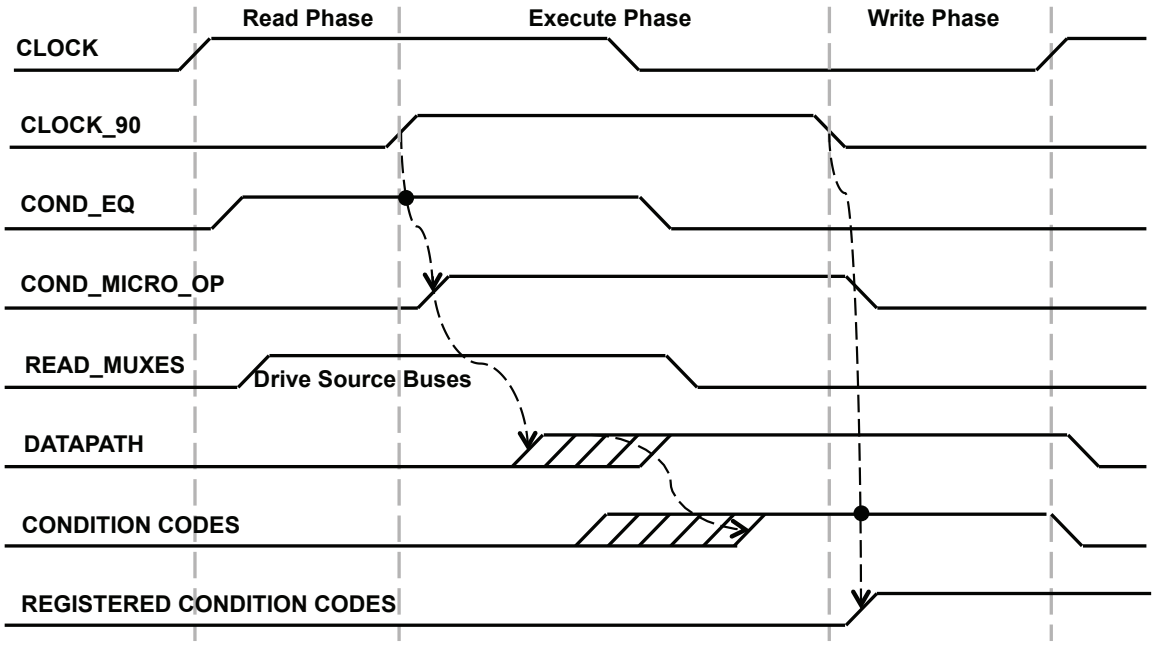


Figure 5.6: Timing Diagram for Condition Code Generation

5.4.2 Special Function Unit (SFU)

The SFU is an algorithmic specific unit that is synthesized using DesignWare® components[1]. In the current implementation the SFU is used to accelerate the Fuzzy

[1] Courtesy of Synopsys Inc.

Logic algorithms. There are two accelerators in the current implementation: 1) MIN/MAX unit and 2) Content Addressable Lookup Table (CLT).

5.4.2.1 MIN/MAX unit

The MIN/MAX unit supports minimum and maximum of the 3 inputs: A_BUS, B_BUS and REFERENCE_DATA. Figure 5.7 below shows the block diagram of the unit.

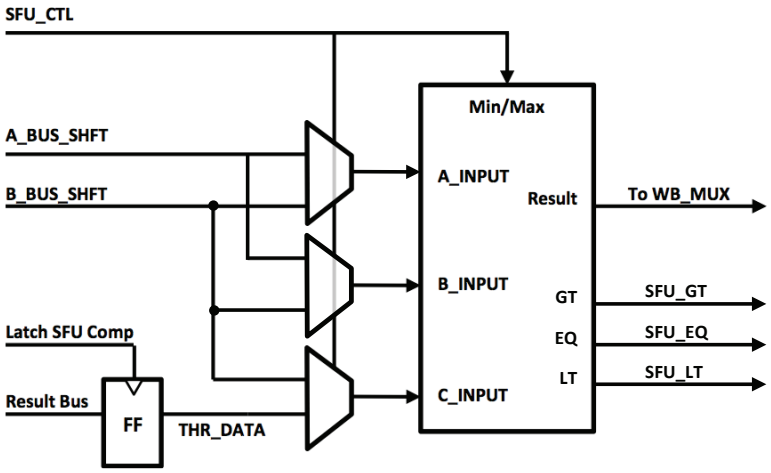


Figure 5.7: Block diagram of MIN/MAX logic

5.4.2.2 Content-Addressable Lookup Table (CLT)

In the current configurations of the DPE, a 64x6 bit memory array coupled to a 64x8 array that is synthesized to provide a low power content-addressable lookup table. The CLT can be dynamically reconfigured via the DPE result bus. Additionally, the Debug Unit can load and unload both memory arrays using the scan test logic. The CLT data would generally reside in an external FLASH memory device connected to the Debug Unit.

Figure 5.8 below shows the basic architecture of the CLT. The output of the 64x6 memory array is used as the address for the 64x8 memory array whose output is then multiplexed onto the result bus through the write-back mux. Each array can be dynamically reconfigured independently of each other. The CLT can be used to linearize sensor data [37], hold route tables for a Network-on-Chip (NOC) topology (Figure 1.8) or provide complex logic functions as is typically done in an FPGA [38]. Additionally, it can be used for Fuzzy Logic operations where in a defuzzify operation the first array holds the antecedent mapping to the fuzzy output value that is in the second array.

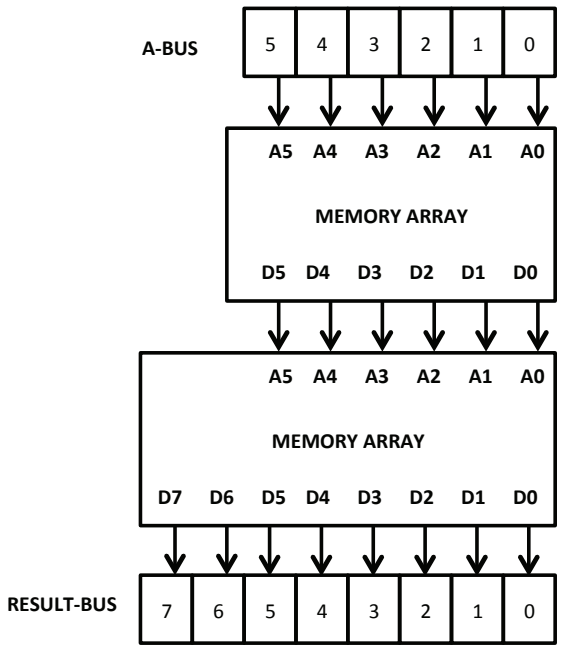


Figure 5.8: Content addressable lookup table (CLT) architecture

5.5 Microcode Engine

The decision to use a microcoded instruction format was primarily driven by the fact that the DPE is not pipelined and there are a number of parallel operations that must

be performed in a single cycle thus eliminating a sequencer to control the various units in the DPE. This also eliminates the need for an instruction decoder, as the output from the Actor/Event queue is a micro-address to a routine in the microcode memory. There are three communicating finite state machines (CFSM) that control the flow DPE as shown below in Figure 5.9.

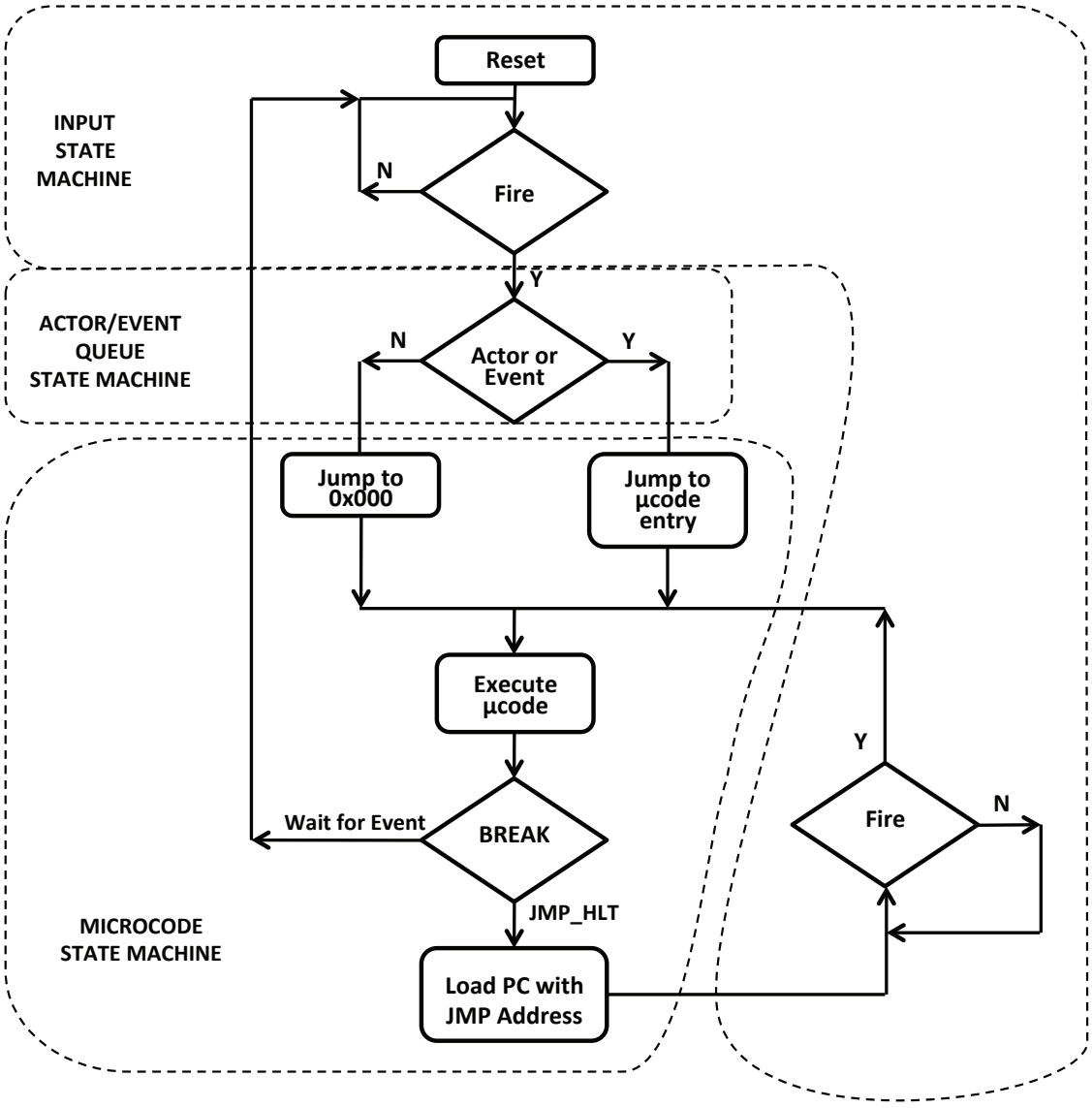


Figure 5.9: DPE Operation Flow Chart

The input state machine tracks the number of input tokens and generates the FIRE signal when all of the tokens are in the IQS. The event queue state machine tracks the availability of actors and events to be executed by the microcode state machine. The microcode state machine tracks the operational state of the DPE as shown below in Figure 5.13.

5.5.1 Microcode fields

There are five control fields in the microcode word as shown below in Figure 5.10. The first field defines specific micro-operations within the microcode engine. These include nested looping, repeat function, branching and conditional execution. Three levels of hardware nested looping [52] are supported. All nested loop offsets are backwards while branch offsets can be both forwards and backwards. The branch operation utilizes the offset field and the loop count fields, which extends the range. The repeat operation further modifies the program flow by providing the capability to execute multiple nested loops. This is useful for operating on multi-dimensional data arrays. There are three repeat counters, one for each level of nesting. A state machine tracks the nesting context of all active loops. Most microcode operations can be conditionally executed. The exceptions are loop returns and the HALT instruction. Conditional execution uses condition codes derived from the arithmetic units in the datapath and the SFU. The first field also contains the immediate data word that can be used as an operand by the datapath or the SFU. The width of this field is determined by the algorithmic requirements of the machine.

Micro-engine control:

IMMED DATA <95:80>	BRANCH OFFSET <79:73>	LOOP COUNTER NUM <72:71>	REPEAT COUNT <70:68>	UCODE OP <67:64>

Datapath control:

SHFT MODE B <45>	SHFT MODE A <44>	SHFT B <43:39>	SHFT A <38:34>	SHFT TC B <33>	SHFT TC A <32>	MULT <31>	SAT <30>	DP EN <29>	ADD/SUB <28>

Queued-Stack control for next operand selection:

READ TOS QS2 <63>	READ TOS QS1 <62>	READ BOS QS2 <61>	READ BOS QS1 <60>	QS2 CTL <59:56>	QS1 CTL <55:52:>	B_BUS MUX <51:50>	A_BUS MUX <49:48>	CHAN MUX QS2 <47>	CHAN MUX QS1 <46>

Queued-Stack control for write-back operations:

QS2 BUS SEL <23:22>	QS1 BUS SEL <21:20>	WB MUX SEL <18:17>	RQS CTL <9:6>	RD RQS <4>	RQS BUS MUX <3>

SFU & I/O control:

LOGIC OP <16:14>	SFU OP <13:11>	WR SFU LAT <10>	WR EQ LAT <5>	FIFO WE <2>	GPIO WE <0>

Figure 5.10: Microcode control fields

The second field controls the multiplexors and the functional units in the datapath using a combination of one-hot control bits and encoded control bits [53]. As can be seen from Figures 5.1 and 5.5 above, there are 10 multiplexors that control the flow of data through the functional units, the write-back mux and the IQS elements.

The third and fourth fields are for Queued-Stack control. The timing diagram for a typical operation is shown above in Figure 5.3. During the first half of the cycle the IQS provides the operands to datapath and the second half of the cycle the address of the IQS can be modified for the write operation. For example, if a PUSH operation is performed the FIFO pointer is shifted right to point to next location on the stack. The result data can be written to this location at the end of the cycle. The pointers in the IQS units and the RQS unit always point to valid data and are only modified during a write cycle.

The fifth field is used to control the special function unit, the general-purpose I/O port and the output FIFO. The block diagram of the micro-engine as shown below in Figure 5.11 consists of four components: a writeable control store, micro-address generation, Actor/Event queue and a finite state machine controller.

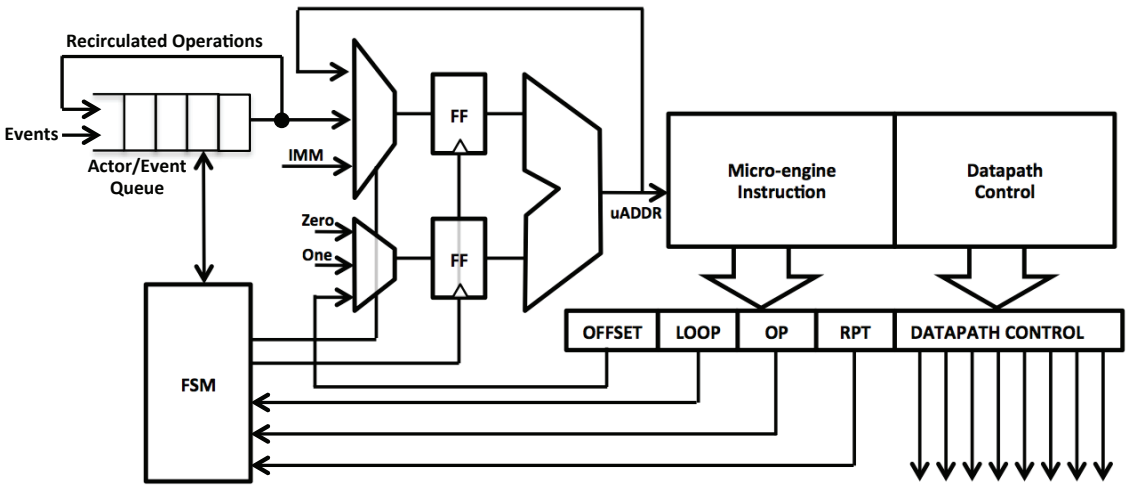


Figure 5.11: Block Diagram of the Microcode Engine

The micro-engine is controlled via a 4-bit micro-opcode field. The micro-engine opcodes supported are shown below in Table 5.3.

Table 5.3: Micro-Engine operation codes

OPCODE	RPT	MICRO-OPERATION
EXEC	Y	UNCONDITIONAL EXECUTION
EXEC_WB_EQ	Y	EXECUTE IF WRITEBACK == REFERENCE DATA
EXEC_WB_GT	Y	EXECUTE IF WRITEBACK > REFERENCE DATA
EXEC_WB_LT	Y	EXECUTE IF WRITEBACK < REFERENCE DATA
WFE	N	HALT AT PC+1; WAIT-FOR-EVENT SIGNAL
JMP	N	JUMP TO ADDRESS SPECIFIED IN IMMED_DATA FIELD
JMP_HLT	N	JUMP TO ADDRESS SPECIFIED IN IMMED_DATA FIELD THEN HALT AND WAIT FOR AN EVENT SIGNAL
LOOP_BACK	Y	LOOP BACK FOR LOOP # (NEGATIVE OFFSET ONLY)
BRA	N	BRANCH UNCONDITIONALLY
BR_WB_EQ	N	BRANCH IF WRITEBACK == REFERENCE DATA
BR_WB_GT	N	BRANCH IF WRITEBACK > REFERENCE DATA
BR_WB_LT	N	BRANCH IF WRITEBACK < REFERENCE DATA
BR_SFU_GT	N	BRANCH IF SFU RESULT > REFERENCE DATA
BR_SFU_LT	N	BRANCH IF SFU RESULT < REFERENCE DATA
BR_SFU_EQ	N	BRANCH IF SFU RESULT == REFERENCE DATA
TXFR	N	TRANSFER OUTPUT FIFO DATA TO CHANNEL

There are two basic execution modes: normal and conditional. In normal mode all micro-operations execute independently of the value of the condition codes. The conditional execution micro-opcode (EXEC_WB_XX) uses the value of the condition codes from the previous cycle to determine if the microinstruction can be executed. If the microinstruction is not executed the datapath retains its original state and the next microinstruction is then fetched.

All branch instructions operate in normal mode and are referred to as merged instructions. A merged instruction is one where the jump/branch can be executed simultaneously with a datapath operation [54].

The repeat function provides the capability of executing a single microinstruction or group of microinstructions 2 to 8 times. Table 5.3 above shows the microinstructions that can be executed using the repeat function and the ones that cannot.

The finite state machine (FSM) controls the nested looping and repeat functions. Three levels of nesting looping are supported in the base architecture. The repeat function is used to control the number of times a loop is repeated. There is a repeat counter for each nested loop that is loaded from the RPT_CNT field. The OFFSET field in the microcode is used to loop backwards in the loop when the LOOP_BACK opcode is executed. The RPT_CNT and OFFSET fields are used by the BRANCH opcode to increase the twos-complement offset range of the branch into the micro-ROM.

Figure 5.12 below shows a typical nested looping microcode sequence. In this sequence there are two nested loops and one conditionally executed branch loop. The nested loops execute 10 times before the branch instruction is executed. Note that the microinstructions are executed in parallel, resulting in zero-overhead loop and branch instructions. Once the conditional branch is not taken the JMP_HLT microinstruction is executed.

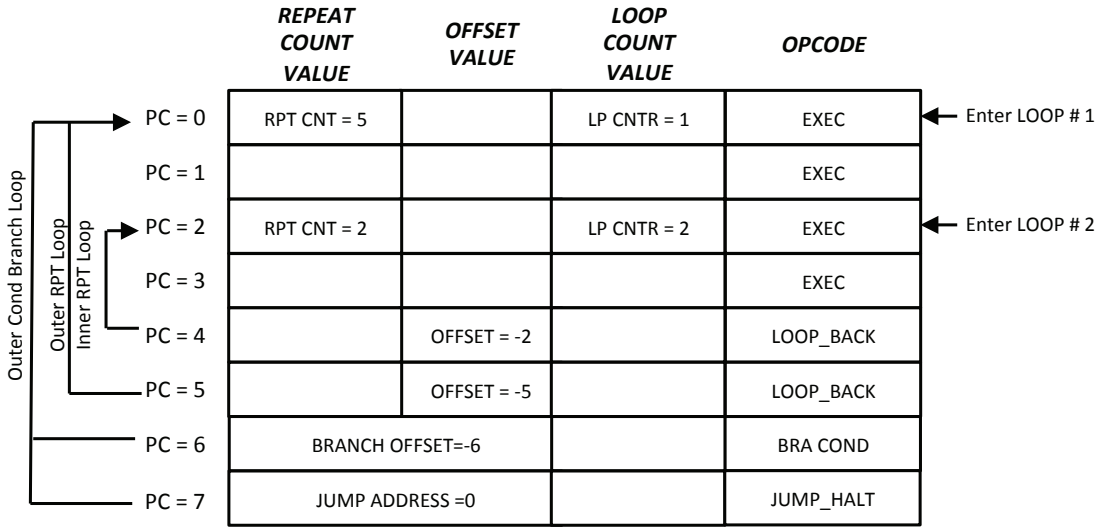


Figure 5.12: Typical nested looping microcode sequence

The JMP_HLT is a merged microinstruction that jumps to the address specified and halts the micro-engine to wait for the next event signal. The microcode engine is 'fired' when the new channel data is inserted into the Queued-Stack. Note: the micro-engine clocks are disabled during idle mode resulting in minimal power dissipation.

5.5.2 Microcode finite state machine

A finite state machine tracks the five possible states that the micro-engine can be operate in. These states are described below in Table 5.4.

Table 5.4: FSM operating modes

State	Operating mode
S0	WAITING FOR EVENT/FIRE SIGNAL
S1	NON-LOOPING EXECUTION STATE
S2	LOOP-1 (INNER MOST) EXECUTION STATE
S3	LOOP-2 EXECUTION STATE
S4	LOOP-3 (OUTER MOST) EXECUTION STATE

The state diagram for the FSM is show below in Figure 5.13.

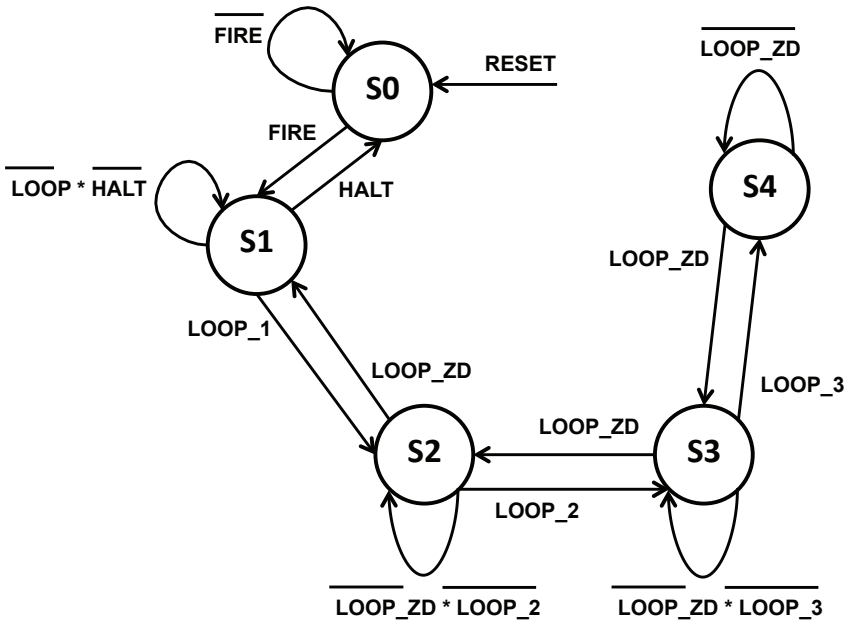


Figure 5.13: State diagram for micro-engine control

The micro-engine remains in State-S0 until the input queue receives all of the tokens or when an event occurs. In either case, a FIRE signal is issued which brings the

micro-engine out of low-power mode and into full execution mode (Figure 4.20). State-S1 is the normal operating mode when the machine is not executing in a nested loop. States S2, S3 and S4 track nested looping operation. State-S2 tracks execution in the outermost loop and State-S4 tracks execution in the innermost loop. The loops are entered sequentially from the outer loop to the inner loop and retire from the inner loop to the outer loop. Change of flow instructions (JUMP/BRANCH/WFE) cannot be executed in a nested loop nor can they enter a nested loop from outside of the loop.

There are three loop counters that track the number of passes through each loop. The counters are loaded with the repeat count value when the first instruction of a loop is executed. The loop counter number and the repeat count value are specified in the microinstruction as shown below in Figure 5.14 (PC = 0 and PC = 2). The repeat counters are 3 bits wide, which allows a maximum repeat count value of 8 per loop. The loop-back address is 7 bits wide, which provides the ability to loop backwards 0 to 64 memory locations. A single instruction can be repeated 8 times by specifying a repeat count of 0x00 and loopback address of 0x00. A single instruction can be repeated 512 times by the following sequence of five microinstructions shown in Figure 5.14.

	REPEAT COUNT VALUE	***OFFSET VALUE***	***LOOP COUNT VALUE***	***OPCODE***	
PC = 0	RPT CNT = 0		LP CNTR = 1	NOP	← Enter LOOP # 1
PC = 1	RPT CNT = 0		LP CNTR = 2	NOP	← Enter LOOP # 2
PC = 2	RPT CNT = 0	OFFSET = 0	LP CNTR = 3	LOOP_BACK & EXEC	← Enter LOOP # 3
PC = 3		OFFSET = -2	LP CNTR = 2	LOOP_BACK	
PC = 4		OFFSET = -4	LP CNTR = 1	LOOP_BACK	

Figure 5.14: Nested looping/repeat example

Note that setting the repeat count value to zero (8 modulo 2) causes an underflow condition before the zero-detect is issued resulting in a full count of 8 operations.

5.5.3 Microcode storage memory

As mentioned above, the microcode storage can be implemented using a low power read-only-memory (ROM) [55] based or writable-control-store (WCS) memory [56]. In either case the clocks to the storage element are controlled by the FSM. For non-looping repeat functions, the latched microcode word is accessed instead of accessing the memory element. This provides additional energy savings as it eliminates pre-charge clocking energy and memory access power. The WCS is loaded via the scan test unit interface and is used in systems where overlaying of microcode is needed due to the size of the code or for debugging microcode before it is committed to ROM. Figure C.3 in Appendix C shows the size difference between a ROM and a WCS.

5.5.4 Actor/Event queue

The microcode engine supports an Actor/Event queue that is used to store recirculated actors and/or asynchronous events. Events are normally generated from external sources such as timers, exceptions and interrupts. This is similar to the SNAP processor [15] as shown above in Figure 2.8. Recirculating actors provides the capability to preload a sequence of operations actors and have them execute until a break occurs. Typically a break condition occurs when new token data is needed. The Wait-For-Event instruction will cause a break condition.

Figure 5.15 below shows the block diagram of the Actor/Event queue and how it is integrated into the microcode address generator.

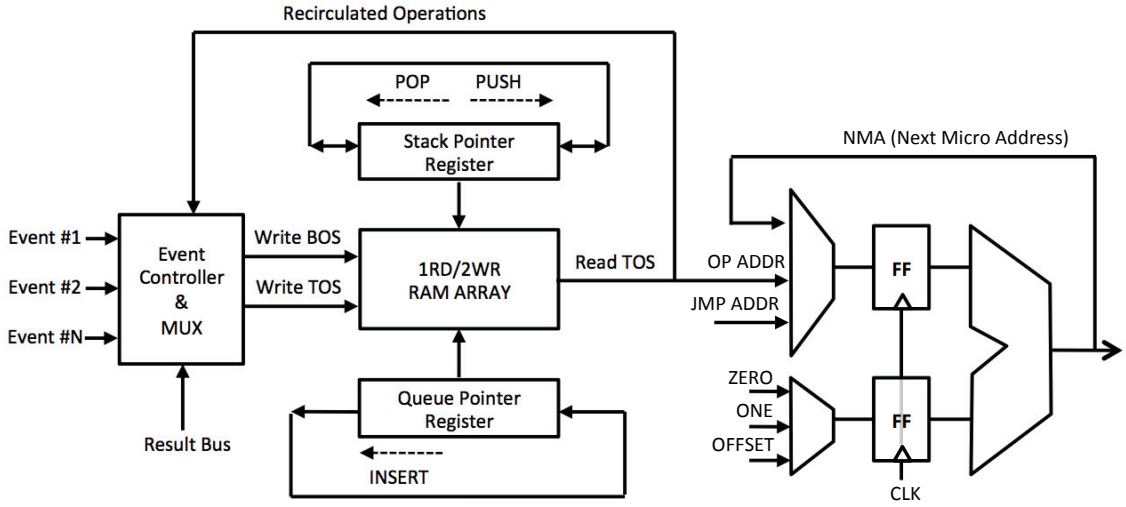


Figure 5.15: Block diagram of the Actor/Event queue

The Actor/Event queue contains entry points (micro-addresses) into the microcode memory. There are two types of entries in the queue: 1) those that recirculate ad infinitum or 2) those that are squashed from the queue once they execute. Asynchronous events such as timer interrupts are Type-2 entries and "actors" are Type-1 entries.

The event controller generates entry points for the asynchronous events being processed. These asynchronous event entry points can be entered on the "Top of Stack" (TOS) if they need to be executed immediately or entered into the "Bottom of Stack" (BOS) if it is not critical when they are executed. Type-2 entries are removed from the queue once they are executed. Type-1 entries are recirculated back into the Actor/Event queue by re-writing them to the BOS.

The FSM in the microcode engine controls the flow of addresses/entry-points to the microcode ROM/WCS. There are three main address sources:

1. Next Micro-Address (NMA)
2. The JUMP Immediate Address
3. The Actor/Event Queue Address

Each address can be modified by three offsets:

1. Zero value (no offset)
2. A value of one for basic incrementing
3. An OFFSET value supplied by the microcode word being executed.

The Actor/Event queue can be loaded from the result bus or via a scan chain controlled by the debug unit. Loading the queue from the result bus provides the ability to use the CLT to look up entry-points and writing them to the queue. The entry-points can be dynamically changed based on algorithmic or Fuzzy Logic based heuristics as the sensor system ages. The debug unit will copy the contents of the queue to external flash memory as needed to preserve the new operating state of the CSP.

Chapter 6. DPE Microprogramming

6.1 Overview

As described in Section 1.3, the CSP is a Synchronous Dataflow (SDF) machine. The DPE is considered an "actor" that processes tokens and generates tokens. Actors can be composed of multiple actors as long as the rules described in Section 1.4 are adhered to. A single DPE can contain one actor or many actors, each of which is implemented in a single microcode routine. Each microcode routine (actor) is terminated using a Jump-Halt or a Wait-for-Event microinstruction. The DPE waits for an event before executing the next actor. The events come from two sources: the Actor/Event queue (Section 5.5.4) or the next tranche of tokens generating a fire signal (Section 5.5.2).

It was an easy decision to use a microcoded implementation to optimize power in the CSP. Each microcode routine (actor) is deterministic which satisfies one of the premises of an SDF machine. This provides the ability to deterministically schedule all transactions that the CSP will perform (another premise of an SDF machine). This also provides the ability to precisely time-stamp all tokens that pass through the CSP (see Section 4.3.2.1).

The CSP does support non-deterministic operation by virtue of the fact that asynchronous events can be inserted into the DPE Actor/Event queue. The ability to dynamically reconfigure the CSP can also cause non-deterministic operation. In both cases the effect can be that the CSP cannot process tokens at the rate the PPU is issuing them to the FSU and/or the DPE. Recall that the CSP supports both Push/Pull modes as described in Section 1.4. The Pull-Mode is recommended for non-deterministic operation, as it will provide backpressure on all tokens that are being generated by the PPU and the FSU. The backpressure can be sensed by the debug unit, which can modify

the sampling rate of the PPU to match the throughput of the SDF machine(s). Of course non-deterministic operation requires dynamic characterization of the machine in order to determine the maximum time-stamp value (Equation 4.1).

The Actor/Event queue can be considered to be program storage sans the immediate data capability that most ISA's provide. A deterministic Actor/Event queue (one without asynchronous events) is considered a collection of actors that the DPE will execute in a deterministic order for every tranche of tokens that are received. The WFE microinstruction is used to halt the machine and wait for the next Actor/Event queue entry to be decoded as an entry point in the microcode memory.

Microcode programming has always been considered to be tedious and that is definitely the case for the DPE. The use of a stack-based machine further complicates the programming paradigm. This is indicative of what it requires to optimize power for low energy embedded applications while maintaining the ability to dynamically reprogram the flow of the machine. The remainder of this chapter discusses how to micro-program the DPE.

Two programming environments were reviewed as a part of this research. These are described below in Section 6.2. Microcode programming has long since fallen out of favor among programmers and the coding tool choices are very limited. The final solution was to use a simple spreadsheet based micro-assembler further validating the fact that spreadsheets can do almost anything.

Section 6.3 below describes the details of the microcode word and Section 6.4 describes the microcode programming syntax. This syntax is unique because of the addition of the stack operations that must be performed for each microcode operation.

6.2 Programing environment

The limitation of many new machine architectures is the availability of a usable and stable programming environment. There are numerous freeware tools to build assemblers and compilers for classical register-based machines but virtually none for stack-based machines. These tools require an extensive overhaul when trying to make them work with a stack-based machine. There are stack-based languages like Forth [58] that compile down to machine code that runs on a register-based machine where a software based stack model is used. In looking at the opcode generation capability of the Forth compiler it was obvious that it was not suited for a Queued-Stack synchronous data-flow model.

The software implementation of the actors is the most challenging because the DPE is a microcoded machine. Ideally this would be accomplished using a high level language that is compiled into microcode routines. As mentioned earlier, this would require a large-scale code generation development effort that is beyond the scope of this research. There are approximately 30 actors that are needed to support the class of workloads that the CSP is designed for. Combinations of these basic actors are used to build more complicated ones. That said, it was decided that the implementation of the actors would be accomplished using a microcode assembler.

Two microprogramming environments were investigated as part of this research. These are described in the following sub-sections and include:

1. Machine independent microcode programming languages [59]
2. Microsoft Excel spreadsheet based micro-assembler.

6.2.1 Microcode Program Languages (MPL)

These languages were developed during the early days of micro-programmable machines to implement microcode using control constructs such as IF-THEN-ELSE, DO-WHILE, etc. These MPLs were generally machine specific, however, machine independent languages were proposed to provide portability from one generation to the next [59]. These languages were compiled into microcode-assembler opcodes/operands and then passed through a machine specific assembler to generate the binary files that can be loaded into a WCS or ROM. After looking for adequate off-the-shelf tools it was determined that the applicability of MPLs for DPE microprogramming was very limited primarily due to the fact that they do not support stack-based machines.

6.2.2 Microsoft Excel spreadsheet assembler

After reviewing numerous microcode assemblers it became obvious that all of them were based on a classic register file based datapath implementation. The distributed Queued-Stack implementation in the DPE with its multiple operation mechanism required extensive rewrite of these assemblers. The most optimal solution was to use a tool that has the ability to do table lookup functions with a broad range of IF-THEN control functions and a bit manipulation functions to build various data fields used in the microcode. Excel provides this capability. Interestingly enough an Excel spreadsheet can also be used to model an SDF algorithm, as it is a reactive system in itself. This would be an interesting Master's project as follow-on to this research. A detailed description of the microcode assembler is presented in Appendix B.

6.3 Microcode Field Descriptions

The microcode word is 96-bits wide and divided into five control blocks: micro-engine control, datapath control, Queued-Stack control for write-back operations, Queued-Stack control for next operand selection operations, SFU and I/O control as shown below. The fields are described in detail in the next four subsections.

Micro-engine control:

IMMED DATA <95:80>	BRANCH OFFSET <79:73>	LOOP COUNTER NUM <72:71>	REPEAT COUNT <70:68>	UCODE OP <67:64>

Datapath control:

SHFT MODE B <45>	SHFT MODE A <44>	SHFT B <43:39>	SHFT A <38:34>	SHFT TC B <33>	SHFT TC A <32>	MULT <31>	SAT <30>	DP EN <29>	ADD/SUB <28>

Queued-Stack control for write-back operations:

QS2 BUS SEL <23:22>	QS1 BUS SEL <21:20>	WB MUX SEL <18:17>	RQS CTL <9:6>	RD RQS <4>	RQS BUS MUX <3>

Queued-Stack control for next operand selection:

READ TOS QS2 <63>	READ TOS QS1 <62>	READ BOS QS2 <61>	READ BOS QS1 <60>	QS2 CTL <59:56>	QS1 CTL <55:52:>	B_BUS MUX <51:50>	A_BUS MUX <49:48>	CHAN MUX QS2 <47>	CHAN MUX QS1 <46>

SFU & I/O control:

LOGIC OP <16:14>	SFU OP <13:11>	WR SFU LAT <10>	WR WB LAT <5>	FIFO WE <2>	GPIO WE <0>

6.3.1 Micro-engine control

Table 6.1 below shows the bit field assignments for the micro-engine control. The five fields control the sequencing of all microinstructions and are described in Section 5.5.

Table 6.1: Micro-engine control bit field assignment

IMMEDIATE DATA AND JUMP ADDRESS <95:80>	LOOP & BRANCH OFFSET <79:73:>	LOOP NUMBER <72:71>	REPEAT COUNT <70:68>	MICRO OPCODE <67:64>
-32768 TO +32767		0-3	0-7	EXEC
-32768 TO +32767		0-3	0-7	EXEC_WB_EQ
-32768 TO +32767		0-3	0-7	EXEC_WB_GT
-32768 TO +32767		0-3	0-7	EXEC_WB_LT
				WFE
-512 TO +511				JMP
-512 TO +511				JMP_HLT
-32768 TO +32767	-64 TO 0	0-3		LOOP_BACK
-32768 TO +32767	-64 TO +63			BRA
-32768 TO +32767	-64 TO +63			BR_WB_EQ
-32768 TO +32767	-64 TO +63			BR_WB_GT
-32768 TO +32767	-64 TO +63			BR_WB_LT
-32768 TO +32767	-64 TO +63			BR_SFU_GT
-32768 TO +32767	-64 TO +63			BR_SFU_LT
-32768 TO +32767	-64 TO +63			BR_SFU_EQ
				TXFR

6.3.1.1 Immediate data – Bit field <95:80>

This bit field is used to provide immediate data to the datapath units. Figure 6.1 below shows the datapath block diagram. There are three consumers of immediate data: 1) the datapath input multiplexor, 2) the Write-Back Unit (WBU) and 3) the Special Function Unit (SFU). The immediate data can be sign extended using control bit <19>. This immediate data field is also used as the two's complement address for the two jump instructions. Consequently jump instructions cannot use the immediate data field for datapath instructions. The branch-always opcode (BRA) can be used for unconditional jumps where the immediate data field is needed by the microinstruction.

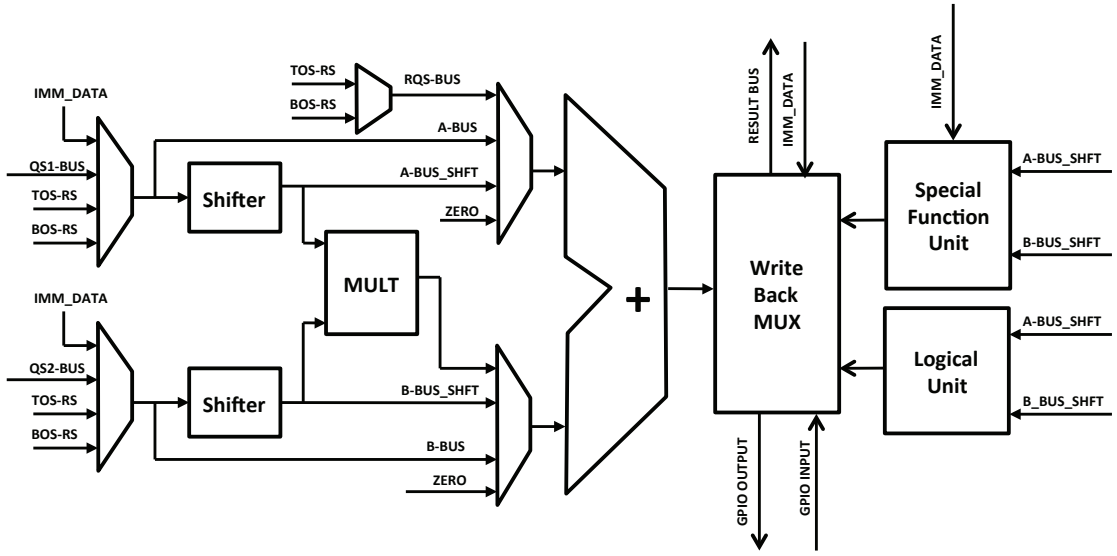


Figure 6.1: Datapath block diagram

6.3.1.2 Branch offset – Bit field <79:73>

This field provides the two's complement offset for the branch instructions and the negative offset for the loop back instruction. The field is limited to 7 bits for the

current implementation of the DPE, providing a -64 to +63 offset for branch instructions and the loop back instruction.

6.3.1.3 Loop number – Bit field <72:71>

This field indicates which loop number the micro-engine should enter. This provides the ability to do 3-deep nested looping (see Section 5.5) and is used in conjunction with the repeat count field. Figure 6.2 below shows that state diagram of the micro-engine as it tracks which loop is active.

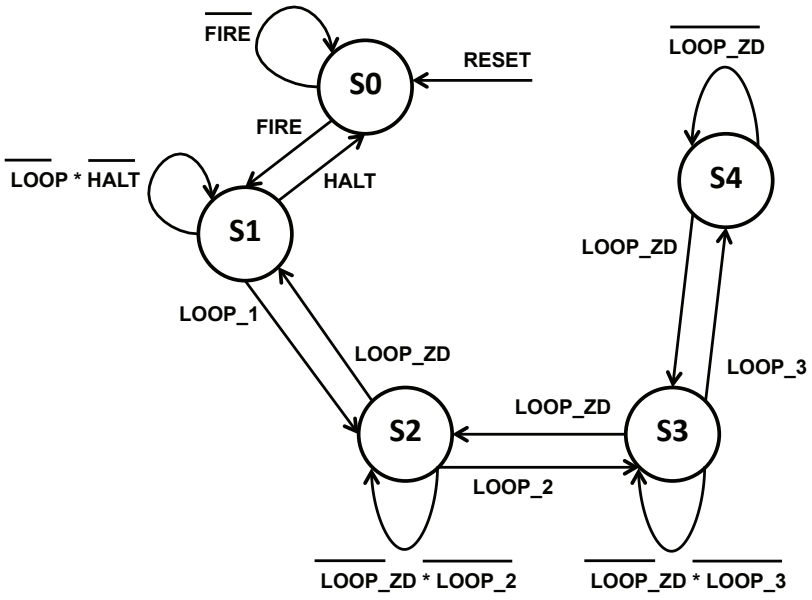


Figure 6.2: Micro-engine state diagram

6.3.1.4 Repeat count – Bit field <70:68>

The repeat count field is used for repeating individual microinstructions or for repeating nested loops. The 3-bit field provides the ability to repeat a loop or an

instruction 1 to 8 times. A single micro-instruction can be repeated 512 times by specifying 3 levels of nesting and a repeat count of 0 (8 modulo 2) for each loop.

6.3.1.5 Micro-engine opcode – Bit field <67:64>

The 4-bit micro-engine opcode field provides up to 16 micro-operations that the micro-engine can execute. Table 6.2 below describes the 16 opcodes that the current implementation of the DPE can execute.

Table 6.2: Micro-Engine Operation Codes

OPCODE	RPT	MICRO-OPERATION
EXEC	Y	UNCONDITIONAL EXECUTION
EXEC_WB_EQ	Y	EXECUTION IF WRITEBACK == REFERENCE DATA
EXEC_WB_GT	Y	EXECUTION IF WRITEBACK > REFERENCE DATA
EXEC_WB_LT	Y	EXECUTION IF WRITEBACK < REFERENCE DATA
WFE	N	HALT AT PC+1; WAIT-FOR-EVENT SIGNAL
JMP	N	JUMP TO ADDRESS SPECIFIED IN IMMED_DATA FIELD
JMP_HLT	N	JUMP TO ADDRESS SPECIFIED IN IMMED_DATA FIELD THEN HALT
LOOP_BACK	Y	LOOP BACK FOR LOOP # (NEGATIVE OFFSET ONLY)
BRA	N	BRANCH UNCONDITIONALLY
BR_WB_EQ	N	BRANCH IF WRITEBACK == REFERENCE DATA
BR_WB_GT	N	BRANCH IF WRITEBACK > REFERENCE DATA
BR_WB_LT	N	BRANCH IF WRITEBACK < REFERENCE DATA
BR_SFU_GT	N	BRANCH IF SFU RESULT > REFERENCE DATA
BR_SFU_LT	N	BRANCH IF SFU RESULT < REFERENCE DATA
BR_SFU_EQ	N	BRANCH IF SFU RESULT == REFERENCE DATA
TXFR	N	TRANSFER OUTPUT FIFO DATA TO CHANNEL

6.3.2 Datapath control

The datapath control is composed of multiplexor control and arithmetic function control. Figure 6.3 below shows a block diagram of the multiplexors and the arithmetic units.

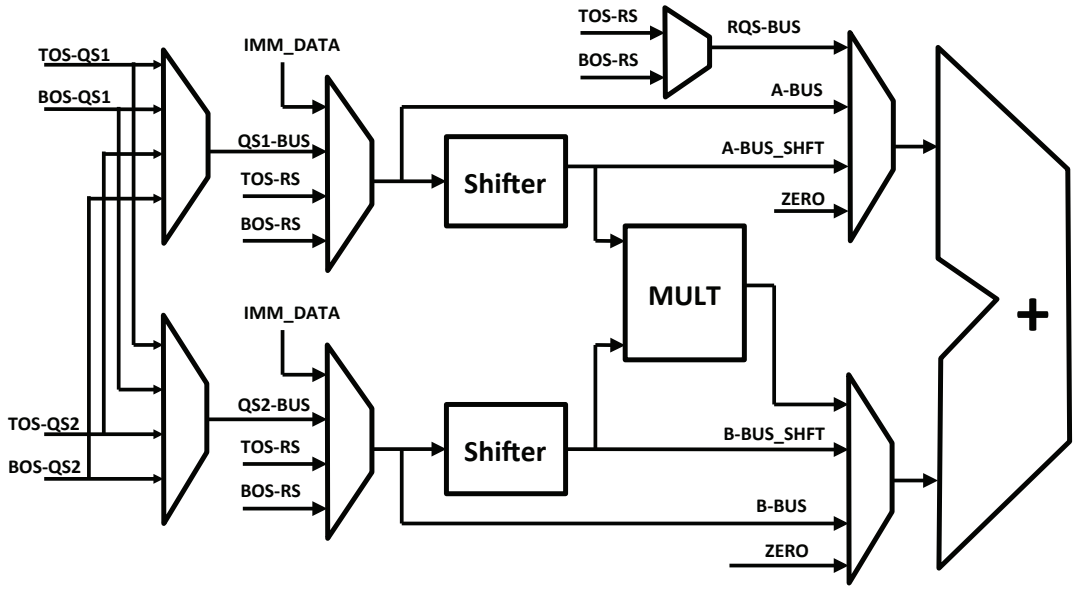


Figure 6.3: Block diagram of the multiplexors and the arithmetic units

6.3.2.1 Multiplexor control for datapath

The first set of multiplexors steer the output data from the IQS1 and IQS2 Queued-Stack elements. The second set multiplexes the output of the RQS unit, immediate data and the output from the first set of multiplexors. The last set multiplexes the output from the second multiplexor, the output from the shifter, the output from the multiplier, an additional output from the RQS unit and lastly a ZERO value. Table 6.3 below shows the bit field assignments for the datapath multiplexor control.

Table 6.3: Datapath multiplexor control bit field assignments

ENCODE	QS2 BUS <23:22>	QS1 BUS <21:20>	B_BUS MUX <51:50>	A_BUS MUX <49:48>	B-MUX SEL <27:26>	A-MUX SEL <25:24>
00	TOS_QS2	TOS_QS1	QS2_BUS	QS1_BUS	ZERO	ZERO
01	BOS_QS2	BOS_QS1	IMM_DATA	IMM_DATA	B_BUS	A_BUS
10	TOS_QS1	TOS_QS2	TOS_RQS	TOS_RQS	B_BUS_SHFT	A_BUS_SHFT
11	BOS_QS1	BOS_QS2	BOS_RQS	BOS_RQS	MULT	RQS_BUS

6.3.2.2 Shifter control

The shifter control bits are shown below in Tables 6.4 and 6.5. The shifter supports arithmetic and barrel shifts. When the control signal SHIFT_TC = 0, the shift value is interpreted as an unsigned positive number, and the shifter performs only left shift operations. When SHIFT_TC = 1, the shift value is a two's complement number, with a negative coefficient performing a right shift and a positive coefficient performing a left shift. The input data, DATA_IN, is interpreted as an unsigned number when DATA_TC=0. When DATA_TC=1, DATA_IN is interpreted as a signed number, and a sign extension is performed for right arithmetic shift operations.

Table 6.4: Datapath control for A-BUS shifter

SHIFT MODE A <44>	SHIFT A <38:34>	SHIFT TC A <32>	DATA TC <19>	MSB OP	OPERATION
1	0-31	0	-	-	LEFT ARITHMETIC SHIFT, LOGIC 0 PADDING
1	0-31	1	-	0	LEFT ARITHMETIC SHIFT, LOGIC 0 PADDING
1	0-31	1	0	0	RIGHT ARITHMETIC SHIFT, LOGIC 0 PADDING
1	0-31	1	1	1	RIGHT ARITHMETIC SHIFT, SIGN EXT. PADDING
0	0-31	0	-	0	LEFT BARREL SHIFT
0	0-31	1	-	1	RIGHT BARREL SHIFT

Table 6.5: Datapath control for B-BUS shifter

SHIFT MODE B <45>	SHIFT B <39:43>	SHIFT TC B <33>	DATA TC <19>	MSB OP	OPERATION
	0-31	0	-	-	LEFT ARITHMETIC SHIFT, LOGIC 0 PADDING
1	0-31	1	-	0	LEFT ARITHMETIC SHIFT, LOGIC 0 PADDING
1	0-31	1	0	0	RIGHT ARITHMETIC SHIFT, LOGIC 0 PADDING
1	0-31	1	1	1	RIGHT ARITHMETIC SHIFT, SIGN EXT. PADDING
0	0-31	0	-	0	LEFT BARREL SHIFT
0	0-31	1	-	1	RIGHT BARREL SHIFT

6.3.2.3 Multiplier control

The multiplier control bits are shown below in table 6.6. The multiplier supports both signed and unsigned operations.

Table 6.6: Datapath control for Multiplier

MULT MODE <31>	DATA TC <19>	OPERATION
0	-	NO OPERATION
1	0	UNSIGNED MULTIPLY
1	1	SIGNED MULTIPLY

6.3.2.4 Adder control

The adder control bits are shown below in table 6.7. The adder supports signed/unsigned, saturating/non-saturating, and add/subtract. The default is to use signed saturating arithmetic for all operations.

Table 6.7: Datapath control bit field assignments for ADD/SUB

ADD/SUB <28>	SAT <30>	DATA TC <19>	OPERATION
0	0	0	UNSIGNED ADD
0	0	1	SIGNED ADD
0	1	0	SATURATED UNSIGNED ADD
0	1	1	SATURATED SIGNED ADD
1	0	0	UNSIGNED SUBTRACT
1	0	1	SIGNED SUBTRACT
1	1	0	SATURATED UNSIGNED SUBTRACT
1	1	1	SATURATED SIGNED SUBTRACT

6.3.2.5 Signed/Unsigned operation

The TC control bit <19> is used to set all data path elements to operate in either signed or unsigned operation. The PPU converts all token data to the appropriate format based on the operating mode of the CSP.

6.3.3 Queued-Stack control

Table 6.8 below shows the microcode encoding for IQS1, IQS2 and RQS fields. A number of the operations are described in the following sub-sections. There are 15 basic operations: PUSH, POP, INS, TOP, BOT and WR/NW. PUSH and POP are stack operations that advance the pointers. INS is a queue operation where the operand is inserted into the queue (bottom of the stack) and advances the pointers. TOP/BOT are operations that write operands to the top and bottom of the stack without advancing the pointers. WR/NW modify the basic operations to either write (WR) or not-write (NW) the operand as specified in the write-back column.

Table 6.8: Queued-Stack operation encoding

ENCODE	IQS2_CTL <59:56>	IQS1_CTL <55:52>	RQS_CTL <9:6>
1001	PUSH	PUSH	PUSH
0100	POP	POP	POP
0111	POP_WR	POP_WR	POP_WR
0010	INS	INS	INS
0011	INS_NW	INS_NW	INS_NW
1100	PUSH_NW	PUSH_NW	PUSH_NW
1101	TOP	TOP	TOP
0001	BOT	BOT	BOT
1110	TOP_BOT	TOP_BOT	TOP_BOT
1011	PUSH_INS	PUSH_INS	PUSH_INS
0101	POP_BOT	POP_BOT	POP_BOT
0110	POP_INS	POP_INS	POP_INS
1000	POP_WR_BOT	POP_WR_BOT	POP_WR_BOT
1010	PUSH_NW_BOT	PUSH_NW_BOT	PUSH_NW_BOT
1111	TOP_INS	TOP_INS	TOP_INS
0000	NOP	NOP	NOP

6.3.3.1 PUSH operation

Figure 6.4 below shows the PUSH operation on a Queued-Stack after reset where TOS pointer is advanced before the write operation.

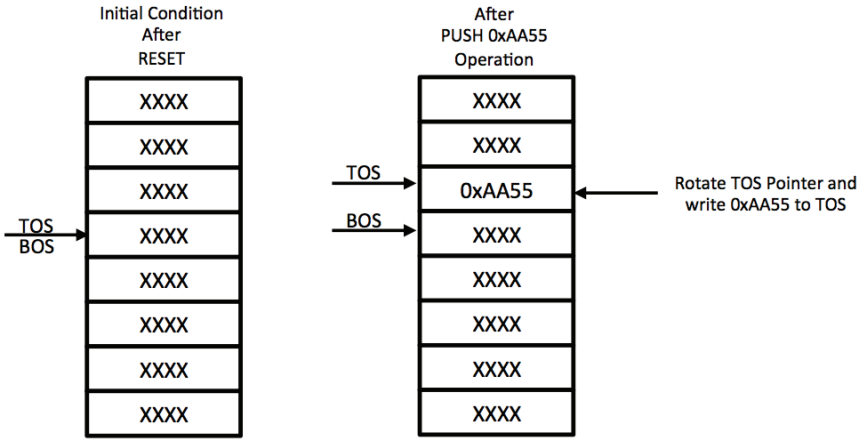


Figure 6.4: PUSH operation

6.3.3.2 BOT operation

Figure 6.5 shows the BOT operation that writes 0x3344 to the BOS without advancing the pointer.

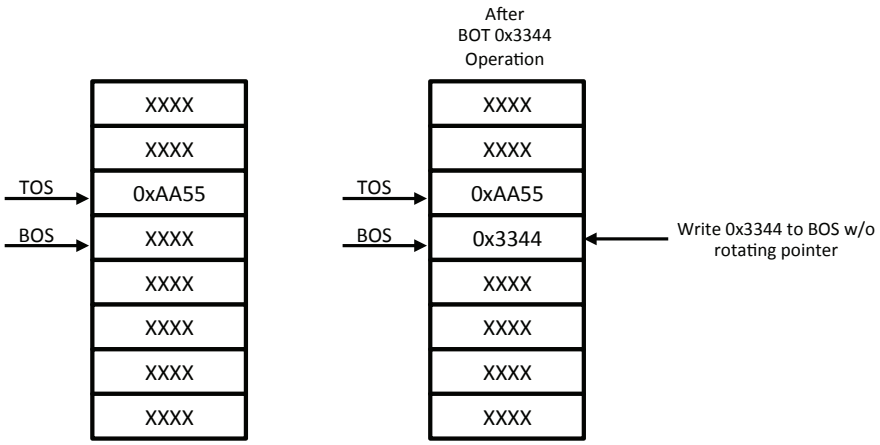


Figure 6.5: BOT operation

6.3.3.3 PUSH_INS operation

Figure 6.6 shows the PUSH_INS operation where 0xABCD is written to the TOS and the BOS after the pointers have been advanced appropriately.

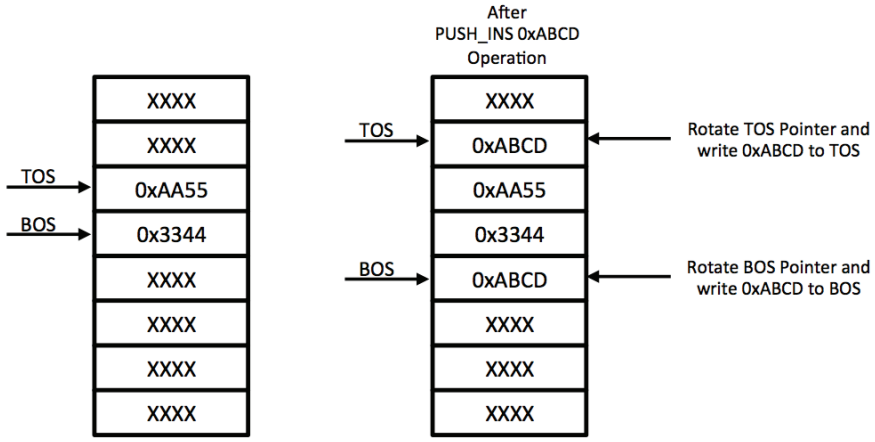


Figure 6.6: PUSH_INS operation

6.3.3.4 TOP_BOT operation

Figure 6.7 below shows the TOP_BOT instruction that writes 0x1234 to both the TOS and the BOS without advancing the pointers.

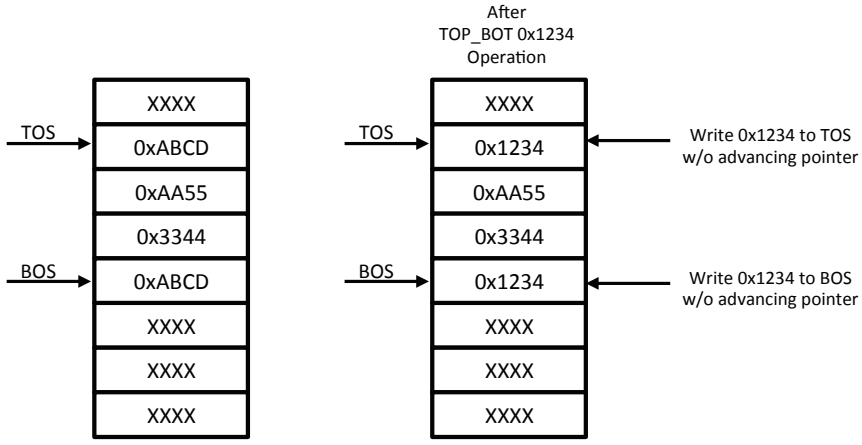


Figure 6.7: TOP_BOT operation

6.3.3.5 POP operation

Figure 6.8 below shows the POP operation that simply advances the TOS pointer appropriately.

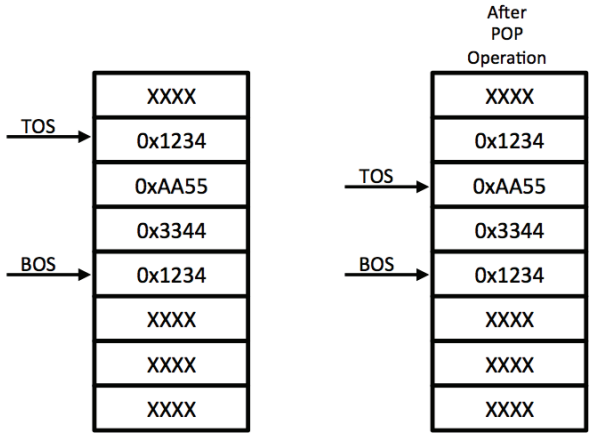


Figure 6.8: POP operation

6.3.3.6 POP_WR operation

Figure 6.9 below shows the POP_WR operation where the TOS pointer is advanced and 0x5678 is written to the new TOS location.

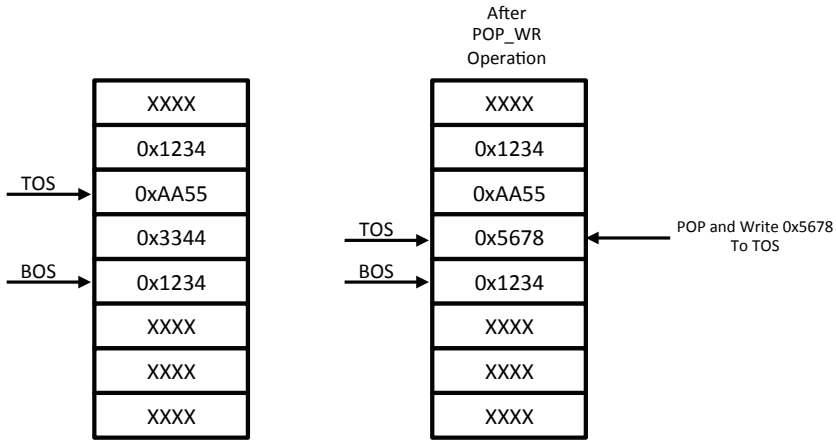


Figure 6.9: POP_WR operation

6.3.3.7 PUSH_NW operation

Figure 6.10 below shows two sequential PUSH_NW instructions where the TOS pointer is advanced without writing the data.

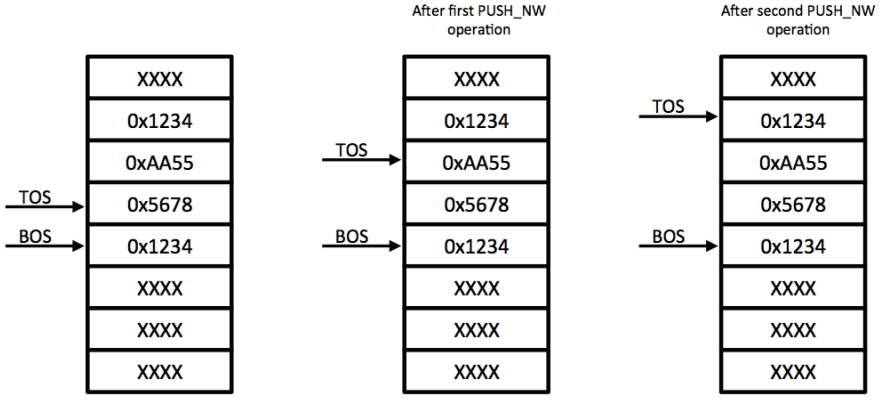


Figure 6.10: PUSH_NW operation

6.3.3.8 TOP_INS operation

Figure 6.11 below illustrates the TOP_INS operation where the TOS is written without advancing the TOS pointer while BOS pointer is advanced and the BOS is written with a value of 0xDEAD.

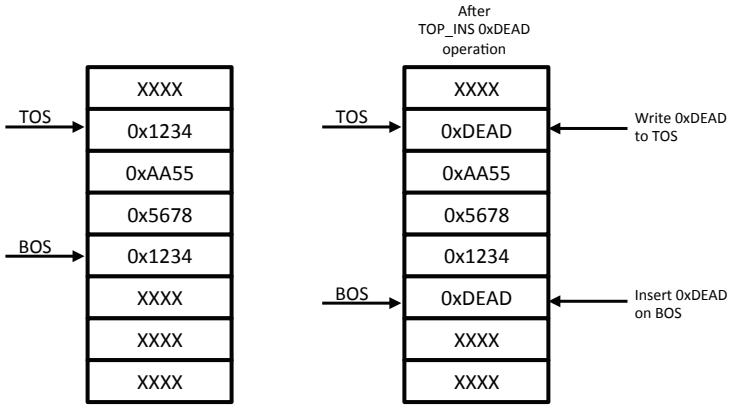


Figure 6.11: TOP_INS operation

6.3.4 Logical Unit, Special Function Unit and I/O Control

The current implementation of the DPE supports a Boolean Logic Unit (BLU) and a special function unit that is used for Fuzzy Logic operations.

6.3.4.1 Logical operations

The DPE supports 8 logical operations as shown below in Table 6.9. Recall that the inputs to the logical unit are from the output of the shifter (Figure 5.5). This provides basic bit field extraction capability.

Table 6.9: Logical operations

OPCODE	DESCRIPTION
A_NOT	INVERT VALUE ON A_INPUT
B_NOT	INVERT VALUE ON B_INPUT
AND	A_INPUT AND B_INPUT
NAND	A_INPUT NAND B_INPUT
OR	A_INPUT OR B_INPUT
NOR	A_INPUT NOR B_INPUT
XNOR	A_INPUT XNOR B_INPUT

6.3.4.2 SFU operations

The SFU implemented in the current version of the DPE is designed to accelerate Fuzzy Logic routines, specifically MIN/MAX functions. Table 6.10 below shows the eight MIN/MAX operations that are supported by the SFU. The REF_DATA input is a constant value that is stored in a register located in the SFU. It is used for threshold detection algorithms.

Table 6.10: MIN/MAX operations

ENCODE	OPCODE	OPERATION
000	MIN_A_B	MIN(SHFT_A_BUS, SHFT_B_BUS, SHFT_B_BUS)
001	MIN_A_REF	MIN(SHFT_A_BUS, REF_DATA, REF_DATA)
010	MIN_B_REF	MIN(SHFT_B_BUS, REF_DATA, REF_DATA)
011	MIN_A_B_REF	MIN(SHFT_A_BUS, SHFT_B_BUS, REF_DATA)
100	MAX_A_B	MAX(SHFT_A_BUS, SHFT_B_BUS, SHFT_B_BUS)
101	MAX_A_REF	MAX(SHFT_A_BUS, REF_DATA, REF_DATA)
110	MAX_B_REF	MAX(SHFT_B_BUS, REF_DATA, REF_DATA)
111	MAX_A_B_REF	MAX(SHFT_A_BUS, SHFT_B_BUS, REF_DATA)

The MIN/MAX logic block in the SFU has three inputs: A_BUS_SHFT, B_BUS_SHFT and REF_DATA as seen below in Figure 6.12. There are two SFU operations that require 3 inputs while the remaining six use only two inputs. In those cases the same input is used for two of the three inputs.

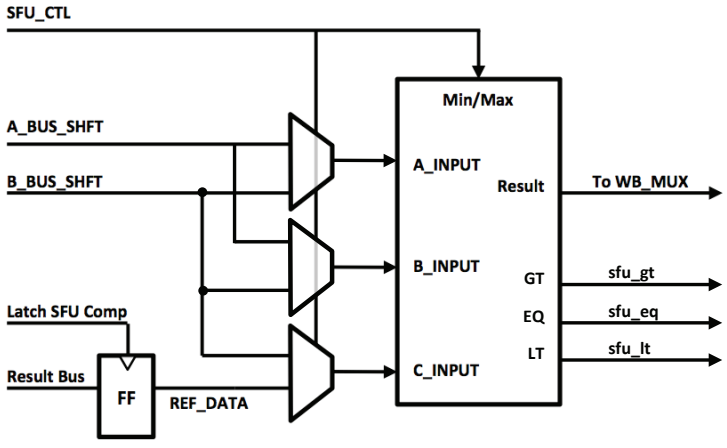


Figure 6.12: Block diagram of MIN/MAX logic

6.4 Microcode programming syntax

There are five operational fields that can be specified for each microinstruction:

1. Arithmetic operations
2. Non-arithmetic operations
3. Write back operations
4. Next-operand operations
5. Micro-engine operations

In most cases the fields do not have to be completely specified. The microcode assembler will set the appropriate fields to the correct state. For example, if a datapath operation is not specified the DP_ENABLE signal will be negated. The syntax for the operations follows the stack based Reverse Polish Notation (RPN) where the source data is presented before the operation [57]. Each operation is delineated using a vertical bar |.

6.4.1 Arithmetic operations

The datapath supports up to 3 concurrent arithmetic operations: shift(scale), multiply and add/subtract. As can be seen in Figure 6.3 above, the two shifters feed the MULT unit and the ADD/SUB unit, while the MULT unit feeds only the B-BUS on the ADD/SUB unit. The shifter instruction is specified first if the shifter is modifying the values being inputted into the MULT unit. The datapath supports both signed and unsigned operations. The same mnemonics apply to both modes of operation.

The basic instruction format for the shifter is:

```
| SOURCE <MNEMONIC=COUNT> |
```

There are four shifter operations: LSR, ASR, LSL and ASL that can be specified.

Examples of shifter instructions for the shifter on the A_BUS:

```
| TOS_QS1 LSLA=0x1A |
| TOS_QS2 ASRA=0x15 |
| BOS_QS2 ASRA=0x18 |
| BOS_RQS ASRA=0x31 |
| IMM_DATA=0x2F LSRA=0x0A |
```

The same instructions are applicable for the shifter on the B_BUS

```
| TOS_QS1 LSLB=0x05 |
| TOS_QS2 ASRB=0x17 |
| BOS_QS1 LSRB=0x1F |
| TOS_RQS ASRB=0x2F |
| IMM_DATA=0x2F LSRB=0x0A |
```

The basic instruction formats for the MULT unit are:

```
| SOURCE_A_BUS SOURCE_B_BUS <MNEMONIC> |     // IMPLIES SHIFT=0x00
| <SHIFTER_A_OP>  <SHIFTER_B_OP>  <MNEMONIC> |   // IMPLIES DEFAULT SOURCE
| SOURCE <SHIFTER_A_OP> | SOURCE <SHIFTER_B_OP> | <MNEMONIC> |
```

Code examples multiplier instructions where the shifter units are not active:

```
| TOS_QS1 TOS_QS2 MULT |
| TOS_QS2 TOS_QS1 MULT |
| BOS_QS1 BOS_QS2 MULT |
| BOS_QS2 BOS_QS1 MULT |
| TOS_RQS IMM_DATA=0x4A MULT |
| BOS_RQS TOS_RQS MULT |
| IMM_DATA=0x66 TOS_QS2 MULT |
| IMM_DATA=0x12 IMM_DATA=0x12 MULT |
| IMM_DATA=0x9A BOS_RQS MULT |
```

Code examples of multiplier instructions where the shifter is active:

```
| TOS_QS1 LSLA=12 | TOS_QS2 LSLB=15 | MULT |
| IMM_DATA=0x7E LSLA=21 | TOS_QS1 LSLB=11 | MULT|
| BOS_QS1 LSLA=0x1A | TOS_QS2 ASRB=0x17 | MULT |
```

The instruction formats for the ADD/SUB unit are:

```
| SOURCE_A_BUS SOURCE_B_BUS <MNEMONIC> |
| SOURCE_A_MUX SOURCE_B_MUX <MNEMONIC> |
```

Code examples of add/subtract instruction include:

```
| ZERO ZERO  ADD |
| A_BUS B_BUS_SHFT  SAT_ADD |
| RQS_BUS=BOS_RQS MULT_BUS SAT_SUB |
| TOS_QS1 IMM_DATA=0x11 SUB |
```

A typical scaled-multiply-saturating-accumulate microinstruction would be written as:

```
1:
| TOS_QS1 LSRB=3                          // SCALE (DIVIDE BY 8)
| IMM_DATA=0x17 LSLB=5                    // SCALE (MULTIPLY 17 x 32)
| MULT                                    // MULTIPLY SCALED VALUES
| RQS_BUS=TOS_RQS MULT_BUS SAT_ADD |:1    // SATURATE ADD
```

6.4.2 Non-arithmetic operations

There are number of special function instructions which are supported by the current implementation of the DPE. These include Boolean logic operations and special function unit (SFU) operations.

6.4.2.1 Boolean logic operations

The inputs to the Boolean Logic Unit are from the output of the shifter. As with previous instructions the inputs to the shifters must be specified. Additionally, the source of data on the write-back bus must be specified (see Section 6.4.3 below). The supported Boolean logic operations are described above in Section 6.3.4.1

The syntax for Boolean logic operations is:

```
| SOURCE_A_BUS SOURCE_B_BUS <MNEMONIC> |       // IMPLIES SHIFT=0x00
| <SHIFTER_A_OPERATION> | <SHIFTER_B_OPERATION> | <MNEMONIC> |
```

Here is an example of a microcode word that performs a basic logical NAND operation and writes the result to the QS2 top of stack:

```
1:
| TOS_QS1                                 // GET ONE OPERAND FROM TOS_QS1
  IMM_DATA=0x55                           // GET FIXED VALUE
```

```
  NAND                                  // TOS_QS1 NAND 0x55 **
| WB=LU                                 // SELECT LOGICAL UNIT AS SOURCE TO WB
| POP_QS1                               // SELECT NEXT OPERAND
| PUSH_QS2 |:1                          // PUSH RESULT ON TOS_QS2
```

** Note: operations can be specified across multiple lines and comments can be inserted in between for clarity.

Here is an example of a more complex logical operation where the GPIO port is read, modified and then re-written.

```
1:
| WB=GPIO                               // SETUP TO READ GPIO PORT
| PUSH_QS1 |:1                          // PUSH ON QS1

2:
| TOS_QS1, LSLB=2                       // GET GPIO VALUE
| TOS_QS2, LSRB=4                       // GET SAVED OPERAND
| NAND                                  // TOS_QS1 NAND TOS_QS2
| WB=LU                                 // SELECT LOGICAL UNIT AS SOURCE TO WB
| POP_QS2                               // SELECT NEXT OPERAND
| GPIO |:2                              // STORE TO GPIO PORT
```

6.4.2.2 SFU syntax

The inputs to the SFU are from the output of the shifter. As with previous instructions the inputs to the shifters and the source of the data on the write-back must be specified. The supported SFU operations are described above in Section 6.2.4.2.

The syntax for SFU operations is:

```
| SOURCE_A_BUS SOURCE_B_BUS <MNEMONIC> |       // IMPLIES SHIFT=0x00
| <SHIFTER_A_OPERATION> | <SHIFTER_B_OPERATION> | <MNEMONIC> |
```

An example of a sequence of SFU microinstructions is shown below. In this example the minimum value of the TOS_QS1 and TOS_QS2 is inserted into the BOS_QS1 and the operation continues until the result is greater than the reference value.

```
1:
| TOS_QS1 LSLB=0                        // READ TOS_QS1
| TOS_QS2 LSRB=0                        // READ TOS_QS2
| MIN_A_B                               // MIN TOS_QS1, TOS_QS2
| WB=SFU                                // SELECT SFU AS SOURCE TO WB
```

```
| INS_QS1                                   // INSERT INTO QS1
| POP_QS1 | POP_QS2  |:1                    // SELECT NEXT OPERANDS

2:
| BR_SFU_GT=-1 |:2                          // BRANCH IF RESULT > REF_DATA
```

Note: the BR_SFU_GT instruction is described below in Section 6.4.4.3

6.4.3 Write-back operations

There are four sources of write-back data. These include:

1. Arithmetic Unit (Shifter, Multiplier, Adder)
2. Boolean Logic Unit
3. Special Function Unit
4. GPIO Port

The instruction format for specifying the write-back source is:

```
| WB=SOURCE |                               // WHERE SOURCE == DP, LU, SFU, GPIO
```

The write-back data can be simultaneously written to any of the 3 Queued-Stacks, the output channel FIFO, the general-purpose I/O (GPIO) port, the two compare latches (WB_LAT and SFU_LAT) and the Operation Queue. The QS1, QS2 and RQS stacks can perform 15 write-back and/or stack manipulation operations as described above in Section 6.3.3. Note: the FIFO only supports the basic insert operation.

The stack operation instruction format is:

```
| <MNEMONIC>_<STACK> |                          // SINGLE QUEUED-STACK OPERATION
| <MNEMONIC>_<STACK> | <MNEMONIC>_<STACK> |    // PARALLEL QUEUED-STACK OPERATION
```

Coding examples of the various stack operations include:

```
| PUSH_QS1 |
| POP_QS2 |
| POP_WR_RQS |
| TOP_QS1, BOT_QS2 |
| TOP_BOT_QS1 | PUSH_INS_RQS |
| POP_BOT_QS2 | POP_INS_QS1 | SFU_LAT | WB_LAT |
| POP_WR_BOT_QS2 | PUSH_NW_BOT_QS1 | TOP_INS_RQS | FIFO | GPIO |  OP_QUEUE
```

As noted above the write-back data can be written to all 3 Queued-Stacks, the output FIFO, the Actor/Event Queue and the GPIO port as shown in last code example. This provides the ability write back result data to a queue while writing it to a stack and outputting it to the next DPE.

Below are examples of arithmetic operations and Queued-Stack operations:

```
1:
| ZERO ZERO ADD                          // ZERO -> ACCUMULATOR
| WB=DP                                  // SELECT DP (DEFAULT)
| PUSH_RQS |:1                           // INSERT AT TOS IN RQS

2:
| TOS_QS1 LSLB=12                        // SCALE
| IMM_DATA=0x55 LSRB=17                  // SCALE
| MULT                                   // MULTIPLY SCALED VALUES
| RQS_BUS=TOS_RQS MULT_BUS SAT_ADD       // SATURATED ADD
| WB=DP                                  // SELECT DP AS SOURCE TO WB
| PUSH_RQS                               // WRITE-BACK TO ACCUMULATOR
| POP_QS1 | POP_QS2 |:2                  // SELECT NEXT OPERANDS

3:
| WB=GPIO                                // SETUP TO READ GPIO PORT
| FIFO                                   // WRITE-BACK TO FIFO
| PUSH_QS1                              // WRITE-BACK TO QS1
| POP_QS1 | POP_QS2 |:3                  // SELECT NEXT OPERANDS
```

The following code sequence sets up the compare latch in the datapath and illustrates the its use as part of conditional execution and conditional branching. Conditional branching is described below in Section 6.4.4.3.

```
1:
| ZERO IMM_DATA=0x55 ADD                 // SETUP COMPARE VALUE 0x55
| WB=DP                                  // SELECT DP AS SOURCE TO WB (DEFAULT)
| WB_LAT |:1                             // WRITE TO WB COMPARE LATCH

2:
| COND_EXEC_EQ                           // CONDITIONALLY EXECUTE
| TOS_QS1 LSLB=0x01                      // READ TOS_QS1 AND SCALE
| TOS_QS2 LSRB=0x02                      // READ TOS_QS2 AND SCALE
| ADD                                    // ADD
| WB=DP                                  // SELECT DP AS SOURCE TO WB (DEFAULT)
| POP_QS1 | POP_QS2 |:2                  // SELECT NEXT OPERANDS

3:
| BR_WB_GT=-2 |:3                        // BRANCH IF RESULT > 0x55
```

6.4.4 Micro-engine operations

The DPE microcode engine can execute sixteen basic opcodes as shown below in Table 6.11. The microcode engine is similar to a VLIW machine where multiple operations can execute simultaneously. In the case of the DPE, branches and datapath operations can execute in the same microinstruction and event operations can execute in parallel with datapath operations. However not all three operations can execute in parallel.

Table 6.11: Mapping of micro-engine opcodes to execution type

OPCODE	EXECUTION TYPE
EXEC	NON-BRANCHING
EXEC_WB_EQ	NON-BRANCHING CONDITIONAL
EXEC_WB_GT	NON-BRANCHING CONDITIONAL
EXEC_WB_LT	NON-BRANCHING CONDITIONAL
WFE	EVENT
JMP	UNCONDITIONAL BRANCHING
JMP_HLT	UNCONDITIONAL BRANCHING/EVENT
LOOP_BACK	UNCONDITIONAL BRANCHING
BRA	UNCONDITIONAL BRANCHING
BR_WB_EQ	CONDITIONAL BRANCHING
BR_WB_GT	CONDITIONAL BRANCHING
BR_WB_LT	CONDITIONAL BRANCHING
BR_SFU_GT	CONDITIONAL BRANCHING
BR_SFU_LT	CONDITIONAL BRANCHING
BR_SFU_EQ	CONDITIONAL BRANCHING
TXFR	CHANNEL

The DPE microinstructions are categorized into three classes of operation:

1. Non-branching datapath execution including conditional and unconditional operations.
2. Branching execution including conditional and unconditional operations.
3. Event and channel handling operations.

The first two classes of instruction support both non-conditional and conditional execution. Conditional execution is predicated on the state of one of the three condition codes that is determined by the previous datapath or SFU operation. As described above in Section 5.4.1 the DPE supports three condition codes: equal, greater-than and less-than. The three condition codes are generated by comparing the value on the write-back bus with a stored reference value. There are two stored values: one for datapath operations and one for SFU operations. The stored values that are used for the comparison operations are written using a standard datapath operation that will be described below.

6.4.4.1 Non-branching operations

EXEC is the only unconditional non-branching microinstruction. The instruction is assumed to be unconditional if EXEC is not specified in the microinstruction. The EXEC instruction must be specified when entering a loop. The syntax for an EXEC instruction that specifies a loop number is:

```
EXEC=<LOOP NUMBER>
```

An example of a typical loop instruction sequence is:

```
1:
| EXEC=1 | RPT=7                          // ENTER LOOP #1, REPEAT 7 TIMES
| ZERO ZERO ADD                          // ZERO -> ACC
```

```
| INS_RQS |:1                          // INSERT AT BOS OF RQS

2:
| POP_QS1 POP_QS2 |:2                  // POP VARIABLES OFF OF INPUT STACKS

3:
| LOOP_BACK=-2 |:3                     // LOOP BACK TO 1:
```

There are three conditional non-branching microinstructions: EXEC_WB_EQ, EXEC_WB_GT and EXEC_WB_LT. An example of a conditional non-branching microinstruction is:

```
1:
| EXEC_WB_EQ                          // CONDITIONAL EXECUTION WB == ZERO
| TOS_QS1 LSLB=12                     // SCALE
| IMM_DATA=0x55 LSLB=2                // SCALE
| MULT                                // MULTIPLY SCALED VALUES
| RQS_BUS=TOS_RQS MULT_BUS SAT_ADD    // SATURATED ADD
| WB=DP,                              // SELECT DP AS SOURCE TO WB (IMPLIED)
| PUSH_RQS |:1                        // WRITE TO ACCUMULATOR
```

This microinstruction will execute the store to the RQS if the result from the previous instruction is a zero.

6.4.4.2 Unconditional branching operations

There are four unconditional branching operations supported by the microcode engine. These are JMP, JMP_HALT, BRA, LOOP_BACK and are described below in Table 6.12

Table 6.12: Unconditional branching operations

OPCODE	MICRO-OPERATION
JMP	JUMP TO ADDRESS SPECIFIED IN IMMED_DATA FIELD
JMP_HLT	JUMP TO ADDRESS SPECIFIED IN IMMED_DATA FIELD THEN HALT
LOOP_BACK	LOOP BACK FOR LOOP # (NEGATIVE OFFSET ONLY)
BRA	BRANCH UNCONDITIONALLY TO ADDRESS SPECIFIED IN BRANCH OFFSET

An example of the JMP_HLT micro-operation is shown below.

```
1:
| ZERO ZERO ADD                          // ZERO -> ACCUMULATOR
| WB=DP                                  // SELECT DP (DEFAULT)
| PUSH_RQS |:1                           // INSERT AT TOS IN RQS

2:
| TOS_QS1 LSLB=12                        // SCALE
| IMM_DATA=0x55 LSRB=17                  // SCALE
| MULT                                   // MULTIPLY SCALED VALUES
| RQS_BUS=TOS_RQS MULT_BUS SAT_ADD       // SATURATED ADD
| WB=DP                                  // SELECT DP AS SOURCE TO WB
| PUSH_RQS,                              // WRITE-BACK TO ACCUMULATOR
| POP_QS1 | POP_QS2 |:2                  // SELECT NEXT OPERANDS

3:
| WB=GPIO                                // SETUP TO READ GPIO PORT
| FIFO                                   // WRITE-BACK TO FIFO
| JMP_HLT=1 |:3                          // JUMP TO 1: AND WAIT FOR NEW EVENT
```

An example of the LOOP_BACK micro-operation is shown below.

```
1:
| EXEC=1 | RPT=7                         // ENTER LOOP #1, REPEAT 7 TIMES
| ZERO ZERO ADD                          // ZERO -> ACC
| INS_RQS |:1                           // INSERT AT BOS OF RQS

2:
| POP_QS1 POP_QS2 |:2                   // POP VARIABLES OFF OF INPUT STACKS

3:
| LOOP_BACK=-2 |:3                       // LOOP BACK TO 1:
```

6.4.4.3 Conditional branching operations

There are six conditional branching operations, 3 for arithmetic operations and 3 for SFU operations. These are described below in Table 6.13.

Table 6.13: Conditional branching operations.

OPCODE	RPT	MICRO-OPERATION
BR_WB_EQ	N	BRANCH IF WRITEBACK == REFERENCE DATA
BR_WB_GT	N	BRANCH IF WRITEBACK > REFERENCE DATA
BR_WB_LT	N	BRANCH IF WRITEBACK < REFERENCE DATA
BR_SFU_GT	N	BRANCH IF SFU RESULT > REFERENCE DATA
BR_SFU_LT	N	BRANCH IF SFU RESULT < REFERENCE DATA
BR_SFU_EQ	N	BRANCH IF SFU RESULT == REFERENCE DATA

The following code shows an example of both conditional execution and conditional branch operations:

```
1:
| ZERO IMM_DATA=0x55 ADD                  // SETUP COMPARE VALUE 0x55
| WB=DP                                   // SELECT DP AS SOURCE TO WB (DEFAULT)
| WB_LAT |:1                              // WRITE TO WB COMPARE LATCH

2:
| COND_EXEC_EQ                            // CONDITIONALLY EXECUTE
| TOS_QS1 LSLB=0x01                       // READ TOS_QS1 AND SCALE
| TOS_QS2 LSRB=0x02                       // READ TOS_QS2 AND SCALE
| ADD                                     // ADD
| WB=DP                                   // SELECT DP AS SOURCE TO WB (DEFAULT)
| POP_QS1 | POP_QS2                       // SELECT NEXT OPERANDS

3:
| BR_WB_GT=-2 |:2                         // BRANCH IF RESULT > 0x55

4:
| WB=GPIO                                 // SETUP TO READ GPIO PORT
| FIFO                                    // WRITE-BACK TO FIFO
| JMP_HLT=1 |:4                           // JUMP TO 1: AND WAIT FOR A NEW EVENT
```

6.4.4.4 Event operations

There are two event operations. JMP_HALT and WFE. Event operations are specified as instructions that execute and then wait for an event to occur. The JMP_HALT is a merged micro-operation where a datapath operation is executed in parallel to a JMP instruction that then halts the micro-engine at the address specified in the immediate data field. The WFE (Wait-for-Event) micro-operation halts the micro-engine at the address to the next microinstruction. Both micro-operations are used to terminate the execution of an actor.

Here is a code example of an FIR actor that is terminated by a JMP_HALT instruction:

```
1:
| ZERO ZERO ADD                  // ZERO -> ACC
| INS_RQS | :1                   // INSERT AT BOS RQS

2:
| TOS_QS1 TOS_QS2 MULT           // A(3,2,1) * X(N-3,2,1)
| BOS_RQS MULT_BUS ADD           // + ACC
| BOT_RQS                        // -> ACC
| POP_QS1 | POP_QS2              // POINT AT NEW VARIABLE
| RPT=3 | :2                     // REPEAT 3 TIMES

3:
| TOS_QS1 TOS_QS2 MULT           // A(0) * X(N)
| BOS_RQS MULT_BUS ADD           // + ACC
| FIFO                          // OUTPUT Y(N) TO FIFO ELEMENT
| POP_QS1 | :3                  // CONSUME X(N) TOKEN

4:
| PUSH_QS1 |PUSH_QS2             // RESET VARIABLE POINTERS
| RPT=4 | :4                     // REPEAT 4 TIMES

5:
| JMP_HALT=1 |:5                 // JUMP AND WAIT FOR NEW X(N) TOKEN
```

Chapter 7. High Level Modeling Environment

The ideal modeling environment for the CSP architecture is one where actors are instantiated and connected in a graphical schematic environment. Analysis and simulation are accomplished by netlisting the schematic into a form that can be input to an SDF aware tool. The tool would do an analysis of the validity of the synchronous data-flow graph (SDFG) generated by the netlister to confirm that it is consistent and does not have any deadlock situations. Simulation of the SDFG would be accomplished using tools such Ptolemy [60] or YAPI [61]. Once the simulation confirms the validity of the algorithm being designed, the netlist is mapped to an implementation on the CSP. The implementation determines the number of DPEs and the network topology that connects the DPEs.

After reviewing many of the options, two modeling environments were investigated as part of this research: SDF3 and SimEvents®. These are described in the following sub-sections.

7.1 SDF3

SDF3 is a tool from Eindhoven University of Technology that provides the ability to analyze, simulate and visualize Synchronous Dataflow Graphs (SDFG) [62]. Additionally it provides transformation services that convert SDFGs to HSDFGs (Homogenous SDFGs) that can be mapped to multi-processor SOCs, specifically NOC based systems [63]. This mapping converts a streaming application onto a NOC-based architecture while determining optimal resource allocation and producing a deterministic timing behavior. This is required for a CSP with multiple DPEs and for deterministic time stamping.

The key limitation of the tool is that it does not comprehend the Push-Pull protocol of the channels in the CSP. The tool assumes infinite resources and schedules all SDF algorithms across the multiple processors accordingly. Recall in Pull-Mode that the DPE can exert backpressure on the flow of tokens thus eliminating channel buffer (Queued-Stack) overflow conditions. This limitation is not an issue with SimEvents® as it allows the various network elements to provide backpressure as a programming option.

SDF3 uses the YAPI [61] tool to provide simulation capability. YAPI is an application programmer's interface to write signal and stream processing applications as process networks. The communication between processes is based on Kahn Process Networks with blocking reads on theoretically unbounded FIFOs.

7.2 SimEvents®

SimEvents® is an event-based simulator from Mathworks [64]. It works in conjunction with Simulink to model both time-based systems and event-driven systems. The sensor and ADC sub-system are described in Matlab or built from Simulink library models. The output of the ADC is converted into a signal-event that is processed by the SimEvents® simulator. SimEvents® does not perform a computational simulation but rather simulates entities propagating through the SDF network. Each resource in the network can be instrumented to determine if there are any errors as the entities propagate. Additionally, the instrumentation enables debug capability by providing visibility to various parameters in a particular network resource.

7.2.1 Entities and Attributes

SimEvents® uses different terminology to describe an SDF system. Rather than use tokens it uses entities[2] where entities are generated by signal-based events or by time-based events as shown below in Figure 7.1. The time-based generator launches entities/tokens into the network at a prescribed rate while the event-based generator only launches an entity/token when an event is detected on the VC pin.

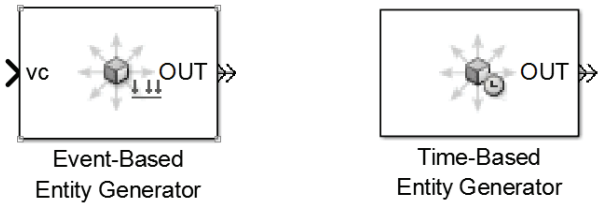


Figure 7.1: Entity generators

Entities can have multiple attributes attached to them. The attributes are tagged with descriptor names. The attributes propagate through the network and can be extracted by referencing the descriptor names. Figure 7.2 below shows the Set-Attribute and Get-Attribute library elements.

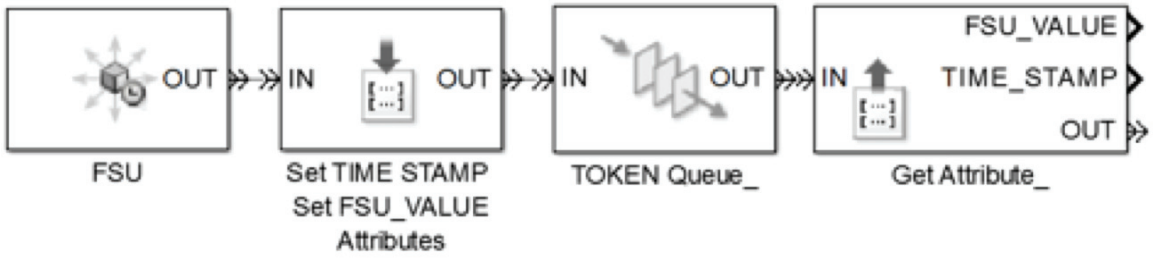


Figure 7.2: Attribute generator and extractor

[2] Entities and tokens will be used interchangeably throughout the rest of this document.

Figure 7.3 shows a typical dialog box that is used to set the multiple attributes to an entity. In this case there are two attributes: FSU_Value and the Time_Stamp value.

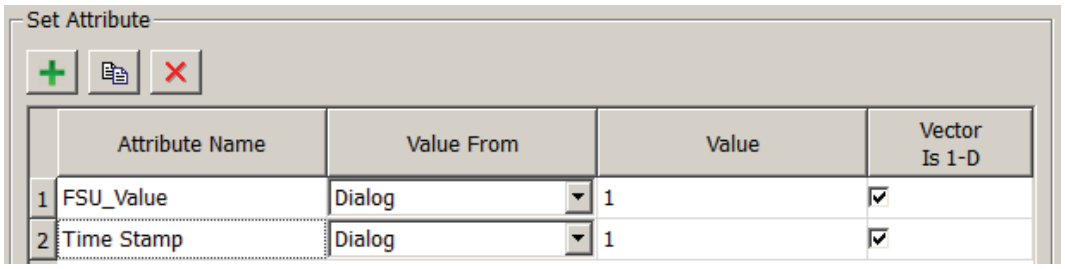


Figure 7.3: Set-Attribute dialog box

Figure 7.4 shows how the attributes are extracted from an entity. Note the ability to do error checking if attribute is missing and the ability to set a default value in the case of a warning.

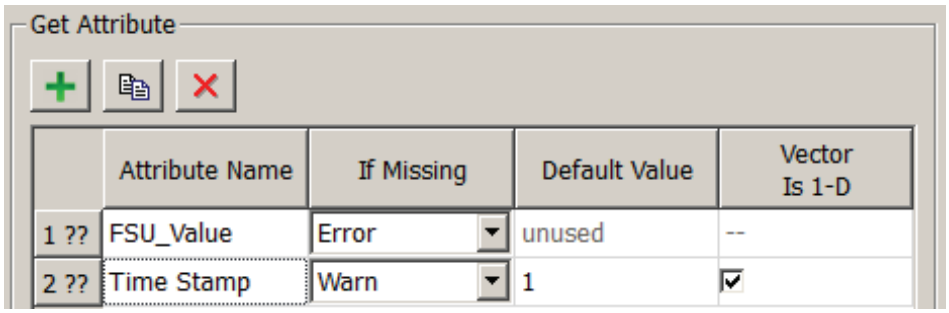


Figure 7.4: Get-Attribute dialog box

The time based entity generator can issue multiple entities/tokens per time period. This simulates a system where the need for queuing is needed to prevent overrun. An event based sequence generator that is set for cyclic repetition is used to trigger the entity generator as shown below in Figure 7.5.

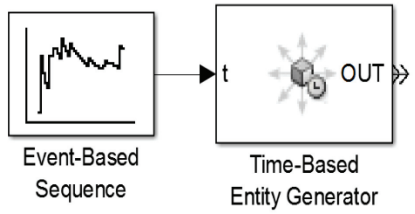


Figure 7.5: Time-based entity generator example

7.2.2 Servers

SimEvents® uses the concept of a Server to process entities (tokens). This is effectively an SDF actor with exception that it does not consume entities and it has only one input. Entities are consumed by using a switch block in front of the server as shown below in Figure 7.6. In this example the number of tokens is divided by 2 before entering the Actor/Server[3].

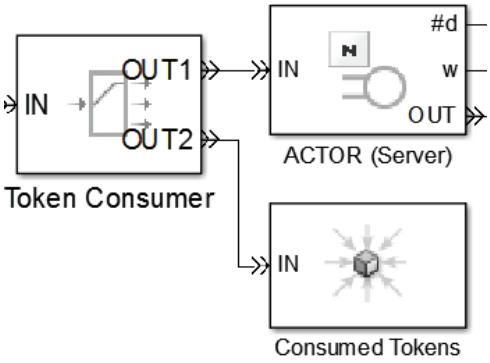


Figure 7.6: Token consumer

The actor processes input tokens and dispatches the output tokens after a prescribed number of event cycles. The actor has a single input so all tokens must be

[3] Servers and actors will be used interchangeably throughout the rest of this chapter.

combined before entering the actor/server. This is accomplished using a token combiner as shown below in Figure 7.7.

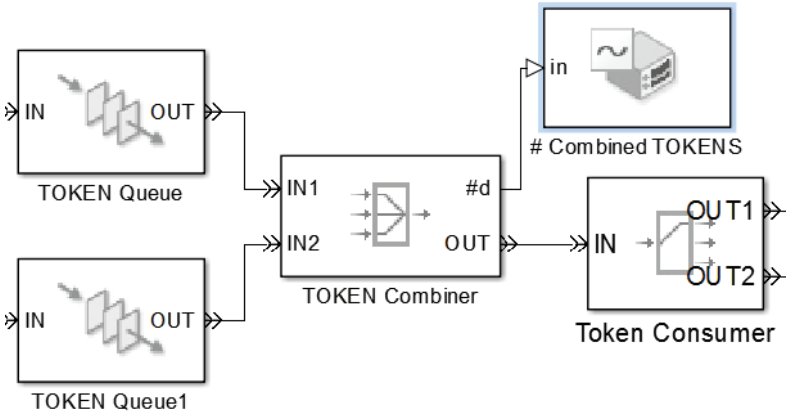


Figure 7.7: Example of a token combiner feeding a token consumer

7.2.3 FIFOs and LIFOs

SimEvents® supports both queues (FIFOs) and stacks (LIFOs). These elements are used to store entities for processing by the server. In Figure 7.6 above, the output the two token queues are combined into a composite token that is fed to the actor. The depth of the queues can be specified using the dialog box and can be instrumented to show statistical data on the entities/tokens entering and leaving the queue as shown below in Figure 7.8.

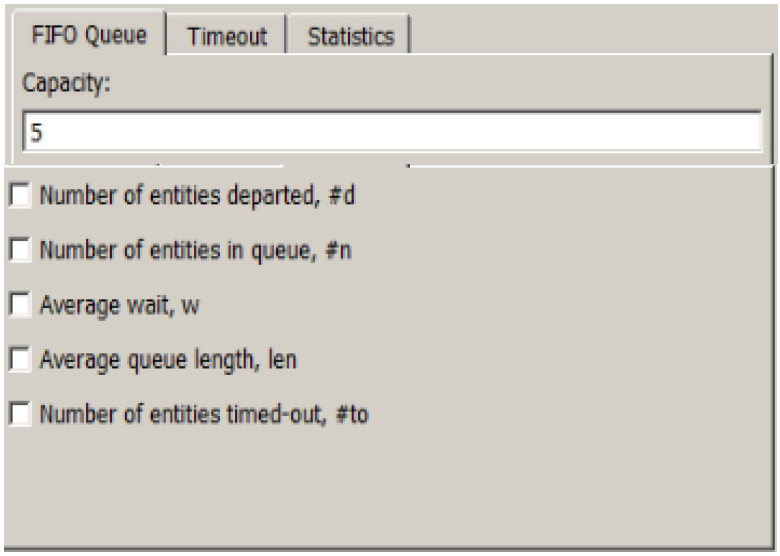


Figure 7.8: Dialog box for a queue element

As mentioned above, SimEvents® uses a Pull-Mode to propagate entities/tokens through the network. If there is a resource that is busy, the queue will continue to fill until it cannot handle new tokens. The elements that are launching tokens into the network need to be configured to produce an error condition when the network cannot process the required number of tokens per unit of time.

7.2.4 CSP Modeling

Figure 7.9 shows a CSP SimEvents® model with two Functional Service Units (FSU) and three Dataflow-Processing Elements. The FSUs are token generators that launch tokens into the network. Each token has two attributes as shown above in Figure 7.3.

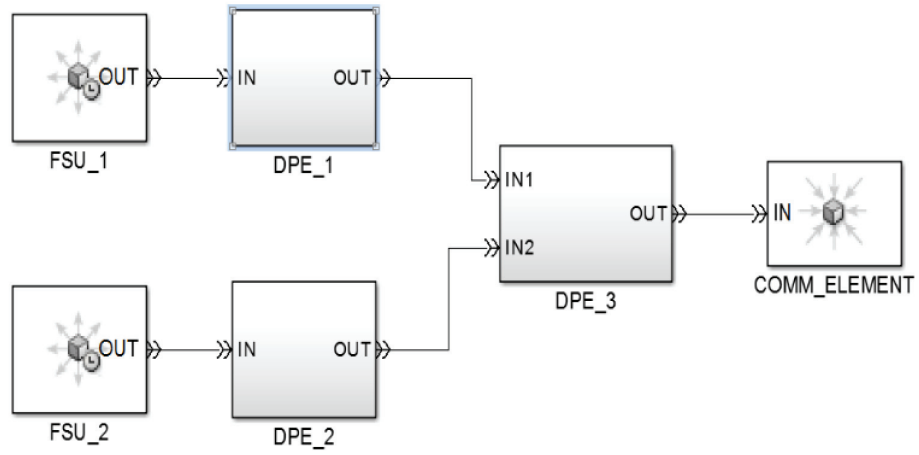


Figure 7.9: CSP SimEvents® Model

The DPEs process the tokens and outputs them to a communications element that is modeled as a token sink. DPE_1 and DPE_2 are modeled with a single queue and a single actor as shown below in Figure 7.10. Note the instrumentation ports on the actor. These are used to determine optimal resource allocation for the single queue DPE.

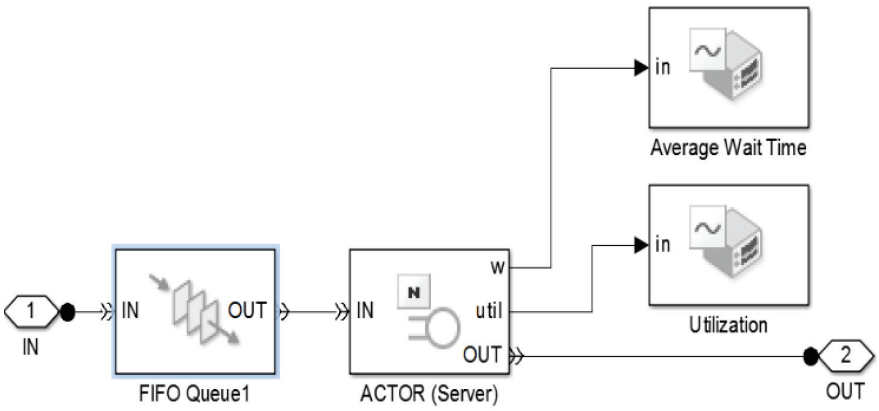


Figure 7.10: Single queue DPE model

DPE_3 is modeled with two queues, a token combiner, token consumer and a single actor as shown below:

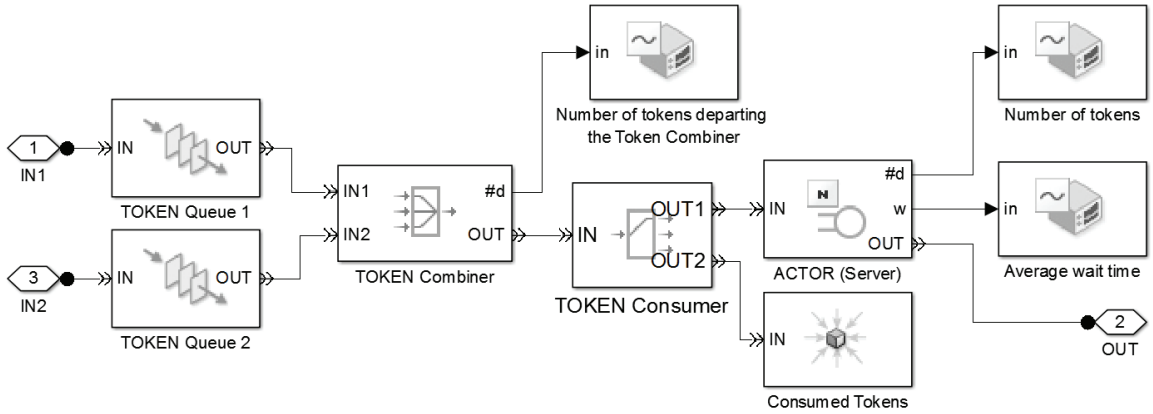


Figure 7.11: Multiple queue DPE model

The actor in this model is configured to measure the number of tokens that are processed and the average wait time for every token. Figure 7.12 shows the output from the average wait time scope and Figure 7.13 shows the total number of tokens that are processed by DPE_3. The average wait time is two time units once the tokens reach the actor at time 7. This indicates that the actor requires two clocks to process the data.

Each dot on a line in the graph indicates a token. In this example there is only one token per time unit. However the token combiner shows two tokens per time unit (Figure 7.14). The actor in DPE_3 is designed to consume two tokens and issue only one and is accomplished by the token consumer as shown above in Figures 7.6 and 7.11.

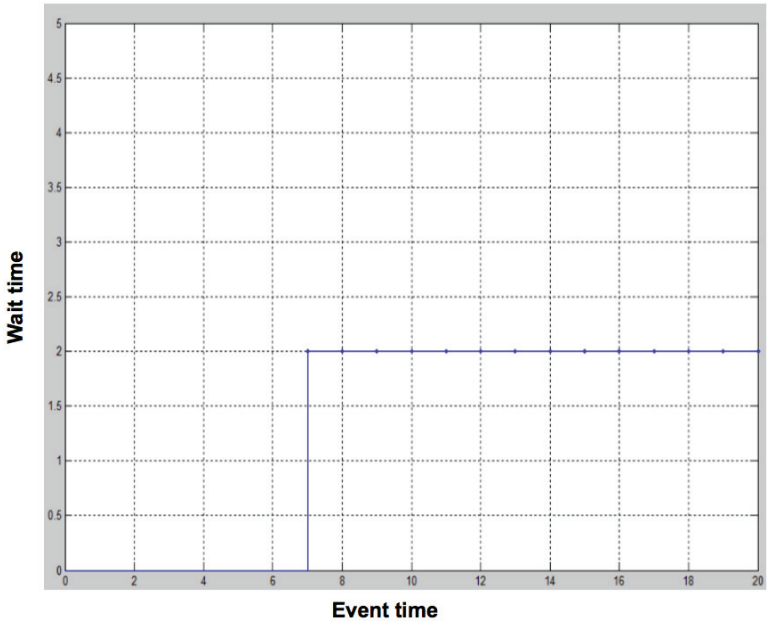


Figure 7.12: Average wait time for tokens entering DPE_3

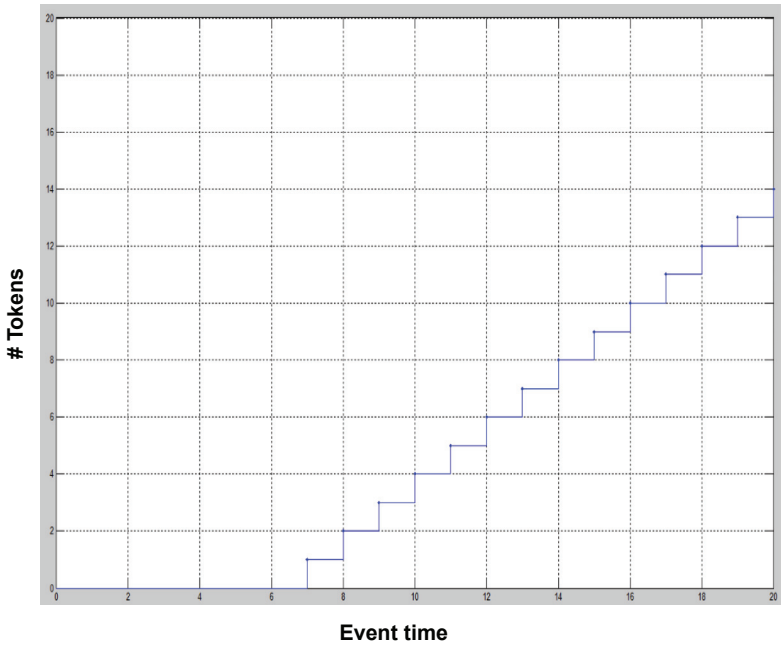


Figure 7.13: Total number of tokens processed by DPE_3

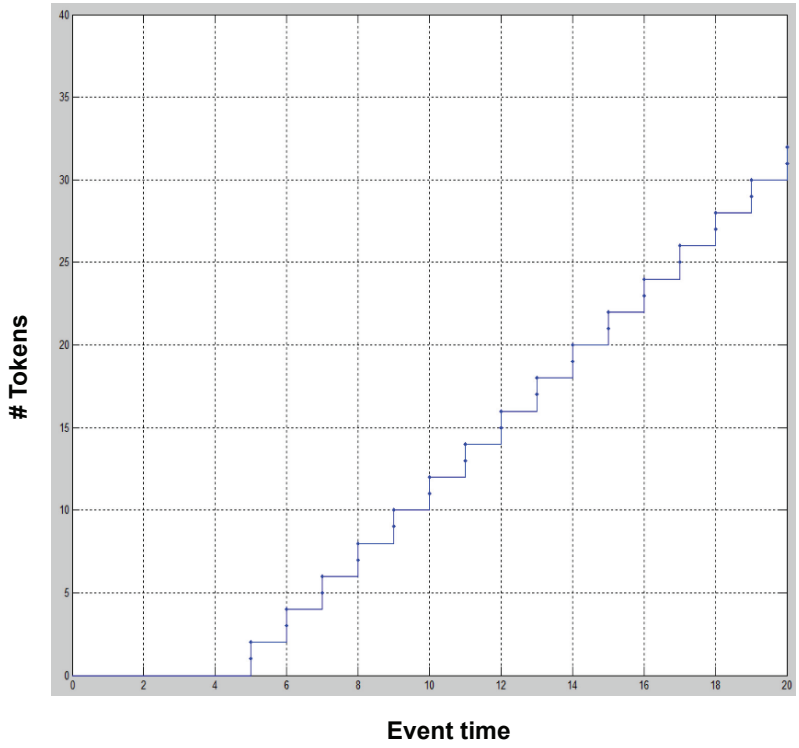


Figure 7.14: Total number of tokens leaving the Token Combiner

7.3 Summary

SimEvents® is a reasonably good tool to model and simulate a deterministic CSP topology. A SimEvents® simulation is primarily used to confirm that there are no overrun situations in the queues and the actors. If a NOC topology is required, a combination of SDF3 and SimEvents® may be needed to generate deterministic results (Note: this has not been researched as part of this work). It is possible though to generate a statistical model of an NOC topology using SimEvents® as it provides the ability to launch and propagate tokens using various types of algorithms that can be specified using MATLAB. The NOC network is modeled as switches that propagate packets through the network attempting to generate a deadlock situation.

Chapter 8. Results

8.1 Overview

Two performance critical workloads were analyzed as part of this research. The workloads are the core routines for FIR and IIR filter applications [65] [66]. These are also used for the energy-delay and energy-performance analysis described below in Section 8.4. The DPE was compared to the ARM Cortex-M3 [67] [68] and the Pleiades [14] processor. The M3 was selected as it is used in a large number of embedded applications and is a general-purpose computer. The Pleiades processor was chosen, as it is an excellent example of a low power application specific processor.

8.2 FIR Filter Performance

Figure 8.1 shows the FIR filter configuration that executes in ten clock cycles and is implemented with 5 microinstructions.

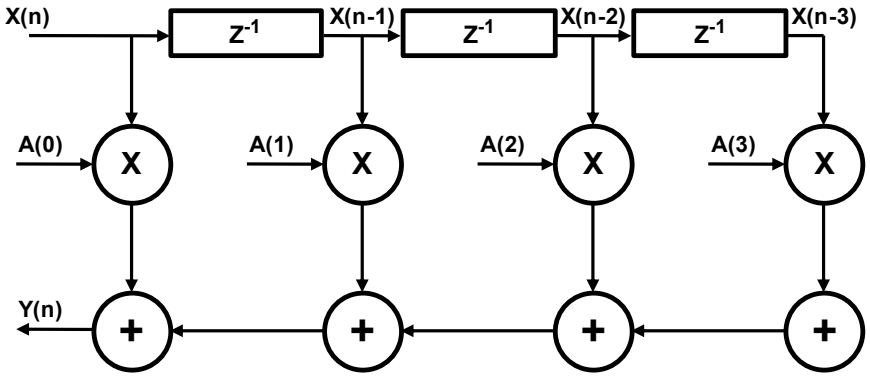


Figure 8.1: FIR filter configuration

The data storage configuration for the FIR filter calculations is shown below in Figure 8.2. IQS1 is used to store the incoming data tokens from the channel node. The

tokens are inserted at the bottom of the stack. The old tokens are overwritten when the BOS pointer recirculates. IQS2 is used to store the filter variables for each of the stage multipliers. The addition results are accumulated in the RQS element. The filter calculations proceed from oldest data token to the most recent. The TOS pointers for IQS1 and IQS2 are popped to point at the next variable and token for each multiplication step. NOTE: The initial conditions for the TOS/BOS pointers for each Queued-Stack are shown below.

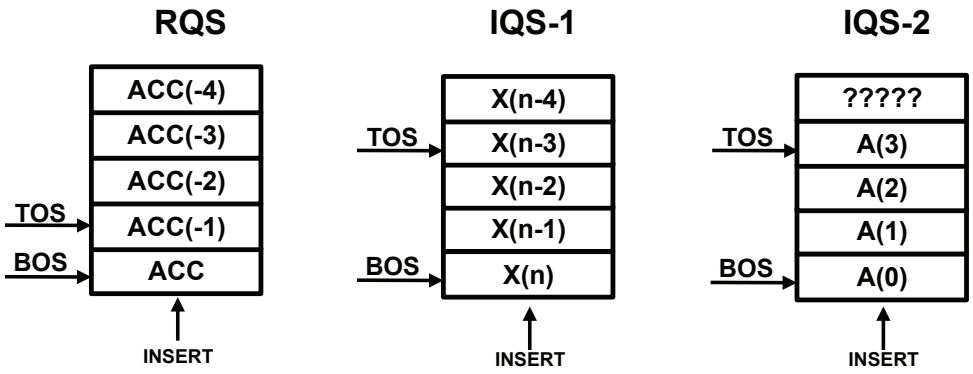


Figure 8.2: Initial data storage configuration for FIR filter routine

The code sequence below does not include the initialization code to set up the variables in in IQS2. This requires 4 clock cycles assuming that the variables are stored as constants in the microcode word.

```
1:
| ZERO ZERO ADD                  // ZERO -> ACC
| INS_RQS | :1                   // INSERT AT BOS RQS

2:
| TOS_QS1 TOS_QS2 MULT           // A(3,2,1) * X(N-3,2,1)
| BOS_RQS MULT_BUS ADD           // + ACC
| BOT_RQS                        // -> ACC
| POP_QS1 | POP_QS2             // POINT AT NEW VARIABLE
| RPT=3 | :2                     // REPEAT 3 TIMES
```

```
3:
| TOS_QS1 TOS_QS2 MULT          // A(0) * X(N)
| BOS_RQS MULT_BUS ADD          // + ACC
| FIFO                          // OUTPUT Y(N) TO FIFO ELEMENT
| POP_QS1 | :3                  // CONSUME X(N) TOKEN

4:
| PUSH_QS1 |PUSH_QS2            // RESET VARIABLE POINTERS
| RPT=4 | :4                    // REPEAT 4 TIMES

5:
| JMP_HALT=1 |:5                 // JUMP AND WAIT FOR NEW X(N) TOKEN
```

The first microinstruction inserts a ZERO into the BOS of the Result-QS, which is used as the accumulator for MULT-ADD instructions. The second microinstruction is repeated 3 times and executes a MULT-ADD of the last 3 stages of the filter, accumulating the result in the RQS. The third microinstruction does a MULT-ADD of the new data token and the A(0) filter variable and issues a POP command to consume the X(n) variable. The result is also sent to the output FIFO using WB_FIFO command in the same microinstruction. The fourth microinstruction resets the TOS pointers to point to the A(3) filter variable and the new X(-3) data token. The JMP_HLT microinstruction branches back to the second instruction that clears the accumulator and waits for the next data token. Once the X(n) variable is inserted into the BOS of IQS1, the channel node issues a FIRE signal to the DPE and the sequence repeats itself.

Table 8.1: FIR Throughput comparison

	Cortex-M3	**Pleiades**	**DPE**
VDD	1.8	1.5	1.8
FREQUENCY (MHZ)	20	14	10
DELAY	50ns	71ns	100ns
THROUGHPUT (CYCLES/FIR)	107	4	10

Table 8.1 above shows the FIR throughput comparisons for the Cortex-M3 processor from ARM, the Pleiades processor and the DPE. The Pleiades uses a special DSP to provide the excellent throughput numbers, however, as described in the next chapter it requires more energy to do so.

8.3 IIR Filter Performance

Figure 8.3 below shows a Bi-Quad IIR filter configuration that can be implemented in 9 microinstructions and 11 clocks.

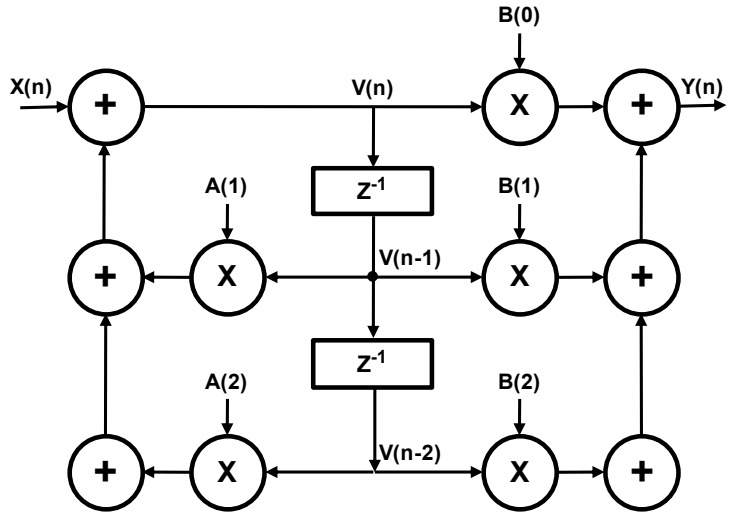


Figure 8.3: IIR filter configuration (Bi-Quad)

The data storage for the IIR filter calculations is shown below in Figure 8.4. This includes the initial conditions of the TOS/BOS pointers. There are two summing nodes. Each one is a separate entry in the RQS. The first sum is inserted into the bottom of the RQS and it becomes the V(n-1) variable the next time the filter is evaluated. The second sum replaces the V(n-2) variable once it is used.

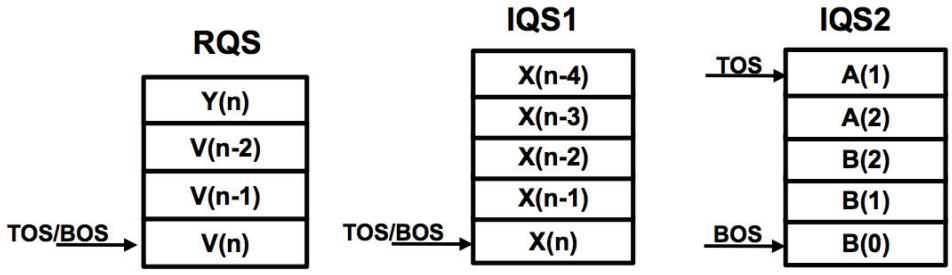
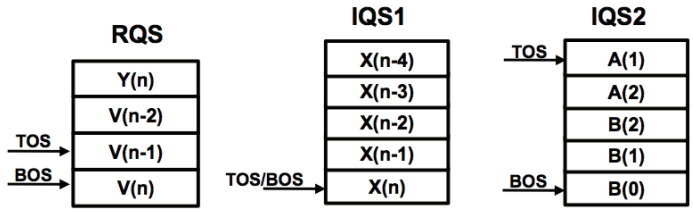


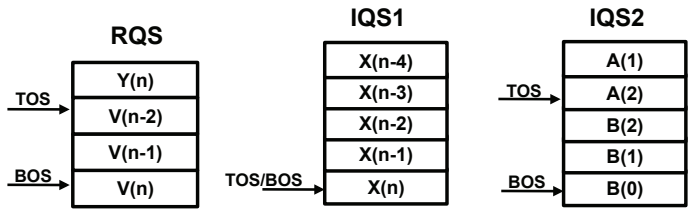
Figure 8.4: Initial data storage configuration for IIR filter routine

In the following IIR code example the status of the three Queued-Stacks will be annotated after the execution of each microinstruction.

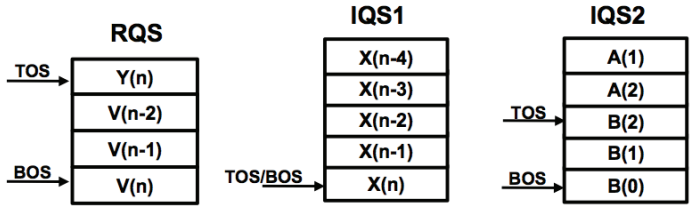
```
1:
| ZERO TOS_QS1 ADD                  // V(N) = X(N)
| PUSH_NW_BOT_RQS  |:1              // WRITE -> V(N) AND POINT @ V(N-1)
```



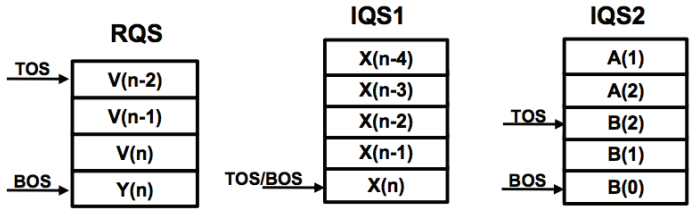

```
2:
| TOS_RQS TOS_QS2 MULT              // V(N-1) * A(1)
| BOS_RQS MULT_BUS ADD              // ADD TO V(N)
| PUSH_NW_BOT_RQS                   // WRITE -> V(N) AND POINT @ V(N-2)
| POP_QS2    |:2                   // POINT @ A(2)
```



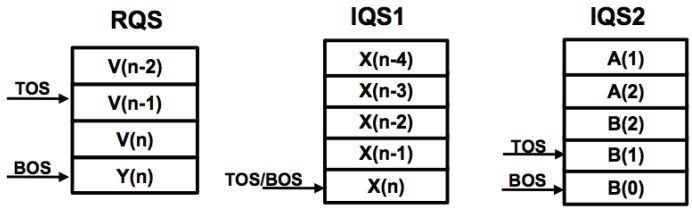

```
3:
| TOS_RQS TOS_QS2 MULT              // V(N-2) * A(2)
| BOS_RQS MULT_BUS ADD              // ADD TO V(N)
| PUSH_NW_BOT_RQS                   // WRITE -> V(N)AND POINT @ Y(N)
| POP_QS2    |:3                   // POINT @ B(2)
```



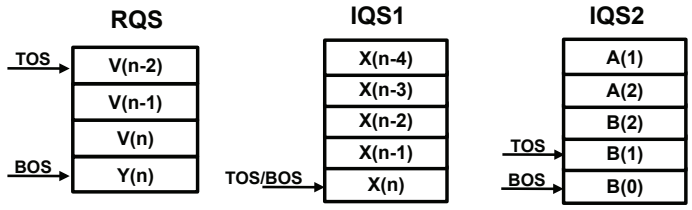

```
4:
| BOS_RQS BOS_QS2 MULT     // V(N) * B(0)
| TOS_RQS MULT_BUS ADD     // ADD TO Y(N)
| POP_INS_RQS    |:4       // WRITE -> Y(N) TO BOS
```



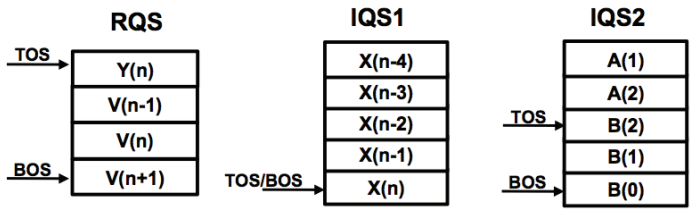

```
5:
| TOS_RQS TOS_QS2 MULT     // V(N-2) * B(2)
| BOS_RQS MULT_BUS ADD     // ADD TO Y(N)
| POP_BOT_RQS              // WRITE -> Y(N)
| POP_QS2    |:5          // POINT @ B(1)
```



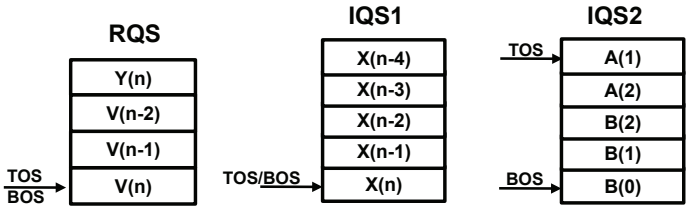

```
6:
| TOS_RQS TOS_QS2 MULT     // V(N-1) * B(1)
| BOS_RQS MULT_BUS ADD     // ADD TO Y(N)
| POP_BOT_RQS              // WRITE -> Y(N) POINT TO V(N-2)
| FIFO       |:6          // OUTPUT TO FIFO
```



```
7:
| ZERO BOS_RQS ADD          // GET Y(N)
| TOP_RQS    |:7           // WRITE -> Y(N)
```




```
8:
| POP_RQS                   // RESET RQS POINTER
| PUSH_NW_QS2               // RESET QS2 POINTER
| RPT=3      |:8           // REPEAT 3 TIMES
```




```
9:
JMP_HALT=1   |:9           // JUMP AND WAIT FOR NEXT TOKEN
```

Table 8.2 below shows the FIR throughput comparisons for the Cortex-M3, the Pleiades processor and the DPE.

Table 8.2: IIR Throughput comparison

	Cortex-M3	**Pleiades**	**DPE**
VDD	1.8	1.5	1.8
FREQUENCY (MHZ)	20	14	10
DELAY	50ns	71ns	100ns
THROUGHPUT (CYCLES/IIR)	129	8	11

The Pleiades processor has greater throughput than the DPE due to the fact that it has two MAC units. As we will see in the next chapter, the DPE is between 1.5X and 3X

better in Energy-Delay/Operation than the Pleiades. This is a benefit for the class of workloads that the DPE is designed to perform. In addition, the benefits of composability will be analyzed where the energy-performance costs of adding multiple DPE's is shown to be minimal.

8.4 DPE energy analysis

There are four energy analysis techniques that were used to benchmark the energy performance of the DPE. These are: Energy-Delay product (E-D), Energy/Instruction, Energy-Delay/Operation and Energy Performance Percentage Ratio (EPPR). Before proceeding it is necessary to review the energy and power relationships that are key to this analysis.

Recall that power is defined as:

$$Power\ (watts) = Vdd * I_{ckt} \qquad (8.1)$$

And that power is also defined as:

$$Power\ (watts)\ = \frac{Energy\ (watt{-}sec)}{T program\ (sec)} \qquad (8.2)$$

Where $T_{program}$ is the program (workload) execution time and is defined as the number of instructions required to execute the program multiplied by the clock period per instruction:

$$T_{program} = Num_{Inst} * T_{clk} \qquad (8.3)$$

Energy[4] can then be defined as:

$$Energy = Vdd * I_{ckt} * Num_{inst} * T_{clk} \quad (8.4)$$

Energy per instruction is simply one instruction multiplied by the clock period:

$$Energy = Vdd * I_{ckt} * 1 * CP \quad (8.5)$$

The current (I_{ckt}) is defined as:

$$I_{ckt} = Cap_{Switch} * \frac{dVdd}{dt} \quad (8.6)$$

Therefore

$$Energy = Cap_{Switch} * Vdd * \frac{d(Vdd)}{dt} * 1\ second \quad (8.7)$$

Where $\frac{d(Vdd)}{dt}$ is generally equal to Vdd (for digital logic). This also assumes that Cap_{switch} is charged and discharged every cycle, which is only true for clock signals. The energy usage is decomposed into two components, one from clock nodes switching and one from logic nodes switching:

$$Energy = \left(\left(\frac{1}{2} Cap_{Logic} * Vdd^2\right) + (Cap_{Clock} * Vdd^2)\right) * 1\ second \quad (8.8)$$

Recall that logic nodes only switch every other cycle, which is accounted for by the ½ term in the equation. The energy equation can be converted to a power equation by multiplying it by 1/sec.

[4] Energy is expressed in Watt-Seconds or Watts/Frequency. One Joule of energy is equivalent to 1 Watt-Second.

$$Power = \left(\left(\frac{1}{2} Cap_{Logic} * Vdd^2\right) + (Cap_{Clock} * Vdd^2)\right) * Freq \qquad (8.9)$$

8.5 DPE Energy-Delay

One figure of merit in the design of the DPE is the Energy-Delay (E-D) product. Figure 8.5 below shows the relationship between energy, delay and the E-D product. The optimal area to operate is between the dashed lines where the energy-delay product is minimal.

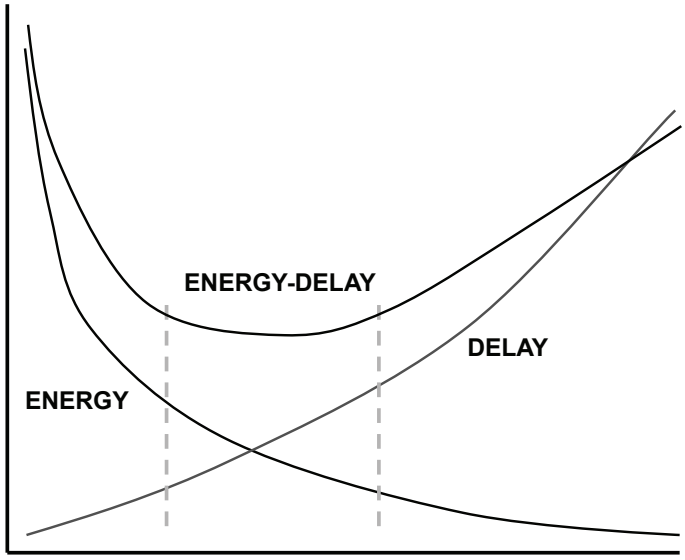


Figure 8.5: Energy-Delay product relationship

To further understand the application of energy-delay, refer to the schematic shown below in Figure 8.6 where one inverter is connected to another inverter via a wire. The input gate capacitance of the inverter is *Cg* and the capacitance of the wire is *Cw*.

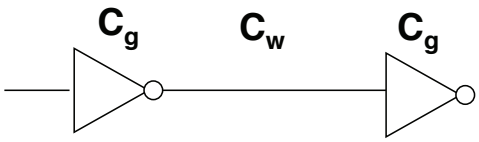


Figure 8.6: E-D Inverter schematic

Recall that delay is proportional to the ratio of gate + wire capacitance to load capacitance [69]:

$$Delay \approx \frac{(Cg+Cw)}{Cg} \tag{8.10}$$

Energy is proportional to the amount of capacitance that is charged in a cycle:

$$Energy \approx Cg + Cw \tag{8.11}$$

The Energy-Delay product is then:

$$E - D \approx \frac{(Cg+Cw)^2}{Cg} \tag{8.12}$$

The optimal Energy-Delay minima occurs when the derivative of the Energy-Delay product with respect to Cg is zero:

$$\frac{d(E-D)}{d(Cg)} \approx \frac{Cg^2 - Cw^2}{Cg^2} = 0 \tag{8.13}$$

The optimal Energy-Delay occurs when Cg = Cw.

There are additional Energy-Delay metrics where the impact of the delay component is emphasized, e.g., Energy-Delay^2 or Energy-Delay^3. Table 8.3 below shows the results of the various optimizations that can be done using the Energy-Delay product.

Table 8.3: Energy-Delay optimizations

OPTIMAL ENERGY	$C_G = MIN$
OPTIMAL ENERGY-DELAY	$C_G = C_W$
OPTIMAL ENERGY-DELAY^2	$C_G = 2*C_W$
OPTIMAL ENERGY-DELAY^N	$C_G = N*C_W$
OPTIMAL DELAY	$C_G = \infty$

The DPE was synthesized for a broad range of frequencies to analyze the synthesis results on energy and delay. Figure 8.7 below shows that the DPE architecture uses approximately the same energy/filter-operation in the operating range of 10–25 MHz. The slight increase in energy usage over this range is due to the impact of the clock tree synthesis tool optimizing setup and hold times. At 30 MHz and above the synthesis tool uses the larger size standard cells that increase the area and the amount of interconnect wiring.

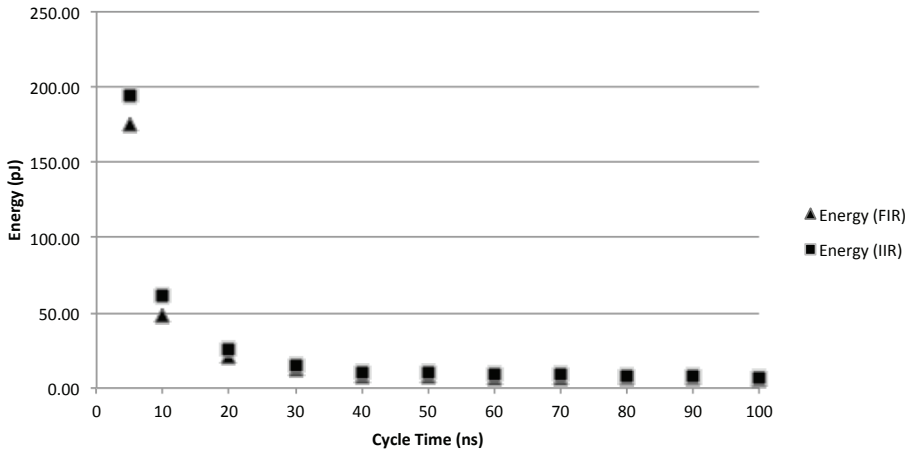


Figure 8.7: Energy vs. Cycle-Time for the IIR and FIR workloads

The energy-delay analysis of the DPE was done using a 180nm mixed-signal process. The DPE logic was synthesized and the layout generated by DC-Topo/ICC from Synopsys using the typical process corner at 85°C. The layout parasitics were extracted using Calibre from Mentor Graphics. The timing and energy values were then derived using PrimeTime (PT) and PrimeTime-PX (PTPX) respectively.

Tables 8.4 and 8.5 below show the energy-delay for the DPE, a reconfigurable DSP (Pleiades) developed at UC-Berkley [14] and the Cortex-M3 from ARM [67]. The Cortex-M3 is widely used in embedded designs as both a general-purpose processor and a DSP. The reconfigurable DSP from UCB is an excellent example of a DSP implementation that is tuned for similar filter applications as the DPE.

The throughput and energy values for the Pleiades DSP were derived from the 600 nm implementation specified in [14] using the scaling calculations defined by the authors for their own benchmarking exercise. The energy calculations for the Cortex-M3 are derived from an 180nm reference design [71]. A DSP library of filter functions designed specifically for the Cortex-M3 [68] was used to determine the throughput. The

DPE energy values are derived from PTPX using a 180nm extracted netlist. All DPE benchmarks use 16-bit integer data tokens and 48-bit integer results.

Table 8.4: FIR Energy-Delay benchmarks

	Cortex-M3	**Pleiades**	**DPE**
VDD	1.8	1.5	1.8
FREQUENCY (MHZ)	20	14	10
DELAY	50ns	71ns	100ns
ENERGY	148.9pJ	61.5pJ	5.64pJ
ENERGY-DELAY (J-S X 10^{-18})	7.44	4.36	.564

Table 8.5: IIR Energy-Delay benchmarks

	Cortex-M3	**Pleiades**	**DPE**
VDD	1.8	1.5	1.8
FREQUENCY (MHZ)	20	14	10
DELAY	50ns	71ns	100ns
ENERGY	169.0pJ	70.1pJ	7.13pJ
ENERGY-DELAY (J-S X 10^{-18})	8.45	4.97	.713

At first glance it looks like the Cortex-M3 is roughly 1/2 the performance of the Pleiades. However when comparing the IIR and FIR operations performed by the two machines the Pleiades is 15-25 times better than the Cortex-M3. This is a limitation of the E-D analysis and is described below where we look at energy-delay per operation.

Tables 8.6 and 8.7 below shows the Energy-Delay product for the DPE data presented above in Tables 8.4 and 8.5 for various operating frequencies at VDD=1.8V.

Table 8.6: FIR Energy-Delay for various cycle times

Cycle-Time	10ns	20ns	40ns	60ns	80ns	100ns
ENERGY/INSTRUCTION	47.9PJ	21.1PJ	8.10PJ	7.13PJ	6.48PJ	5.64
ENERGY-DELAY (J-S X 10^{-18})	.48	.421	.324	.428	.518	.564

Table 8.7: IIR Energy-Delay for various cycle times

Cycle-Time	10ns	20ns	40ns	60ns	80ns	100ns
ENERGY/INSTRUCTION	61.5PJ	25.9PJ	10.37PJ	9.07PJ	8.10PJ	7.13PJ
ENERGY-DELAY (J-S X 10^{-18})	.616	.518	.415	.544	.648	.713

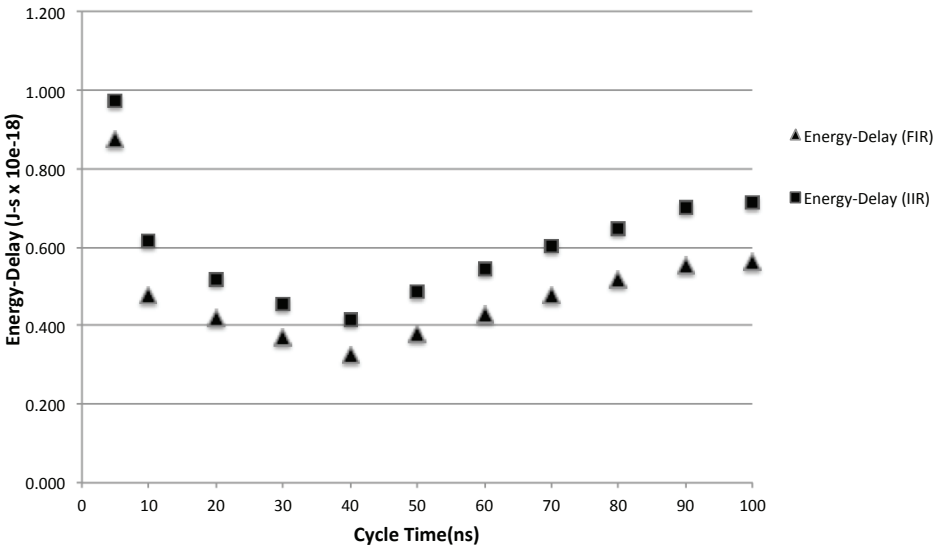


Figure 8.8: DPE Energy-Delay/Actor for cycle-time design points

Figure 8.8 above shows the graph of the Energy-Delay/Actor. Note that 25MHz is the optimal operating point for the DPE from an E-D perspective. This data needs to be considered in context with other key system metrics. For example, executing an IIR 2.5

times faster (25MHz vs. 10 MHz) at a better E-D is beneficial if the 25MHz clock is shut off when the DPE is idle. Otherwise the idle clock power will negate the benefits of the lower E-D during normal operation.

8.6 DPE Energy/Instruction vs. Energy-Delay/Operation

Energy per instruction is another measurement of the efficiency of computational element and is typically measured in Joules/Instruction. A variation of this measurement is Watts/IPS or the more widely used variant: MIPS/Watt. The derivation of this equality is shown below:

$$\frac{Joules}{Instruction} = \frac{\frac{Joules}{Second}}{\frac{Instructions}{Second}} = \frac{Watts}{IPS} \qquad (8.14)$$

Analyzing energy per instruction can provide an interesting mechanism for measuring the impact of control logic on the overall energy usage of a particular microarchitecture. The complexity of the control logic is a function of the instruction set architecture (ISA), the depth of the pipeline and complex features such as register renaming, branch prediction, prefetching, etc. For energy sensitive processors like the DPE, increasing the complexity of the control logic has minimal impact on performance and consumes more area, power in addition to impacting cycle time [70]. Ideally the ratio of control logic to datapath logic for this class of machines is much less than 1:1. The DPE has a ratio of 0.36:1 due in large part to the single cycle microcoded control logic design. Table 8.8 below shows the Energy/Instruction for the various computational elements that are used in embedded sensor applications. Note: the Pleiades numbers are for a 600nm process and were scaled to 180nm for this energy analysis exercise.

Table 8.8: Energy/Instruction for various computational elements

Processor	Arch Style	Data Path Width	Event Driven	Memory (KB)	Process (nm)	Voltage	MIPS	Energy (pJ/Inst)
SNAP	RISC+ Accel	16	Y	8	180	1.8 0.6	200 23	218 24
BitSNAP	RISC	16	Y	8	180	1.8 0.6	54 6	152 17
Subliminal	GP	8	Y	0.256	130	~0.360	0.8	2.6
Pleiades	DSP	16	N	0.512	600	1.5	14	205
SmartDust	RISC	8	N	3.125	250	1.0	.5	12
Atmel 128L	GP	8	N	132	350	3.0	7.3	3200
Intel XScale	GP	32	N	8	130	1.65	400	1100

There are a number of limitations with using the energy/instruction metric to benchmark different microarchitectures because it does not comprehend how much work is accomplished with each instruction, nor does it account for the different technologies and power supply voltages that a particular architecture uses. The first limitation can be addressed by measuring Energy-Delay/Operation instead of Energy/Instruction if the frequency of machine and the number of instructions it takes to complete an operation are known. The second limitation can be partially addressed by normalizing the impact of technology scaling on the energy-delay [73].

Tables 8.9 and 8.10 below show the results of the Energy-Delay/Operation for the IIR and FIR operations including frequency and throughput numbers. As mentioned above, the M3 and the DPE are implemented in 180nm technology and the Pleiades is implemented in 600nm and scaled to 180nm.

Table 8.9: FIR Energy-Delay/Operation Benchmarks

	Cortex-M3	**Pleiades**	**DPE**
VDD	1.8	1.5	1.8
FREQUENCY (MHZ)	20	14	10
CYCLE-TIME	50NS	71.4NS	100NS
THROUGHPUT (CYCLES/FIR)	107	4	10
SWITCHED-CAPACITANCE/FIR	4.92NF	126PF	17.4PF
ENERGY/FIR	15,941PJ	285PJ	56.4PJ
ENERGY-DELAY/FIR (J-S X 10^{-18})	85,284	81.3	56.4

Table 8.10: IIR Energy-Delay/Operation Benchmarks

	Cortex-M3	**Pleiades**	**DPE**
VDD	1.8	1.5	1.8
FREQUENCY (MHZ)	20	14	10
CYCLE-TIME	50NS	71.4NS	100NS
THROUGHPUT (CYCLES/IIR)	129	8	11
SWITCHED-CAPACITANCE/IIR	6.73NF	295PF	22.6PF
ENERGY/IIR	21,805PJ	659PJ	78.3PJ
ENERGY-DELAY/IIR (J-S X 10^{-18})	140,642	376.8	86.1

The DPE is obviously better than the M3 and the Pleiades processors for these particular workloads. The limitation of the DPE is that it has very little local storage and is suited primarily for low energy streaming data applications. For workloads that require large amounts of storage, the Pleiades processor is a better choice; however, the CSP is easily scaled to handle larger workloads by composing multi-DPE systems. This is described in the following section.

8.7 Energy Performance Percentage Ratio

Another interesting figure of merit is the scaling relationship (ratio) between energy and performance; in other words, the impact on performance when energy is increased or decreased. Ideally a 1% increase in energy results in a 1% increase in performance. If one considers that energy is a proxy for the number of transistors then adding 1% more transistors should result in a 1% improvement in performance. Taking this to an extreme, adding an additional processor core should double the performance. We know that is generally not the case especially as the number of cores increases beyond a certain limit. The problem of course goes back to equation 1.1 where we need to impedance match the application domain, the software compiler domain, the microarchitecture domain and the lastly the transistor technology domain.

This section discusses three basic areas of optimization: energy optimized, energy-performance optimized and performance optimized. Each area has an impact on resulting the micro-architectural optimizations and environmental conditions i.e., VDD, frequency, temperature, etc. Performance can be specified many different ways. It is generally in the form of a standard benchmark like SPEC [74], TPC [75], EEMBC [76] or something more algorithmic specific such as Multiply-Accumulates/Second. In all cases, performance has a temporal component where each benchmark is measured by how quickly it completes the benchmark. Figure 8.9 shows the relationship between energy and performance and introduces the concept of Energy Performance Percentage Ratio (EPPR).

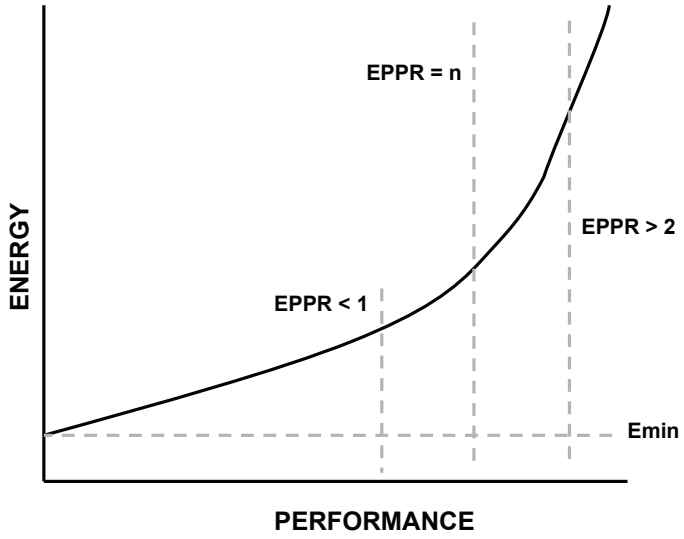


Figure 8.9: Energy-Performance relationship

There are three regions of interest: 1) EPPR < 1 where an increase in energy results in a proportional increase in performance and 2) the 1<= EPPR <= 2 region where a moderate increase in energy produces reasonable performance increase or 3) EPPR > 2 where an increase in energy provides a marginal increase in performance while asymptotically approaching a finite performance limit. EPPR= n is the point of inflection between region 1 and 3 and is derived below.

In this derivation *E(x)* is the energy expended modifying a particular variable *(x)* such as supply voltage, number of processor cores, number of pipeline stages, cache size, etc. *P(x)* is the resulting performance improvements that modifying each variable provides. This is accomplished by taking the derivative of energy vs. performance^n and then solving for *n*.

$$d/d(x)\ (\ E(x) / P(x)^{n}\ ) = 0 \qquad (8.15)$$

This results in the following relationship:

$$E'(x)\,P(x) = n\,E(x)\,P'(x) \tag{8.16}$$

Solving for the energy and performance ratios:

$$E'(x) / E(x) = n\,P'(x) / P(x) \tag{8.17}$$

Solving for *n:*

$$(E'(x) / E(x))/(P'(x) / P(x)) = n \tag{8.18}$$

The ratio is then expressed as:

$$EPPR(x) = (E'(x) / E(x))/(P'(x) / P(x)) = n \tag{8.19}$$

This indicates that a 1% increase in performance costs n% increase in additional energy.

The three EPPR operating regions roughly map to the following characteristics in general purpose computation elements [77]:

1. Energy Optimized: EPPR < 1
   - Run at low VDD
   - Mostly small devices
   - Shallow pipeline
2. Energy – Performance: 1<= EPPR <= 2
   - Run at nominal VDD
   - Moderate pipeline depth, moderate instruction level parallelism

3. Performance optimized: EPPR > 2
    - Run at maximum VDD
    - Little concurrency in application
    - Deep pipelines

An example of EPPRs for two different variables is shown in Figure 8.10. The system designer would use these EPPRs to determine which region to operate in.

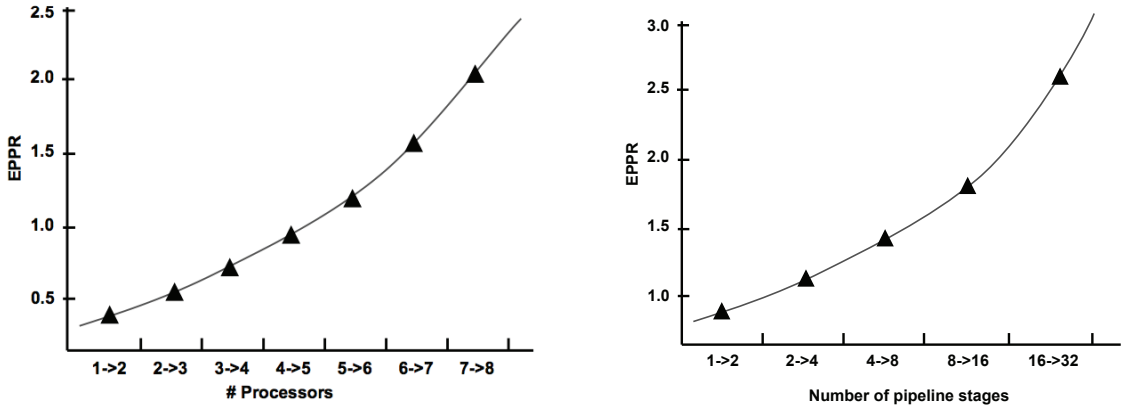


Figure 8.10: EPPRs for different design variables

One of the key advantages of this platform is that it is easy to compose a system of multiple DPEs. Figure 8.11 below shows the four configurations that were analyzed to determine the impact of multiple DPEs on EPPR. The performance metric for this evaluation is "token throughput" where the throughput increase is approximately linear with the additional number of DPEs. The system is tuned for minimal token wait time for all four configurations.

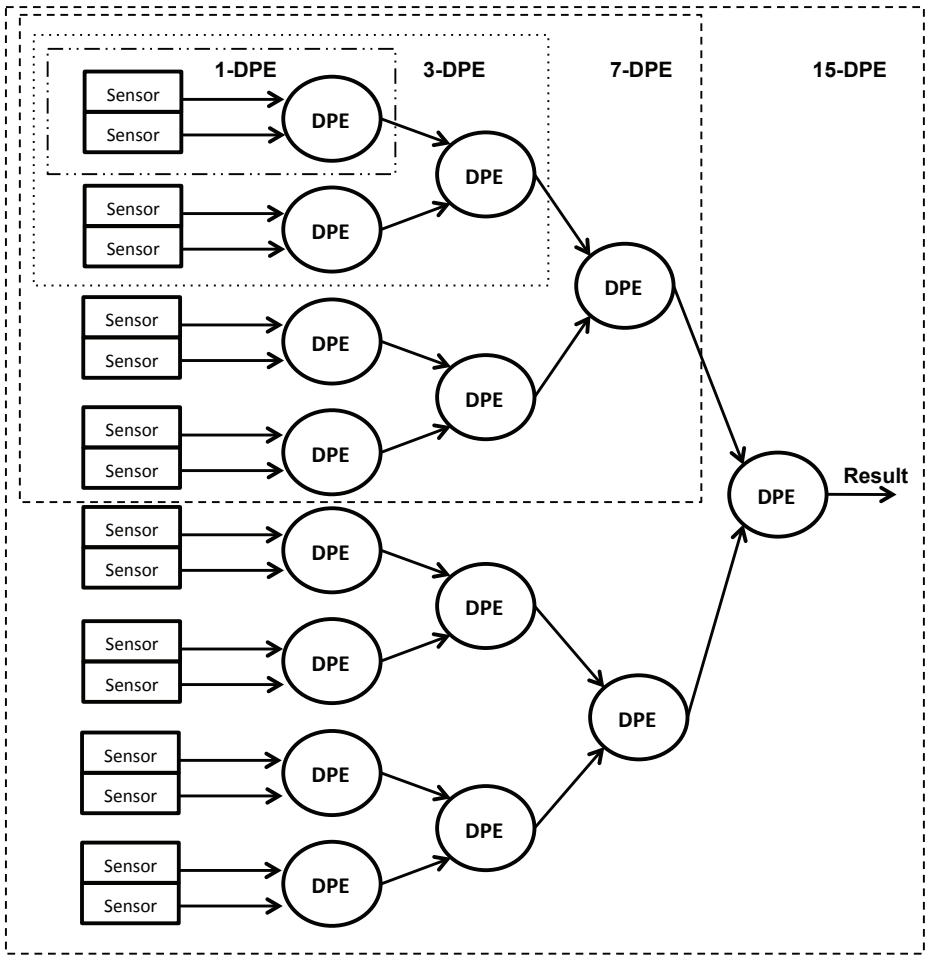


Figure 8.11: DPE topology for EPPR evaluation

Each system configuration has a single output and executes a Fuzzy Logic based data fusion algorithm on each DPE. The fusion algorithm is described above in Section 3.6.1. The algorithm is implemented using four Actors as shown below in Figure 8.12. The tokens passed between Actors are annotated in the figure as they propagate through the SDF network. The tokens are stored in IQS1, IQS2 and RQS as shown below in Figure 8.13. IQS-1 receives tokens from Channel-1 and IQS-2 from Channel-2. RQS is used to store the Change, Rate-of-Change, Weight and Drift data. The fusing algorithm requires 35 clock cycles per operation.

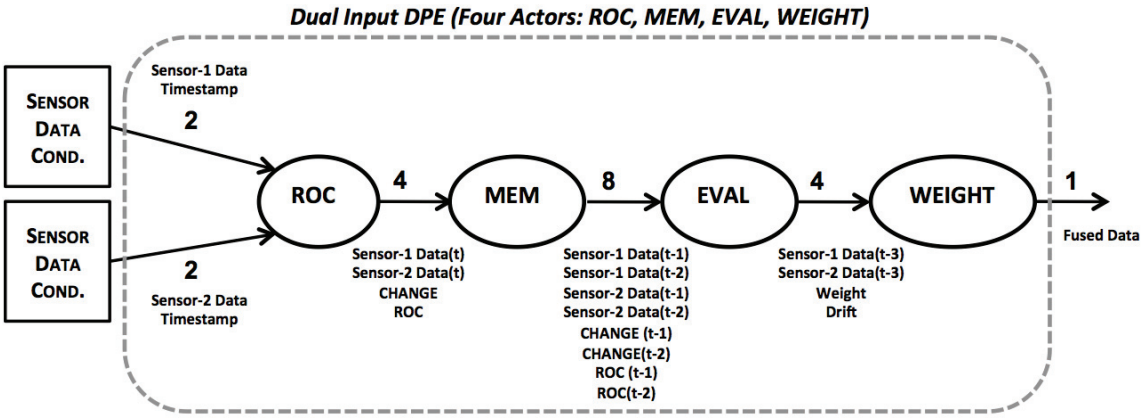


Figure 8.12: Actors used in Fuzzy Logic fusing algorithm

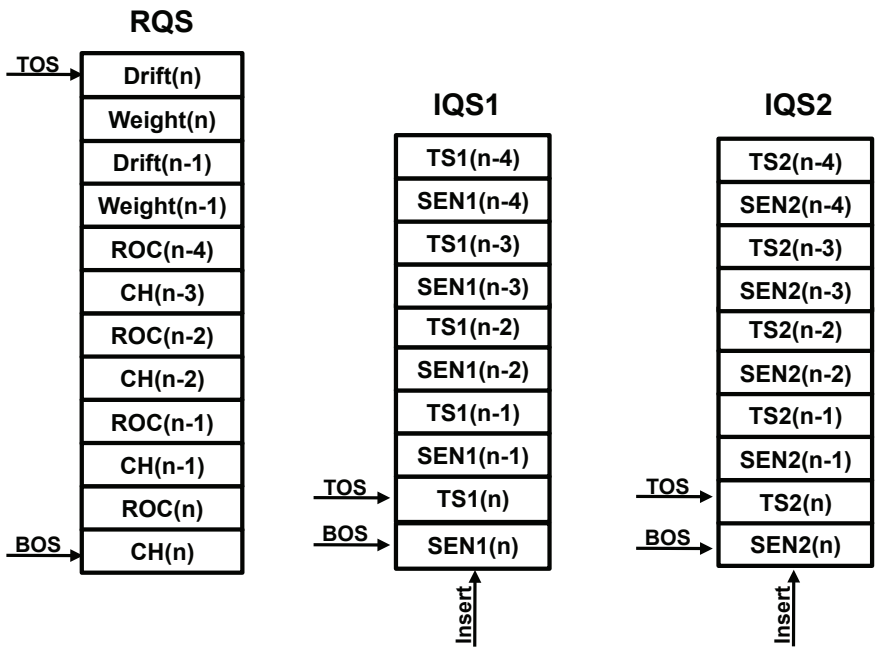


Figure 8.13: Queued-Stack storage for Fuzzy Logic fusing algorithm

Table 8.11 below shows the Energy-Delay/Operation for the four multi-DPE topologies.

Table 8.11: Energy-Delay/Operation benchmarks for fusing algorithm

	1-DPE	3-DPE	7-DPE	15-DPE
VDD	1.8	1.8	1.8	1.8
FREQUENCY (MHZ)	10	10	10	10
CYCLE-TIME	100NS	100NS	100NS	100NS
# SENSOR CHANNELS	2	4	8	16
THROUGHPUT (CYCLES/OP)	35	35	35	35
SWITCHED-CAPACITANCE/OP (PF)	67.6	212.8	548.5	1357.8
ENERGY/OP (PJ)	218.9	689.4	1772.2	4596.1
ENERGY-DELAY/OP (J-S X 10^{-18})	766.1	804.3	888.6	1072.4
EPPR	0.35	0.42	0.62	1.93

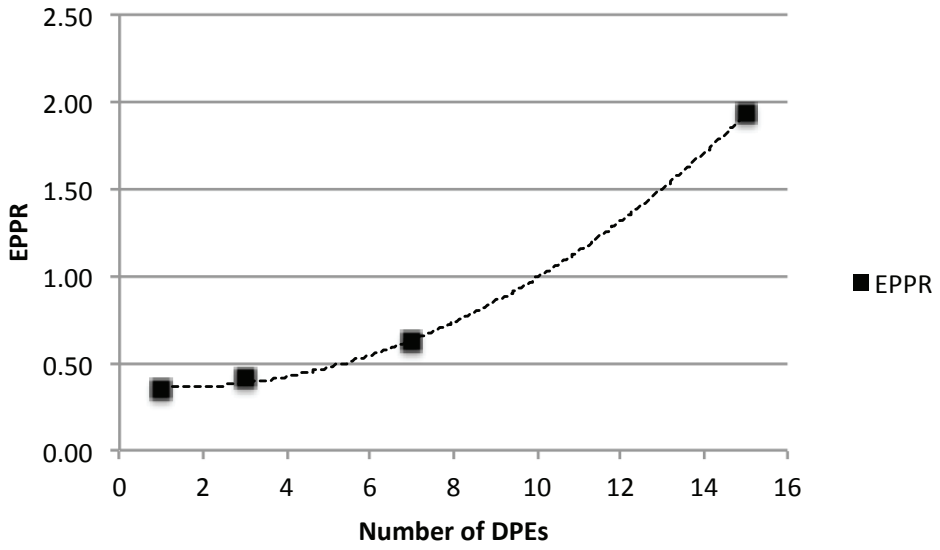


Figure 8.14: Impact of multiple DPEs on EPPR for a tree topology

Figure 8.14 above shows the EPPR for the four multi-DPE systems described above. The variable in this case is the number of cores. For ten DPEs the EPPR is still less than 1. At 15 DPEs the EPPR approaches 2. The decrease in energy efficiency is

primarily due to the limitations of the physical implementation of a multiple DPE substrate resulting in a larger die size and increased wire capacitance (Figure C.2 below). Additionally, the global clocking and signal routing power increases proportionally to the X-Y growth of the resulting die.

Figure 8.15 below shows the impact of token wait times on EPPR for the 4 different DPE platform configurations. Looking at token wait times and the impact on the energy utilization is a useful EPPR to consider as there are a number of variables which impact token wait time including: algorithmic mapping to actors, actor placement on multiple DPE's, clock frequency, etc. For this evaluation the clock frequency is constant and optimized for maximum performance for each configuration. The token delays are randomly introduced into the network by using an event triggered delay element.

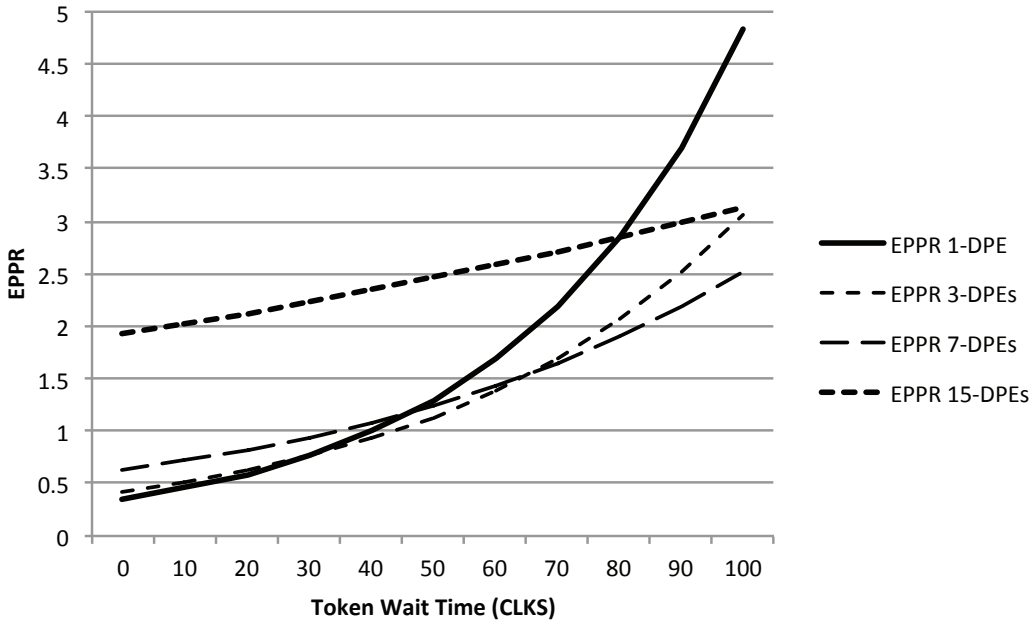


Figure 8.15: Impact of Token Wait Time on EPPR

As expected the impact of token wait times impacts a single DPE more than a 15 DPE platform. Obviously the system designer should optimize the number of DPE's, the clock frequency and actor placement to minimize the effects of token wait time on EPPR.

8.8 DPE Performance and Energy Analysis Summary

Energy-Delay is not a good indicator of energy usage from a benchmarking perspective. It is good for optimizing the physical implementation of a computational element for a specific architectural implementation. Energy-Delay/Operation on the other hand provides a composite view of the impact of architectural and physical implementation decisions on energy usage vs. performance for various benchmarks. EPPR is very useful for determining the sensitive variables of a particular architecture or implementation with respect to the workloads targeted for the platform. These variables are optimized as needed to generate an optimal composite EPPR.

As can be seen from the energy analysis data presented above, the DPE has been optimized for low energy operation with the optimal level of performance for embedded sensor workloads. The composability of the DPE provides the capability to build higher performance systems with minimal degradation of EPPR for certain workloads.

A number of micro-architectural features were instrumental in achieving optimal energy-performance/operation. These have been described above and include:

- One-hot control signals that eliminate decode logic and associated glitching power.
- Novel stack based register file system that uses one-hot shifters that eliminate decoding logic.
- Single cycle pipeline depth.

- Multiplexed latches in the datapath to eliminate spurious transactions
- Single cycle RPT instruction that requires only one access to the microcode memory for each loop.
- Nested looping, which eliminates branch instruction execution.
- Conditional execution, which eliminates branch instruction execution overhead.
- Self timed operation using the SDF “firing” mechanism
- Extensive clock gating.

Chapter 9. Final Observations and Future Work

It is a virtual certainty that battery technology will not improve at the same rate as transistor technology in the foreseeable future. This will require system designers to continuously improve the energy efficiency of energy-limited systems. The challenge is both technical and economical. This dissertation illustrated a number of technical methods to improve energy and computational efficiency. What it did not do is look at the economical feasibility of crafting a new platform for a specific class of workloads. For low volume applications the costs of a COTS design can be prohibitive. Adding flexibility to a platform modifies the computational efficiency by producing an impedance mismatch between the hardware and software domains. Yet it may provide the ability to increase the production volume of the platform and make it feasible to pay for the NRE (non-recurring expense) of designing and validating a new platform. The NOC implementation of the CSP provides some of this flexibility at the expense of increased communication overhead. It can be economically feasible if the number of supported workloads is sufficient [78].

The processing element presented in this dissertation is a unique amalgam of micro-architectural features from 30 years of computer design. These include microcode controlled dataflow engine, Fuzzy Logic acceleration, lookup table capability, Queued-Stack based register file and scale-multiply-accumulate ALU functionality. The low overhead composability of the platform provides excellent scalability that can be matched to the algorithmic workload of a particular sensor system. The ability to directly map and execute SDF based algorithms eliminates the overhead of an operating system and requisite middleware. The mapping of SDF based algorithms to the platform is

accomplished by simply instantiating actors using a graphical modeling and simulation environment.

There are four patentable ideas that resulted from the amalgamation process. The first one is the merged Queued-Stack with its unique control functions to perform multiple stack/queue operations in a single cycle. The second is the Actor/Event queue that uses a variant of the Queued-Stack to dynamically control the sequencing of the DPE while allowing asynchronous events to be inserted into the actor stream and squashed upon execution. The third is the low-energy microcode engine with its n-way looping, repeat function and conditional execution capabilities. The last includes sending actors and/or events with data tokens. The actors or events can be used to modify the operation of the down-stream SDF processing element(s).

There are a number of research areas that can be pursued based on the work in this dissertation. These include:

- Analyze the EPPR for various Network-on-Chip topologies. NOC's are ideal for some algorithms but not all. The EPPR of an NOC is highly dependent on the utilization of the processing elements and the network. There are numerous challenges mapping and scheduling network traffic in NOCs. There is also high level modeling challenges associated with NOCs. As mentioned in Chapter 7, SimEvents® can be used to statistically model the network traffic to determine throughput, however, it may not be adequate to detect potential deadlock or live-lock situations.
- Develop an SDFG model using YAPI [61] to validate the algorithms that will be mapped to an NOC or to a fixed function topology. This model would detect deadlock or live-lock situations. These situations can arise in implementations

where the DPEs are algorithmically changing their operating mode based on the environmental changes that are occurring to sensors, battery, etc.

- Develop an energy modeling environment in SimEvents® that uses attributes attached to tokens to propagate cumulative energy usage data through the network. Each actor can be well characterized as to the amount of energy required to perform a particular function. This information can be attached to the output tokens and analyzed as the tokens enter the communication element.
- Advanced communication protocols such as ANT™ [41] should be researched. Most of the energy in an embedded sensor system is consumed by the communication system. Should a star configuration be used, or would a distributed network system provide better energy utilization?
- The microcode programming environment needs lots of help. While using Excel is a unique method for writing microcode, there are potentially better solutions that should be considered. For systems with a small number of actors, it may be feasible to synthesize the entire microcode control unit while saving area and reducing power.
- The system level programming of the CSP can be done a number of different ways. Algorithms can be specified graphically or in textual format. Which way works the best for an SDF based platform? How is the resulting program mapped to the CSP? Many of the UC Berkley tools look like they could be modified to work in conjunction with YAPI and SimEvents.
- Research additional SFU functionality for algorithmic specific support. Are there better accelerators for data fusing?
- Research QDI (Quasi-Delay0Insensitive) and NDI (Non-Delay-Insensitive) implementations of the DPE.

Appendix A. Fuzzy Logic Tutorial

The flow diagram for a Fuzzy Logic system [30] is shown below in Figure A.1. There are two steps in designing a Fuzzy Logic system. First, the system designer must design a set of rules and membership functions that the evaluation engines will use. Secondly, the system designer has to design a fuzzy inference kernel that takes the system inputs and produces outputs based on the rules and membership functions.

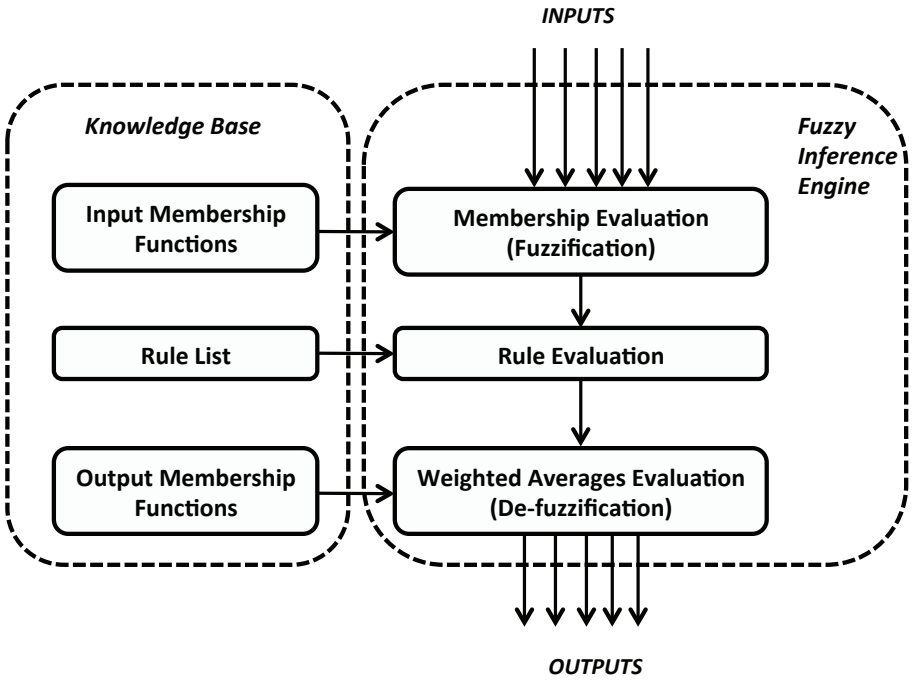


Figure A.1: Flow diagram of a Fuzzy Logic system

A.1.0 Membership Evaluation (Fuzzification)

During membership evaluation, the system input values are compared against stored input membership functions to determine the degree of membership. This is accomplished by finding the y-axis intercept point for the current input value on a trapezoidal membership function as shown below in Figure A.2. The y-axis represents the

degree of membership and the x-axis represents the input value. In this example for an input value of 0.25 the degree of membership is 0.17 (17%). A trapezoidal membership function defines a fuzzy set (the foundation of Fuzzy Logic). To describe a trapezoidal membership function, you need four values: (1) the start point of the trapezoid, (2) the first slope, (3) second slope, and (4) the endpoint of the trapezoid.

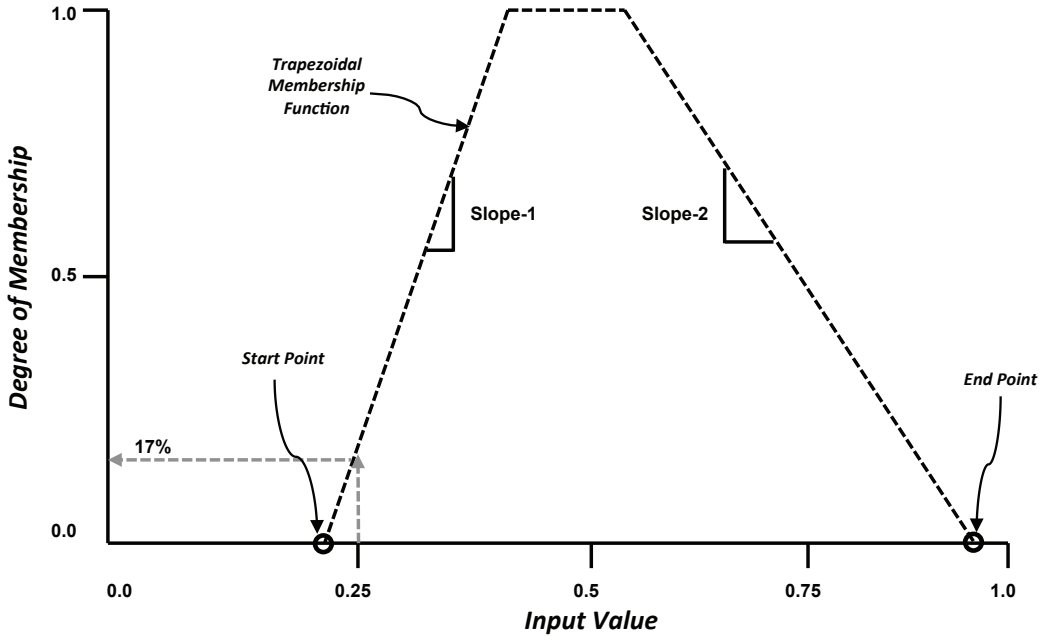


Figure A.2: Trapezoidal Membership Function Example

A fuzzy set is a set without a crisp, clearly defined boundary. It can contain elements with only a partial degree of membership. For example, the membership function for HOT could equal 0.5 for temperatures above 100 degrees and 1.0 for 120 degrees. Any input falling in this range would be considered HOT. There are typically many membership functions that must be evaluated to determine the state of the inputs. Figure A.3 shows an example where there are 3 membership functions: HOT, WARM and COLD. For an input of 64 degrees, the outputs from the membership evaluation are

0% for HOT, 48% for WARM and 83% for COLD. This indicates that it is more COLD than it is WARM and it is definitely not HOT.

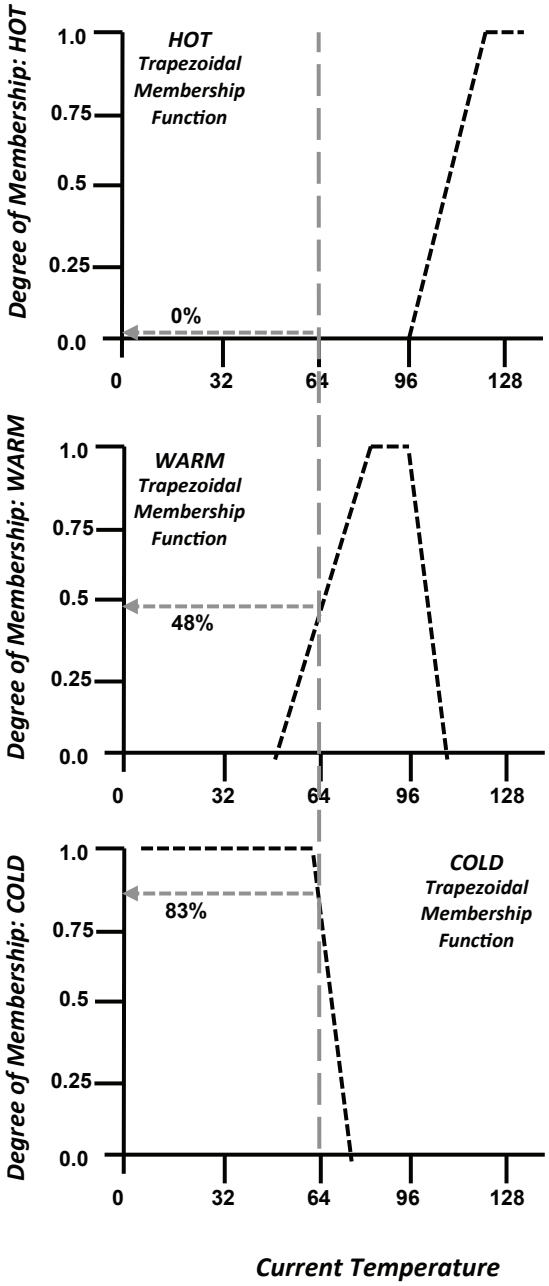
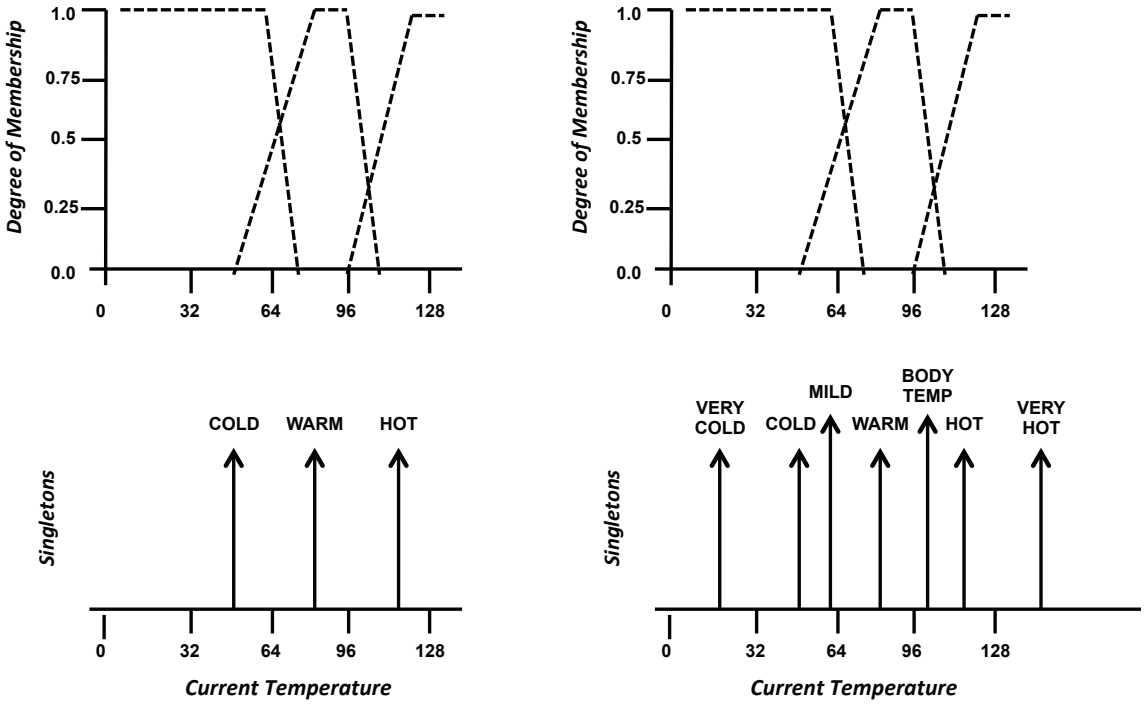


Figure A.3: Output from Fuzzification Operation

The output from the fuzzification operation can be in the form of singletons that are discrete outputs used during the de-fuzzification process. Figure A.4 (a) below shows what this would represent. In this case there are only three conditions, however, there can be multiple singletons as shown below in Figure A.4 (b). The singletons are given linguistic variables that are used in rule generation and evaluation.



(a) (b)

Figure A.4: Singleton output from the Fuzzification process

A.1.1 Rule Evaluation

There are three basic Fuzzy Set operators: Union, Intersection and Complement. The union of two membership functions is calculated using a MAX function:

$$Union\ (\ f_A(x)\ and\ f_B(x)\ ) = max\ (f_A(x)\ ,\ f_B(x)\ ) \qquad A.1$$

An example of this is shown below in Figure A.5 where the union of COLD and WARM membership functions is illustrated.

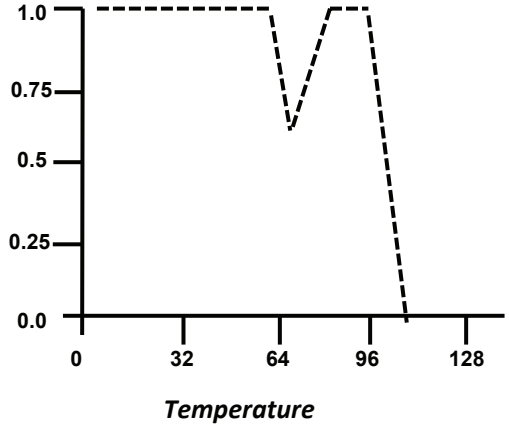


Figure A.5: Union of COLD and WARM Membership Functions

The intersection of WARM and HOT membership functions is shown below in Figure A.6 and is calculated using the MIN function:

$$Intersection\ (\ f_A(x)\ and\ f_B(x)\ ) = min\ (f_A(x)\ ,\ f_B(x)\ ) \qquad A.2$$

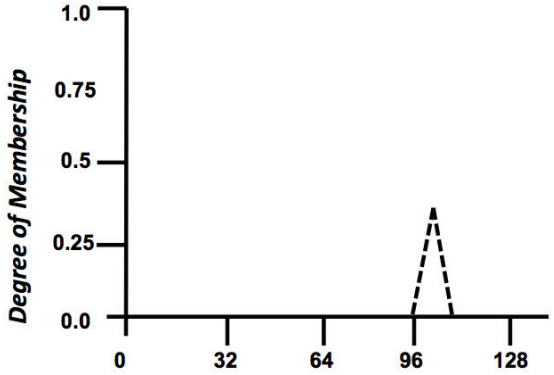


Figure A.6: Intersection of WARM and HOT Membership Functions

The complement of COLD membership functions is shown below in Figure A.7 and is calculated using the following function:

$$Complement\ (f_A(x)) = 1 - (f_A(x)) \qquad A.3$$

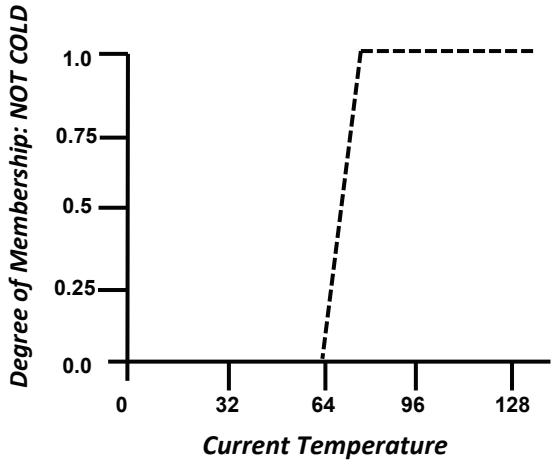


Figure A.7: Complement of the COLD Membership Function

The three Fuzzy Set operators can be also be described using the following truth tables.

Table A.1: Truth Table for Fuzzy Set Complement Operator

F(a)	Complement F(a)
0.0	1.0
0.25	0.75
0.5	0.5
0.75	0.25
1.0	0.0

Table A.2: Truth Table for Fuzzy Set Intersection Operator (MIN)

F(a)	F(b)				
	0.0	**0.25**	**0.5**	**0.75**	**1.0**
0.0	0.0	0.0	0.0	0.0	0.0
0.25	0.0	0.25	0.25	0.25	0.25
0.5	0.0	0.25	0.5	0.5	0.5
0.75	0.0	0.25	0.5	0.75	0.75
1.0	0.0	0.25	0.5	0.75	1.0

Table A.3: Truth Table for Fuzzy Set Union Operator (MAX)

F(a)	F(b)				
	0.0	**0.25**	**0.5**	**0.75**	**1.0**
0.0	0.0	0.25	0.5	0.75	1.0
0.25	0.25	0.25	0.5	0.75	1.0
0.5	0.5	0.5	0.5	0.75	1.0
0.75	0.75	0.75	0.75	0.75	1.0
1.0	1.0	1.0	1.0	1.0	1.0

Rule evaluation performs the actual calculations on the results from the membership evaluations. An example of a rule list is shown below:

```
IF TEMPERATURE IS COLD AND WIND IS HIGH, THEN HEAT IS ON HIGH.
IF TEMPERATURE IS WARM AND WIND IS LOW, THEN HEAT IS ON LOW.
IF TEMPERATURE IS HOT AND WIND IS LOW, THEN HEAT IS OFF.
```

After the fuzzy inputs are evaluated, the system's fuzzy outputs indicate the degree to which an output should have a specific value. These outputs must then undergo de-Fuzzification before their values are useful. Creating the rule list is actually very straightforward. The antecedents (left side of the rule) are the fuzzy inputs created by the membership evaluation (e.g., a temperature reading evaluated with the COLD, WARM and HOT membership functions). The consequents (right side of the rule) are the fuzzy outputs of the system. Each antecedent is joined using the fuzzy Intersection Operator (MIN). This minimum value is compared to the current fuzzy output of each consequent using the fuzzy Union Operator (MAX), and the maximum of these two values is stored in each consequent (fuzzy output). In other words, the overall truth of a rule is stored in the fuzzy outputs and if a subsequent rule is truer, then the fuzzy outputs are updated to reflect this new value.

A.1.2 Defuzzification

The next step in the Fuzzy Logic calculation is defuzzification, where the raw fuzzy outputs are evaluated to create a crisp system output. Defuzzification is performed according to the membership function of the output variable. There are different algorithms for defuzzification as shown below in the following equations:

$$Center\ of\ Gravity \qquad U = \left(\int_{min}^{max} u \times \mu(u)\ du\right) \div \int_{min}^{max} \mu(u)\, du \qquad (A.4)$$

$$Center\ of\ Gravity\ for\ Singletons \qquad U = \left(\sum_{i=1}^{n} u_i \times \mu_i\right) \div \sum_{i=1}^{n} \mu_i \qquad (A.5)$$

$$Left\ Most\ Maximum \qquad U = inf(u')\,, \mu(u') = sup\ (\mu(u)) \qquad (A.6)$$

$$Right\ Most\ Maximum \qquad U = sup(u')\,, \mu(u') = sup\ (\mu(u)) \qquad A.7$$

The variables are described below in Table A.4

Table A.4: Variables used in equations A.4 – A.7

Variable	Meaning
U	RESULT OF DEFUZZIFICATION
U	OUTPUT VARIABLE
N	NUMBER OF SINGLETONS
μ	MEMBERSHIP FUNCTION AFTER ACCUMULATION
I	INDEX
MIN	LOWER LIMIT OF DEFUZZIFICATION
MAX	UPPER LIMIT OF DEFUZZIFICATION
SUP	LARGEST VALUE
INF	SMALLEST VALUE

Rather than using the equations above to calculate the system outputs, a table lookup function can also be used. The output values are predetermined and loaded in the lookup table when the system is initially configured. As the system ages the values in the table can be modified to handle sensor aging, etc.

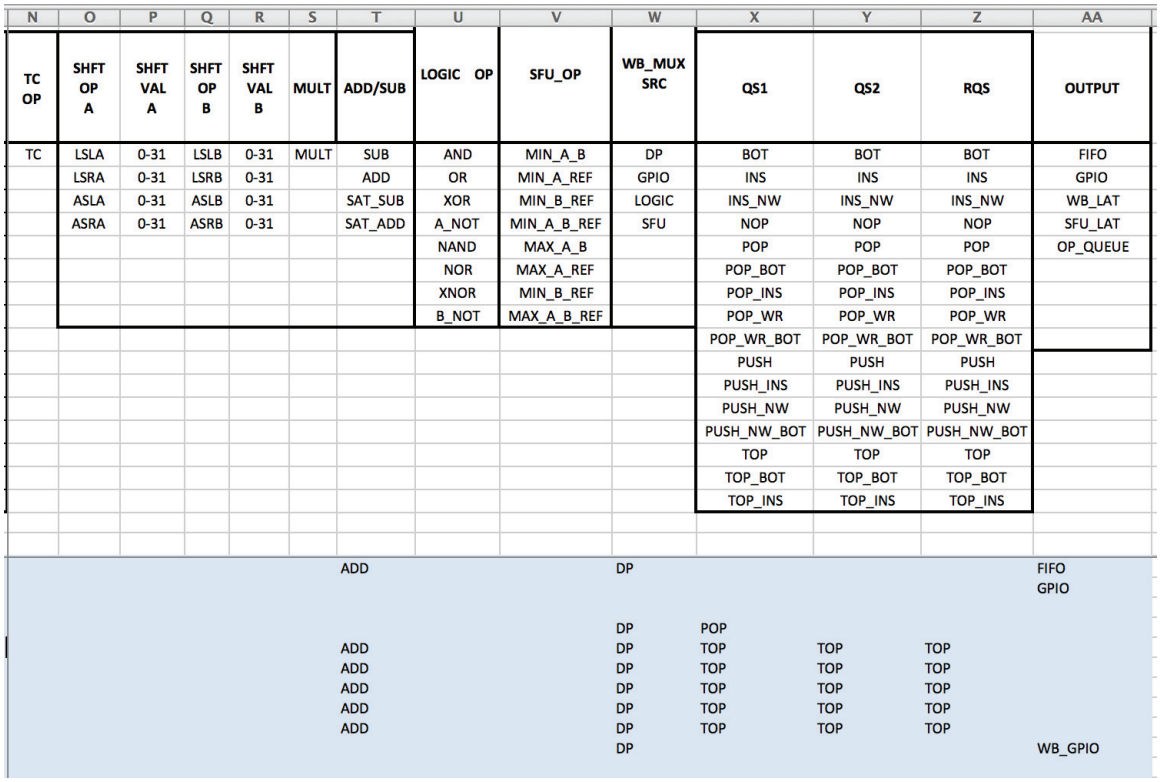
Appendix B. Microcode Assembler

The microcode assembler for the DPE was implemented using a Microsoft Excel spreadsheet. Excel has the ability to do table lookup functions and a broad range of IF-THEN control functions. It also has a concatenate function that is used to compose each microcode word. Additionally, it has the capability to do very useful bit manipulation functions to build various data fields used in the microcode word. Figure B.1 shows how the microcode is entered into the spreadsheet. The grey section is the user entry area. The section above shows the valid entries for each field.

B	C	D	E	F	G	H	I	J	K	L	M
IMMED DATA & JMP ADDR <95:80>	BRANCH OFFSET <79:73>	REPEAT COUNT <70:68>	LOOP NUMBER <72:71>	UCODE OP <67:64>	QS1_BUS	QS2_BUS	A_BUS	B_BUS	RQS_BUS	A_MUX	B_MUX
-32768 to 32767		0-7	0-3	EXEC	TOS_QS1	TOS_QS2	QS1_BUS	QS2_BUS	TOS_RQS	ZERO	ZERO
-32768 to 32767		0-7	0-3	COND_EXEC_GT	BOS_QS1	BOS_QS2	IMMED	IMMED	BOS_RQS	A_BUS	B_BUS
-32768 to 32767		0-7	0-3	COND_EXEC_EQ	TOS_QS2	TOS_QS1	TOS_RQS	TOS_RQS		A_BUS_SHFT	B_BUS_SHFT
-32768 to 32767		0-7	0-3	COND_EXEC_LT	BOS_QS2	BOS_QS1	BOS_RQS	BOS_RQS		RQS_BUS	MULT_BUS
			N/A	WFE							
-512 to 511			N/A	JMP							
-512 to 511			N/A	JMPHALT							
-32768 to 32767	-64 to +63		N/A	BR_WB_GT							
-32768 to 32767	-64 to +63		N/A	BR_WB_EQ							
-32768 to 32767	-64 to +63		N/A	BR_WB_LT							
-32768 to 32767	-64 to +63		N/A	BR_SFU_GT							
-32768 to 32767	-64 to +63		N/A	BR_SFU_EQ							
-32768 to 32767	-64 to +63		N/A	BR_SFU_LT							
-32768 to 32767	-64 to +63		N/A	BRA							
	-64 to 0		0-3	LOOP_BACK							
			N/A	TXFR							
1				EXEC	TOS_QS1	TOS_QS2	IMMED	IMMED	TOS_RQS	A_BUS	B_BUS
0				TXFR	TOS_QS1	TOS_QS2	QS1_BUS	TOS_RQS	TOS_RQS	A_BUS	B_BUS
0				WFE							
0				EXEC	TOS_QS1	BOS_QS2	QS1_BUS	IMMED	BOS_RQS	A_BUS	B_BUS
1		7	1	EXEC	TOS_QS1	TOS_QS2	QS1_BUS	IMMED	TOS_RQS	A_BUS	B_BUS
1	0	7	2	EXEC	TOS_QS1	TOS_QS2	QS1_BUS	IMMED	TOS_RQS	A_BUS	B_BUS
1	0	7	3	LOOP_BACK	TOS_QS1	TOS_QS2	QS1_BUS	IMMED	TOS_RQS	A_BUS	B_BUS
1	-2		2	LOOP_BACK	TOS_QS1	TOS_QS2	QS1_BUS	IMMED	TOS_RQS	A_BUS	B_BUS
1	-4		1	LOOP_BACK	TOS_QS1	TOS_QS2	QS1_BUS	IMMED	TOS_RQS	A_BUS	B_BUS
0				EXEC	TOS_QS1	TOS_QS2	QS1_BUS	IMMED	TOS_RQS	A_BUS	B_BUS
3				JMPHALT							

Figure B.1: Microcode entry example.

Figure B.2 below shows the remaining entry fields for the spreadsheet.



N	O	P	Q	R	S	T	U	V	W	X	Y	Z	AA
TC OP	SHFT OP A	SHFT VAL A	SHFT OP B	SHFT VAL B	MULT	ADD/SUB	LOGIC OP	SFU_OP	WB_MUX SRC	QS1	QS2	RQS	OUTPUT
TC	LSLA	0-31	LSLB	0-31	MULT	SUB	AND	MIN_A_B	DP	BOT	BOT	BOT	FIFO
	LSRA	0-31	LSRB	0-31		ADD	OR	MIN_A_REF	GPIO	INS	INS	INS	GPIO
	ASLA	0-31	ASLB	0-31		SAT_SUB	XOR	MIN_B_REF	LOGIC	INS_NW	INS_NW	INS_NW	WB_LAT
	ASRA	0-31	ASRB	0-31		SAT_ADD	A_NOT	MIN_A_B_REF	SFU	NOP	NOP	NOP	SFU_LAT
							NAND	MAX_A_B		POP	POP	POP	OP_QUEUE
							NOR	MAX_A_REF		POP_BOT	POP_BOT	POP_BOT	
							XNOR	MIN_B_REF		POP_INS	POP_INS	POP_INS	
							B_NOT	MAX_A_B_REF		POP_WR	POP_WR	POP_WR	
										POP_WR_BOT	POP_WR_BOT	POP_WR_BOT	
										PUSH	PUSH	PUSH	
										PUSH_INS	PUSH_INS	PUSH_INS	
										PUSH_NW	PUSH_NW	PUSH_NW	
										PUSH_NW_BOT	PUSH_NW_BOT	PUSH_NW_BOT	
										TOP	TOP	TOP	
										TOP_BOT	TOP_BOT	TOP_BOT	
										TOP_INS	TOP_INS	TOP_INS	
						ADD			DP				FIFO
													GPIO
									DP	POP			
						ADD			DP	TOP	TOP	TOP	
						ADD			DP	TOP	TOP	TOP	
						ADD			DP	TOP	TOP	TOP	
						ADD			DP	TOP	TOP	TOP	
						ADD			DP	TOP	TOP	TOP	
									DP				WB_GPIO

Figure B.2: Microcode entry example (continued)

Figures B.3 and B.4 below illustrate how the microcode field data is generated.

AD	AE	AF	AG	AH	AI	AJ	AK	AL	AM	AN	AO	AP	AQ	AR
IMMED DATA <95:80>	BRANCH OFFSET <79:73>	LOOP COUNTER # <72:71>	REPEAT COUNT <70:68>	UCODE OP <67:64>	READ TOS QS2 <63>	READ TOS QS1 <62>	READ BOS QS2 <61>	READ BOS QS1 <60>	QS2 CTL <59:56>	QS1 CTL <55:52:>	B_BUS MUX <51:50>	A_BUS MUX <49:48>	CHAN MUX QS2 <47>	CHAN MUX QS1 <46>
1111111111111111	0000000	00	000	0000	0	0	0	0	0000	0000	00	00	0	0
0000000111111111	0000000	00	000	0000	0	0	0	0	0000	0000	00	00	0	0
0000000000000001	0000000	01	111	1111	1	1	0	0	0111	0111	01	01	1	1
0000000000000001	0000000	00	000	0000	1	1	0	0	0000	0000	01	01	0	0

Figure B.3: Microcode field generation

AS	AT	AU	AV	AW	AX	AY	AZ	BA	BB	BC	BD	BE	BF	BG	BH	BI	BJ	BK	BL	BM	BN	BO	BP	BQ	BR
SHFT MODE B <45>	SHFT MODE A <44>	SHFT B <43:39>	SHFT A <38:34>	SHFT TC B <33>	SHFT TC A <32>	MULT <31>	SAT <30>	DP EN <29>	ADD SUB <28>	B MUX SEL <27:26>	A MUX SEL <25:24>	QS2 BUS SEL <23:22>	QS1 BUS SEL <21:20>	TC <19>	WB MUX SEL <18:17>	LOGIC OP <16:14>	SFU OP <13:11>	WR SFU LAT <10>	RQS CTL <9:6>	WR WB LAT <5>	RD RQS <4>	RQS BUS MUX <3>	FIFO WE <2>	OP QUEUE <1>	GPIO WE <0>
0	0	00000	00000	0	0	0	0	0	0	00	00	00	00	0	00	000	000	0	0000	0	0	0	0	0	0
0	0	00000	00000	0	0	0	0	0	0	00	00	00	00	0	00	000	000	0	0000	0	0	0	0	0	0
0	0	00000	00000	0	0	0	0	1	0	01	01	00	00	0	00	000	000	0	0000	0	1	0	0	0	0
0	0	00000	00000	0	0	0	0	1	0	01	01	00	00	0	00	000	000	0	0000	0	1	0	1	0	0

Figure B.4: Microcode field generation (continued)

B.1.0 Field generation equations

The field generation equations for each of the microcode fields are presented below:

IMMEDIATE DATA <95:80>:

```
=IF(MID(DEC2BIN($B22,10),1,1)=1,CONCATENATE(111111,DEC2BIN($B22,10)),(CONCATENATE(000000,DEC2BIN($B22,10))))
```

BRANCH OFFSET <79:73>:

```
=MID(DEC2BIN($C22,10),4,7)
```

LOOP COUNTER # < 72:71>

```
=IF($E22<>0,(DEC2BIN($E22,2)),00)
```

REPEAT COUNT <70:68>:

```
=IF($F22=EXEC,(DEC2BIN($D22,3)),(IF($F22=LOOP_BACK,(DEC2BIN($D22,3)),000)))
```

UCODE OP <67:64>:

```
=IF($F22<>0,(VLOOKUP(F22,$BW$4:$BX$19,2)),0000)
```

READ TOS QS2<63>:

=(IF($H22=TOS_QS2,1,(IF($G22=TOS_QS2,1,0))))

READ TOS QS1<62>:

=(IF($G22=TOS_QS1,1,(IF(H22=TOS_QS1,1,0))))

READ BOS QS2<61>:

=(IF(G22=BOS_QS2,1,(IF(H22=BOS_QS2,1,0))))

READ BOS QS1<60>:

=(IF(G22=BOS_QS1,1,(IF(H22=BOS_QS1,1,0))))

QS2 CTL<59:56>:

=IF($Y22<>0,(VLOOKUP($Y22,$CT$4:$CU$11,2)),0000)

QS1 CTL<55:52>:

=IF($X22<>0,(VLOOKUP($X22,$CT$4:$CU$11,2)),0000)

B-BUS MUX CTL<51:50>:

=IF(J22<>0,(VLOOKUP(J22,$CE$4:$CF$7,2)),00)

A-BUS MUX CTL<49:48>:

=IF(I22<>0,(VLOOKUP(I22,$CC$4:$CD$7,2)),00)

QS2 CHANNEL MUX CTL<47>:

=IF(Y22<>0,1,0)

QS1 CHANNEL MUX CTL<46>:

```
=IF(X22<>0,1,0)
```

B-BUS SHIFTER CTL<43:39>:

```
=MID(DEC2BIN(R22,10),6,10)
```

A-BUS SHIFTER CTL<38:34>:

```
=MID(DEC2BIN(P22,10),6,10)
```

B-BUS SHIFTER TC CTL<33>:

```
=IF(Q22=LSRB,1,(IF(Q22=ASRB,1,0)))
```

A-BUS SHIFTER TC CTL<32>:

```
=IF(O22=LSRA,1,(IF(O22=ASRA,1,0)))
```

MULTIPLIER ENABLE<31>:

```
=IF(S22=MULT,1,0)
```

ADDER SATURATION MODE CTL<30>:

```
=IF(T22=SAT_ADD,1,(IF(T22=SAT_SUB,1,0)))
```

DATAPATH ENABLE<29>:

```
=IF(T22<>0,1,(IF(S22<>0,1,(IF(Q22<>0,1,(IF(O22<>0,1,0)))))))
```

ADD/SUB CTL<28>:

```
=IF(T22=SUB,1,(IF(T22=SAT_SUB,1,0)))
```

B-MUX SEL<27:26>:

```
=IF($M22<>0,(VLOOKUP($M22,$CI$4:$CJ$7,2)),00)
```

A-MUX SEL<25:24>:

```
=IF($L22<>0,(VLOOKUP($L22,$CG$4:$CH$7,2)),00)
```

QS2 BUS MUX SELECT<23:22>:

```
=IF($H22<>0,(VLOOKUP($H22,$CA$4:$CB$7,2)),00)
```

QS1 BUS MUX SELECT<21:19>:

```
=IF($G22<>0,(VLOOKUP($G22,$BY$4:$BZ$7,2)),00)
```

TWO'S COMPLEMENT CTL<19>:

```
=IF(N22=TC,1,0)
```

WRITEBACK MUX SELECT<18:17>:

```
=IF(W22<>0,(VLOOKUP(W22,$CO$4:$CP$7,2)),00)
```

LOGIC OPERATION CTL<16:14>:

```
=IF(U22<>0,(VLOOKUP(U22,$CM$4:$CN$11,2)),000)
```

SPECIAL FUNCTION UNIT CTL<13:11>:

```
=IF(V22<>0,(VLOOKUP(V22,$CQ$4:$CR$9,2)),000)
```

WRITE SFU REGISTER< 10>:

```
=IF(AA22=SFU_LAT,1,0)
```

RQS CONTROL<9:6>:

```
=IF($Z22<>0,(VLOOKUP($Z22,$CT$4:$CU$11,2)),0000)
```

WRITE WB REGISTER<5>:

```
=(IF($AA22=WB_LAT,1,0))
```

READ RQS CONTROL<4>:

```
=IF(I22<>0,1,(IF(J22<>0,1,(IF(K22<>0,1,0)))))
```

RQS BUS MUX CONTROL<3>:

```
=IF(K22=BOS_RQS,1,0)
```

FIFO WRITE ENABLE<2>:

```
=IF($AA22=FIFO,1,0)
```

OPERATION QUEUE WRITE ENABLE<1>:

```
=IF($AA22=OP_QUEUE,1,0)
```

GPIO WRITE ENABLE<0>:

```
=IF($AA22=GPIO,1,0)
```

B.1.1 Lookup tables

The lookup tables used in the equations above are shown below:

CI	CJ	CK	CL	CM	CN	CO	CP	CQ	CR	CT	CU
B_MUX		**RQS_BUS**		**LOGIC OP**		**WB_MUX SRC**		**SFU_OP**		**QS & RQS Encoding**	
B_BUS	01	BOS_RQS	1	A_NOT	011	DP	00	MIN_A_B	100	BOT	0001
B_BUS_SHFT	10	TOS_RQS	0	AND	000	GPIO	01	MIN_A_REF	110	INS	0010
MULT	11			B_NOT	111	LOGIC	10	MIN_B_REF	101	INS_NW	0011
ZERO	00			NAND	100	SFU	11	MIN_A_B_REF	111	NOP	0000
				NOR	101			MAX_A_B	000	POP	0100
				OR	001			MAX_A_REF	010	POP_BOT	0101
				XNOR	110			MIN_B_REF	001	POP_INS	0110
				XOR	010			MAX_A_B_REF	011	POP_WR	0111
										POP_WR_BOT	1000
										PUSH	1001
										PUSH_INS	1011
										PUSH_NW	1100
										PUSH_NW_BOT	1010
										TOP	1101
										TOP_BOT	1110
										TOP_INS	1111

BW	BX	BY	BZ	CA	CB	CC	CD	CE	CF	CG	CH	CI	CJ
Lookup table for Micro OPCODES		**QS1_BUS**		**QS2_BUS**		**A_BUS**		**B_BUS**		**A_MUX**		**B_MUX**	
BR_SFU_EQ	1100	BOS_QS1	01	BOS_QS1	11	BOS_RQS	11	BOS_RQS	11	A_BUS	01	B_BUS	01
BR_SFU_GT	1011	BOS_QS2	11	BOS_QS2	01	IMMED	01	IMMED	01	A_BUS_SHFT	10	B_BUS_SHFT	10
BR_SFU_LT	1101	TOS_QS1	00	TOS_QS1	10	QS1_BUS	00	QS2_BUS	00	RQS_BUS	11	MULT	11
BR_WB_EQ	1001	TOS_QS2	10	TOS_QS2	00	TOS_RQS	10	TOS_RQS	10	ZERO	00	ZERO	00
BR_WB_GT	1010												
BR_WB_LT	1000												
BRA	1110												
COND_EXEC_EQ	0101												
COND_EXEC_GT	0100												
COND_EXEC_LT	0110												
EXEC	0000												
JMP	0010												
JMPHALT	0011												
LOOP_BACK	1111												
TXFR	0111												
WFE	0001												

Appendix C. DCT/ICC Implementation Details

The DPE was synthesized in a 180nm TSMC process using Design Compiler Topographical (DCT) from Synopsys. The synthesized output was placed and routed using IC Compiler (ICC) from Synopsys. Figure C.1 below shows the layout of the DPE from ICC. The microcode RAM is the regular structure in the bottom center of the layout. Figure C.2 shows the layout for a 15 DPE implementation.

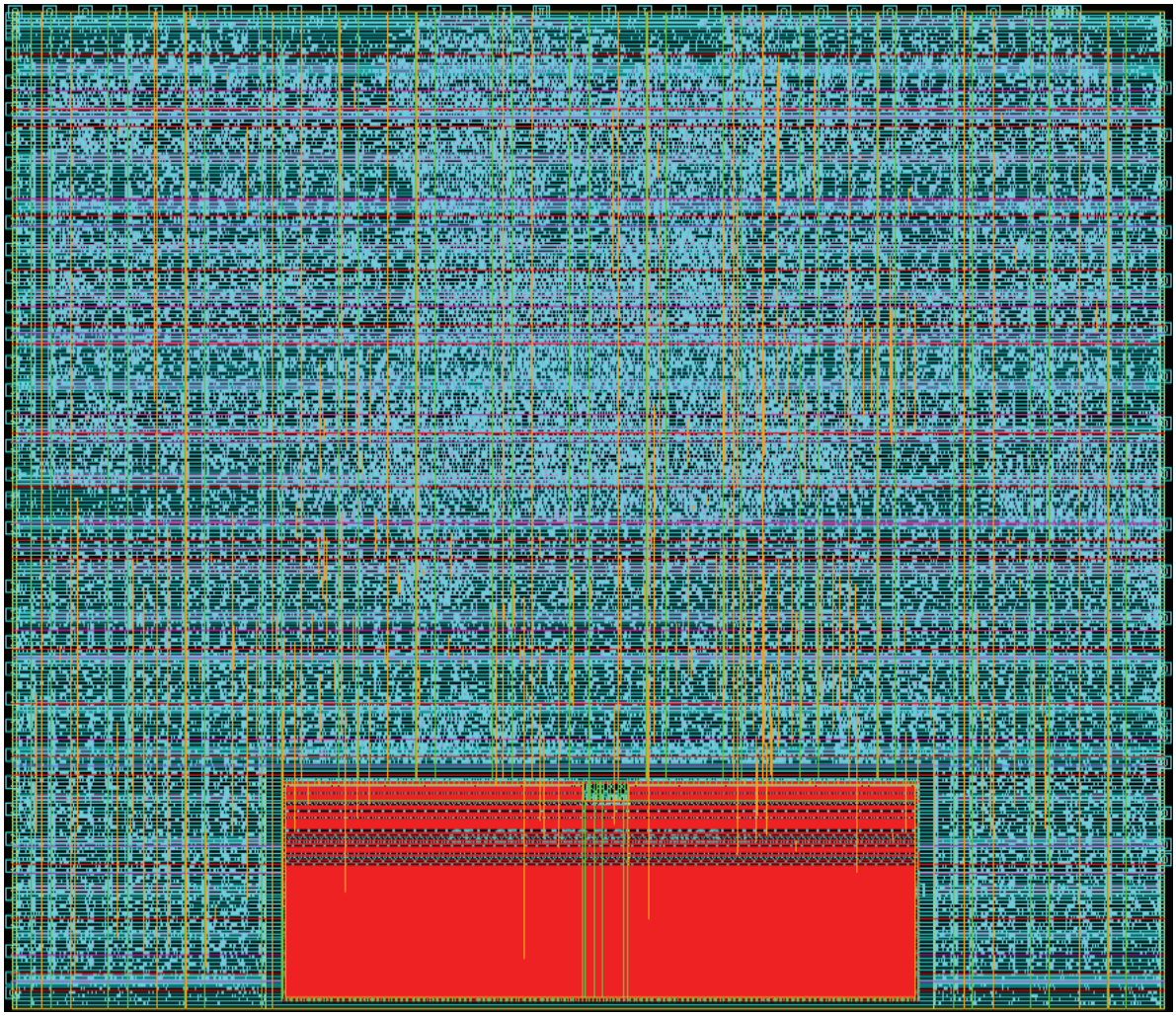


Figure C.1: Layout of a single DPE (from IC Compiler)

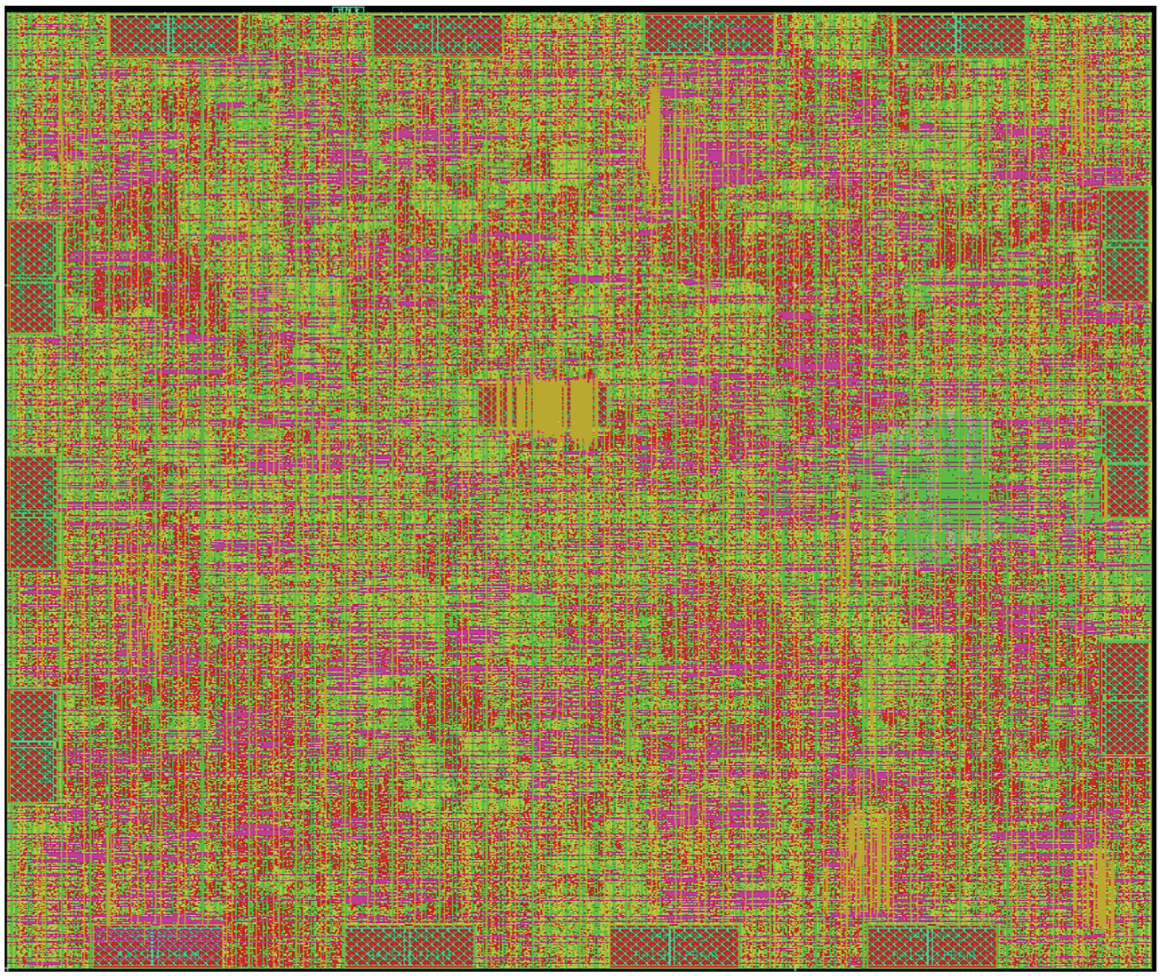


Figure C.2: Layout of a 15-DPE implementation (from IC Compiler)

C.1.0 Synthesis Constraints

The synthesis constraints for a 25MHz (single DPE) implementation are shown below. The DPE was synthesized for a range of cycle times ranging from 5ns-100ns.

```
# Set cycle time in nanoseconds

create_clock -period 40 clk -waveform {0 20}
create_clock -period 40 clk90 -waveform {10 30}

#  Set driving cell on all inputs
set_driving_cell -lib_cell INVX2 [all_inputs]

#  Isolate ports on all outputs
set_isolate_ports [all_outputs] -force

#  Set load on all outputs
set_load 0.20 [all_outputs]


set_input_delay -max[expr 0.5 * ($CCT)]  -clock clk  [ get_nets cpe_hold_in ]
set_input_delay -max[expr 0.5 * ($CCT)]  -clock clk  [ get_nets cpe_ready_in ]
set_input_delay -max[expr 0.1 * ($CCT)]  -clock clk  [ get_nets reset_b ]
set_input_delay -max[expr 0.1 * ($CCT)]  -clock clk  [ get_nets cpe_token_ck_in ]
set_input_delay -max[expr 0.2 * ($CCT)]  -clock clk  [ get_nets scan_enable ]
set_input_delay -max[expr 0.3 * ($CCT)]  -clock clk  [ get_nets scan_in ]


set_output_delay -max  [expr 0.1 * ($CCT)]  -clock clk [ get_nets cpe_token_ck_out]
set_output_delay -max  [expr 0.2 * ($CCT)]  -clock clk [ get_nets cpe_hold_out]
set_output_delay -max  [expr 0.2 * ($CCT)]  -clock clk [ get_nets cpe_ready_out]

remove_attribute [get_lib_cells */*FF*] dont_use

set_dont_use [get_lib_cells */*XL ]
set_dont_use [get_lib_cells */*CLK* ]

set_max_transition 1.0 [get_designs]
set_fix_hold clk
set_fix_hold clk90


compile_ultra -no_autoungroup -timing_high_effort_script
compile_ultra  -incremental -only_design_rule -no_autoungroup

 set j 0;
 while {$j<5} {
   set x [get_timing_path]
   set slack [get_attribute $x slack]
   if {$slack < 0 } {
     compile_ultra -incremental -no_autoungroup
     incr j
   } else {
     set j 10
   }
 }
```

C.1.1 Critical Timing Paths

The worse case timing path in the DPE is from the SFU compare register write, the MIN/MAX compare and to the microcode finite state machine. It is a quarter cycle path: CLK (fall) to CLK90 (fall). This path is a false path as “writes” to the compare register followed by a compare is not valid. If this register-write operation is followed by a conditional branch then there is a chance that the branch operation may fail as the validity of the condition code bit is questionable.

```
****************************************
Report : timing
        -path full
        -delay min
        -max_paths 1
        -transition_time
Design : dpe
Version: E-2010.12
Date   : Oct 26 13:40:58 2012

  Startpoint: dp/dp_sfu_0/ref_data_reg_13_
              (rising edge-triggered flip-flop clocked by clk')
  Endpoint: ctl/ctl_ucode_1/ucode_fsm/wb_lt_lat_reg
            (falling edge-triggered flip-flop clocked by clk90)
  Path Group: clk90
  Path Type: max

  Point                                                   Incr       Path
  --------------------------------------------------------------------------
  clock clk (fall edge)                                   5.00       5.00
  clock network delay (ideal)                             0.00       5.00
  dp/dp_sfu_0/ref_data_reg_13_/CK (DFFSX4)                0.00       5.00 r
  dp/dp_sfu_0/ref_data_reg_13_/QN (DFFSX4)                0.24       5.24 r
  dp/dp_sfu_0/U207/Y (NAND2X2)                            0.05 *     5.28 f
  dp/dp_sfu_0/U292/Y (AND2X4)                             0.13 *     5.42 f
  dp/dp_sfu_0/U248/Y (NOR2X4)                             0.08 *     5.49 r
  dp/dp_sfu_0/min_max_unit_minmax41_0_prefix13_UGT1_1_6_0/Y (OAI21X2)
                                                          0.06 *     5.55 f
  dp/dp_sfu_0/min_max_unit_minmax41_0_prefix13_UGT0_2_3_1/Y (AOI21X1)
                                                          0.12 *     5.67 r
  dp/dp_sfu_0/min_max_unit_minmax41_0_prefix13_UGT1_3_1_3/Y (OAI21X1)
                                                          0.07 *     5.74 f
  dp/dp_sfu_0/min_max_unit_minmax41_0_prefix13_UGT0_4_0_7/Y (AOI21X2)
                                                          0.10 *     5.84 r
  dp/dp_sfu_0/U151/Y (XOR2X2)                             0.19 *     6.03 r
  dp/dp_sfu_0/min_max_unit_minmax41_0_U1/Y (OAI2BB2X4)
                                                          0.14 *     6.17 r
  dp/dp_sfu_0/U43/Y (BUFX20)                              0.12 *     6.29 r
  dp/dp_sfu_0/U374/Y (MXI2X4)                             0.11 *     6.41 f
  dp/dp_sfu_0/sfu_out[2] (dp_sfu)                         0.00       6.41 f
  dp/dp_wbmux_0/lut_in[2] (dp_wbmux)                      0.00       6.41 f
  dp/dp_wbmux_0/U187/Y (OAI2BB1X4)                        0.15 *     6.55 f
  dp/dp_wbmux_0/wb_cmp/wb_data[2] (wb_cmp)                0.00       6.55 f
  dp/dp_wbmux_0/wb_cmp/cmp_eq/A[2] (cmp6)                 0.00       6.55 f
```

```
  dp/dp_wbmux_0/wb_cmp/cmp_eq/gt_x_5631_2_U184/Y (NAND2X1)
                                                          0.08 *     6.63 r
  dp/dp_wbmux_0/wb_cmp/cmp_eq/U132/Y (OAI2BB1X2)          0.10 *     6.73 r
  dp/dp_wbmux_0/wb_cmp/cmp_eq/U131/Y (OAI21X1)            0.06 *     6.79 f
  dp/dp_wbmux_0/wb_cmp/cmp_eq/gt_x_5631_2_U178/Y (AOI21X2)
                                                          0.11 *     6.89 r
  dp/dp_wbmux_0/wb_cmp/cmp_eq/gt_x_5631_2_U163/Y (OAI21X2)
                                                          0.06 *     6.95 f
  dp/dp_wbmux_0/wb_cmp/cmp_eq/gt_x_5631_2_U132/Y (AOI21X2)
                                                          0.18 *     7.13 r
  dp/dp_wbmux_0/wb_cmp/cmp_eq/gt_x_5631_2_U1/Y (OAI21X4)
                                                          0.06 *     7.19 f
  dp/dp_wbmux_0/wb_cmp/cmp_eq/U128/Y (NOR2X1)             0.10 *     7.29 r
  dp/dp_wbmux_0/wb_cmp/cmp_eq/LT (cmp6)                   0.00       7.29 r
  dp/dp_wbmux_0/wb_cmp/wb_lt (wb_cmp)                     0.00       7.29 r
  dp/dp_wbmux_0/wb_lt (dp_wbmux)                          0.00       7.29 r
  dp/wb_lt (dp)                                           0.00       7.29 r
  ctl/wb_lt (ctl)                                         0.00       7.29 r
  ctl/ctl_ucode_1/wb_lt (ctl_ucode)                       0.00       7.29 r
  ctl/ctl_ucode_1/ucode_fsm/wb_lt (ctl_ucode_fsm)         0.00       7.29 r
  ctl/ctl_ucode_1/ucode_fsm/U70/Y (MXI2X1)                0.06 *     7.35 f
  ctl/ctl_ucode_1/ucode_fsm/wb_lt_lat_reg/D (DFFNSX1)     0.00 *     7.35 f
  data arrival time                                                  7.35

  clock clk90 (fall edge)                                 7.50       7.50
  clock network delay (ideal)                             0.00       7.50
  ctl/ctl_ucode_1/ucode_fsm/wb_lt_lat_reg/CKN (DFFNSX1)
                                                          0.00       7.50 f
  library setup time                                     -0.15       7.35
  data required time                                                 7.35
  ---------------------------------------------------------------------------
  data required time                                                 7.35
  data arrival time                                                 -7.35
  ---------------------------------------------------------------------------
  slack (MET)                                                        0.00
```

C.1.2 PTPX Power Analysis Results

The results from PTPX for a single DPE core running at 25MHz operation are shown below. A SAIF (switching activity interchange format) pattern was used to generate the power numbers for the FIR filter operation. Energy was then derived using equation 8.12 and used in the analysis presented in Chapter 8.

```
****************************************************************************
Report : Averaged Power
        -cell_power
        -verbose
        -sort_by cell_internal_power
        -power_greater_than       0
Design : dpe
Version: H-2012.12
Date   : Nov  5 14:28:54 2012

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

```
Library(s) Used:

    typical (File: /IMPLEMENTATION/Artisan/synopsys/typical.lib)
    typical (File: /IMPLEMENTATION/Artisan/synopsys/typical.db)
    dw_foundation.sldb (File:
/usr/local/packages/synopsys_2012/syn/libraries/syn/dw_foundation.sldb)
    RA1SHD (File: /home/ecelrc/faculty/mcdermot/IMPLEMENTATION/WCS/RA1SHD.db)


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

Operating Conditions: typical   Library: typical
Wire Load Model Mode: Parasitic

Power-specific unit information:
    Voltage Units             = 1 V
    Capacitance Units         = 1 pf.
    Time Units                = 1 ns
    Dynamic Power Units       = 1 W
    Leakage Power Units       = 1 W
    Cycle-time                = 40 ns


  Attributes
  ----------
      a  -  Annotated internal & leakage power
      b  -  Black-box (unresolved) cell
      c  -  Clock pin internal power only
      d  -  Does not include clock pin internal power
      h  -  Hierarchical cell

                    Internal  Switching  Leakage    Total
Cell                Power     Power      Power      Power   (     %)  Attrs
--------------------------------------------------------------------------------
dp                  4.106e-06 4.587e-06 5.107e-09 8.699e-06 (43.74%)  h
ctl                 3.504e-06 6.223e-07 9.070e-08 4.217e-06 (21.20%)  h
qs1                 6.966e-07 1.462e-06 9.770e-09 2.168e-06 (10.90%)  h
rqs                 6.373e-07 1.797e-06 2.670e-08 2.461e-06 (12.37%)  h
qs2                 3.994e-07 1.287e-06 9.483e-09 1.696e-06 ( 8.53%)  h
fifo                1.416e-07 1.168e-07 6.984e-10 2.591e-07 ( 1.30%)  h

--------------------------------------------------------------------------------
Totals (6 cells)    9.560e-06 1.019e-05 1.425e-07 1.989e-05 (100.0%)
```

Note that the ratio of datapath power to control power is 43.7% to 21.2%. This is key to the energy efficiency of the DPE. In this implementation the CTL power is dominated by the RAM microcode memory. This can be reduced by about 35% by using a read-only memory (ROM). Figure C.3 below shows the layout differences between the ROM and the RAM layouts.

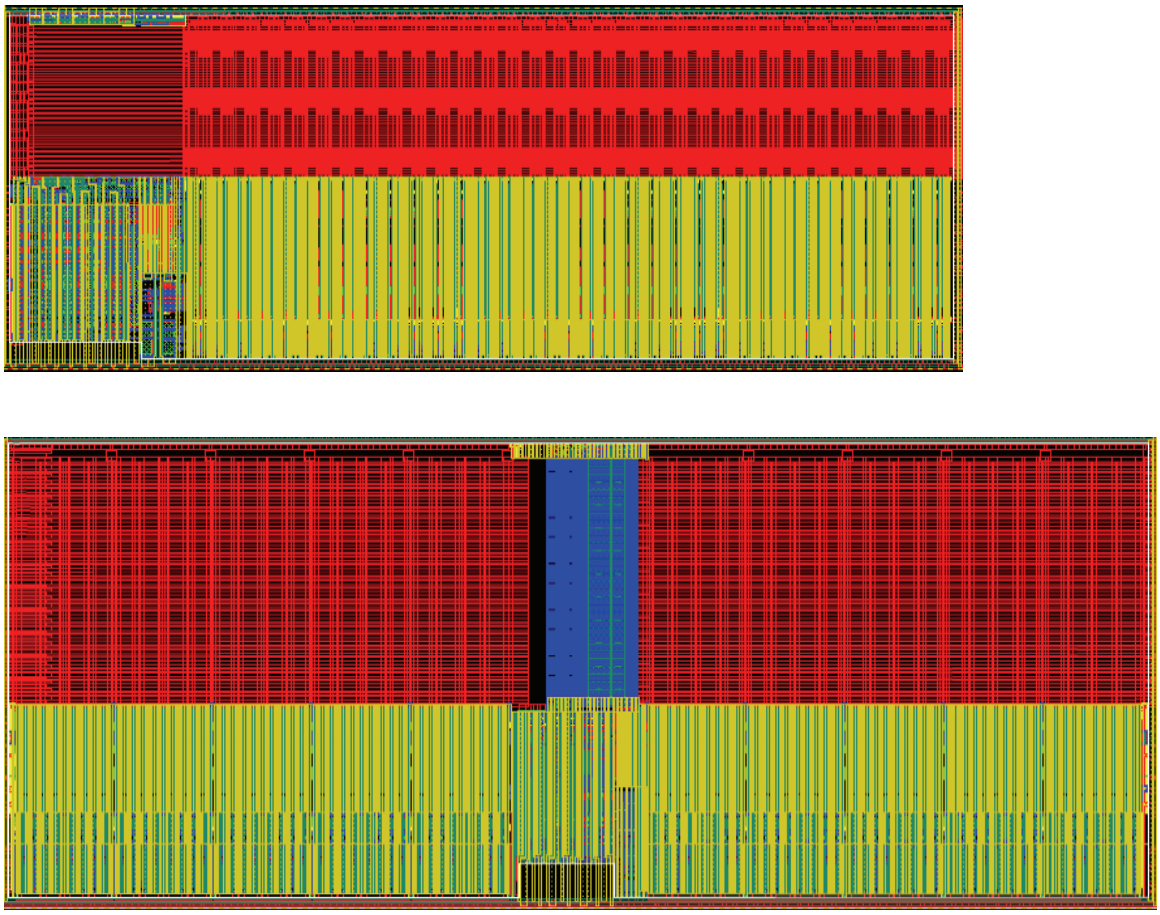


Figure C.3: ROM (top) vs. RAM (bot) layout comparison

Appendix D. FPGA Implementation Details

The TLL-5000 System Design Platform was used to implement and test the DPE [79]. The TLL5000 is a hardware and software design platform that consists of a Xilinx Spartan-3 XC3S1500 FPGA (field-programmable gate array) and a large assortment of peripherals including the following:

- 16 MB Flash
- 16MG DRAM
- LEDs (light-emitting diodes)
- LCD (liquid crystal display)
- SD (secure digital) card/MMC (Multi-Media-Card) I/F
- Video encoder/decoder with video input/output ports
- VGA output
- Audio codec with audio amplifier and output connection
- Microphone and audio input connection
- Mouse and keyboard ports,
- Ethernet interface
- User switches and push buttons

These peripherals are powered by on-board power supplies and driven by the FPGA and internal clocks. The TLL500 has two 80-pin mezzanine expansion connectors for additional processing or interfacing capabilities. Currently, one of these mezzanine connectors is used to interface the TLL6219 ARM9 Module to the Baseboard. Figure D.1 below shows a photograph of the TLL5000 Baseboard and its various on-board components.

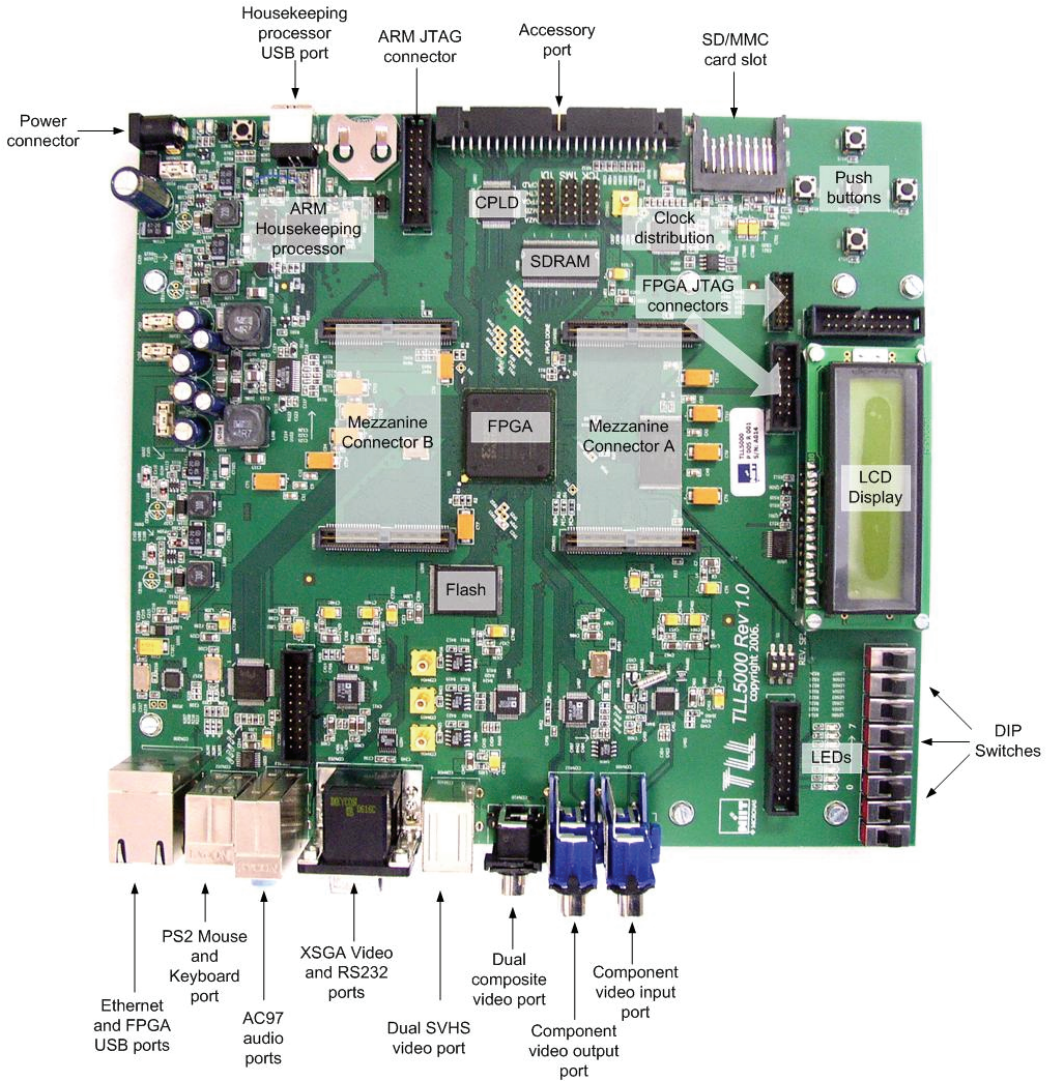


Figure D.1: TLL5000 System Development Platform

The synthesized gate level netlist from DCT was used as the netlist for the FPGA implementation of the DPE. The netlist was instantiated into a bus controller Verilog model that provides the interface between the ARM SOC and the DPE. Figure D.2 shows the block diagram of the DPE test environment.

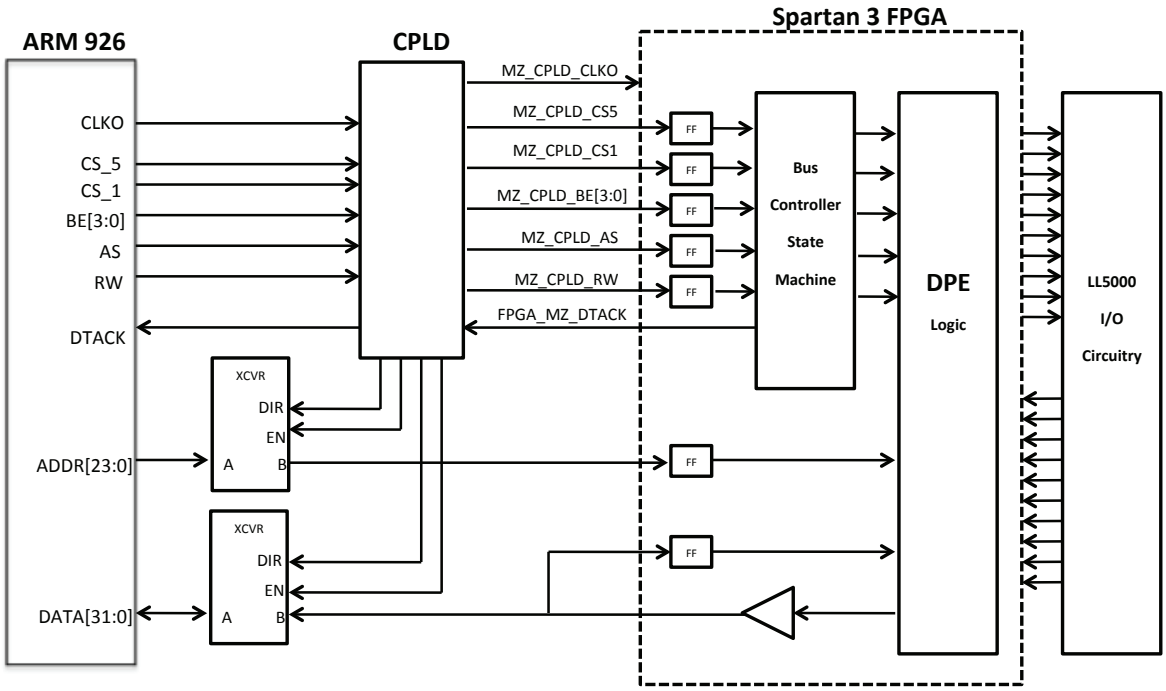


Figure D.2: Block diagram of the DPE test environment

The ARM processor is used to load the microcode in the DPE. The microcode memory is memory mapped into the ARM processor memory map. The ARM processor is uses a memory mapped control register in the FPGA to "fire" the DPE. The output of the DPE FIFO is accessible via the accessory port on TLL5000 and was used to verify correct system operation using a logic analyzer. Additional internal control and clock signals were routed to the accessory connector. These were compared to the original Verilog simulation to further confirm correct internal operation. Using the synthesized gate level netlist was instrumental in making sure the silicon and FPGA designs functioned identically. The same self-checking test that was used to verify the RTL functionality of DPE was used to verify the FPGA implementation. The IQS-1 is used to

seed the test by loading tokens and triggering the FIRE signal. The output of the FIFO is recirculated back into IQS-2 and used as the feedback mechanism for the test. The ARM processor can read the result of the test via a register in the bus controller. The microcode WCS is memory mapped into the ARM address space allowing for dynamic changing of the microcode for debugging, reloading, etc.

Figure D.3 shows a block diagram of the test environment for the FPGA implementation of the DPE.

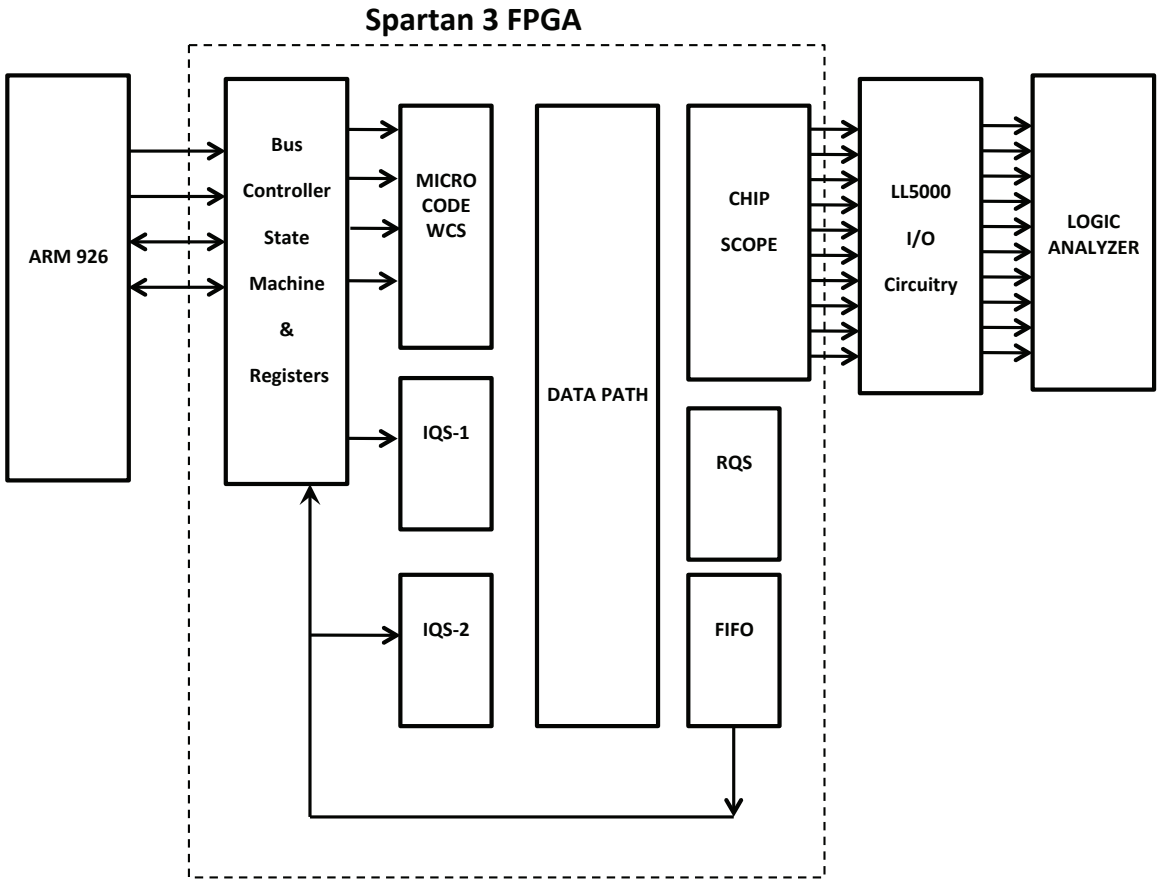


Figure D.3: Test configuration for FPGA implementation of the DPE.

D.1.0 FPGA Implementation Details

Figure D.4 below shows the post synthesis, mapping and place & route implementation summary.

top Project Status (12/12/2013 - 09:57:04)			
Project File:	TOP_GATE.xise	**Parser Errors:**	No Errors
Module Name:	top	**Implementation State:**	Placed and Routed
Target Device:	xc3s1500-4fg676	• **Errors:**	No Errors
Product Version:	ISE 14.4	• **Warnings:**	3591 Warnings (372 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Total Number Slice Registers	3,612	26,624	13%	
Number used as Flip Flops	332			
Number used as Latches	3,280			
Number of 4 input LUTs	8,093	26,624	30%	
Number of occupied Slices	5,873	13,312	44%	
Number of Slices containing only related logic	5,873	5,873	100%	
Number of Slices containing unrelated logic	0	5,873	0%	
Total Number of 4 input LUTs	8,106	26,624	30%	
Number used as logic	8,093			
Number used as a route-thru	13			
Number of bonded IOBs	134	487	27%	
Number of RAMB16s	4	32	12%	
Number of BUFGMUXs	8	8	100%	
Average Fanout of Non-Clock Nets	4.15			

Performance Summary			[-]
Final Timing Score:	0 (Setup: 0, Hold: 0, Component Switching Limit: 0)	**Pinout Data:**	Pinout Report
Routing Results:	All Signals Completely Routed	**Clock Data:**	Clock Report
Timing Constraints:	All Constraints Met		

Detailed Reports					[-]
Report Name	**Status**	**Generated**	**Errors**	**Warnings**	**Infos**
Synthesis Report	Current	Thu Dec 12 09:53:00 2013	0	3340 Warnings (157 new)	5 Infos (5 new)
Translation Report	Current	Thu Dec 12 09:55:42 2013	0	6 Warnings (3 new)	0
Map Report	Current	Thu Dec 12 09:56:01 2013	0	125 Warnings (125 new)	4 Infos (1 new)
Place and Route Report	Current	Thu Dec 12 09:56:56 2013	0	112 Warnings (83 new)	1 Info (1 new)
Power Report					
Post-PAR Static Timing Report	Current	Thu Dec 12 09:57:04 2013	0	8 Warnings (4 new)	5 Infos (0 new)
Bitgen Report					

Secondary Reports		[-]
Report Name	**Status**	**Generated**

Date Generated: 12/12/2013 - 09:57:05

Figure D.4: FPGA implementation summary

D.1.1 Synthesis Timing Constraints

The timing constraints for the FPGA synthesis of the DPE and the bus controller logic is shown below:

```
# ***************************************************
# RESET PIN
# ***************************************************
NET SYS_RST_N LOC=AB11;
NET SYS_RST_N IOSTANDARD = LVCMOS33;
NET SYS_RST_N TIG;   # IGNORE TIMING. RST IS SYNCHRONIZED INTERNALLY
# ***************************************************
# 24 MHz CLOCK INPUT
# ***************************************************
NET SYS_CLK LOC=AE14;
NET SYS_CLK IOSTANDARD = LVCMOS33;
NET SYS_CLK   TNM_NET  = SYS_CLK;
TIMESPEC TS_SYS_CLK = PERIOD SYS_CLK   41.6666 NS HIGH 50 %;
# ***************************************************
# 100MHZ/N CLOCK INPUT FROM AD9510 CLOCK GENERATOR
# ***************************************************
NET FPGA_CLK3 LOC = AF14;
NET FPGA_CLK3 IOSTANDARD = LVCMOS33;
NET FPGA_CLK3 TNM_NET    = "FPGA_CLK3";
TIMESPEC "TS_FPGA_CLK3"  = PERIOD "FPGA_CLK3"  10 NS HIGH 50 %;
# ***************************************************
# FPGA_CLK4 CONFIGURED FOR VE_CLK (27MHZ) ON BOARD
# ***************************************************
NET FPGA_CLK4 LOC = AE13;
NET FPGA_CLK4 IOSTANDARD = LVCMOS33;
NET "FPGA_CLK4" TNM_NET = "FPGA_CLK4";
#TIMESPEC "TS_FPGA_CLK4"    = PERIOD "FPGA_CLK4"  37 NS HIGH 50 %;
# ***************************************************
# CPLD CLK FROM ARM CORE
# ***************************************************
NET "MZ_CPLD_CLKO" LOC = "AA18";
NET MZ_CPLD_CLKO IOSTANDARD = LVCMOS33;
#NET "MZ_CPLD_CLKO" TNM_NET = "MZ_CPLD_CLKO";
TIMESPEC "TS_MZ_CPLD_CLK0"  = PERIOD "MZ_CPLD_CLKO" 20 NS HIGH 50 %;
# ***************************************************
TIMEGRP "EB"        OFFSET = IN 6 NS    BEFORE "SYS_CLK";
TIMEGRP "ADDR"      OFFSET = IN 6 NS    BEFORE "SYS_CLK";
TIMEGRP "DATABUS"   OFFSET = OUT 11 NS  AFTER  "SYS_CLK";
# ***************************************************
```

The DPE was synthesized for 12 MHz operation using the SYS_CLK clock, which is driven by an external 24MHz oscillator on the TLL5000. The SYS_CLK is divided by 2 and phase shifted 90 degrees to generate the two clocks for the DPE.

D.1.2 Synthesis Timing Report

The resulting timing report is shown below. The synthesized DPE netlist is able to meet the 12 MHz timing constraint.

```
--------------------------------------------------------------------------------
Release 14.4 Trace  (lin)
Copyright (c) 1995-2012 Xilinx, Inc.  All rights reserved.

/misc/linuxws/packages/Xilinx/14.4/ISE_DS/ISE/bin/lin/unwrapped/trce -intstyle
ise -v 3 -s 4 -n 3 -fastpaths -xml top.twx top.ncd -o top.twr top.pcf

Design file:              top.ncd
Physical constraint file: top.pcf
Device,package,speed:     xc3s1500,fg676,-4 (PRODUCTION 1.39 2012-12-04)
Report level:   verbose report

--------------------------------------------------------------------------------

INFO:Timing:3224 - The clock SYS_CLK associated with TIMEGRP "ADDR" OFFSET
   = IN 6 ns BEFORE COMP "SYS_CLK";
INFO:Timing:3225 - Timing constraint TIMEGRP "ADDR" OFFSET = IN 6 ns BEFORE
   COMP " SYS_CLK ";
INFO:Timing:3391 - Timing constraint TIMEGRP "ADDR" OFFSET = IN 6 ns BEFORE
   COMP "SYS_CLK";
INFO:Timing:3224 - The clock SYS_CLK associated with TIMEGRP "EB" OFFSET =
   IN 6 ns BEFORE COMP " SYS_CLK ";
INFO:Timing:3225 - Timing constraint TIMEGRP "EB" OFFSET = IN 6 ns BEFORE
   COMP " SYS_CLK ";
INFO:Timing:3391 - Timing constraint TIMEGRP "EB" OFFSET = IN 6 ns BEFORE
   COMP " SYS_CLK ";
INFO:Timing:3412 - To improve timing, see the Timing Closure User Guide (UG612).
INFO:Timing:2752 - To get complete path coverage, use the unconstrained paths
   option. All paths that are not constrained will be reported in the
   unconstrained paths section(s) of the report.
INFO:Timing:3339 - The clock-to-out numbers in this timing report are based on
   a 50 Ohm transmission line loading model.  For the details of this model,
   and for more information on accounting for different loading conditions,
   please see the device datasheet.
 INFO:Timing:3389 - This architecture does not support 'Discrete Jitter' and
   'Phase Error' calculations, these terms will be zero in the Clock
   Uncertainty calculation.  Please make appropriate modification to
   SYSTEM_JITTER to account for the unsupported Discrete Jitter and Phase
   Error.
================================================================================
Timing constraint: TIMEGRP "EB" OFFSET = IN 6 ns BEFORE COMP "SYS_CLK";
For more information, see Offset In Analysis in the Timing Closure User Guide (UG612).
 3412 paths analyzed, 2998 endpoints analyzed, 0 failing endpoints
 0 timing errors detected.
--------------------------------------------------------------------------------

  All constraints were met.

================================================================================
```

D.1.3 Place and Route Details

Figure D.5 below shows the resulting placement and routing of the DPE and the bus controller in the XC3S-1500-4FG676. The overall utilization is about 40% and is described in detail in Figure D.4 The three Queued-Stack implementations consume a large number of the available resources due to the latch based implementation. Future implementations should be done using the RAMB16 modules. This will require an extensive rewrite of the DPE Verilog code.

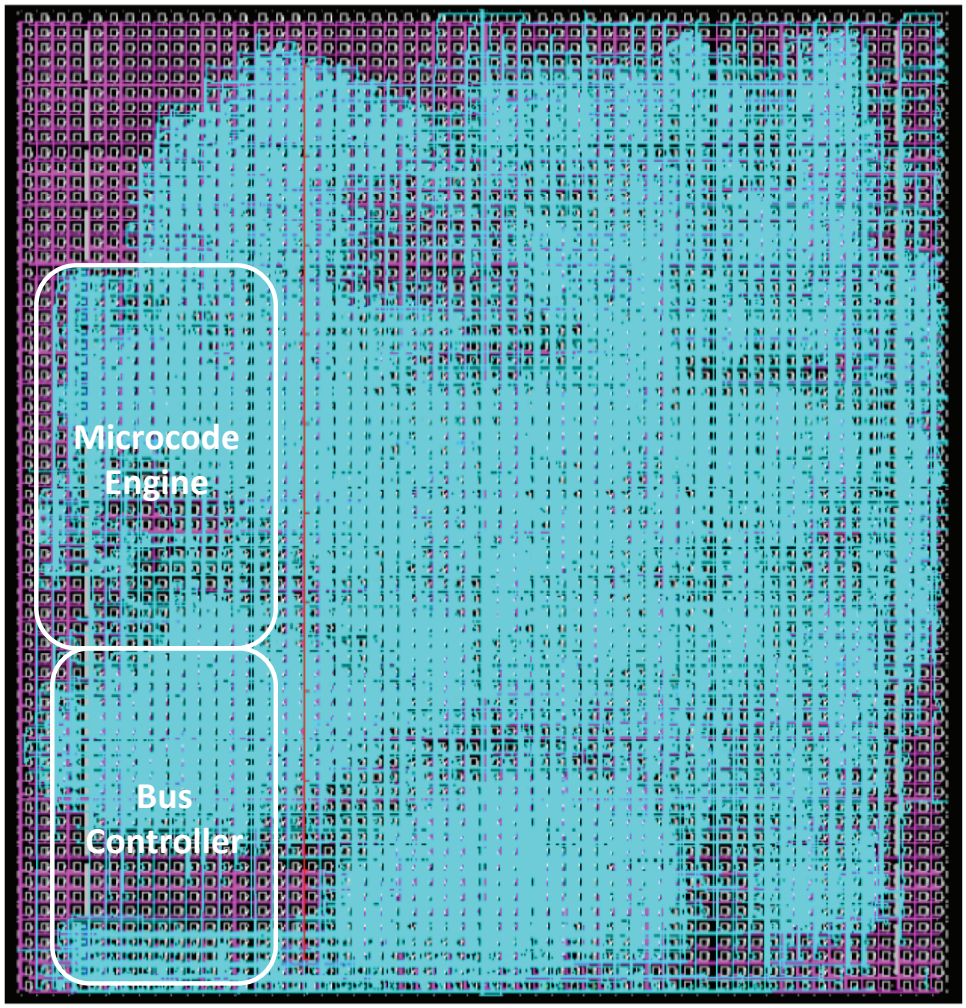


Figure D.5: Placement and routing of the DPE and bus controller

Glossary

CSP – Cognitive Sensor Platform
DPE – Dataflow-Processing Element
FSU – Functional Services Unit
PPU – Preprocessing Unit
IQS – Input Queued Stack
RQS – Result Queued Stack
DSP – Digital Signal Processor
GPP – General Purpose Processor
ASP – Application Specific Processor
ASIC – Application Specific Integrated Circuit
ISA – Instruction Set Architecture
FPGA – Field Programmable Gate Array
FSM – Finite State Machine
SDF – Synchronous Data Flow
TOS – Top of Stack
BOS – Bottom of Stack
ADC – Analog-to-Digital Converter
SDC – Sensor Data Conditioning Unit
FIFO – First-in-First-out (queue)
LIFO – Last-in-First-out (stack)
ROM – Read only memory
SRAM – Static Random Access Memory
WCS – Writeable Control Store
NOC – Network-on-Chip
FLIT – Flow Control Unit or Flow Digits
COTS – Commercial-off-the-Shelf

References

Chapter 1: Introduction to Sensor Systems

[1] A. Howard and E. Tunstel, Development of Cognitive Sensors, NASA TECH BRIEF Vol. 26, No. 4, August 2008

[2] J. Koomey, S. Berard, M. Sanchez, H. Wong, Implications of Historical Trends in the Electrical Efficiency of Computing, Annals of the History of Computing, IEEE, Volume: 33 Issue: 3, pp. 46 – 54, March 2011

[3] D. Harel and A. Pnueli, On the Development of Reactive Systems, in Logics and Models of Concurrent Systems (K. R. Apt, ed.), NATO ASI Series, Vol. F-13, Springer-Verlag, New York, pp. 477-498, 1985

[4] B. Lee, Specification and Design of Reactive Systems, Ph.D. Thesis, Memorandum UCB/ERL M00/29, Electronics Research Laboratory, University of California, Berkeley, May 2000

[5] E. A. Lee and D. G. Messerschmitt, Synchronous Data Flow, IEEE Proceedings, Vol.75, No.9, pp.1235-1245, September 1987

[6] B. Lee, and A. R. Hurson, Issues in dataflow computing, Adv. in Computing, Vol. 37, pp. 285-333, 1993

Chapter 2: Survey of Sensor Systems

[7] J. Polastre, R. Szewczyk, and D. Culler, Telos: Enabling ultra-low power wireless research, In Proc. IEEE/ACM Information Processing in Sensor Networks (IPSN) - Track on Platforms, Tools and Design Methods for Networked Embedded Systems (SPOTS), 2005

[8] S. Park, I. Locher, A. Savvides, M. B. Srivastava, A. Chen, R. Muntz, S. Yuen, Design of a Wearable Sensor Badge for Smart Kindergarten, Proceedings of the International Symposium on Wearable Computing, 2002

[9] A. Savvides and M. B. Srivastava, A Distributed Computation Platform for Wireless Embedded Sensing, Proceedings of International Conference on Computer Design (ICCD), Freiburg, Germany, September 2002

[10] J. Polastre, R. Szewczyk, C. Sharp, and D. Culler, The Mote Revolution: Low Power Wireless Sensor Network Devices, Hot Chips 2004, August 22-24, 2004

[11] J. Rabaey, M. Ammer, T. Karalar, S. Li, B. Otis, M. Sheets, and T. Tuan, Pico-Radios for Wireless Sensor Networks: The Next Challenge in Ultra-Low-Power Design, Proc. Of Int'l Solid-State Circuits Conference (ISSCC), February 3-7, 2002

[12] B. Calhoun, et al, Design Considerations for Ultra-low Energy Wireless Micro-sensor Nodes, IEEE Transactions on Computers, pp. 727-749, June 2005

[13] M. Hempstead, M. Lyons, D. Brooks and G. Wei, Survey of Hardware Systems for Wireless Sensor Networks, ASP Journal of Low Power Electronics, Vol. 4., No. 1, April 2008

[14] A. Abnous et al., Evaluation of a Low-Power Reconfigurable DSP Architecture, Proceedings of the Reconfigurable Architectures Workshop, Orlando, Florida, USA, March 1998

[15] C. Kelly, V. Ekanaya, and R. Manohar, SNAP: A Sensor-Network Asynchronous Processor, Proceedings of the 9th International Symposium on Asynchronous Circuits and Systems, Vancouver, BC, May 2003

[16] B. Zhai, L. Nazhandali, J. Olson, A, Reeves, M. Minuth, R. Helfand, S. Pant, D. Blaauw, T. Austin, ″A 2.60pJ/Inst. Sub-threshold Sensor Processor for Optimal Energy Efficiency,″ IEEE Symposium on VLSI Circuits (VLSI-Symposium), Honolulu, Hawaii USA, June 2006

[17] A. Eswaran, A. Rowe, R. Rajkumar, Nano-RK: an Energy-aware Resource-centric RTOS for Sensor Networks, 26th IEEE International Real-Time Systems Symposium RTSS05, 2005

[18] S. Han, R. Rengaswamy, R. Shea, E. Kohler, M. Srivastava, SOS: A Dynamic Operating System for Sensor Nodes, In Third International Conference on Mobile Systems, Applications and Services (Mobisys), June 2005

[19] A. Dunkels, B. Gronvall, and T. Voight, Contiki – a lightweight and flexible operating system for tiny networked sensors, In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, 2004

[20] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. S. J. Pister, System architecture directions for networked sensors, Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp. 93–104, 2000

[21] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer and D. E. Culler, The Emergence of Networking Abstractions and Techniques in TinyOS, Networked System Design and Implementation (NSDI), 2004

[22] M. Molla and S. Ahamed, A Survey of Middleware for Sensor Network and Challenges, Proceedings of the 2006 International Conference on Parallel Processing Workshops, 2006

[23] P. Levis, and D. Culler. Maté: a Tiny Virtual Machine For Sensor Networks, Architectural Support for Programming Languages and Operating Systems, 2002

[24] C. Curino, M. Giani, M. Giorgetta, A. Giusti, TinyLIME: Bridging mobile and sensor networks through middleware, In Proc. the 3rd IEEE Int. Conf. Pervasive Computing and Communications, Kauai Island, Hawaii, March 8-12, 2005

[25] P. J. Marrón, D. Minder, A. Lachenmann, and K. Rothermel. TinyCubus: An Adaptive Cross-Layer Framework for Sensor Networks, Information Technology, vol. 47(2), 2005

[26] C. Fok, G. Roman C. Lu. Agilla: A Mobile Agent Middleware for Sensor Networks, Tech. Rep. WUCSE-06-16, Washington University, Department of Computer Science and Engineering, St. Louis, 2006

[27] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, TinyDB: An acquisitional query processing system for sensor networks, ACM Transactions on Database Systems, 30(1), pp.122–173, 2005

Chapter 3: Cognitive Sensor System Requirements

[28] J. Carretero, P. Chaparro, X. Vera, J. Abella, and A. González, Implementing End-to-End Register Data-Flow Continuous Self-Test, Presented at IEEE Trans. Computers, 2011, pp.1194-1206.

[29] C. Lee, Fuzzy Logic in Control Systems, IEEE Trans. on Systems, Man, and Cybernetics, SMC, Vol. 20, No. 2, 1990, pp. 404-435

[30] CPU-12 Reference Manual, Rev. 4.0, Chapter 9, pp. 341-380, Motorola Inc., May 2003

[31] D. Hall and J. Llinas, An introduction to multi-sensor data fusion, Proceedings of the IEEE, vol. 85, No. 1, pp. 6-23, January 1997

[32] M. Akhoudi, E. Valav, Multi-Sensor Fuzzy Data Fusion Using Sensors with Different Characteristics, Unpublished, submitted to The CSI Journal on Computer Science and Engineering (JCSE)

[33] G. Ahmed, A Fuzzy Logic-Based Multi-sensor Data Fusion for Maritime Surveillance - Real Data Testing, Proceedings of the 26th National Radio Science Conference (NRSC2009), March 17-19, 2009

[34] Y. Vershinin, A Data Fusion Algorithm for Multi-sensor Systems, Proc. of ISIF, pp. 341-345, July 2002

[35] E. Punskaya, Bayesian approaches to multi-sensor data fusion, Master's thesis, Cambridge University Engineering Department, 1999

[36] P. Escamilla-Ambrosio and N. Mort, Hybrid Kalman Filter Fuzzy Logic Adaptive Multi-sensor Data Fusion Architecture, Proc. of The IEEE Conference on Decision and Control, 2003, pp. 5215-5220.

[37] R. Francis, A tutorial on logic synthesis for lookup-table based FPGAs, IEEE/ACM International Conference on Computer-Aided Design (ICCAD-92), pp. 40-47, 1992

[38] T. Sawkar, Area and delay mapping for table-look-up based field programmable gate arrays, 29th ACM/IEEE Design Automation Conference, pp.368-378, 1992

[39] J. Hauser, J. Wawrzynek, GARP: a MIPS processor with a reconfigurable coprocessor, Proceedings, The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, pp.12-21, April 16-18, 1997

[40] Q. Zhang, A.B.J. Kokkeler and G.J.M. Smit, Dynamically reconfigurable FFTs for a Cognitive Radio on a multiprocessor platform, International Conference on Engineering of Reconfigurable Systems and Algorithms (ERSA2008), July 2008

[41] ANT™ Protocol, available at: www.thisisant.com

Chapter 4: CSP Architecture

[42] J. Brignell, and N. White, Intelligent Sensor Systems, Institute of Physics Publishing (Sensors), pp. 51, 1996

[43] J. Brignell, and N. White, Intelligent Sensor Systems, Institute of Physics Publishing (Sensors), pp. 73, 1996

[44] A. Manickam, A. Chevalier, M. McDermott, A. D. Ellington, A. Hassibi, A CMOS electrochemical impedance spectroscopy biosensor array for label-free bio-molecular detection, International Solid State Circuits Conference, pp. 130-131, 2010

[45] J. Elson, D. Estrin, Fine-Grained Network Time Synchronization using Reference Broadcast, The Fifth Symposium on Operating Systems Design and Implementation (OSDI), p. 147-163, December 2002

[46] S. Ganeriwawal, R. Kumar, M. Srivastava, Timing-Sync Protocol for Sensor Networks, The First ACM Conference on Embedded Networked Sensor Systems (SenSys), pp. 138-149, November 2003

[47] M. Maroti, B. Kusy, G. Simon, A. Ledeczi, The Flooding Synchronization Protocol, Proc. of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys), November 2004

[48] IEEE Standard for a Smart Transducer Interface for Sensors and Actuators - Network Capable Application Processor (NCAP) Information Model, IEEE STD 1451.1-1999

Chapter 5: Dataflow-Processing Element

[49] P. Koopman, Stack Computers The New Wave, Ellis Horwood Ltd, ISBN 0-7458-0418-7, 1989

[50] M. Schoeberl, Design and implementation of an efficient stack machine, Proceedings of the 12th IEEE Reconfigurable Architecture Workshop (RAW2005), Denver, Colorado, USA, April 2005

[51] K. Nakamura, K. Sakai, T. Ae, Real-time multimedia data processing using VLIW hardware stack processor, Parallel and Distributed Real-Time Systems, 1997. Proceedings of the Joint Workshop on, pp. 296-301, 1-3 April 1997

[52] Y. Tsao, et al, Hardware nested looping of parameterized embedded DSP core, IEEE International SOC Conference, pp. 49-52, Sep 17, 2003

[53] L. Mengibar, L. Entrena, M. Lorenz, E. Millan, Partitioned state encoding for low power in FPGAs, Electronics Letters, vol.41, no.17, pp. 948- 949, Aug 18, 2005

[54] Paya-Vaya, G., Martin-Langerwerf, J., Giesemann, F., Blume, H., Pirsch, P.; Instruction merging to increase parallelism in VLIW architectures, System-on-Chip, 2009. SOC 2009 International Symposium, pp.143-146, 5-7 Oct. 2009

[55] M. Seok, S. Hanson, S. Jae-sun, D. Sylvester, D. Blaauw, Robust ultra-low voltage ROM design, Custom Integrated Circuits Conference (CICC 2008), pp.423-426, September 21-24, 2008

[56] P. Liu, F. Mowle, Techniques of Program Execution with a Writable Control Memory, IEEE Transactions on Computers, vol. 27, no. 9, pp. 816-827, Sept. 1978

[57] Reverse Polish Notation: http://www.hpmuseum.org/rpn.htm

Chapter 6: DPE Microprogramming

[58] Forth Language: http://www.forth.com/

[59] D. DeWitt, Extensibility - a New Approach for Designing Machine Independent Microprogramming Languages, MICRO-9 Proceedings, pp. 33-41, September 1976

Chapter 7: Modeling and Simulation

[60] J. Buck, S. Ha, E. Lee, and D. Messerschmitt, Ptolemy: A Framework for Simulating and Prototyping Heterogeneous Systems, International Journal of Computer Simulation, special issue on Simulation Software Development, vol. 4, pp. 155-182, April 1994

[61] E. de Kock, et al, YAPI: Application Modeling for Signal Processing Systems, Proceedings 37th Design Automation Conference, Los Angeles, 2000

[62] S. Stuijk, M. Geilen, T. Basten, SDF3: SDF For Free, Proceedings of the Sixth International Conference on Application of Concurrency to System Design (ACSD'06), pp. 276-278, June 2006

[63] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, A. Sangiovanni-Vincentelli. Addressing the System-on-a-Chip Interconnect Woes Through Communication-Based Design, DAC 2001, June 18-22, 2001

[64] Matlab: SimEvents®: http://www.mathworks.com/products/simevents/

Chapter 8: Results

[65] FIR C-code example: http://iowahills.com/A7ExampleCodePage.html

[66] IIR C-code example: http://iowahills.com/A7ExampleCodePage.html

[67] Cortex-M3 Technical Reference Manual, ARM Corporation, 2010

[68] Cortex-M3 DSP library filter functions Application Note STM32F10X, ST Microelectronics, 2010

[69] I. Sutherland, R. Sproull, D. Harris, Logical Effort: Designing Fast CMOS Circuits, Morgan Kaufmann, 1999

[70] Y. Chang, B. Park, I. Park, C. Kyung, Customization of a CISC processor core for low-power applications, International Conference on Computer Design (ICCD'99, pp.152-157, 1999

[71] Cortex-M3-implementation-specifications: http://www.arm.com/products/processors/cortex-m/cortex-m3.php

[72] Synopsys Prime Time Power Analysis (PTPX): http://www.synopsys.com/Tools/Implementation/SignOff/CapsuleModule/ptpx_wp.pdf

[73] M. Horowitz, T. Indermaur, R. Gonzalez, Low-power digital design, Symposium. Low Power Electronics, pp. 8-11, Oct. 1994

[74] SPEC Benchmark: http://www.spec.org/benchmarks.html

[75] TPC Benchmark: http://www.tpc.org/information/benchmarks.asp

[76] EEMBC Benchmark: http://www.eembc.org/products/

[77] P. Hofstee, K. Nowka, VLSI-2 Class Notes, ECE Department, University of Texas at Austin, 2005-2013

Chapter 9: Final Observations and future work

[78] R. McIvor, Managing for Profit in the Semiconductor Industry, Prentiss Hall, 1989

Appendix D: FPGA Implementation

[79] TLL5000 Electronic System Design Base Module V1.1 User Manual. The Learning Labs, Inc., 2008.