International Journal of Reconfigurable and Embedded Systems (IJRES)
Vol. 1, No. 3, November 2012, pp. 75~86
ISSN: 2089-4864

Queued-Stack Dataflow Processing Element for a Cognitive Sensor Platform

Mark McDermott
Department of Electrical and Computer Engineering, University of Texas at Austin

Article Info

Article history:

Received Aug 5, 2012
Revised Oct 1, 2012
Accepted Oct 13, 2012

Keyword:

Queued-Stack
Dataflow Processor
Composable System
Synchronous Dataflow
Cognitive Sensor Platform

ABSTRACT

This paper describes a Queued-Stack (QS) Dataflow Processing Element (DPE) that is used in a cognitive sensor platform. The queued-stack is used for buffering input data to the DPE and for storage of variables and results. The queuing mechanism and dataflow protocol provides the capability to compose multi-node computational systems where communication between elements is via non-blocking FIFO channels. System composition is achieved using synchronous dataflow tools such as SDF3 or Ptolemy. The dataflow-processing element is implemented using single cycle micro-coded engine where the ratio of datapath transistors to control logic is optimized for programmable energy-performance sensitive applications.

Corresponding Author:

Mark McDermott
Department of Electrical and Computer Engineering,
University of Texas at Austin
1 University Station C0803, Austin, Texas 78712-0214
Email: mcdermot@ece.utexas.edu

1. INTRODUCTION

The next step in the evolution of intelligent sensors is towards cognitive sensors. Cognitive sensor platforms have the capability to reason about both their external environment and internal conditions and to modify their processing behavior and configuration in an ongoing effort to optimize their operation relative to the device's relevant optimization criteria [1]. This requires improved computational capability to perform the additional tasks, within the same energy budget as an intelligent sensor system. General-purpose computing platforms do not have the energy-performance characteristics needed for low energy sensor systems. Hard-coded logic would provide the most optimal energy-performance but at the expense of re-programmability. The basic architecture of the sensor platform is shown below in Figure 1. The processing element used in this platform is implemented using an energy efficient stack-based microcoded engine.

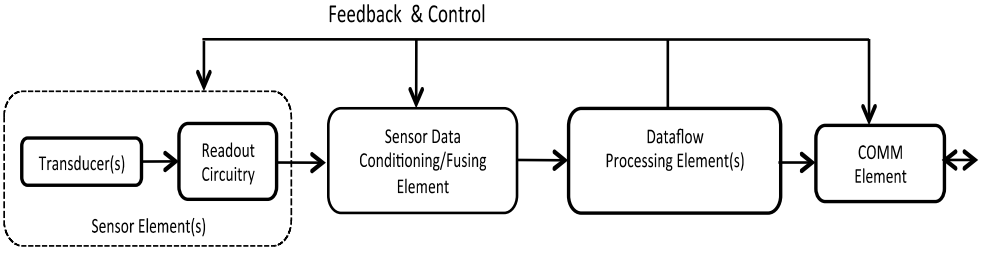

Figure 1. Sensor Platform Architecture

A key charac teristic of a sensor system is that it is primarily a reactive system where a change in the value of a sensor input will automatically force recalculation of the values of other variables in the system. The processing of sensor data in a reactive system is optimally done using a dataflow-processing element (DPE) [2]. The DPE is "fired" once all the data tokens have been received. Upon completion of the computation the dataflow-processing element is idled, waiting for the next tranche of tokens from the channel node.

The passing of tokens is best accomplished using a queuing (FIFO) mechanism, which is insensitive to the variable latency of the input sensor data; thus eliminating the need for the DPE to fetch and store the sensor data on the stack (LIFO). This architecture merges a queuing mechanism into a stack-based processing element. The result is queued-stack that is used to store input data from a sensor channel as well as results from the computation operations in the datapath.

The merged queued-stack element provides an ideal mechanism for building a synchronous dataflow system. Processing of data is triggered when the correct numbers of tokens are inserted into the queue. The ability to simultaneously read or write to the queued-stack prevents unnecessary stalling of the dataflow-processing element. The zero-overhead nested-looping, repeat function and the conditional execution micro-operations result in excellent energy-performance efficiency.

This paper presents a unique implementation of a synchronous dataflow-processing element for use in reactive systems. The queued-stack architecture and microcoded control provide optimal energy-performance for energy limited intelligent/cognitive sensor platforms. The token-based channel communication protocol provides the capability to compose complex systems of heterogeneous data-flow processing elements.

2. SYSTEM COMPOSABILITY

As mentioned above the queued-stack architecture provides a mechanism to easily compose systems by connecting processing nodes via channel nodes. The channel nodes implement a synchronous dataflow protocol where data tokens are transferred from one node and consumed by another node using a non-blocking FIFO [9]. Synchronous dataflow tools such as SDF3 [10] and Ptolemy [11] can be used to compose and simulate the sensor system.

Figure 2 below shows a "composed system" consisting of a cluster of DPEs, sensor elements and associated channel nodes, an output communications element and a debug element. This configuration is used for basic dataflow processing of data from multiple sensors. The DPEs can be synthesized to have heterogeneous computational capabilities based on the algorithmic requirements of the system. Each DPE can be configured to have one or two input-QS elements depending on how many channels need to be processed and what the storage requirements are for the algorithms being executed by the DPE.

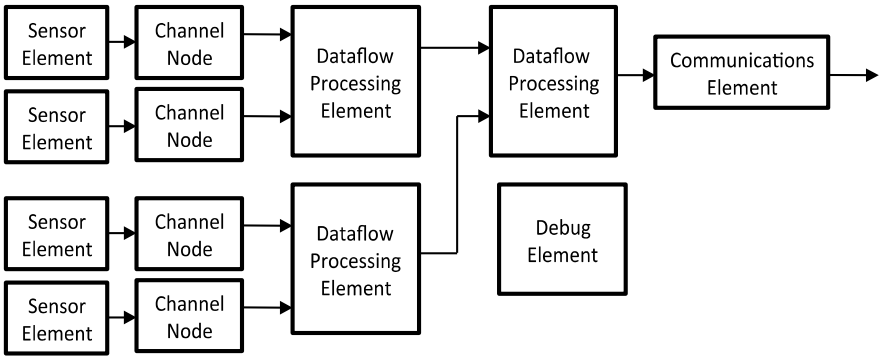

Figure 2. Example of a "Composed System"

The channel nodes are used to connect a DPE (or clusters of DPEs) to the sensor element(s). The channel nodes buffer the transactions from the sensors and forward the data tokens to the DPE. Each DPE has an integrated output channel node that is used to connect the output of a DPE to the input of another DPE.

3. DATAFLOW PROCESSING ELEMENT

The processing element used for this platform is implemented as a stack-based microcoded engine with advance features such as nested looping, conditional execution, repeat execution and a programmable lookup table for reconfigurable functional operations. Figure 3 below shows a block diagram of the DPE implemented using three QS elements and one output FIFO channel node. The input QS elements are used to receive channel data from two sources or they can be configured such that one QS element is receiving data while the other is processing data from a previous transaction. The Result-QS is used to store the higher precision results of the datapath operations.

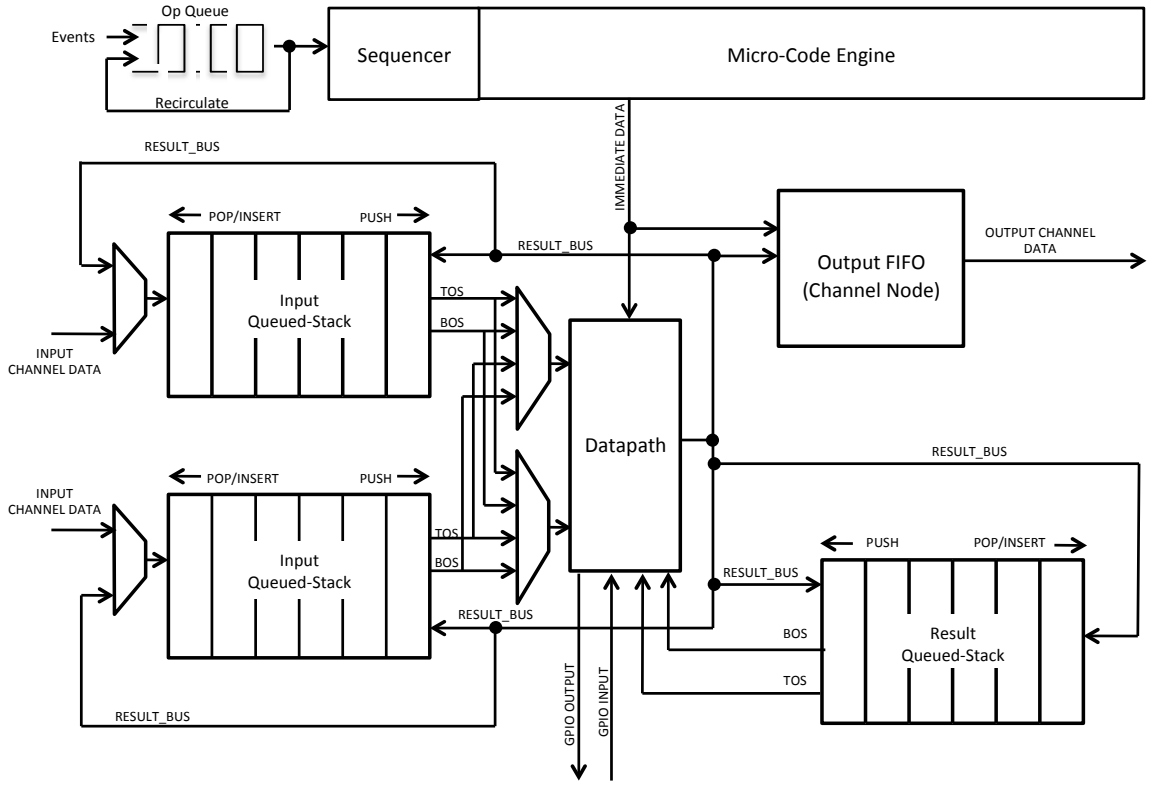

Figure 3. Dataflow Processing Element

The DPE is implemented using a parameterized synthesized model where the width and depth of the stacks, functional units, and data-paths 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 task or group of tasks during the synthesis process by selecting the optimal parameters.

The microcode storage is implemented using a standard single port ROM or WCS (writeable control store) memory compiler. The WCS configuration is useful for systems where the microcode needs to be updated from an external source such as flash memory [3].

4. QUEUED-STACK ARCHITECTURE

The Queued-Stack (QS) is implemented as a circular buffer that has two circular pointers, one to track the LIFO (stack) data and the other one to track the FIFO (queue) data. A three-deep QS is shown below in Figure 4. The FIFO pointer tracks data "inserted" into the queue from either the channel data or the result data from the datapath operations. The LIFO pointer is used to track pushes and pops from the stack. The QS 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 microcode engine handles the QS data management. The engine can select if the QS is available to receive inserted data while the computational engine is active ("push mode") or if the QS will fetch the data from the channel once the computation task is completed ("pull mode"). Pull-mode allows the QS to be used as a circular buffer to store intermediate data used in many filtering algorithms. Note: in pull-mode, the output data must be stored in an output FIFO channel node to prevent stalling the processing element.

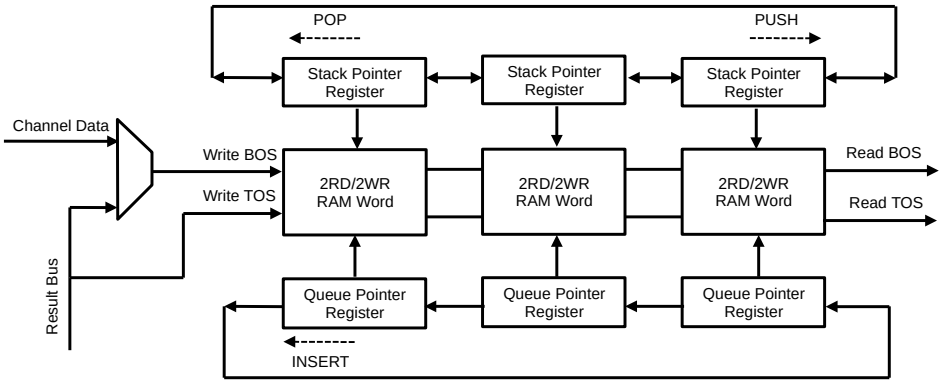

Figure 4. Three-deep Input Queued-Stack

As shown above in Figure 4, both the top-of-stack (TOS) data and the bottom-of-stack (BOS) data can be accessed simultaneously. In addition, the pointers can be manipulated by the microcode to select data anywhere in the circular buffer. For example, the LIFO pointer can be rotated five positions to the right and the FIFO pointer can be rotated 5 positions to the left in a single instruction by executing a non-write POP command, a non-write INSERT command with a REPEAT of 5 while executing a data path instruction and storing the result in the RESULT-QS, the Input-QS, and the Output-FIFO.

Table 1 below shows the operations that the QS support. The basic operations include PUSH, POP/POP_WR and INSERT that write data to either the TOS or BOS and selectively rotate the appropriate pointers. The same operations can also be merged to provide the ability to store results to the TOS and BOS in a single write cycle.

Table 1. Queued-Stack Operations

Operation	Description
PUSH	Rotate TOS pointer "right" and write result-bus value to new TOS
POP	Rotate TOS pointer "left" (no 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" without writing
PUSH_NW	Rotate TOS pointer "right" (no 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_BOT	Rotate TOS pointer "right" and write result-bus to BOS and to new TOS
TOP_INS	Rotate BOS pointer "left" and write result-bus to TOS and to new BOS
NOP	No Operation

5. DPE DATAPATH

The DPE datapath is composed of five major elements: shifter, multiplier, adder, logical unit (LU) and a special function unit (SFU). The latter two elements can be eliminated during system synthesis if they are not needed. Figure 5 below shows the datapath configuration including the multiplexing units that control the data flow through the datapath.

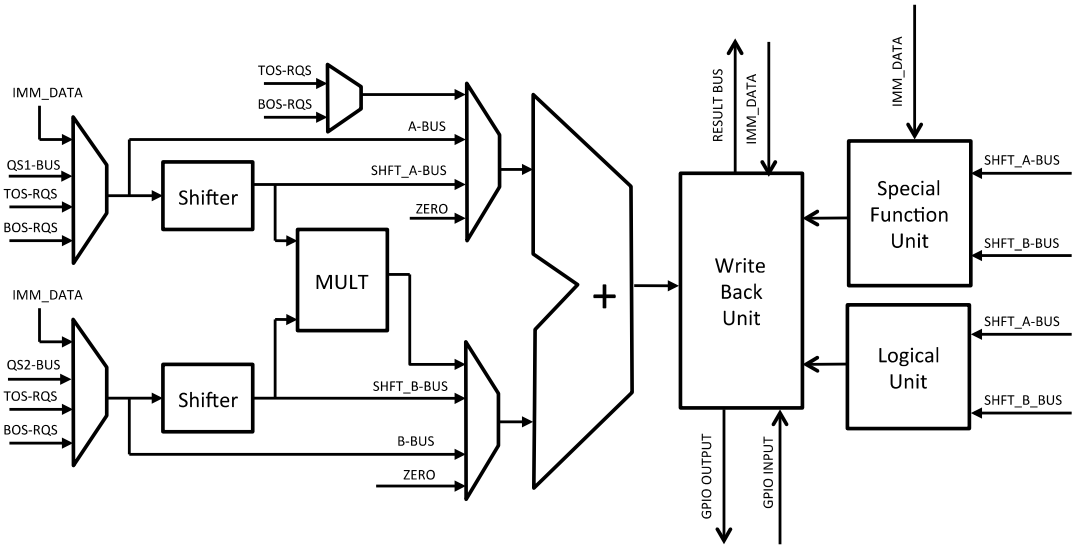

Figure 5. DPE Datapath

The datapath multiplexers have latched outputs, which can store intermediate data and are used to prevent spurious transactions propagating through the shifter-multiplier-adder paths thus reducing power. The Write-Back-MUX multiplexes data from the Adder, SFU, logical unit and the GPIOs to the result bus. The result bus is connected to the three QS elements and the output FIFO via the multiplexing scheme shown above in Figure 3. The Result-QS is used to store the results of the datapath transactions and is synthesized to be the width of the Adder. An Input-QS can also be used to store results, however it is limited to storing data that is the width of the incoming channel data. Table 2 below shows the operand sources for the computational instructions.

Table 2. Datapath Operand Sources

Shifter	Multiplier/LU/SFU	Adder
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 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 calculations, data compression, data fusion, time stamping, edge detection, threshold detection, period measurements, etc.

The microcode engine individually controls each element in the datapath resulting in a large combination of parallel operations including: Shift-Multiply-Add, Shift-Multiply-Saturating Add, Multiply-Add, Multiply-Saturating Add, Shift-Add, Shift-Saturating Add, Arithmetic & Logical Shift, Bit Clear, Bit Set, Boolean Logic functions, table lookup, interpolation, linearization, absolute value, etc.

6. 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. This eliminates the need for an instruction decoder and sequencer to control the various units in the DPE. This results in an optimal ratio of control logic to datapath logic.

There are four control fields in the microcode word as shown below in Figure 6.

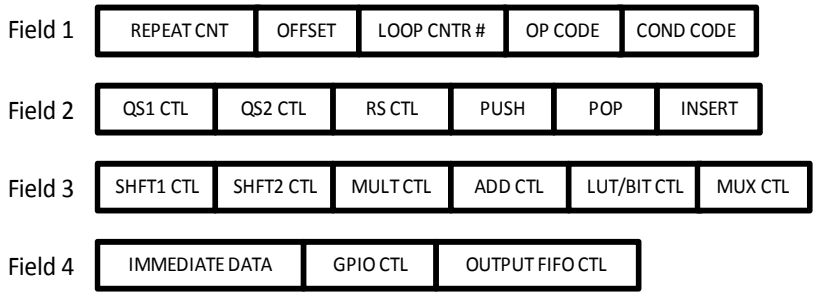

Figure 6. Microcode Control Fields

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 [4] 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 second field is the queued-stack control. A typical operation is shown below in Figure 7. During the first half of the cycle the QS provides the operands to datapath and the second half of the cycle the address of the QS 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.

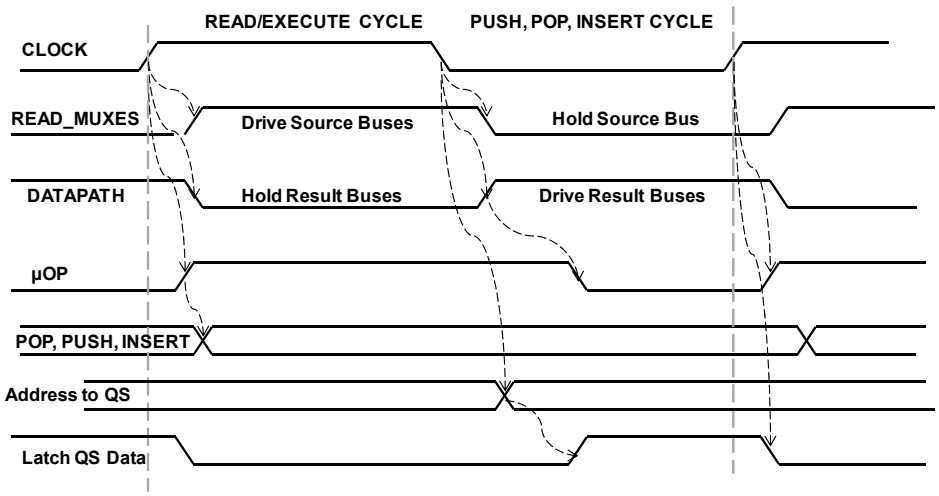

Figure 7. Typical QS read-write Cycles

The third field controls the datapath using “one-hot” control bits. The bits control the multiplexers and the functional units. As can be seen from Figures 3, 4 and 5, there are 10 multiplexers that control the flow of data through the functional units, the write-back mux and the QS elements.

The fourth field is used for immediate data and to address data in the lookup table. The width of this field is determined by the width of the microcode word. In some implementations of the DPE, sub-fields in the micro-engine control field can be used to extend the width of word.

The block diagram of the micro-engine is shown below in Figure 8 and consists of four components: WCS, micro-address generation, operation queue and a finite state machine controller.

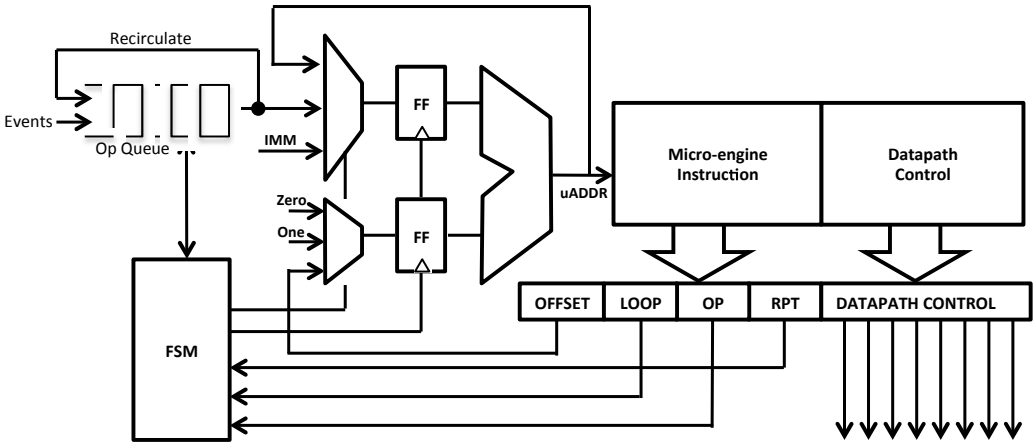

Figure 8. Block Diagram of The Microcode Engine

The micro-engine is controlled via a 4-bit micro-opcode field. The micro-engine opcodes supported are shown below in Table 3. There are two basic execution modes: normal and conditional. In normal mode all micro-operations execute independently of the value of the condition codes. In conditional execution the value of the condition codes from the previous cycle are compared with value in micro-code word. If the two codes are not identical the micro-operation is not executed.

Table 3. Micro-Engine Opcodes

OPCODE	REPEAT	OPERATION
EXEC	Y	Normal Execution
COND_EXEC	N	Conditional Execution
WFE	N	Wait for Event
JMP	N	Jump to address specified in IMMED_DATA field
JUMP_HALT	N	Jump to address specified in IMMED_DATA field THEN Halt
LOOP_BACK	Y	Loop back for Loop # {1,2,3}
BRA	N	Branch Unconditionally
BR_WB_GT	N	Branch Writeback GREATER THAN REFERENCE DATA
BR_WB_LT	N	Branch Writeback LESS THAN REFERENCE DATA
BR_WB_EQ	N	Branch Writebakc EQUAL toREFERENCE DATA
BR_SFU_GT	N	Branch SFU result GREATER THAN REFERENCE DATA
BR_SFU_LT	N	Branch SFU result LESS THAN REFERENCE DATA
BR_SFU_EQ	N	Branch SFU result EQUAL to REFERENCE DATA
TXFR	N	Transfer Output FIFO data to Channel

The finite state machine (FSM) controls the nested looping and repeats 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 9 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 JUMP_HALT microinstruction is executed.

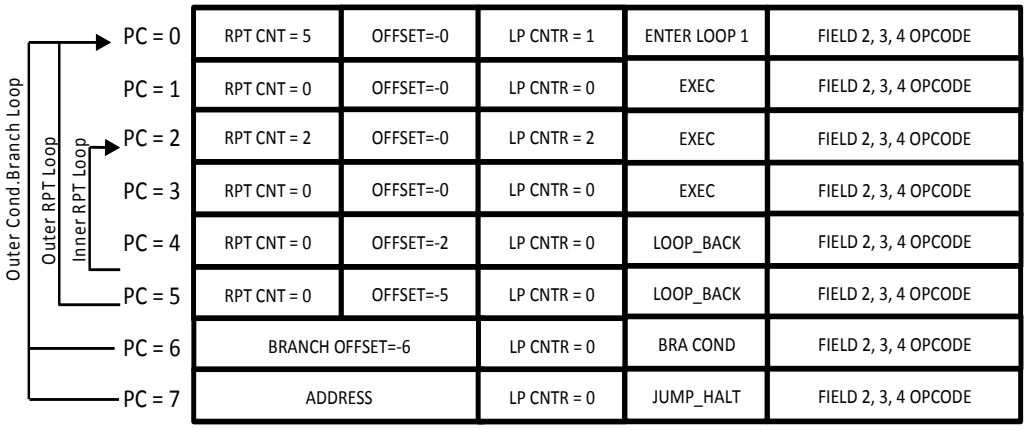

Figure 9. Typical "Nested Looping" Microcode Sequence

The JUMP_HALT is a merged microinstruction that jumps to address specified and halts the micro-engine to wait for the next "fire" 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.

The microcode engine contains an operation queue that is used to register events and trigger microcode operations in response to the events. The operation queue provides a mechanism to interrupt the normal flow of the microcode at specific entry points. The queue is circular and can also be used to store a sequence of macroinstructions for systems that use ROM storage for the microcode. The FSM controls the flow of events/macro-instructions to the micro-engine.

As mentioned above, the microcode storage can be either read-only-memory (ROM) or writable-control-store (WCS) based. 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. The WCS is loaded via the JTAG interface and is used in systems where overlaying of microcode is needed due to the size of the code or for debugging microcode before the microcode is committed to ROM.

7. DPE OPERATION EXAMPLES

The DPE micro-architecture is designed to primarily support sensor data conversion and conditioning. Typical operations include: digital filtering, decimation, linearization, averaging, data compression, feature extraction, data fusing, edge detection, threshold detection, etc. Figure 11 below shows an FIR filter configuration that can be implemented with five microinstructions and executes in ten clock cycles.

IQS-1 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 over-written when the BOS pointer recirculates. IQS-2 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 IQS-1 and IQS-2 are "popped" to point at the next variable and token for each multiplication step.

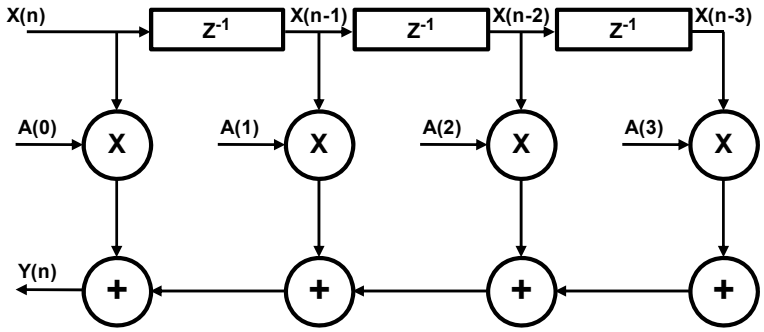

Figure 11. FIR Filter Configuration

The data storage for the FIR filter calculations is shown below in Figure 12.

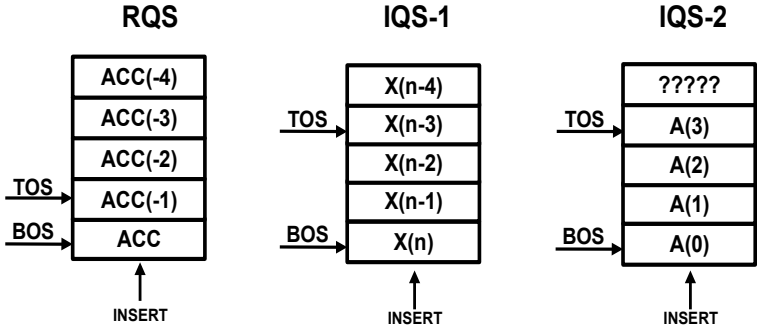

Figure 12. Data Storage For FIR Filtercalculations

The microinstructions to execute one filter cycle are:

```
1:    ADD, ZERO, ZERO,                  ; ZERO -> ACC
      POP_INS_RQS                  ; INSERT AT BOS RQS and POP TOS

2:    MULT, TOS_QS1, TOS_QS2,           ; A(3,2,1) * X(N-3,2,1)
      ADD, BOS_RQS,                ; + ACC
      WB[BOS_RQS],                 ; WRITEBACK-> ACC
      POP_QS1, POP_QS2,                 ; POINT AT NEW VARIABLE
      RPT=3                        ; REPEAT 3 TIMES

3:    MULT, TOS_QS1, TOS_QS2            ; A(0) * X(N)
      ADD, BOS_RQS,                ; + ACC
      WB_FIFO,                          ; WRITEBACK TO FIFO ELEMENT
      POP_QS1                           ; CONSUME X(N)

4:    PUSH_NW_QS1, PUSH_NW_QS2,         ; RESET VARIABLE POINTERS
      RPT=4                        ; REPEAT 4 TIMES

5:    JUMP_HALT 1;                 ; JUMP and WAIT FOR 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 JUMP_HALT microinstruction branches back to the first instruction that clears the accumulator and waits for the next data token. Once the X(n) variable is inserted into the BOS of IQS-1. The channel node issues a FIRE signal to the DPE and the sequence repeats itself.

Figure 13 below shows a Bi-Quad IIR filter configuration that can be implemented in 9 microinstructions and 13 clocks.

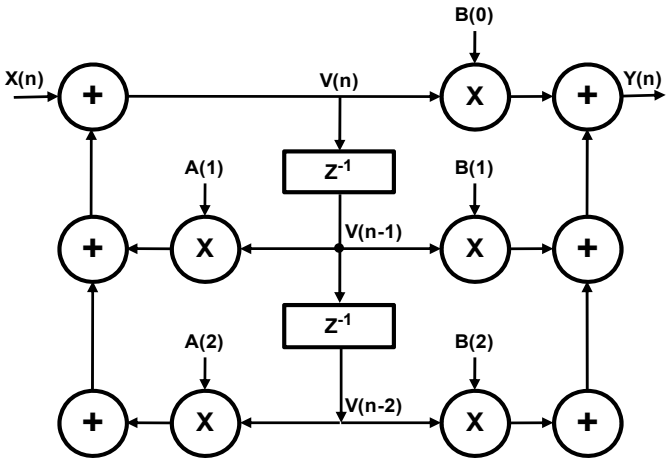

Figure 13. IIR filter Configuration (Bi-Quad)

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. It will become 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. The data storage for the IIR filter calculations is shown below in Figure 14.

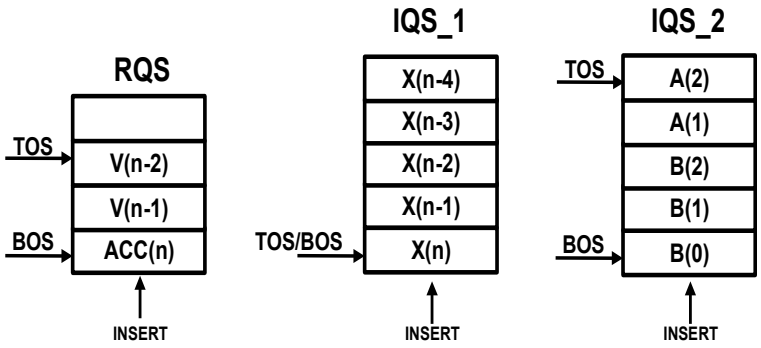

Figure 14. Data Storage For IIR Filter Calculations

The microinstructions to execute one IIR filter cycle are:

```
1:      MULT, TOS_RQS, TOS_QS2,              ; V(N-2) * A(2)
        WB[BOS_RQS],                 ; WRITEBACK -> ACC_1
        INSERT_RQS,                  ; INSERT BEFORE WRITE
        POP_QS2,                           ; POINT @ A(1)
        POP_RQS                      ; POINT @ V(N-1)

2:      MULT, TOS_RQS, TOS_QS2,              ; V(N-1) * A(1)
```

```
        WB_BOS_RQS,                 ; WRITEBACK -> ACC_1
        POP_QS2,                          ; POINT @ B(2)
        PUSH_NW_RQS                 ; POINT @ V(N-2)

3:      ADD, BOS_RQS, BOS_QS1,      ; X(N) + ACC
        WB[BOS_RQS]                 ; WRITEBACK -> ACC_1

4:      MULT, TOS_RQS, TOS_QS,      ; V(N-2) * B(2)
        WB[TOS_RQS],                ; WRITEBACK -> ACC_2 (V(N-2))
        POP_QS2                     ; TOS => @ B(1)

5:      POP_RQS                     ; POINT AT V(N-1)

6:      MULT, TOS_RQS, TOS_QS2,           ; V(N-1) * B(1)
        WB[TOS_RQS],                ; WRITEBACK -> ACC_2
        POP_QS2                     ; POINT @ B(0)

7:      MULT, BOS_RQS, TOS_QS2,           ; ACC_1 * B(0)
        ADD, TOS_RQS                ; + ACC_2
        WB[FIFO]                          ; WRITEBACK TO FIFO ELEMENT
        POP_RQS                     ; POINT AT NEW V(N-1)

8:      PUSH_NW_QS2,                ; RESET VARIABLE POINTER
        RPT=5                       ; REPEAT 5 TIMES

9:      JMP_HALT 1;                 ; JUMP and WAIT FOR TOKEN
```

8. ENERGY-PERFORMANCE ANALYSIS AND RESULTS

The energy-performance analysis of the DPE was done using an 180nm mixed-signal process. The DPE logic was synthesized and the layout generated using DCT/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 numbers were then derived using Prime Time and Prime Time-PX respectively.

Tables 4 and 5 below show the energy-performance for the DPE, a reconfigurable DSP (Pleiades) developed at UC-Berkley [5] and the Cortex-M3 from ARM [6]. The Cortex-M3 is widely used in embedded designs as both a general-purpose processor and as 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.

Table 4. FIR (4-TAP) Energy-Performance Benchmarks

	Cortex-M3	**Pleiades**	**DPE**
Frequency (MHz)	20	14	10
Throughput (cycles/FIR)	107	4	10
Switched-Capacitance/FIR	4.92nF	126pF	17.4pF
Energy/FIR	15,947pJ	285.2pJ	56.4pJ
Energy-Delay/FIR (J-s x 10^{-18})	85,317	82	56

Table 5. IIR Energy-Performance Benchmarks

	Cortex-M3	**Pleiades**	**DPE**
Frequency (MHz)	20	14	10
Throughput (cycles/IIR)	129	8	13
Switched-Capacitance/IIR	5.93nF	465pF	25.6pF
Energy/IIR	19,226pJ	1,046pJ	82.9pJ
Energy-Delay/IIR (J-s x 10^{-18})	43,042	299	107

The throughput and energy values for the Pleiades DSP were derived from the 600nm implementation specified in [5] 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 [7] [8]. A DSP library of filter functions designed specifically for the Cortex-M3 [8] was used to determine the throughput. The DPE energy and capacitance numbers are derived from an 180nm extracted netlist. All DPE benchmarks use 16-bit integer data tokens and 48-bit integer results.

9. CONCLUSION

The queued-stack dataflow-processing element presented in this paper is ideally suited for low-energy data-driven deeply embedded applications such as remote sensing, medical implants and structural implants. The addition of cognitive processing capabilities to the sensor platform is necessary for these types of unattended applications where it is not feasible to routinely replace the batteries or sensors in these applications. The key metric for this class of embedded processing elements is energy-performance-volume where battery volume is the limiting factor as it determines the number of joules available for system operation. The micro-architecture of the dataflow-processing element is optimized such that the ratio of datapath transistors to control logic is maximized. The result as shown above is excellent energy-performance characteristics.

REFERENCES

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

[2] Bruno Preiss and V.C. Hamacher, “Data flow on a queue machine”, *ISCA '85 Proceedings of the 12th annual International Symposium on Computer Architecture*, ISBN:0-8186-0634-7

[3] Phillip Koopman, “Writable instruction set, stack oriented computers”, *Proceedings of the 1987 Rochester Forth Conference.*

[4] Ya-Lan Tsao, et al, “Hardware nested looping of parameterized embedded DSP core”, *IEEE International SOC Conference*, pp. 49-52, Sep 17, 2003

[5] Arthur Abnous, et al, “Evaluation of a low-power reconfigurable DSP architecture”, *Proceedings of the Reconfigurable Architectures Workshop*, Orlando, Florida, USA, March 1998.

[6] Cortex-M3 Technical Reference Manual, ARM Corporation, 2010

[7] Cortex-M3 implementation specifications, http://www.arm.com/products/processors/cortex-m/cortex-m3.php

[8] “Cortex-M3 DSP library filter functions” Application Note STM32F10x, ST Microelectronics, 2010

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

[10] S. Stuijk, M. Geilen, and T. Basten, "SDF3: SDF For Free," in *6th International Conference on Application of Concurrency to System Design, ACSD 2006, Proceedings. IEEE*, pp. 276-278, June 2006.

[11] E. A. Lee, H. Zheng, "Leveraging Synchronous Language Principles for Heterogeneous Modeling and Design of Embedded Systems," *EMSOFT '07*, September 30 - October 3, 2007, Salzburg, Austria.

BIOGRAPHY OF AUTHOR

Mark McDermott is adjunct faculty member in the ECE Department at the University of Texas where he teaches graduate level courses in VLSI design, Embedded System Design and System-on-Chip design. His research interests are in low-energy embedded system design.