RISA: A Hardware Platform for Evolutionary Design

Andrew J. Greensted
Department of Electronics
University of York
Heslington, York, UK
Email: ajg112@ohm.york.ac.uk

Andy M. Tyrrell
Department of Electronics
University of York
Heslington, York, UK
Email: amt@ohm.york.ac.uk

***Abstract*— The Reconfigurable Integrated System Array is a new form of field programmable digital device incorporating both hardware and software reconfigurable elements. RISA was developed specifically for implementing biologically inspired electronic systems. The architecture is particularly suited for investigating evolvable hardware due to the flexibility of RISA's configuration system. Fine grained partial reconfiguration is supported and random configuration bitstreams will not cause device damage. This paper describes the new RISA chip and the features that make it suitable for bio-inspired applications.**

I. Introduction

The vast majority of electronic components are designed by humans and are intended for use in human designed systems. Component datasheets provide specific operational instructions allowing a designer to make use of a device and avoid their misuse. Devices that incorporate programmable elements, whether FPGA, microcontroller or the most basic PLD use specific configuration systems; tailored hardware and protocols provide access to the user configurable parts. Overall, commercial electronic components are heavily dependant on being utilised in a constrained and correct manner.

If the trained designer is removed from the system design process and the operational constraints of components are ignored, the likely result is at best, a non-working system and, at worst, damaged components. Ignoring even the simplest procedure or instruction can lead to problems such as incorrect package alignment, bad device programming, wiring contention, poor power planning and clock skew problems.

Adhering to operational constraints avoids component damage. However, manufacturer imposed constraints also limit the potential functionality of a device. Many programmable devices can only be configured in a particular fashion. It is not always possible to program a device in what could be considered the most flexible manner.

The power of evolutionary design techniques is their ability to search throughout the problem solution space, evaluating novel solutions that conventional design would not consider. Unfortunately, this process is not immediately compatible with the very constrained manner in which electronic components should be used. Electronics will allow you to do things you should not, and in contrast, it is not always possible to achieve the flexibility that evolution can make use of. In essence, commercial devices are not designed to be utilised using such an unconstrained and exploitive technique.

Clearly electronic systems have been designed using evolutionary processes. However, unless evaluation is undertaken by simulation, or the potential of damage is inconsequential, or the inflexibility of device manipulation can be tolerated, it is necessary to constrain solutions such that their evaluation is allowable within the operational constraints of the underlying hardware. Constraining evolution in such a manner restricts the search space and the ability to find a novel solution.

The highly configurable nature of field programmable devices has made them a very popular choice for evolving electronic systems [1]–[3]. Unfortunately, they suffer from the constraints problems outlined above. Reconfiguration means that multiple candidate solutions can be tested. However, solutions must be constrained to safe configurations and the flexibility of reconfiguration is limited to the options available in the device.

Ideally, the evolutionary process should have the ability to search as widely, freely and safely as possible. What evolvable hardware really needs are devices that have very flexible and inherently safe configuration systems. Ideally, devices need fine grained configuration and be random configuration safe. However, commercial devices either do not posses these properties or only in a very limited manner. For example, current Xilinx® devices only have course partial reconfiguration capabilities [4]. The older XC6200 [5] series was an exception, having a more flexible configuration system, however, it is no longer manufactured.

Two approaches can be undertaken to overcome this problem. The first, being the easiest, is to implement a custom reconfigurable architecture *on-top* of a standard device [6], [7]. This allows the designer to create a reconfigurable platform with features suitable for evolution. Unfortunately, implementing one reconfigurable circuit upon another is inefficient, limiting the size of circuit available for use.

The second approach is to design and manufacture a new device. An ASIC solution provides the maximum flexibility to the designer. However, the greater cost of ASIC fabrication in comparison to FPGA implementation makes this approach less popular, but certainly not unheard of [8]–[10]. This is the approach taken and described in this paper. The Reconfigurable

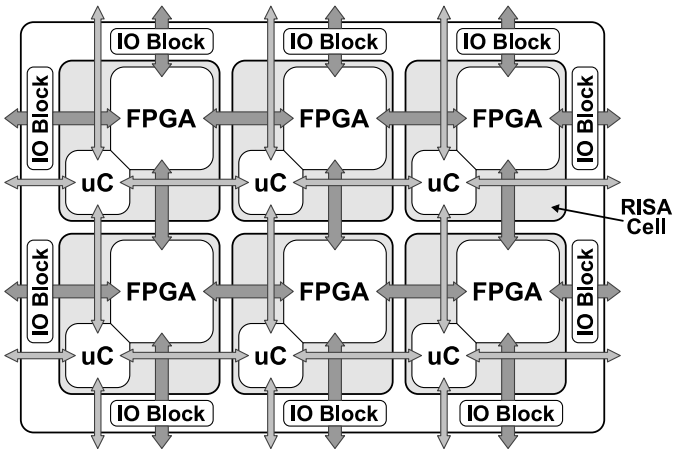

Fig. 1. The RISA architecture. A two dimensional array of highly reconfigurable RISA Cells.

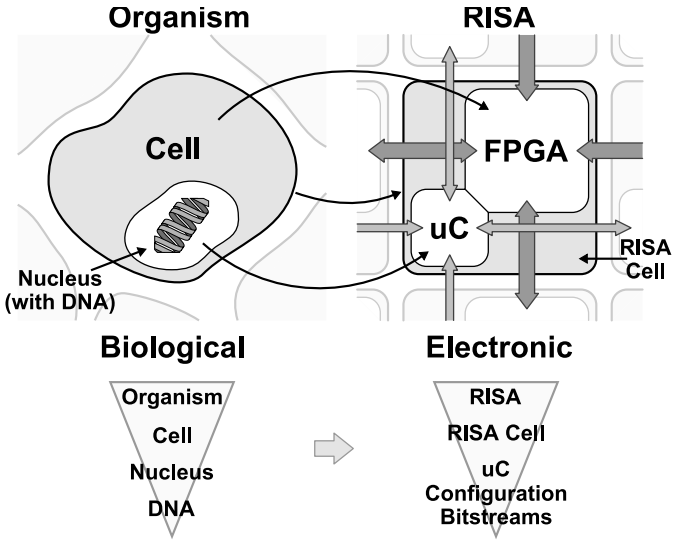

Fig. 2. The RISA cell is inspired by the structure of biological cells. The cell's functional flexibility is achieved by the integration of a microcontroller and FPGA.

Integrated System Array (RISA) architecture is a new type of field programmable electronic platform designed to support bio-inspired research, including evolutionary design.

Section II introduces the RISA architecture. Sections III & IV describe its two main components: a custom microcontroller core and an FPGA fabric. A disucssion in Section V concludes the paper.

II. The RISA Architecture

The Reconfigurable Integrated System Array (RISA) architecture is a new form of reconfigurable digital device. The architecture incorporates both hardware and software reconfigurability in the form of FPGA blocks and a microcontroller array. The architecture is illustrated in Figure 1.

The project aim was to create a new device more appropriate to implementing bio-inspired applications, in particular those with a cellular structure. Each node of the architecture's array is constructed from a microcontroller and FPGA, forming the *Integrated System* referred to in the architecture's name. Figure 2 illustrates one of these nodes, called a RISA cell, and how its structure is inspired by that of a biological cell. The microcontroller provides a simple method of performing cellular specialisation, being able to control the configuration of the FPGA. The flexibility of the FPGA fabric makes it ideal for implementing a variety of cellular functions.

The architecture's cellular inspiration does not limit the device to implementing only cell-based systems; the architecture is still applicable to other applications such as evolvable hardware. The following features highlight the devices benefits over commercial devices:

- A fine grained partial reconfiguration system. Bitstream loading occurs without disrupting circuit operation and subsequent reconfiguration occurs in a single clock pulse.
- The architecture's multiplexer based configurable fabric can not be configured into a contentious state. Therefore, it is possible to use random bitstreams without risk of device damage.
- Each RISA cell's microcontroller can be used to perform intrinsic reconfiguration of the FPGA fabric.
- Each microcontroller has a dedicated full-duplex, hardware flow-controlled communication link with its four nearest neighbours.

The RISA microcontroller and FPGA fabric, the two main parts of the architecture, are described in Sections III and IV respectively.

III. SNAP

A custom microcontroller core, called the Simple Networked Application Processor (SNAP), was developed for the RISA architecture. As the name suggests, the microcontroller is suited to applications requiring a loosely coupled processor network. Dedicated modules provide an easy method of inter-core communication. However, the microcontroller is also designed for close integration with the hosting RISA cell's FPGA fabric. FPGA configuration can be driven from the microcontroller, and the core's instruction set simplifies bitstream creation and manipulation. A simplified illustration of the SNAP core is shown in Figure 3.

SNAP is a 4 stage pipelined RISC core. It has a 16 bit data width and a 16 bit address space. A Von-Neumann memory architecture is used in order to make the most efficient use of the core's memory space, since a single memory space avoids unused program space that could be used for data storage. As well as standard computational operations, a number of additional function modules are accessible via the core's register file. There are 128 register file locations, the assignment of these being deliberately very flexible. In the configuration used in the first RISA chip, 32 locations are used as working registers, the rest are available for connecting to peripheral modules. This approach allows designers to specialise the SNAP core to their purpose by including custom

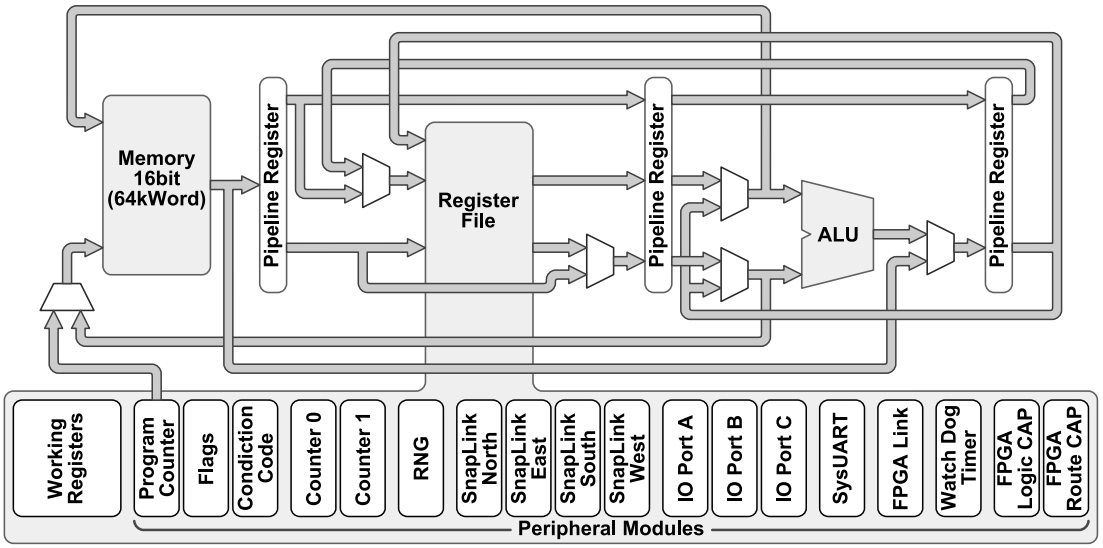

Fig. 3. The SNAP microcontroller, a 16 bit data width, 16 bit address space, 4 stage pipelined, RISC, Von Neumann memory design. Peripheral modules, accessed via the register file provide extra functionality.

peripheral modules. Descriptions of the default set of modules are given below:

Configuration Access Port - The Configuration Access Ports (CAPs) are used to control the microcontroller's neighbouring section of FPGA fabric. As described further in Section IV-A, the two CAPs provide access to the logic and routing configuration and selection chains. The two CAPs are independent and can be operated concurrently to speed-up FPGA configuration time.

Random Number Generator - A 8x8 cellular automata (CA), based on Shackleford's design [11], is used to generate a stream of pseudo-random numbers. The CA connectivity and next state logic were evolved to find a configuration that generated a number sequence satisfying the most tests from the DIEHARD [12] battery of randomness tests.

SNAPLink - The SNAPLink is the main inter-core communication port. Each core has four independent modules for north, south, east and west connections. Hardware flow control is provided to simplify the transfer of data between cores, which can be achieved as easily as writing to, or reading from, the appropriate SNAPLink register.

SysLink - The SysLink is a general purpose full-duplex serial interface. The link's UART is compatible with a standard PC serial port. With appropriate line-driving the SysLink can be directly connected to a PC for debugging purposes.

FPGA Link - The FPGA Links provide general purpose connections between the SNAP core and the neighbouring FPGA fabric. These bidirectional ports can be configured into the FPGA routing.

Counters and WDT - The 16-bit Counters add timing and counting functionality. These modules can be used solely in the SNAP core or to supply different digital waveforms to the FPGA fabric. The Watchdog Timer (WDT) provides a simple method of failure recovery.

IO Port - General purpose connectivity external to the RISA chip is made available via the bidirectional IO Ports.

A. Instruction Set & Encoding

The design of the SNAP instruction set and instruction encoding is key to the core's successful operation for applications such as evolutionary design and dynamic FPGA configuration control. The core's reduced instruction set contains 19 instructions which are listed in Table I. Included in this set are instructions important to both FPGA bitstream and evolutionary genotype data manipulation. For example, `bs` and `bc` instructions are particularly useful for performing evolutionary mutation, whereas FPGA bitstream alignment can be quickly performed using `rol` and `ror` instructions. As shown in Table I each instruction can be used to operate on two register stored values, type 0, or a register and a instruction encoded literal value, type 1.

The power of what initially seems a limited instruction set is greatly enhanced by a set of execution controls. As shown in Figure 4 the instruction encoding includes a set of control flags that determine whether an instruction executes and how it affects the state of the microcontroller. These control flags are described below:

insnEnable *(bit 30)* - This is a global instruction execution enable flag. When set to zero the instruction will still pass through the microcontroller pipeline but will not execute, no results are stored, and no status flags are set. This single bit flag can easily be toggled without affecting any other instruction parameter and therefore provides a powerful control for self-modifying code.

conditional *(bit 29)* - The conditional flag declares whether the instruction's execution should be dependant on a microcon-

TABLE I
SNAP INSTRUCTION SET

Mnemonic	Op. Code	Instruction (Type 0)	Instruction (Type 1)	Operation
`add`	0	$\mathbf{reg_A} = \mathbf{reg_A} + \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A} + \mathbf{lit}$	Addition
`addc`	1	$\mathbf{reg_A} = \mathbf{reg_A} + \mathbf{reg_B} + c$	$\mathbf{reg_A} = \mathbf{reg_A} + \mathbf{lit} + c$	Addition with carry
`sub`	2	$\mathbf{reg_A} = \mathbf{reg_A} - \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A} - \mathbf{lit}$	Subtraction
`subc`	3	$\mathbf{reg_A} = (\mathbf{reg_A} - \mathbf{reg_B}) - b$	$\mathbf{reg_A} = (\mathbf{reg_A} - \mathbf{lit}) - b$	Subtraction with carry
`neg`	4	$\mathbf{reg_A} = -\mathbf{reg_A}$	$\mathbf{reg_A} = -\mathbf{lit}$	Negation
`and`	5	$\mathbf{reg_A} = \mathbf{reg_A}$ & $\mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A}$ & $\mathbf{lit}$	Bitwise and
`or`	6	$\mathbf{reg_A} = \mathbf{reg_A} \mid \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A} \mid \mathbf{lit}$	Bitwise or
`xor`	7	$\mathbf{reg_A} = \mathbf{reg_A} \wedge \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A} \wedge \mathbf{lit}$	Bitwise exclusive or
`rol`	8	$\mathbf{reg_A} = \mathbf{reg_A} \ll \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A} \ll \mathbf{lit}$	Rotate left (towards MSB)
`ror`	9	$\mathbf{reg_A} = \mathbf{reg_A} \gg \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{reg_A} \gg \mathbf{lit}$	Rotate right (towards LSB)
`bs`	10	$\mathbf{reg_A} = \mathbf{reg_A} \mid (1 \ll \mathbf{reg_B})$	$\mathbf{reg_A} = \mathbf{reg_A}$ & $(1 \ll \mathbf{lit})$	Bit set
`bc`	11	$\mathbf{reg_A} = \mathbf{reg_A}$ & $!(1 \ll \mathbf{reg_B})$	$\mathbf{reg_A} = \mathbf{reg_A}$ & $!(1 \ll \mathbf{lit})$	Bit clear
`mov`	12	$\mathbf{reg_A} = \mathbf{reg_B}$	$\mathbf{reg_A} = \mathbf{lit}$	Move
`ld`	13	$\mathbf{reg_A} = MEM[\mathbf{reg_B}]$	$\mathbf{reg_A} = MEM[\mathbf{lit}]$	Load
`st`	14	$MEM[\mathbf{reg_B}] = \mathbf{reg_A}$	$MEM[\mathbf{lit}] = \mathbf{reg_A}$	Store
`bra`	15-0	$PC = \mathbf{reg_A}$	$PC = \mathbf{lit}$	Branch
`reti`	15-1	$PC = \mathbf{reg_A}; GIE = 1$	$PC = \mathbf{lit}; GIE = 1$	Return from interrupt, set global interrupt enable bit
`fset`	15-2	$Flags = \mathbf{reg_A}$	$Flags = \mathbf{lit}$	Set Flags
`cset`	15-3	$Condition = \mathbf{reg_A}$	$Condition = \mathbf{lit}$	Set Condition for Type 1 instructions

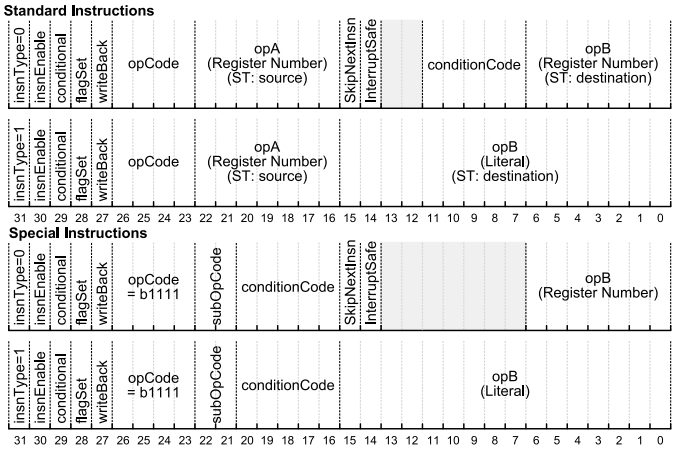

Fig. 4. Instruction Encoding.

TABLE II
SNAP STATUS FLAGS AND EXECUTION CONDITIONS

#	Status Flag	Condition	$\overline{\text{Condition}}$
0	Z	Zero/Equal	Not Zero/Not equal
1	CB	Carry/borrow	No Carry/No Borrow
2	N	Negative	Positive
3	O	Overflow	No Overflow
4	ET0	Less than	Greater or equal
5	ET1	Less or equal	Greater than
6	ET2	Less than (unsigned)	Greater than (unsigned)
7		Global Interrupt Enable	Not GIE
8-15		External flags 0-7	Not EF 0-7

troller status condition. The conditions are listed in the third and fourth column of Table II. Type 0 and special instructions have the condition code encoded in the instruction, whereas Type 1 standard instruction use a separate condition code register, set via the `cset` instruction.

flagSet *(bit 28)* - By default, instructions do not alter the microcontroller status flags. This bit causes the flags to be updated to reflect the status of an instruction's result. The flags are those listed in the second column of Table II. The default of not setting flags means many instructions can be made conditional on the same instruction result.

writeBack *(bit 27)* - The writeBack flag controls whether the result of the operation is stored. Disabling this flag and enabling the flagSet flag converts any instruction into a non-destructive test operation.

skipNextInsn *(bit 15)* - When set, the skipNextInsn flag causes the microcontroller to disable the next instruction. This flag is especially useful when used with a conditional instruction and the next instruction is a branch.

interruptSafe *(bit 14)* - The interruptSafe flag is used to stop an interrupt branch during an instruction's execute stage. This is required when an instruction accesses a read-sensitive register such as a FIFO. Such a source would return a different value when re-read after return from an interrupt. Setting the interruptSafe bit postpones the interrupt and allows the read value to be stored.

B. SNAP-FPGA Integration

Being able to perform intrinsic FPGA reconfiguration is one of the main motivations for coupling a microcontroller with the FPGA fabric. The dedicated CAP modules make this a simple operation. However, there are a number of methods that further integrate the two RISA Cell components. As mentioned, the FPGA Link provides a general purpose interface. In addition,

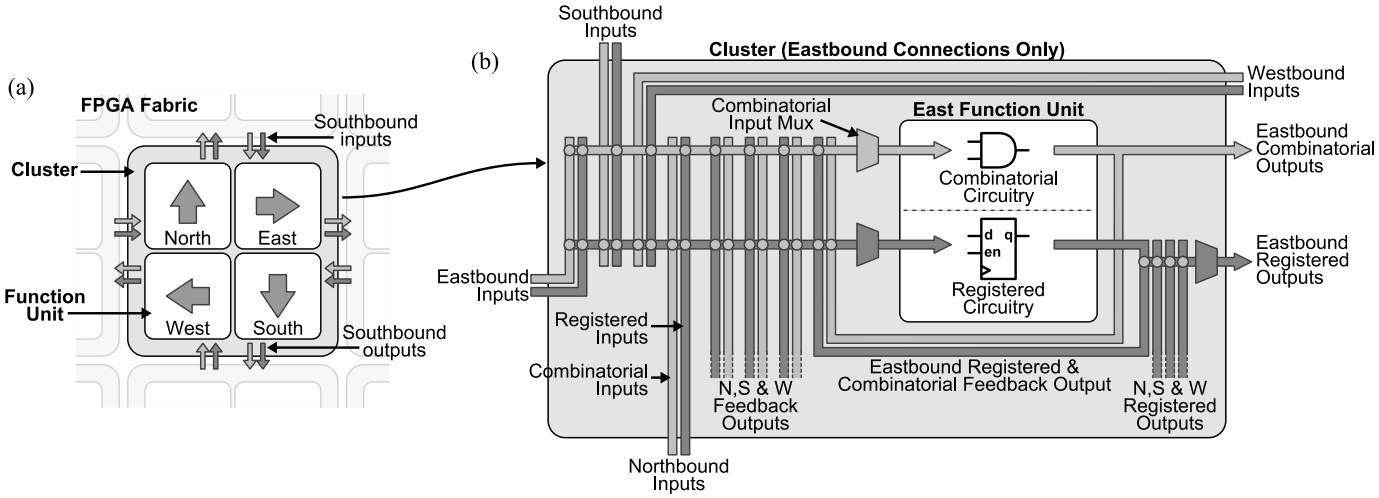

Fig. 5. The FPGA fabric is formed by an array of Clusters. Each Cluster contains four function units that contain the configurable logic elements.

the SNAP counters can be driven by the FPGA and also generate waveforms that drive FPGA signals.

The SNAP core has a number of interrupt sources that indicate events triggered in the peripheral modules. Among these interrupt sources are interrupt request lines connected to the FPGA fabric. This allows the FPGA to trigger an interrupt service routine, for example, to read data presented at the FPGA Link ports.

Execution of microcontroller code can be further controlled by FPGA signals that drive SNAP condition codes. As shown in Table II, conditions 7 to 15 are driven from external sources. Depending on the configuration, these sources are a mixture of off-chip inputs and signals driven by the FPGA fabric. Using this technique the execution of blocks of SNAP code can be made dependant of FPGA signal states.

IV. FPGA Fabric

The FPGA fabric provides a hardware reconfigurable element to the RISA architecture. The FPGA architecture does not attempt to compete with commercial devices in terms of density and flexibility of circuit implementation but aims to provide a more appropriate configuration system for bio-inspired systems. In particular, as mentioned earlier, by offering fine grained partial reconfiguration and being random bitstream safe.

The main configurable component of the gate array is the Cluster. Each cluster contains logic circuitry and the routing for interconnecting internal circuity and clusters. The Function Unit is the main configurable logic element. There are four Function Units in each Cluster. Figure 5a illustrates the FPGA structure.

The circuitry of the Function Unit contains three core elements: a Function Generator, a 2-1 multiplexer, and a single D-type flip-flop. These may be used individually, but can be combined to provide extra functionality such as the configurations listed below.

- 4 input, 1 output Look-Up Table (LUT)
- 16x1 bit RAM
- 1 to 16 bit variable length Shift register block (extendible via a dedicated shift chain to other clusters)
- 4-1 multiplexer
- 1 bit full adder with fast carry chains for expansion into other clusters

The routing scheme is critical to creating a configurable system that can tolerate random configurations. The RISA FPGA uses multiplexer based routing as opposed to a bus based scheme. This avoids the possibility of configuring a wiring resource into contention; driving the same wire with different logic values.

A second potential problem is the creation of combinatorial feedback loops. A feedback path can create fast oscillations, which unnecessarily consume power, cause noise in neighbouring signals via crosstalk and can cause metastability issues on connected registers. The RISA FPGA uses directed combinatorial routing so that unregistered feedback paths can not be created. As indicated in Figure 5a, each of the Cluster's four Function Units are assigned a different direction. This refers to the Function Unit's combinatorial routing direction. Using this scheme, a north Function Unit can only have a purely combinatorial connection to the north Function Units of Clusters to the north. Conversely, a registered signal, one output from a flip-flop, is free to connect in any direction.

Figure 5b illustrates the connections for a Cluster's East Function Unit, those for the North, South and West Function Units being removed for clarity. Any input can be connected to a registered logic function, whereas the purely combinatorial circuitry can only accept inputs from eastbound combinatorial signals and registered signals.

A. FPGA Configuration

The RISA FPGA Fabric uses two separate serial data chains to configure the logic circuitry and the routing connectivity. This approach provides for a faster configuration system. Both chains can be utilised simultaneously therefore allowing

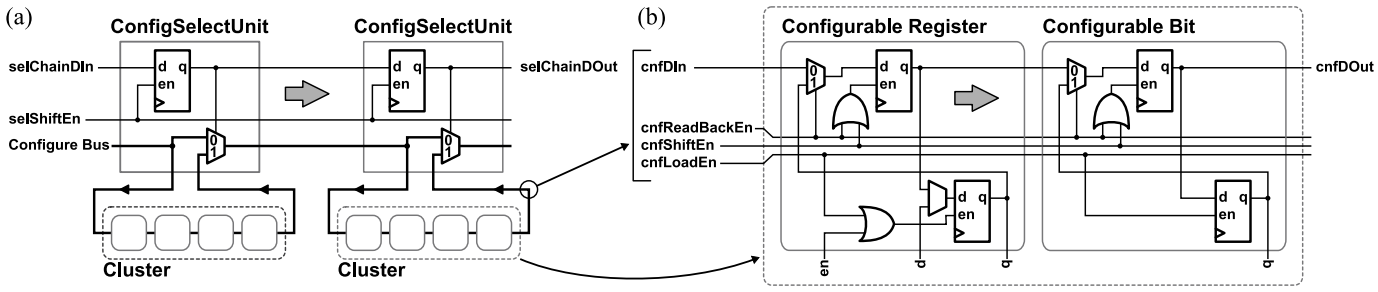

Fig. 6. The RISA FPGA configuration chains are divided into switchable sub-chains that can be targeted for configuration individually or in combination. Configuration data is loaded without affecting current device operation.

concurrent configuration of logic and routing. Additionally, two shorter chains means that targeting an area of the FPGA fabric using RISA's partial configuration process is also faster.

Rather than creating a configuration chain from the registers that control the FPGA's configurable resources, a separate serial data chain is used. This dedicated shift chain can be operated without affecting the currently configured circuit. Once data is loaded, it can be simultaneously loaded into the controlling registers.

Figure 6b shows the two types of configurable element used in the FPGA fabric. The *Configurable Bit* has a single control output, q, that is used to drive components such as a routing multiplexer. The *Configurable Register* acts like a normal D-type Flip-flop, but with a configurable initial output value. From the circuit diagram it is clear to see the separation between the configuration chain registers, those connected to cnfDIn and cnfDOut, and the registers that are configuration targets.

Further inspection of Figure 6b shows that the values of target registers can be loaded back into those of the configuration chain. This is useful in a number of ways. It is possible to take a snapshot of the current target register values and read the values back. This is more useful for the Configurable Register elements whose values will reflect the states of the configured circuitry. Another use is to read back configurations for subsequent adjustment. Use of the configuration chain removes previous content, therefore, after adjusting one area of the device, it is possible to load back another area's configuration onto the chain for read back and adjustment. Consequently, it is not always necessary to keep an alternate copy of the complete configuration data.

Partial reconfiguration is achieved by dividing the configuration chains into sub-chain units. Each sub-chain can be individually selected for inclusion into the main configuration chain. Using this system, it is possible to select a single sub-chain, a combination, or the complete device for configuration. A single sub-chain represents either all the configurable logic or all the routing resources in a Cluster.

Figure 6a shows how sub-chain connectivity is controlled. Selection units, labelled *ConfigSelectUnit*, are used to either switch a sub-chain into the main chain, or to cause it to be bypassed. The state of these selection units is controlled by a separate chain. This chain is considerably shorter than the main configuration chain it controls, therefore, the selection of target areas can be set quickly.

In summary, the RISA configuration system uses four serial data chains. These are used in pairs, one pair configures routing connectivity, the other, the logic circuity. One chain of the pair is used to shift the actual configuration data, the other is used to select which parts are to be targeted. This approach provides a mechanism for quick and fine grained partial reconfiguration.

V. Discussion

The RISA architecture offers a new reconfigurable device for investigating bio-inspired systems. The architecture's design offers end-users maximum flexibility in device operation. The configuration system is simple, quick and provides a level of control not found in commercial FPGA devices. The integration of a microcontroller array adds a new dimension to reconfigurable architectures, providing a distributed reconfigurable software element.

A variety of evolutionary design techniques can be undertaken using the RISA architecture. Intrinsic evolution of electronic circuits can be performed using the FPGA fabric for implementing candidate solutions and the microcontroller to drive the evolutionary algorithm. What is more, the evolutionary process can be accelerated by performing multiple evaluations of candidate solutions in parallel, using the SNAP network to transfer fitness information. The hardware random number generator further simplifies the task by removing the need for a time consuming software based number generator.

Performing genetic programming is also possible using the SNAP microcontroller. The Von-Neumann memory architecture allows access to the stored program which provides a means to create self-modifying code. This, coupled with the instruction execution control inherent in the instruction's encoding, provides methods for implementing evolutionary variation operators. Furthermore, the microcontroller array once more provides a platform for accelerating evolution by supporting parallel fitness evaluations.

Further application areas include artificial neural networks [13], where the architecture is ideal for implementing time-multiplexed neuron models. RISA's cellular structure is also

appropriate for implementing Embryonic Arrays [14], [15], the nucleus inspired microcontroller can control cellular specialisation and positioning whereas the FPGA fabric performs the role of cellular function.

The first RISA device has only recently been fabricated, therefore its full potential is still to be established. Further work will uncover new application areas. Further information on the RISA device can be found on the project website: `http://www.bioinspired.com/users/ajg112`. The site includes schematics and VHDL code.

ACKNOWLEDGEMENTS

The authors would like to thank the EPSRC (grant number GR/R74512/01) and the MoD for funding this project and members of Europractice's Microelectronics Support Centre for their expert assistance with ASIC development.

REFERENCES

[1] K. A. Vinger and J. Torresen, "Implementing evolution of FIR-filters efficiently in an FPGA," in *In Proceedings of 2003 NASA/DoD Conference on Evolvable Hardware EH-03*. Washington, DC, USA: IEEE Computer Society, July 2003, pp. 26–29.

[2] C. Lambert, T. Kalganova, and E. Stomeo, "FPGA-based systems for evolvable hardware," in *International Conference on Computer Science, ICCS-06*, March 2006.

[3] A. Upegui and E. Sanchez, "Evolving hardware by dynamically reconfiguring Xilinx FPGAs," in *Proceedings of 6th International Conference on Evolvable Hardware, ICES-2005,*, ser. LNCS, J. M. Moreno, J. Madrenas, and J. Cosp, Eds. Springer Verlag, June 2005, pp. 56–65.

[4] *Two Flows for Partial Reconfiguration: Module Based or Difference Based - XApp 290*, Xilinx, 2004.

[5] *XC6200 Field Programmable Gate Arrays - Data Sheet*, Xilinx, 1997.

[6] L. Sekanina, "Towards evolvable IP cores for FPGAs," in *Proceedings of the 3rd NASA/DoD Conference on Evolvable Hardware, EH-03*. Washington, DC, USA: IEEE Computer Society, 2003, pp. 145–154.

[7] ——, "Virtual reconfigurable circuits for real-world applications of evolvable hardware," in *Proceedings of 5th International Conference on Evolvable Hardware, ICES-03*, ser. LNCS, no. 2606. Springer Verlag, March 2003, pp. 186–197.

[8] N. J. Macias, "The PIG paradigm: The design and use of a massively parallel fine grained self-reconfigurable infinitely scalable architecture," in *Proceedings of the 1st NASA/DOD Conference on Evolvable Hardware, EH-99*. Washington, DC, USA: IEEE Computer Society, 1999, p. 175.

[9] I. Kajitani, T. Hoshino, N. Kajihara, M. Iwata, and T. Higuchi, "An evolvable hardware chip and its application as a multi-function prosthetic hand controller," in *Proceedings of the 16th National Conference on Artificial intelligence and the 11th Innovative Applications of Artificial Intelligence Conference Innovative Applications of Artificial Intelligence, AAAI-99/IAAI-99*. Menlo Park, CA, USA: American Association for Artificial Intelligence, 1999, pp. 182–187.

[10] Y. Thoma and E. Sanchez, "A reconfigurable chip for evolvable hardware," in *Genetic and Evolutionary Computation, GECCO 2004*, ser. LNCS, vol. 3102. Springer Verlag, 2004, pp. 816–827.

[11] B. Shackleford, M. Tanaka, R. Carter, and G. Snider, "FPGA implementation of neighborhood-of-four cellular automata random number generators," in *Proceedings of FPGA 2002, 10th International Symposium on Field Programmable Gate Arrays*. ACM Press, February 2002, pp. 106–112.

[12] G. Marsaglia, "DIEHARD statistical tests." [Online]. Available: http://stat.fsu.edu/~geo/diehard.html

[13] B. Linares-Barranco, A. Andreou, G. Indiveri, and T. Shibata, "Guest editorial - special issue on neural networks hardware implementations," *IEEE Transaction on Neural Networks*, vol. 14, no. 5, pp. 976–979, 2003.

[14] D. Mange, M. Sipper, A. Stauffer, and G. Tempesti, "Towards robust integrated circuits: The embryonics approach," *Proceedings of the IEEE*, vol. 88, no. 4, pp. 516–541, April 2000.

[15] C. Ortega-Sánchez and A. Tyrrell, "A hardware implementation of an embryonic architecture using Virtex® FPGAs," in *Proceedings of ICES 2000, 3rd International Conference on Evolvable Hardware*, ser. LNCS, no. 1801. Springer Verlag, 2000, pp. 155–164.