Go Functional Model for a RISC-V Asynchronous Organisation - ARV

Marcos L. L. Sartori, Ney L. V. Calazans
PUCRS-FACIN - Ipiranga Av., 6681 - Porto Alegre - Brazil, 90619-900
marcos.sartori@acad.pucrs.br, ney.calazans@pucrs.br

***Abstract*— This work presents ARV, an asynchronous super-scalar organisation for the RISC-V architecture. As far as the authors could verify, this is the first proposal of an asynchronous version for this recent open source processor architecture. The organisation is modelled using Google's Go as a high level hardware description language. Go has proved adequate to model the refined handshake structures present in the asynchronous design of complex super-scalar structures. Preliminary performance data obtained using the Go model enabled a detailed evaluation of the organisation, providing design exploration of several points to further improve the organisation before committing to its implementation at lower abstraction levels.**

I. Introduction and Related Work

Processors are complex circuits where active paths are highly dependent on the instruction execution flow. They are overwhelmingly employed in multiple domains, including in low and ultra-low power mobile applications with tight power consumption budgets. These circuits benefit from recent technology advances and also provide a design challenge where novel techniques and flows can be explored.

Asynchronous design has over the years demonstrated advantages over synchronous design in several application domains [1]. These advantages derive from, e.g.: (i) eliminating the clock distribution network and its overheads, which can account for 40-70% of the overall circuit power consumption; (ii) widening the spectrum of electromagnetic emission, reducing noise in mixed signal applications; (iii) executing operations based on average case delay, instead of the worst case delay required by synchronous design. This improves performance and efficiency; (iv) making circuits more robust to ageing, process and operating conditions variations, enabling more aggressive voltage scaling to reduce power consumption. The asynchronous design of processors has a potential to create new, advantageous tools for solving problems. This has been claimed in several efforts in the past, as described e.g. in [2].

Synchronous digital circuits rely on clock signals that provide a discrete common timing reference to ensure correct operation. Asynchronous circuits are digital circuits with no such discrete timing reference. Instead, in an asynchronous design correct operation is accomplished using *local handshakes* between communicating entities [3]. Local handshakes signal when new data is ready and signal back when inputs can change after being processed. A commonly used handshake protocol develops in four steps: (i) a producer announces data availability, by issuing a request to the consumer; (ii) when ready, the consumer acknowledges the request, storing data and replying with an acknowledge signal; (iii) the producer acknowledges the consumer reception by lowering its request signal; (iv) the consumer announces its availability to receive new data by lowering the acknowledgement signal. This is an example of the so-called *4-phase handshake protocol*. Another frequently used choice is the *2-phase protocol*, in which data can be made available after the consumer rises its acknowledge signal [3]. This is usually more complex to implement.

The use of a handshake protocol between processing elements form a *handshake channel*. *Logic elements* intended to transform data must be transparent to the handshake mechanism. The combination of storage and logic elements forms a *logical stage*. Logical stages are chained using channels to form a *pipeline* in which *tokens* flow in *wavefronts* carrying data. A circular pipeline is called a *loop*. Loops can be used to store information and perform iterative computation. Closed loops must have *bubbles* to allow token propagation, otherwise deadlock conditions can arise.

There are different asynchronous design templates used to implement several variations of 4-phase or 2-phase protocols. Templates employing strict channel timing assumptions and explicit request signals are part of the *Bundle Data (BD)* family, while templates using *Delay Insensitive (DI)* encoding to eliminate channel timing assumptions are often part of the *Quasi Delay Insensitive (QDI)* family.

Modelling is an important step in any processor design. This is true in asynchronous design as well. Technologies used to model asynchronous circuits should typically reflect the behaviour of handshake channels.

Since asynchronous circuits are highly concurrent and communication between components relies on handshake channels, languages designed with concurrency and communication in mind provide a better formalism to describe asynchronous circuits than traditional HDL languages like Verilog and VHDL, both of which assume an underlying synchronous hardware paradigm. The basis for several such languages is the *Communicating Sequential Processes* (CSP) formalism proposed by Hoare [4], [5]. Message passing in CSP abstracts the handshake protocol and underlying encoding details, allowing behavioural validation and optimisation of complex asynchronous constructs. This enables the detection and elimination of deadlock conditions early in the design, furthering overall correctness from the design start.

Some previous works proposed flows starting with high-level models written in specialised languages based on CSP to implement asynchronous circuits. Two noteworthy examples are: (i) Balsa [6], an asynchronous circuit description language and synthesis system based on direct syntactical translation targeting both QDI and BD templates; (ii) and Proteus [7], a performance guided asynchronous ASIC design flow targeting QDI templates for high performance applications.

Balsa instantiates asynchronous netlist macros implement-

978-1-5386-1911-7/17/$31.00 ©2017 IEEE

ing high-level functions defined in the language and later optimises the generated netlist. The Proteus flow compiles CAST, a language previously used in a flow proposed by Martin et al. [8], to RTL-level Verilog descriptions. Traditional logic synthesis tools are then used to generate synchronous netlists which are optimised and transformed into asynchronous netlists of a target template.

This article explores the use of Go, a modern programming language proposed by Google, which implements CSP-inspired primitives. Go here is seen as a high-level hardware description language. The choice to implement the RISC-V architecture [9] comes from its simplicity, versatility and development tools availability. Unlike other open instruction set architectures (ISAs), RISC-V was designed from the start to be simple, flexible and extensible, which facilitates its port to different technologies and application domains. As far as the authors could verify, this is both the first proposed asynchronous RISC-V organisation and the first use of Go as a hardware description language. Original contributions are thus two: (i) giving the first results of investigating the suitability of Go to model asynchronous circuits; (ii) and provide a first step towards an asynchronous design for the RISC-V ISA.

II. Asynchronous Circuit Modelling using Go

Go is a new programming language designed by Robert Griesemer, Rob Pike and Ken Thompson at Google [10]. It is a compiled, structured, strongly typed, imperative language focused in concurrency.

Concurrency is built into Go using two primitives: (i) *goroutines*, a form of lightweight threads sharing the program address space managed by the Go runtime environment; and (ii) *channels*, typed conduits in which data can be transferred as messages using send and receive operations. These primitives constitute a valid implementation of CSP, making Go an interesting candidate for the high-level modelling of asynchronous circuits.

Also, the use of Go allows modern software verification techniques, e.g. unit testing, static code analysis, deadlock detection, displaying potential advantages over other languages traditionally employed to model asynchronous circuits.

By default, sending and receiving messages on channels blocks the execution of goroutines. Non-blocking channels are possible, by using buffered sends and conditional reading. However, this is not a feature originally available in CSP .

Figure 1 illustrates a conditional reading of the Set channel. Stage 0 only reads the Set channel when it provides valid data, otherwise it reads s0. Channels that are only read when complete are called *uncoupled channels*. Loops which feature uncoupled channels are called *uncoupled loops*.

Channels provide synchronisation analogous to fully buffered handshake channels; goroutines concurrency is analogous to pipelining parallelism; and messages are analogous to tokens flowing in a pipeline carrying data.

An asynchronous pipeline stage, composed of logic and a handshaking storage element, is modelled in Go using a goroutine containing an eternal loop performing the following steps: (i) receive data from input channels, blocking execution if the channel is empty; (ii) perform data transformation on the input; (iii) send data to the output channels, possibly blocking execution until the next stage receives from the channel. Initial (reset) states are modelled by sending the initial conditions to the corresponding channel prior to entering the receive-process-send loop (see the $s1 <\!\!-0$ command in Stage 0 of Figure 1).

To capture the nature of asynchronous circuits, the goroutine loop should only operate on information received through channels, meaning that goroutines should not rely on internal state variables. State variables inside a goroutine would imply instantiating an implicit loop of handshaking elements, breaking the correlation between goroutines and pipeline stages. It is preferable to model state-holding loops manually, using multiple goroutines and channels as in Figure 1.

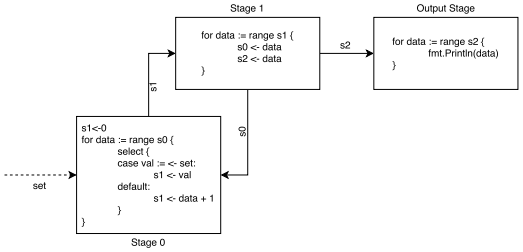

Fig. 1. An asynchronous counter with parallel load, implemented with a closed loop and an uncoupled set channel in Go. It uses conditional reading with the select statement.

III. The ARV processor organisation

The organisation presented in Figure 2 was designed using the modelling technique described in Section II. It is an asynchronous super-scalar pipeline implementing the *RISC-V 32-bit Integer Architecture*, also known as RV32I. The pipeline employs speculative execution and a register locking scheme to deal with branches and hazards.

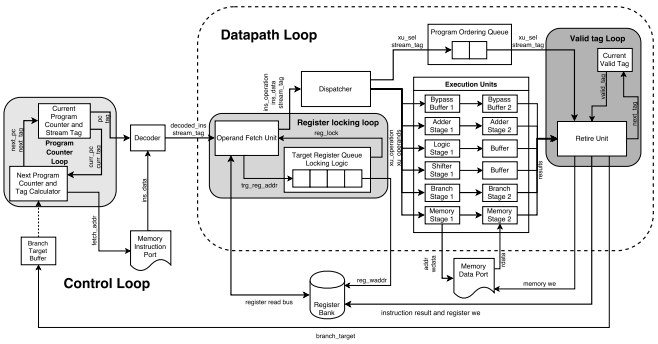

Fig. 2. Block diagram representation of the ARV execution pipeline organisation. Arrows represent handshake channels and blocks represent pipeline stages.

ARV is composed by communicating elements organised in two nested loops, forming the *Execution Pipeline*: (i) the outermost loop is an uncoupled *Control Loop* (CL), responsible for fetching, decoding and terminating instructions; and (ii) the *Datapath Loop* (DL), which retrieves, manipulates and stores data.

Additionally, three other auxiliary loops are present: (i) the *Program Counter Loop* is responsible for keeping track of the

Program Counter (PC) and *stream tag*; (ii) the *Valid Tag Loop* is responsible for holding the *valid tag* identifying the current valid instruction flow; (iii) and the *Register Locking Loop* (RLL), responsible for identifying registers being modified by instructions currently under execution.

As usual, the PC is used for fetching new instructions and as an operand in some instructions, while the valid tag is used by the Retire Unit to identify and cancel instructions that have been invalidated by branches or exceptions.

The basic flow of instructions in the pipeline is a follows: (i) the *fetch address* feeds the *Memory Instruction Port*; (ii) the fetched *opcode*, PC and *stream tag* values associated with the instruction feed the *Decoder Unit*, which identifies the *operands* and the *operation*; (iii) the *Operand Fetch Unit* is fed. It reads the operands from registers, possibly holding the instruction execution to avoid hazards. Once ready, the instruction target register address feeds the Target Register Unit, which locks the target register for reading, while the operation feeds the Dispatcher Unit; (iv) the *Dispatcher Unit* records the instruction in the *Program Ordering Queue* and sends the instruction to the appropriate *Execution Unit*; (v) the selected Execution Unit is where the instruction is in fact executed and results wait for collection; (vi) the *Retire Unit* reorders instructions as defined by the *Program Ordering Queue* and verifies conditions for the achievement of instructions by asserting their *validity tag*; (vii) the instruction finishes execution as the *Register Bank* stores the result from the Retire Unit to the address read from the Target Register Queue, unlocking the target register for reading; (viii) optionally, branches are taken, updating the PC and the Tag.

The Control Loop (CL) is the system outermost loop. It is responsible for controlling the execution flow of programs. Its entry point is the Program Counter Loop and its end point is the Valid Tag Loop. It shares a path with the Datapath Loop (DL) from the Operand Fetch Unit to the Retire Unit.

Every instruction is associated with a PC and Stream Tag value. The PC is used as address to fetch instructions from memory and as an operand in some instructions. The Stream Tag identifies instructions that must be cancelled due to branching and/or exceptions. It does so by counting the number of times the program counter is set at both ends, sending the value kept at the Program Counter Loop along the instruction in the pipeline. Next it compares the tag received with the instruction to the tag kept in the Valid Tag Loop. If the Stream Tag received from the Program Ordering Queue by the Retire Unit does not match the updated value kept by the Valid Tag Loop, any program counter updates, memory or register writes performed by the instruction are not issued, effectively cancelling the instruction.

The Retire Unit increments the valid tag and issues the *branch target* to the *Next Program Counter and Tag Calculator* (NPC) through a buffered uncoupled channel when a branch or exception occurs. When the NPC receives the branch target, it increments the Stream Tag and sets the PC accordingly.

The branch target channel is uncoupled, since NPC checks the validity of the input and acts accordingly, instead of blocking while it waits for a valid input. As the path closing the loop contains an uncoupled channel, the loop itself is called an uncoupled loop. A buffer is used in the branch target channel to avoid deadlocks created by race conditions in the uncoupled input, possibly eliminating the available bubble that allows token mobility in the loop.

The branch target uncoupled channels close the CL, making it an uncoupled loop. This is important due to how the DL deals with pipeline hazards, which occur because the DL stalls decoding and fetching of instructions while it inserts bubble pseudo-instructions. This fact increases the number of tokens and saturates the CL.

The *Decoder Unit* is responsible for three tasks: (i) determining the instruction format, used by the Operand Fetch Unit to identify instruction operands; (ii) decoding the three register fields to a one-hot register address used by the Execution Pipeline; (iii) identifying the instruction operation the Execution Unit performs.

The Datapath Loop (DL) is a closed loop with six logical stages: (i) the *Operand Fetch Unit*; (ii) the *Dispatcher*; (iii) first and (iv) second stages of the *Execution Unit* and *Program Ordering Queue*; (v) the *Retire Unit*; (vi) the *Register Bank*.

The DL holds a constant amount of 5 tokens. New instructions are admitted in the loop as old instructions are retired.

The *Operand Fetch Unit* is the entry point of instructions into the DL. It is responsible for retrieving data from the Register Access Controller and completing immediate operands. It also stalls the pipeline in case of data hazards.

Data hazards are avoided by locking registers that are waiting for data to be written. If an instruction attempts to read a locked register, the Operand Fetch Unit stalls the decoding of new instructions and inserts bubbles in the pipeline.

The Register Locking Loop (RLL), detailed in Figure 3 tracks the currently locked registers. It is composed by a 4-stage *Target Register Queue*, an OR logical stage and the Operand Fetch Unit. The DL and RLL are closed loops of the same length, running in parallel. Every token in the RLL corresponds to a token in the DL.

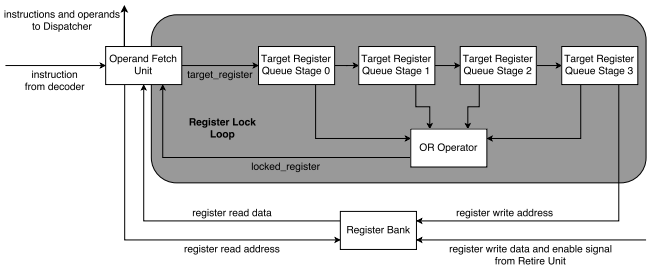

Fig. 3. Block diagram detailing the Register Locking Loop and the Operand Fetch mechanism.

After retrieving all operands, the instruction has all the information to complete execution. It then follows to the dispatcher unit, where it is steered to the execution unit responsible for accomplishing it.

The pipeline uses a fan-out distribution characteristic of asynchronous circuits. Instructions are only sent to units involved in their execution. An interesting aspect of this is the potential for energy saving, since units not operating on an instruction need not produce switching activity.

The fan-out unit selection scheme introduces problems when there is a discrepancy in the individual delays of distinct

TABLE I
PERFORMANCE ESTIMATES EXTRACTED FROM SIMULATION

	riscv-test	test-hanoi
Bubble insertion loss	94.4%	120%
Branch misprediction loss	48.8%	15%
Bypass Unit Utilisation	50%	59.8%
Adder Unit Utilisation	18%	26.4%
Logic Unit Utilisation	1.6%	0.6%
Shifter Unit Utilisation	4.1%	0.9%
Branch Unit Utilisation	13%	8.8%
Memory Unit Utilisation	12.9%	3.3%

units. This may cause instructions to execute out of order. As a consequence, instruction reordering techniques are required to guarantee correct execution.

The *Program Ordering Queue* helps to perform the reordering of instructions. When the Dispatcher issues an instruction to one of the execution units, it registers which unit was selected along with the stream tag in the Program Ordering Queue. The Retire Unit reorders instructions by collecting results from the Execution Units in the order determined by the Program Ordering Queue.

Since an instruction reordering mechanism is introduced, it becomes simple to improve throughput by implementing parallel execution. To achieve a theoretical best case of 2-instruction parallelism, the Program Ordering Queue is two-stage deep. Besides, to accommodate the scenario of two instructions being dispatched to a same Execution Unit in sequence, units are each two-stage long.

IV. VALIDATION AND EXPERIMENTS

The model was validated to correctly implement the RISC-V ISA by running two bare-metal applications: (i) a port of the official RISC-V RV32I compliance test (riscv-test); and (ii) a simple C application implementing a non-recursive solution for the tower of Hanoi problem (test-hanoi). The RV32I compliance test is a set of assembly routines designed by the RISCV-V Foundation to test the correct implementation of the RISC-V ISA. It does so by testing each instruction at corner cases and printing on the screen the result of each test. Tower of Hanoi is a simple C program employing function calls, loops and memory access. It was compiled using GCC with optimisation flags tuned to our processor implementation. The idea of this test was to stress the pipeline with a regular program workload, checking for unpredicted hardware bugs and providing real world performance estimations.

To allow software execution, the external memory was modelled using a Go byte vector mapped to an image file containing the program. The memory vector was encapsulated using goroutines that provide the asynchronous interface expected by the processor model. Two virtual registers were included in the processor memory address space to print results and terminate execution. Pipeline performance estimations were extracted using counters in key pipeline stages. From this data it is possible to establish losses due to branch misprediction, bubble insertion and execution unit utilisation.

Each test was repeated 100 times and a simple arithmetic mean was used to account for the non-deterministic nature of parallel execution. The results are presented in Table I.

V. CONCLUSIONS AND FUTURE WORK

The ARV processor model took three months to go from conception to validation. Results demonstrate that the model correctly implements the RISC-V RV32I architecture.

Go proved a language adequate to model complex handshake channel-based circuits. The Authors thus believe justified its use for design validation as a hardware description language. The source code, tools and experiments described here and other material (such as the first Author's end-of-term dissertation) are available openly in the Github development platform (see https://github.com/marlls1989/arv). Ongoing work comprises developing tools to automatically transform the high level Go description into lower level netlists using asynchronous templates.

Preliminary performance results obtained from the model indicate there is room to optimise it, even though no precise timing measurements can in fact be extracted at this point. The timing analysis required for performance evaluation depends on the proposal of a more detailed model, with features that allow estimating signal propagation delays. Such finer-grain models are the target of ongoing work. One of the points to address for optimising the model are in reducing branch mispredictions. Ongoing work addresses the development of more sophisticated branching prediction mechanisms.

The Authors suggest that the cost of bubbles in asynchronous templates are inferior to the cost of additional hardware to avoid hazards, ongoing work is evaluating this hypothesis. Improvements to the register locking and operand fetching mechanisms can further reduce the cost of bubbles.

The Authors are also currently exploring the necessary steps to implement the proposed organisation in real silicon using asynchronous design techniques.

REFERENCES

[1] S. M. Nowick and M. Singh, "Asynchronous Design – Part I: Overview and Recent Advances," *IEEE Design and Test*, vol. 32, no. 3, pp. 5–18, 2015.

[2] K. Slimani, J. Fragoso, M. Es Sahliene, L. Fesquet, and M. Renaudin, *Low-Power Asynchronous Processors*. Boca Raton, FL: Taylor and Francis, 2006, ch. 5, pp. 5.1–5.20.

[3] J. Sparsø and S. Furber, *Principles of Asynchronous Circuit Design – A Systems Perspective*. Springer, 2001.

[4] C. A. R. Hoare, *Communicating Sequential Processes*. Prentice Hall International, 1985. [Online]. Available: http://www.usingcsp.com/cspbook.pdf

[5] ——, "Communicating Sequential Processes," *Communications of the ACM*, vol. 21, no. 8, pp. 666–677, Aug. 1978.

[6] A. Bardsley, "Balsa: An asynchronous circuit synthesis system," Master's thesis, University of Manchester, 1998. [Online]. Available: http://apt.cs.manchester.ac.uk/ftp/pub/apt/theses/bardsley_mphil.pdf

[7] P. A. Beerel, G. D. Dimou, and A. M. Lines, "Proteus: An asic flow for ghz asynchronous designs," *IEEE Design Test of Computers*, vol. 28, no. 5, pp. 36–51, Sept 2011.

[8] A. J. Martin and M. Nyström, "CAST: Caltech Asynchronous Synthesis Tools," California Institute of Technology (CALTECH), Tech. Rep., June 2004. [Online]. Available: http://www.async.caltech.edu/Pubs/PS/2004_turkuacid_CAST.ps

[9] A. Waterman, Y. Lee, D. A. Patterson, and K. Asanović, "The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Version 2.1," University of California, Berkeley, Tech. Rep. UCB/EECS-2016-118, May 2016. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2016/EECS-2016-118.html

[10] Google Inc., "The Go Programming Language," 2012. [Online]. Available: https://golang.org