AN ELEMENTAL COMPUTING ARCHITECTURE FOR SD RADIO

Steven Kelem, Brian Box, Stephen Wasson, Robert Plunkett, Joseph Hassoun, Chris Phillips
(Element CXI, Milpitas, CA, US)
{steve.kelem, brian.box, stephen.wasson, bob.plunkett, joseph.hassoun, chris.phillips}
@elementcxi.com

ABSTRACT

This paper described a new reconfigurable architecture that lends itself to parallelizable tasks, such as Software-Defined Radio, while providing a new level of reliability for systems built with this architecture. A new computing paradigm called Elemental Computing efficiently combines four computational styles: sequential, data-flow, message-passing, and DMA in a rapidly-reconfigurable distributed system on a chip. We call this an Elemental Computing Array (ECA). Elemental code can be placed and routed in real time to work around defects on a device. This extends the useful lifetime of applications using this device, allowing graceful degradation of the system instead of catastrophic failure.

1. INTRODUCTION

The omnipresent goals in building electronic systems are to meet the design goals of Power, Performance, and Price. By concentrating on Reconfiguration and Resilience we produced an architecture that also satisfies these power, performance, and price goals.

Reconfiguration is defined as *The ability of a system to easily and quickly change its functionality.*

Resilience is defined as *The ability of a system, in the presence of long-term aging, to continue to operate with minimal disruption, provided sufficient computing resources exist, and to degrade gracefully while warning the user as resources become insufficient.*

One of Element CXI's requirements is an architecture that is tolerant of silicon defects. Defects in electronic hardware typically follow a "bathtub curve". This curve plots defect rate vs. time. Defects are most prominent at the infancy of a device and at the end of the life of a chip. Defects during the "infancy" period are attributed to process, handling, or installation defects[1]. Most of these defects are caught early in a chip's lifetime with the remaining devices working for many years. However, a fact that is not well known, is that silicon devices do not last forever. Many factors contribute to a finite lifetime for silicon chip, such as dielectric breakdown of the gate oxide, hot carrier damage, and electro-migration[2]. Building devices that are tolerant of failures in the field meant re thinking how a reconfigurable architecture is tested, programmed, and monitored throughout a device's lifetime.

1. A TASK-BASED ARCHITECTURE

The Element CXI architecture is a multi-tasking architecture that allows tasks to be built from sequential and parallel building blocks. The Elemental building blocks are non-homogeneous, highly pipelined and able to process many waves of data simultaneously. Tasks are built from these blocks. Tasks can be loaded onto the chip while other tasks are running, even in the same region of the chip as a running task. A task may have exclusive use of an Element or time-share an Element with other tasks, depending on throughput requirements. A portion of a task requiring high throughput might require exclusive use of an Element. Portions with lower throughput requirements can timeshare Elements. Tasks can be distributed across computing Elements for maximum speed and parallelism or "folded" onto a smaller number of Elements, thus time-sharing the Element with other portions of the same, or other tasks.

1.1 Hardware O/S

A multi-tasking chip needs an operating system to manage the tasks. The ECA includes hardware support for system-level functions:

- Task Loading, including rapid location of available (and working) computing and routing resources
- Task Binding, rapid configuration of selected computing and routing resources
- Interface Binding: binding the inputs and outputs of one task to other tasks
- Task Run-Time Control: start, halt, suspend, single-step and free a task.

These steps are done by distributed hardware resources that perform multiple local searches simultaneously. This enables the Hardware O/S (HWOS) to be distributed across the ECA, improving performance and resiliency by removing the HWOS as a single-point-of-failure.

Providing hardware support for task binding means that place and route can be done as a task is loaded into the device. Components of a task are bound to computational and interconnect resources in real-time. Run-time Binding takes into account which resources are broken and which ones are already in use, so that only free, working resources are allocated to new tasks. This is in contradiction to devices where the portions of a task to be loaded into a device must

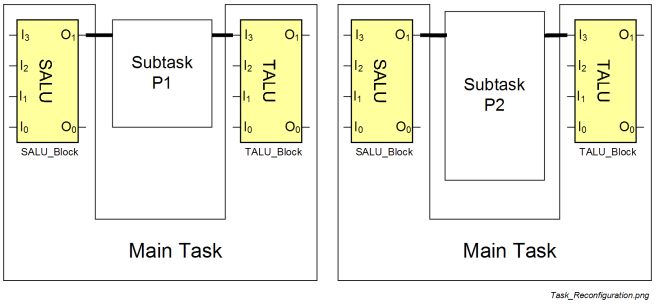

Illustration 1.1.: Subtask Reconfiguration

be carefully "placed" and "routed" in anticipation of fitting together snugly on the device. In the ECA, the Elemental code is easily relocatable.

1.2 Multi-Tasking

The Elemental device is a fine-grained, multi-tasking device. The device has a task space of up to 1,024 tasks per "cluster". A task may be as small as a single Element instruction or may occupy an entire Cluster of 16 Elements. Tasks may be made up of parallel, data-flow, instructions and sequential code. Data-flow instructions are mapped to data-flow elements, which are high-speed, self-timed, pipelined execution units that have streaming inputs and streaming outputs. Sequential code is mapped to sequential elements, which are specialized RISC processors. Data is brought onto and off of the chip via PCI-e.

Tasks can communicate in one of two ways. One way is through a shared memory. The Memory Unit Element transfers blocks efficiently and notifies any downstream tasks of the completion so that these tasks won't use data before it is valid.

The other way for tasks to communicate is through the streaming interfaces. When a task has a streaming input, it receives streams of data. Tasks can be programmed to "listen" to the output of another task making the broadcast of data efficient.

Another important feature for run-time reconfigurable systems is Run-Time Rebinding. In Illustration 1.1, we have two tasks that are connected in sequence. At some point, the Main task determines that subtask PL1 is no longer needed and that subtask PL2 should be used instead. The main task then does the following:

1. Suspend the streams that connect the Main task to Task PL1. All processing inside the main task continues running, except for the transmission of data to Task PL1.
2. Free Task PL1. This halts the task and frees its resources.
3. Load Task PL2 directing it to listen to the main task's output stream.
4. Redirect the main task's TALU to listen to PL2's output stream.
5. Run Task PL2.
6. Restart the Main Task's output stream.

These steps are performed very quickly. PL1 and PL2 do not have to occupy the same resources. If sufficient resources exist on the device for all three tasks, then steps 2 and 3 are omitted.

2. ELEMENTAL COMPUTING

Elemental Computing is a parallel, distributed, data-flow paradigm. The execution units are 16- and 32-bit operators. All of the Elements are self-timed, so no timing closure is required. Data transfers in every stage in a task are controlled automatically by a handshaking protocol that is hidden from the user. Their only concern is the levels of latency between Elements. Elements are connected to other Elements in a pipelined fashion. Elemental interconnect is word-based, and includes a broadcast capability so a single Element can transmit to multiple destination Elements.

Elements contain up to eight dataflow instructions, called contexts. These contexts are not executed sequentially as in a traditional processor. Instead, they are executed when the data arrives at the appropriate instruction. Each of the instructions in an Element can be allocated to only a single task for the lifetime of that task. Instructions from the same or different tasks can occupy other contexts of the same Element.

One of the keys to high-speed execution and resiliency is keeping data close to the execution units. This allows multiple data to be fetched simultaneously, instead of sequentially as from a central memory. Distributed memories have the added intentional implication of being fault tolerant.

2.1 Elements

Elements are non-homogeneous data-flow computational engines. All Elements have the same form, but different capabilities. This allows us to provide efficient implementations of each of the Element type, without having to provide all the capabilities of all the Elements in each computing node. There are currently seven Element types as shown in Table 1.

Table 1: Element Types

BREO	Bit Reorderer	SALU	Super ALU
BSHF	Barrel Shifter	TALU	Triple ALU
MEMU	Memory Unit	SME	State Machine Engine
MULT	Multiplier		

Each Element Type has four 16-bit inputs and two 16-bit outputs, as shown in Illustration 2.1. Some Elements have the capability of ganging a pair of inputs or outputs

together to perform 32-bit functions. Each input and output of an Element is queued, isolating the Element from the interconnect delays and bounding the maximum clock delay. Every Element executes an operation in one clock cycle.

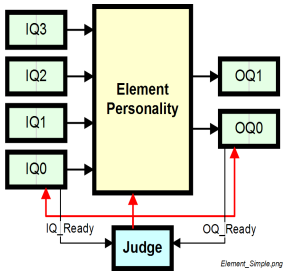

Illustration 2.1.: Element Interfaces

Illustration 2.1 also shows the Judge that is necessary for the data flow operation of the Element. A Judge monitors the input and output queues of its Element. Other Elements supply this Element's input queues with data operate asyncronous to this Element. A Judge ensures that its Element will not operate unless there is a valid word in each of the input queues and room to store the result in an output queue. Thus, when the Element begins execution, it is guaranteed to read valid data and have a place to store the results, all in one clock cycle.

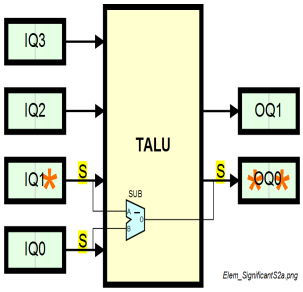

Illustration 2.2.: Element with Insignificant Inputs & Outputs

All (but one of) the Elements have four inputs and two outputs. The Elements are highly configurable, and may be reconfigured on a clock-by-clock basis, as we will discuss below. Some of these configurations may not need all of their inputs to perform a calculation. Such a configuration is shown in Illustration 2.2. Inputs `I0` and `I1` and output `O0` are used by the configuration, while the other inputs and output are not used. To avoid having to wait for unused inputs and outputs, we mark the inputs and outputs that are used in a calculation as *Significant*. Unused inputs and outputs are termed *insignificant*. The Judge does not check insignificant inputs or outputs. In Illustration 3.2, `I0`, `I1`, and `O0` are significant. `I1` contains one valid value, and `O0` contains two. Therefore the Judge will not execute this configuration for two reasons: (a) it does not have data in the significant input `I0`, and (b) its output `O0` is full.

The Element, as described so far, can suffer from being idle while waiting for inputs to arrive. To circumvent this, the execution portion of the Element is shared across multiple *contexts*. Each context contains a *configuration*, which defines the function that the Element will perform, the four input queues and two output queues, and their configuration data. This is shown in Illustration 2.3. In each clock cycle, the Judge examines the significant inputs and outputs for each context, looking for the Execution Criteria:

a) Every significant input must have at least one data value, and

b) *Every significant output must have at least one empty location in its queue.*

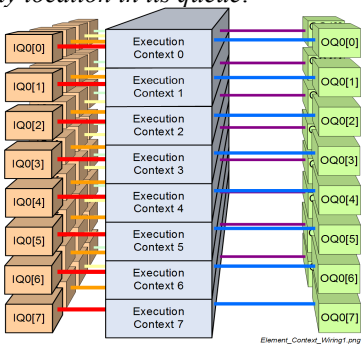

Illustration 2.3.: An Element with Eight Contexts

After a context has executed:

a) The results are stored in the output queue for that context.

b) *The inputs are consumed, removing them from the input queues for that context.*

Because every Element has its own Judge it can be operating, independently, on every clock cycle, on its own local data. (The SME and MEMU operate a little differently, as will be described in a later section.)

An Input Queue may be written to while the Element is reading data. Similarly, an Output Queue may be written to while it is transmitting data. It is natural for an Element to context switch on every cycle, and, at the same time, collect data to be operated upon in future cycles.

2.2 Hierarchy

Elements are organized hierarchically. Four elements are grouped into a Zone. Elements in a Zone are tightly bound, communicating within a single clock cycle. Four Zones are grouped into a Cluster. All of the Zones in a Cluster communicate with each other through a number of special queues, called Through Queues. All Clusters are identical. The Cluster is the unit of replication on the die.

Up to sixteen Clusters are grouped into a Super-Cluster. Clusters within a Super-Cluster can communicate resiliently through a hierarchical bus structure, or more expediently and less resiliently through local interconnect.

Up to sixteen Super-Clusters are grouped into a Matrix. Super-Clusters within a Matrix communicate exactly

as Clusters and this method of interconnecting levels of hierarchy extends indefinitely. ECA devices communicate via PCI-e in the same hierarchical fashion, extending the hierarchy to the board level.

2.3 Element Types

An Elemental Computing Array contains seven Element types designed with digital signal processing in mind, but these are not your father's traditional DSP.

2.4 SME – State Machine Engine

Each Cluster contains a SME. The SME is a custom RISC processor. The SME has a small instruction set that can be augmented by Elements in its Cluster. The SME is used to:

- Execute sequential code. This is useful for blocks of code that either do not lend themselves to parallelism or are executed infrequently.
- Process Interrupts from the Cluster's Elements and the Message Manager.
- Extend Elemental Instructions by emulating a data flow Element.
- Extend the SME's instructions by using the data flow Elements to make arbitrary instructions.
- Bind new tasks to resources and connecting tasks to other tasks.
- To manage tasks by starting, suspending, halting, and single-stepping tasks and their Elements. Task management is asyncronous to and independent of any other already-running tasks.

The SME's instructions and data come from the Cluster RAM. The SME's instruction memory is protected from accidental access by the Elements using the RAM as data storage.

2.5 MM – Message Manager

Each Cluster sends and receives data packets through its Message Manager. The Message Manager communicates between Clusters and, to the PCI-e interface to the ECA at the device level. The messages are posted and non-posted, and come in three forms.

Write Messages – Inside a Cluster, this can be Cluster RAM, the memory-mapped configuration addresses, or memory-mapped control for the Elements and the SME. Outside a Cluster, it can access other Clusters, transfer RAM contents, or configure/control tasks in other Clusters. It is also used to transmit data off-chip to peripherals.

Read Message – This is the mechanism for reading data from off-chip or from other Clusters.

User Messages – Communicate between user tasks.

Messages, whichever form is used, operate on the Cluster via DMA that is asyncronous to, and independent of, the operation of the SME and the Elements and executing tasks.

2.6 BREO – Bit Reorderer

The BREO can be configured to perform:

- Interleaving bits from up to four data streams.
- Select bits from each of four 16-bit words.
- Puncture (remove) bits from input words.
- Choose input words based on a masking functions.
- Control structure in FOR and WHILE loops.

2.7 BSHF – Barrel Shifter

The Barrel Shifter performs the following shifting functions of 1 to 32 bits.

- Logical
- Circular
- Arithmetic; with and without saturation
- Word reversal
- Shifting functions for variable-length encoding.

2.8 MULT – Multiplier

The multiplier is based on a partitionable 16-bit multiplier that performs a 16x16 multiplication or two 8-bit multiplications in one cycle. Additionally, the multiplier can perform a 32-bit multiplication in four cycles. The multiplier also performs the following modes.

- Unsigned Integer
- Unsigned Fraction (Q.15 and Q.31)
- Signed Integer (two's complement)
- Signed Fraction (Q.15 and Q.31)

The MULT contains a 64-bit accumulator for 32-bit and 16-bit modes. This accumulator is split in half when executing in 8-bit mode so that two MACs can be performed simultaneously.

2.9 TALU – Triple ALU

The TALU contains three configurable 16-bit ALUs, two optional ABS, and three optional data aligners, as shown in Illustration 2.4.

The ALUs can be configured to perform any of ten different logical operations or any of six signed arithmetic functions.

The TALU can also be configured to perform a data-steering function called *conditional mode* because it evaluates a conditional of I2 and I3 in ALU2. Based on that value, the values from I0 and I1 are either passed to O0 and O1 or swapped, being passed to O1 and O0. This provides a

powerful and flexible way of merging or interleaving two data streams.

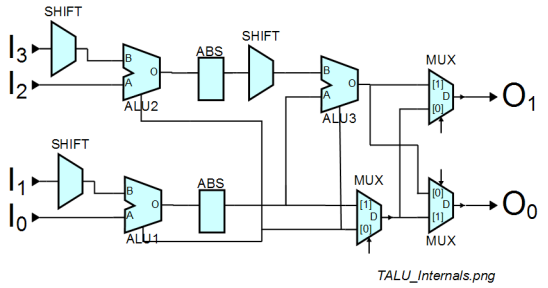

Illustration 2.4.: TALU Internals

2.10 SALU – Super ALU

The SALU performs any of fourteen logical operations or ten arithmetic operations on either 16- or 32-bit signed or unsigned data. When configured to perform 16-bit arithmetic, it uses the four inputs as four data sources. When configured in 32-bit mode, the four 16-bit inputs are paired to form two 32-bit numbers. Signed arithmetic is two's complement, and may be saturated or not. The SALU also computes a number of useful functions, such as counting the number of leading ones and the number of leading zeros in a word, and counting the total number of ones and the total number of zeros in a word.

An alternate configuration of the SALU is used in control-flow. The SALU can be configured to control CASE statements and FOR loops.

2.11 MEMU – Memory Unit

The MEMU provides data flow access to Cluster RAM. The RAM can be accessed a word at a time, or as FIFOs or as blocks of RAM. The MEMU contains six powerful Address Generators (AG) that may be tied to individual execution contexts. Each AG can be used in either FIFO or Block mode. An AG allows a single execution context to access a portion of RAM as a block, reading or writing a block as an atomic instruction. The words in a block can be accessed sequentially, or strided through in a positive or negative direction. Two AGs can be ganged together to provide two-dimensional addressing, each AG using its own stride to define the rows and columns of the array.

The Cluster RAM is composed of eight 2K short sub-blocks. Each is a single-ported RAM, but can be accessed independently from the other RAM blocks. The MEMU Judge allows up to eight simultaneous RAM accesses, but only one access per block. These accesses include the SME's instruction and data fetch as well as the Message Manager's DMA access. The Judge makes sure that only one context per Input Queue and only one context per Output Queue is made. Thus, the MEMU can execute six data flow instructions in one clock cycle, providing data streams to the other Elements and providing data storage for stream data from the other Elements.

2.12 Memory Hierarchy

The ECA has a hierarchical, distributed memory. Distributed memory allows many calculations to proceed in parallel. Placing memory with the calculating circuitry removes the path of getting data out of the memory and into the processing components. Distributed memory also provides a form of redundancy, which contributes to greater resiliency.

2.13 Queues

Small amounts of memory in the form of queues, are located on the inputs and outputs of each Element. Each queue is two words deep, but is enough to separate the Elemental and interconnect timing domains and guarantee fast cycle times. There are 1,984 short words in all the queues in a Cluster. The queues can be configured as a 2-word dynamic FIFO or as a 1- or 2-word constant FIFO. A dynamic FIFO is commonplace. An Input Queue is written into by the interconnect (driven by the Output Queues) and read independently by the Element. A Constant FIFO is one whose values are never consumed (destroyed) by the Element. If there is one value in a Constant FIFO, it will be read forever. If there are two constants in the Constant FIFO, then they will be read alternately forever. Constant FIFOs can thus be used to supply constants, coefficients, initial values, etc. instead of taxing a central memory.

The next level of memory comes from grouping all contexts of an Input Queue together to get 16 words of data. When all eight contexts on an IQ are configured into a 16-word queue, all contexts of the Element may read from the head of this queue. Somc only one context can execute at a time there's no contention for reading the queue. This mode is used for a 16-word FIFO, to equalize unequal-latency branches of a computation, and to supply a small number (3–16) of constants to an Element. A Constant Queue can be written to as part of the data flow, or by the SME or the Message Manager for use in adaptive filters.

2.14 RAM

Most algorithms need RAM or ROM. Every Cluster in the ECA has 16K short words (16-bits wide). The RAM is built from 2K short word RAMs that can be accessed and allocated independently. The flexibility allows a user/ designer to trade off memory for SME instructions, data, and MEMU blocks and FIFOs. Some algorithms require more sequential code, others more memory. This trade-off can be made at compile or run-time.

When a MEMU address generator accesses the RAM as a FIFO, the size of the FIFO can be anywhere from two words to the entire size of the RAM. The same is true of the MEMU blocks. Blocks and FIFOs can exist anywhere in memory.

2.15 Interconnect

A novel interconnect scheme is used within each Elemental Zone. A Zone contains four Elements, each with four input queues and two output queues. Each Zone also has eight Through Queues (TQs) that connect to adjacent Zones. A subset of these TQs connect to adjacent Clusters. There is full interconnectivity within a Zone. Every source within a Zone, that is, the Elemental Output Queues and incoming TQs, broadcast to every destination within that Zone. A destination is a context of each Input Queue and a context of an outgoing TQ in that Zone. Every destination *subscribes* to one, and only one, source within that Zone. Multiple destinations can subscribe to the same source, making replication of data transparent.

The converse of data broadcast is data merging. Data merging is provided in each Element, between the Elemental function and the Output Queue. Every execution context normally writes its outputs to the same context of its Output Queue. However, this is programmable. A context can write to *any* specific context of its Output Queues, as is shown in Illustration 3.5. Here, execution contexts 0 and 1

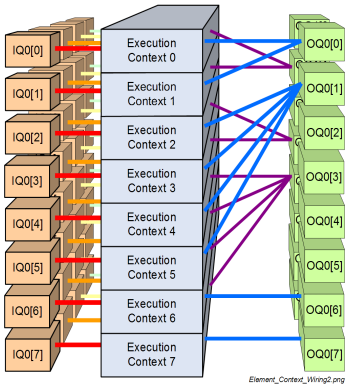

Illustration 2.5.: Stream Merging through Output Queue Mapping

write their outputs to OQ0 context 0, thus merging the two resultant data streams. In the same illustration, execution contexts 2–5 write to OQ0 context 1, merging four data streams. Note that this OQ mapping may be different for the two Output Queues.

3. TESTING STRATEGIES

Two traditional means for testing a device in the field are built-in testing and programmable testing.

Built-in Self Testing [3] uses special circuitry to test the device (primarily) at power-on. When an error is found the device can no longer be used (or trusted). But, what if the error is in the BIST circuitry?

Programmable Testing is nothing new. Every time you boot your PC, the BIOS runs programmable self-test routines.

Our approach is to use the Elemental circuitry as form of programmable testing. It can do this at power-on or, because the ECA is multi-tasking, tests can be run while application tasks are running. ECA testing is a matter of running specific tasks at the desired frequency.

The frequency with which you should test a device is dictated by the bathtub curve. Frequent tests are the rule when a device is new and when it is “old”. Testing in-between can be infrequently. The testing frequency is, therefore, up to the designer, and the devices environment.

4. APPLICATIONS

We implemented a 20 MHz version of the AES cipher on the Elemental Architecture. The target data rate for H.264 was met in a small number of resources, 31 contexts in a total of two Zones. A second benchmark is a 64-point FFT, computing one cycle per butterfly. This runs at 200 Mhz with a latency of 192 cycles. This FFT is parameterized up to a 4K-point FFT. The performance of this FFT is suitable for WiMAX, Wi-Fi, DVD, and ISDBT.

5. SUMMARY

The Elemental Computing Array was designed to meet reconfigurable and resiliency goals, and ended up as a powerful computing paradigm that is fault tolerant, yet power and performance efficient. The multi-tasking device allows relocatable tasks to communicate efficiently with one another. Rapid reconfiguration of the device at the Elemental level provides a high-level of silicon utilization and throughput.

6. REFERENCES

[1] D. Wilkins, “The Bathtub Curve and Product Failure Behavior,” *weibull.com Reliability HotWire*, http://www.weibull.com/hotwire/issue21/hottopics21.htm.

[2] D.L. Goodman, *IEEE Transactions on Components and Packaging Technologies*, Volume 24, Issue 1, March 2001, pages 109–111.

[3] B. Koenemann, J. Mucha, and G. Ziehoff, "Built-in test for complex digital integrated circuits," *IEEE J. Solid State Circuits*, vol. SC-15, pp. 315-319, 1980.