A Synthesisable Quasi-Delay Insensitive Result Forwarding Unit for an Asynchronous Processor

Luis A. Tarazona, Doug A. Edwards and Luis A. Plana
Advanced Processor Technologies Group, School of Computer Science
The University of Manchester, Manchester, M13 9PL, United Kingdom
Email: {tarazonl,doug,plana}@cs.man.ac.uk

***Abstract*—The implementation of an efficient result forwarding unit for asynchronous processors faces the problem of the inherent lack of synchronisation between result producer and consumer units. An efficient, full-custom solution to this problem has been proposed and implemented before (in the AMULET3 asynchronous processor) with the consequent limitations on design-space exploration and technology portability. The use of automatic synthesis to describe asynchronous systems is attractive in terms of rapid development, technology mapping transparency and design space exploration. This paper presents the description of a synthesisable result forwarding unit for an asynchronous microprocessor, using the syntax-directed synthesis approach and targeting a robust quasi-delay-insensitive implementation. The description of such a system also serves as a complex case study to evaluate the capabilities and limitations of syntax-directed synthesis when used as a tool to automate the synthesis of performance-demanding asynchronous systems.**

I. Introduction

Result forwarding [1] is a method used in pipelined microprocessors to reduce the penalty caused by inter-instruction data dependencies. The forwarding mechanism can also be used to allow partial overtaking of (normally slow) memory operations, but making sure that the instructions complete in the same order as they appear in the program. Figure 1 depicts some potential performance benefits of the result forwarding mechanism.

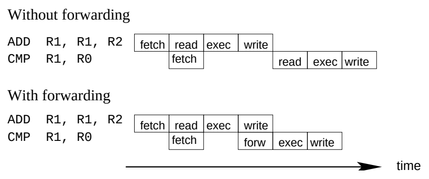


Figure 1: Potential performance benefits of result forwarding in a 4-stage pipeline.

In synchronous systems, the problem of result forwarding can be easily solved because the clock signal serves as a reference that allows synchronisation between result producing and consuming units. In an asynchronous environment, the problem of implementing a result forwarding mechanism is more complicated due to the lack of synchronisation between producers and consumers. In this case, one cannot rely on a control signal that indicates which cycle an instruction is in. This would require a lockstep operation of the pipeline that would heavily penalise the performance. An asynchronous implementation is attractive because asynchronous systems have some advantages over their synchronous counterparts such as: no clock distribution or clock skew problems, better composability, lower power consumption, lower electromagnetic interference and robustness towards variations in supply voltage, temperature and fabrication process parameters [2]. An efficient, full-custom solution to the problem of result forwarding within an asynchronous environment has been proposed and implemented before in the AMULET3 asynchronous processor [3], targeting a bundled-data implementation (see section II), with the consequent limitations on design-space exploration, technology portability and similar timing closure problem as synchronous designs. In order to overcome such limitations and reduce the impact of increasingly difficult timing closure within modern fabrication processes variability, it is desirable to have a synthesisable asynchronous description, which can be mapped into a quasi-delay-insensitive implementation (QDI - presented in section II). This paper presents the description of a synthesisable result forwarding unit for an asynchronous microprocessor, using the syntax-directed synthesis approach and targeting a robust QDI implementation. The description of such a system also serves as a complex case study to evaluate the capabilities and limitations of syntax-directed synthesis when used to automate the synthesis of performance-demanding asynchronous systems.

This paper is organised as follows: Section II presents an overview of asynchronous design and introduces the motivation towards its use in modern VLSI designs. Section III introduces the syntax-directed syntax approach and the synthesis system used. Section IV briefly introduces the target processor. Section V presents related work on result forwarding. Section VI presents the architecture of the proposed forwarding unit and discusses the challenges faced when such architecture is mapped into a QDI implementation and the proposed solutions. Simulation results and discussion are given in Section VII. Conclusion and future work are presented in Section VIII.

II. Overview of Asynchronous Design

Synchronous digital systems, which are the basis of most of today's digital designs, are based on two major assumptions: all signals are binary and time is discrete, defined by the system's clock signal which controls all communication and event sequencing. These assumptions reduce greatly the task of design but also lead to clock distribution and clock skew problems, increased power consumption and electromagnetic emissions (EMI) and forcing all parts of the circuit to work at the same (worst-case) rate. As opposed to synchronous, asynchronous systems does not rely on a global clock signal. Instead, these systems use a form of local communication that comprises handshake signals to request (initiate) and acknowledge (indicate) the reception of a request and that the operation can proceed. Asynchronous circuits have some advantages over their synchronous counterparts that make them attractive to use in large VLSI designs, including: no clock distribution or clock skew problems, better modularity and composability, lower EMI, lower power, average-case performance and robustness towards variations in supply voltage, temperature and fabrication process parameters [2].

A. Handshake protocols and data encoding

As introduced in the previous section, asynchronous circuits communicate using request (*req*) and acknowledge (*ack*) handshake signals. These signals together with the data signals form a *handshake channel* between two units. The unit that requests an operation is called the *active* party (or *active port*) and the unit that responds is referred to as *passive*. If the sender of data is the active party the channel is called a *push channel*. If it is the receiver who initiates the communication, the channel is called a *pull channel*. In abstract diagrams, it is common to identify the active end of a channel using a black dot. There are several common asynchronous handshake protocols named according to the encoding used for data and handshake signals. Here we briefly present the most common of them.

1) Four-phase bundled-data protocol: In this protocol the data signals use binary levels to encode information (one bit per wire) and there are separate wires for *req* and *ack* signals. These are bundled with the data wires to form the channel, as shown in figure 2(a). This protocol is also called *single-rail*. The term four-phase refers to the number of actions that take place during a handshake communication, which are: (1) the active party issues the data and initiates the handshake by setting the *req* signal high. (2) The passive party reads the data and sets *ack* high. (3) Upon receiving the acknowledge, the active party returns to zero (RTZ) its *req* signal, after this, data is not longer guaranteed to be valid. (4) The receiver detects the return to zero of *req* and acknowledges this by taking *ack* low (RTZ), allowing a new handshake to start. Figure 2(b) shows a timing diagram for this protocol.

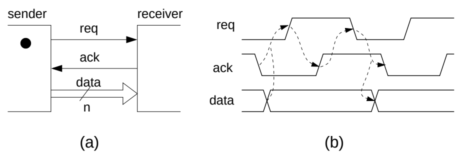
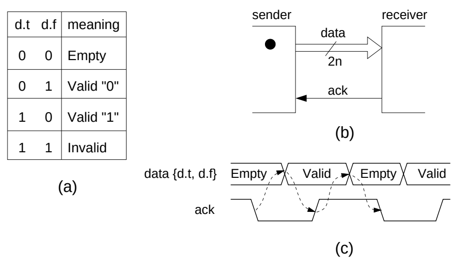


Figure 2: (a) Bundled-data push channel. (b) Four-phase bundled-data protocol.

2) Two-phase Bundled data protocols: In this protocol each transition on the *req* and *ack* signals correspond to an event in the channel. In the first phase the active party initiates the handshake with a transition in *req*. In the second phase the other party terminates the handshake by changing the value of the *ack* signal. After that, the active channel can transition its *req* signal again, initiating a new handshake. For this reason, two-phase protocols are also known as *non-return-to-zero* (NRZ) protocols.

3) Four-phase dual-rail protocol: In this protocol the request signal and data are encoded together using two wires per bit of information. Each data bit d requires two signals: $d.t$ for signalling a logic 1 (*true*) and $d.f$ for signalling a logic 0 (*false*). In this way, the pair of wires $\{d.t, d.f\}$ form a code whose codewords are shown in figure 3(a). This encoding scheme allows the easy extension to an n-bit channel by concatenating n bits coded in dual-rail as above. Figure 3(b) shows an n-bit dual-rail channel. Using this convention, the four phases are: (1) The sender issues a valid codeword on each pair of wires. (2) The receiver identifies when *all* bits have become valid (completion detection), reads the data and takes *ack* high. (3) The sender detects the acknowledgement and changes the bits to the empty state (RTZ). (4) The receiver identifies when all the bits have become empty and takes *ack* low (RTZ completion detection). Figure 3(c) shows a timing diagram for this protocol. The dual-rail coding is a member of the family of delay-insensitive codes [4]. This encoding method allows a reliable communication between two parties regardless of the delay in the wires. This property makes dual-rail encoding very attractive despite the fact of using more wires.



d.t	d.f	meaning
0	0	Empty
0	1	Valid "0"
1	0	Valid "1"
1	1	Invalid

Figure 3: (a) 1-bit Dual-rail encoding. (b) n-bit Dual-rail channel. (c) Four-phase dual-rail protocol.

4) Two-phase dual-rail protocol: This protocol also uses two wires per bit but the information is encoded as transitions instead of logic levels. On an n-bit channel, a new codeword is received when exactly one wire per bit has made a transition. In this case there is no empty value: a valid codeword is acknowledged and the sender can change again one wire per bit to send another codeword.

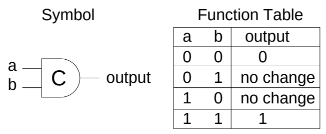
B. *Delay models*

In order to design and implement asynchronous circuits some delay assumptions and timing constraints are used, generating a number of delays models. Delays assumptions allow simplifications to the modelling of the systems and timing constraints specify the restrictions the circuit is subject to in order to operate correctly. Here we will discuss bundled-data and delay-insensitive models.

1) Bundled-data circuits: As introduced in section II-A1 bundled-data uses binary levels to encode information and there are separate *req* and *ack* signals. All protocols using bundled-data rely on delay matching to preserve the order of events at the sender's and receiver's end. For instance, in the push channel of figure 2 (a), valid data must precede the *req* signal in order to guarantee correct operation.

2) Delay-insensitive (DI) circuits: In this model all wires and circuit elements can have positive, unbounded delay. With this assumption, an element that receives an input signal is forced to *indicate* (acknowledge) to the sender when it has received the information. No new changes can occur at the input before receiving the acknowledge signal. In DI circuits, completion detection circuitry is used to generate the acknowledge signal. In a DI system, communication between different modules is made using a DI protocol such as the dual-rail protocol described in section II-A3. The DI model is a very robust model, however, it has limitations if applied to general circuit design due to its heavy restrictions. The only n-input, single-output gate that can be safely used in DI circuits is called the *Muller C-element* [5]. Due to this restriction, the class of delay-insensitive circuits is very limited. It has been demonstrated that only circuits composed of C-elements and inverters can be delay insensitive [6]. Figure 4 shows the symbol and the specification for a two-input C-element.

3) Quasi-delay insensitive (QDI) circuits: This model uses the DI assumptions with the addition of *isochronic forks*. Isochronic forks are forking wires where the difference in delays between the destinations is negligible. This allows a signal to go to different places and be safely acknowledged at only one of the ends, simplifying the design of the circuits.



a	b	output
0	0	0
0	1	no change
1	0	no change
1	1	1

Figure 4: The Muller C-Element.

III. Syntax-directed synthesis

The syntax-directed approach to synthesise asynchronous circuits is based in the compilation of descriptions written in a high-level language into a communicating network of pre-designed modules called handshake components [7]. The compilation process performs a one-to-one mapping of each language construct into the network of components that implements it. This transparent mapping gives a high degree of flexibility in the design as incremental changes to the specification generates predictable changes in the resulting circuit, allowing the designer to optimise the circuit in terms of performance, power or area, at the description language level. The compiled network of handshake components constitutes an intermediate representation that can be subsequently replaced by a gate netlist.

Currently there exist two fully automated CAD systems that use this approach for the synthesis of asynchronous systems: *Haste* (formerly called *Tangram*) [8] and *Balsa* [9], an open-source system developed at the University of Manchester that closely follows the Tangram philosophy. Syntax-directed synthesis has been used successfully in the synthesis of several VLSI systems, including the SPA processor [10], an asynchronous MIPS microprocessor [11] and the ARM996HS, the first commercially-available synthesisable asynchonous ARM.

A. *The Balsa synthesis system*

Balsa is the name for both the framework for synthesising asynchronous circuits and the language used to describe such systems. Balsa uses the syntax-directed synthesis approach to generate *handshake circuits* from a description written in the Balsa language. Originally introduce by van Berkel [7], a handshake circuit is a communicating network of handshake components connected point-to-point using *handshake channels*. Each channel connects exactly one passive port of a handshake component to an active port of another handshake component. As an example, consider the following Balsa piece of code, which describes a simple 1-place buffer (register):

```
procedure buffer (
  parameter DataType : type;
  input  in  : DataType;
  output out : DataType
) is
    variable buf : DataType
begin
    loop
        in -> buf ;
        out <- buf
    end
end
```

The specification is parameterised in the type of data the register can hold. The register has an input channel `in` and an output channel `out`. The variable `buf` stores the data and the operation consists of an unbounded repetition (`loop`) of two actions: input data (`->`) from channel `in` into `buf` sequenced (`;`) with output (`<-`) of the data stored in `buf` to channel `out`. Figure 5 shows the handshake circuit generated by Balsa from the code above, where the *Loop* component is labelled with a star ($*$).

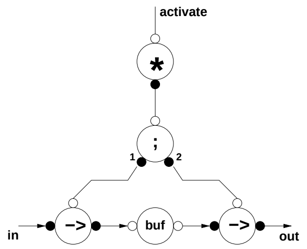


Figure 5: 1-Place buffer handshake circuit.

1) Sequencing events in asynchronous circuits: In synchronous circuits, the sequencing of events can be easily controlled using the global clock. In an asynchronous environment, sequencing handshake events must follow the protocol rules in order to avoid data or control hazards that may cause malfunction and deadlock. Balsa generates two types of sequencers, based on the S-element and T-element respectively [12]. Figure 6 presents a block diagram of such components with their respective specifications as Signal Transition Graphs (STG - a class of Petri net used to specify the operation of a circuit [2]). In an STG, a transition from 0 to 1 in signal x is represented by $x+$. Similarly, a transition from 1 to 0 in x is represented by $x-$. Note how in the S-element the RTZ phases of the left (I) and right side (O) do not overlap, whereas in the the T-element they can occur cuncurrently, which increases the speed of the sequencing. Unfortunately, it is not always possible to use this type of overlapping due to the possibility of introducing write-after-write (WAW) and write-after-read (WAR) hazards. For a complete discussion of these issues, the interested reader can refer to [12].

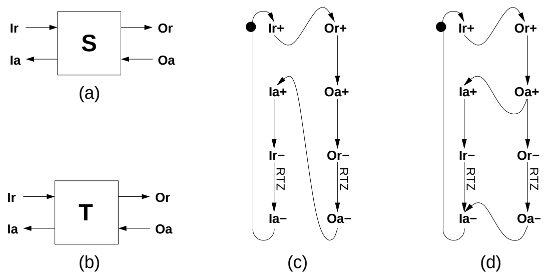


Figure 6: (a) S-element. (b) T-element. (c) S-element STG. (d) T-element STG.

IV. The target processor: nanoSpa

The forwarding unit described in this paper was designed to be used in the nanoSpa processor [13]. NanoSpa is an experimental, new specification of the SPA processor [10], a fully synthesised asynchronous implementation of the 32-bit ARM v5T ISA. As opposed to SPA, whose description focused on security, nanoSpa description uses highly optimised Balsa code targeting high performance. NanoSpa is currently under development and the initial version shares the same architecture organisation as SPA: an ARM-style 3-stage Fetch-Decode-Execute pipeline with a Harvard-style memory interface. To date, nanoSpa has the following functional differences with respect to SPA: it does not have support for Thumb instructions, interrupts, memory aborts or coprocessors. In a new description of nanoSpa, the pipeline depth has been increased to enhance the performance, but in order to avoid the performance loss caused by inter-instruction dependencies, this new description requires the use of a result forwarding unit. This non-trivial problem presents interesting challenges for the architecture, description techniques, language expressiveness and performance of the synthesised circuits. Figure 7 shows a simplified diagram of the nanoSpa pipeline.

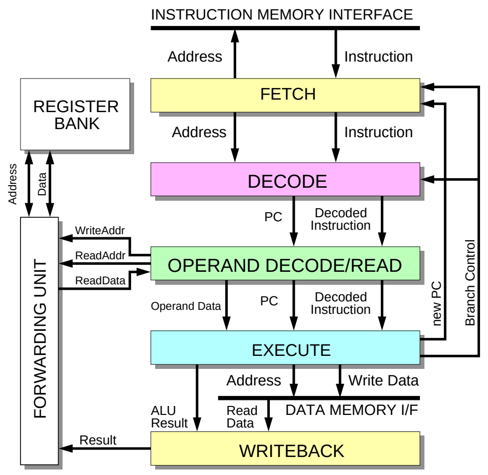


Figure 7: The 5-stage nanoSpa pipeline.

V. Related work

Earlier asynchronous techniques for resolving dependencies include: the register locking mechanism for the AMULET 1 processor [14], register locking plus "last result" register used in the AMULET2 processor [14], the last result bypass mechanism of the Caltech asynchronous MIPS [15], the scoreboard-like Data Hazard Detection Table (DHDT) of the SAMIPS processor [11], the counterflow pipeline architecture [16] and the asynchronous "queue" FIFO [3] for the AMULET3 processor [17]. The "queue" was an efficient solution to the problems of result forwarding and exception handling within an asynchronous pipeline, with the disadvantage of being a full-custom design implemented using matched-delay-based, bundled-data encoding, which limits the possibility of design-space exploration and technology portability. The ARM996HS processor by

Handshake Solutions, is a commercially-available synthesizable asynchronous 32-bit CPU that was implemented using the Haste tools [8]. The processor core is a five-stage asynchronous pipeline but no information has been published about the dependency avoidance technique used. Similar to AMULET3, the implementation uses bundled-data encoding.

As nanoSpa is also an ARM core, the AMULET3 asynchronous "queue" FIFO (AQF from herein) was used as the reference model for the nanoSpa forwarding unit (nFU). The AQF is a circular buffer that acts both as a forwarding unit and a reorder buffer. Figure 8 shows a diagram of the AQF process model. The queue operation consist of 5 processes: *Lookup*, *Allocation*, *Forward*, *Arrival* and *Writeout*.

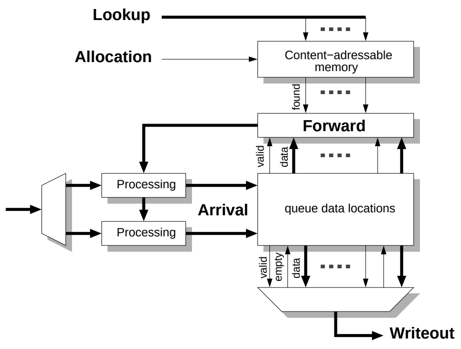


Figure 8: AQF process model.

Lookup: This process receives the source register names from the decoder, examines the queue to see if they are present, and returns a bit mask indicating the possible data source positions in the queue. This is performed using a CAM (Content Addressable Memory) that holds the previously allocated destination registers.

Allocation: After obtainning the lookup source mask, the instruction's own destination address can be written into the CAM. The writing position is allocated cyclically within the circular buffer structure.

Forward: Concurrently with *allocation*, this process receives the mask generated during *lookup*, examines each of the possible sources (starting at the most recent), waits until the data are present and then checks for validity. Valid data is forwarded, otherwise the process examines the next most recent possibility. If all the possibilities are exhausted (or if there were no data sources) the forwarding process gives up and the default value read from the register bank is used.

Arrival: Results arriving at the queue come along with the allocated queue address. The allocation process guarantees non-conflicting allocations even in the event of multiple writes. When the data arrive, the previous data is known to have been both written back to the register bank and forwarded as required. If the instruction was abandoned due to conditional execution, then the result will be marked as invalid.

Writeout: This process copies valid results back to the register bank. It examines the queue locations cyclically and waits until the valid result arrives, then copies the data to the register bank and mark the location as "empty" so it can be reallocated.

In order to improve the speed of the lookup process, the AMULET AQF uses a small CAM to hold the information about the registers written in the buffer. Speculative read of the default value from the register bank is also performed in case the source operand is not present in the buffer. The AQF has a centralised control and features three read ports for forwarding and two write ports for arrival. As mentioned earlier, the AMULET3 and the AQF were implemented using bundled-data encoding together with a token-passing asynchronous control.

VI. Architecture of the nanoForward Unit

The nFU has been designed around the process model of the AMULET3 AQF and it has the same number of read ports (3) and write ports (2), but as nanoSpa does not execute instructions out of order, it is not used as a reorder buffer. Figure 9 shows the architecture of the nFU and its location within the new nanoSpa pipeline. The figure shows details of the communication interface between the various processes, the queue and the processor units.

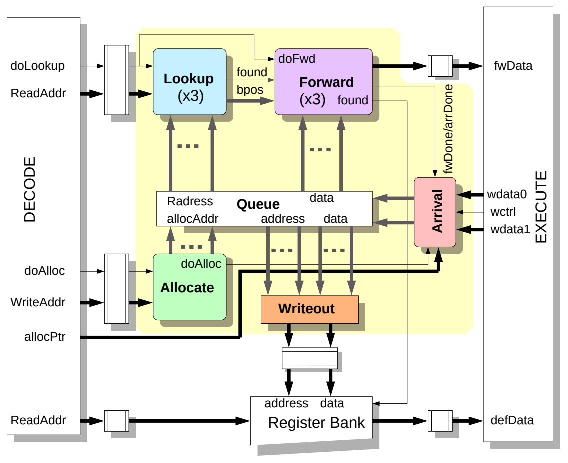


Figure 9: The nanoForward Unit architecture

A. nanoFU implementation

The nFU implementation targeted a delay-insensitive dual-rail encoding implementation as this is a robust data encoding scheme that facilitates timing closure, which is advantageous in current fabrication technologies affected by process variability. This advantage comes at the cost of higher area, energy consumption and, in some cases, restricted solution alternatives. Similar to nanoSpa, the style used for the Balsa description was data-driven, with performance as the main goal.

B. Implementation issues

ARM instructions are particular in the sense that any instruction can be executed conditionally, which adds extra complication to the result forwarding mechanism. In order to improve the efficiency of the pipeline in both the AQF and the nFU, allocation is done regardless of the instruction being or not conditional. If a conditional instruction fails its condition code tests, a token is sent through the pipeline to indicate that the instruction has been processed and the allocated queue slots are marked as invalid. This introduces some wastage in the queue but figures reported in [3] give 90% of queue utilisation for typical ARM programs.

1) Synchronisation between processes: To guarantee correct operation, on each instruction the nFU must perform sequentially some operations as shown in figure 10. To allow synchronisation among handshake modules, the Balsa language provides special non-data channels called *sync* channels. The initial description used *sync* channels to synchronise the processes but this caused a large performance penalty, so alternatives were looked for. A solution that reduced dramatically this penalty was to perform synchronisation using data instead of sync tokens: to decouple *forward* from *arrival*, the queue contents are read speculatively and sent through data channels to the *forward* process. *Lookup* and *allocation* were decoupled using an "allocation mask" that blocks the reading of the queue locations that are about to be modified by the allocation/arrival process during the current instruction. This masking mechanism reduces the effective length of the queue in 1 or 2 locations, depending on the number of results to be written (one or two). These mechanisms obviously dissipate more power and require larger area. Another alternative is to implement a less concurrent operation by grouping the processes according to the information that they read or write: *lookup*/*allocate* are sequenced as they read/write the register names and the *valid* flag. Similarly, *forward/arrival* are sequenced because they read/write results. In this way, *lookup/allocation* can now run concurrently with *forward/arrival*. Synchronisation between *lookup* and *forward* is done with data tokens carrying the lookup result. *Allocation* and *arrival* completion must be synchronised and this information triggers the *writeout* process. This grouping and sequencing prevents an instruction from overtaking the previous one but reduces the area and energy penalty.

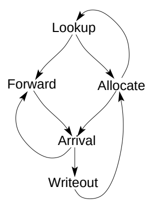
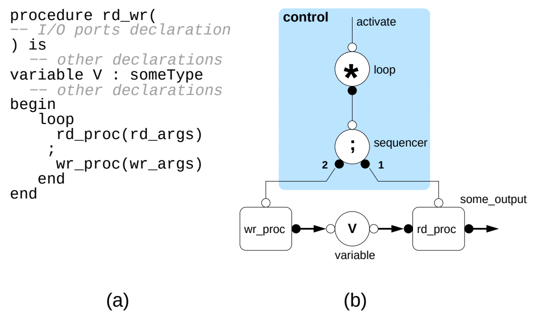


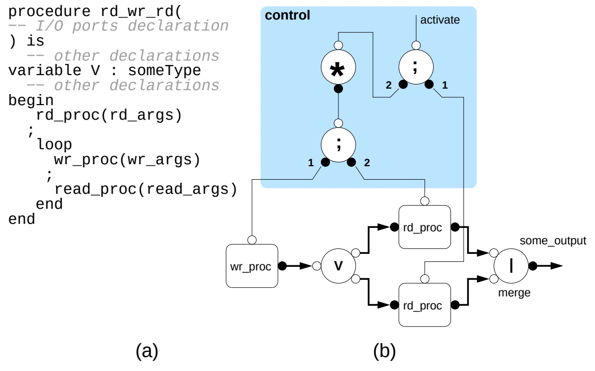
Figure 10: Inter-process dependencies in the nFU.

2) Performance of sequenced operations: One performance problem that arises with the grouping scheme presented in section VI-B1 is that those read-then-write operations require the use of sequencers based on the *S-element* which fully sequences processing and return-to-zero (RTZ) phases in order to avoid the risk of write-after-read (WAR) hazards. To allow a more concurrent operation with decoupled RTZ phases, the processes can be rearranged as *allocation/lookup* and *arrival/forward*. This write-then-read operation permits the safe use of a sequencer based on the *T-element* but requires requires an initial empty token to be sent to *allocate* and *arrival* before the nFU begins to process instructions. In Balsa, a write-then-read sequence to a variable within a procedure generates a sequencer based in the *T-element* but in the nFU the write and read processes reside in separate modules with multiplexed/demultiplexed accesses to a global variable. In this situation, the Balsa compiler is conservative and inserts a safe *S-element*. It is clear that in such cases, the designer's knowledge about the system behaviour should ideally be expressible at the language level so that the compiler can make better decisions. In our case, the S-elements for the mentioned sequencers were manually changed to T-elements in the intermediate handshake netlist (before technology mapping).



```
procedure rd_wr(
-- I/O ports declaration
) is
  -- other declarations
variable V : someType
  -- other declarations
begin
   loop
     rd_proc(rd_args)
    ;
     wr_proc(wr_args)
   end
end
```

Figure 11: Read-write loop (a) Code. (b) Handshake circuit.



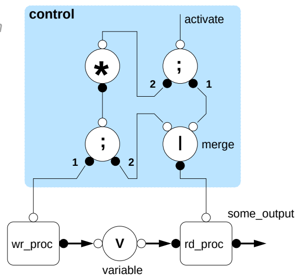
```
procedure rd_wr_rd(
-- I/O ports declaration
) is
  -- other declarations
variable V : someType
  -- other declarations
begin
   rd_proc(rd_args)
  ;
   loop
     wr_proc(wr_args)
    ;
     read_proc(read_args)
   end
end
```

Figure 12: First-read-unfolded version of circuit in figure 11.

An improvement to the above solution is to take advantage of the unbounded repetition of read-then-write actions over

```
procedure rd_wr(
-- I/O ports declaration
) is
  -- other declarations
variable V : someType
  -- other declarations
begin
    loop
      rd_proc(rd_args)
     ;
      wr_proc(wr_args)
    end
end
```

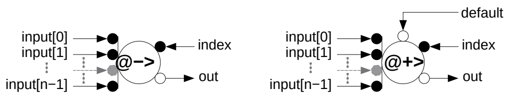
(a)



(b)

Figure 13: Optimised First-read-unfolded read-write loop.

common variables and let the compiler automatically generate an optimised control tree, without the need for reordering the operations. An unbounded repetition of sequenced read-write operations can be described in Balsa as shown in the piece of code in figure 11(a) where `rd_proc()` and `wr_proc()` access the common variable V. Figure 11(b) shows the resulting handshake circuit. If we unfold the first read operation off the loop construct as the code shown in figure 12(a), the behaviour will remain the same, but now the operation inside the loop is a write-then-read, which does not have WAR hazards. This source-level optimisation has the disadvantage of requiring the use of multiplexers in the datapath to merge the reads and duplicate blocks (larger area, energy and latency) as shown in the resulting circuit of figure 12(b). To avoid hardware duplication, Balsa allows the use of *shared procedures* with the limitation of not being able to access local channels [9]. The proposed solution is to automatically substitute the loop-sequencer control structure obtained for unbounded loop descriptions (as in figure 11(b)) by the optimised control shown in figure 13(b), which allows write and read RTZ overlapping and does not have the restrictions of shared procedures. Simulation results show that this automatic first-read-unfold optimisation has similar performance gain as the operation reorder described earlier.



(a) CaseFetch (b) CaseFetchDefault

Figure 14: Symbols for the *CaseFetch* and *CaseFetchDefault* handshake components.

3) Lookup CAM and forward process implementation: In the AMULET3 AQF, the *lookup* process uses a small, very fast custom CAM to determine if the source registers of the decoded instruction are written or have been allocated in the buffer. Balsa does not provide a way to describe a CAM and generate an efficient circuit structure. The Balsa synthesised circuit consists of a number of logic comparators that, despite being relatively simple, do not perform as well as an optimised CAM, resulting in some performance penalty for the *lookup* process. In the AMULET3 AQF the *forward* process iteratively examines the possible data sources until valid data is found or, if all possibilities are exhausted, the default value read form the register bank is used. This operation was efficiently implemented at the signal-level. As Balsa is a behavioural language, no signal-level operations can be described and attempting to replicate this behaviour in the nFU would require extensive use of sequenced operations that penalise the performance. The implemented solution is to wait for data validity during the allocation process and attach this information to the register number before writing it to the CAM. In this way the CAM will report zero or the single most recent valid source to the forwarding process, avoiding the need for iteration.

4) Register bank operation: Due to the use of dual-rail encoding in the nFU implementation, speculative reading of the register bank concurrently with the *writeout* process is not safe, unless the required register is not present in the forwarding buffer. An alternative, that has been avoided in this design, is the use of arbitration to resolve possible conflicts. Instead, a modified version of the Balsa *CaseFetch* component, that provides a default value when a potential conflict can occur, was designed. In Balsa circuits, the *CaseFetch*, shown in figure 14(a), is used to access a single element from an array of variables: it receives an *index* value that indicates which of the *inputs* must be sent to the *out* port. The new *CaseFetchDefault* in figure 14(b) has an additional input named *default* which indicates whether the selected input or a default (and valid) constant value will be sent, preventing the propagation of spurious values that may cause deadlock.

VII. Results

The nanoSpa with the nFU was synthesised in 180nm technology. After a series of pre-layout, transistor level simulations it was found that the optimum queue size is 4. Different architectures of the nFU were tested and compared running the Dhrystone program. Tables I and II show that performance increases were 10%, with area and energy overheads of 13% in spite of the limitations faced with the Balsa description. These results also show that the techniques used for desynchronising the processes achieve close to 40% increase in performance relative to the use of sync channels. Unfortunately it is not possible to make a relative comparison of the performance gain with respect to the AMULET3 AQF, because there are no published figures with and without the AQF. Pre-implementation, simulation results in [18] suggest that the AQF in AMULET3 would increase its performance by 22.5% when running the Dhrystone benchmark. Notice also that the AMULET3 pipeline has a decoupled memory stage and this feature is not currently present in nanoSpa.

nanoSpa device	*DMIPS*	*speedup* (%)	*overhead* in area (%)
no nFU	78.37	0.00	0.00
nFU (sync signals)	61.22	-28.80	5.2
nFU (allocation mask)	82.03	4.67	15.71
nFU (grouping)	81.86	5.86	11.20
nFU (gruping + unfolding)	86.27	10.08	11.21

Table I: Performance results for nanoSpa using the nFU

nanoSpa device	*Energy for a Dhrystone loop(μJ)*	*overhead* (%)
no nFU	0.360	0.00
nFU (allocation mask)	0.491	36.23
nFU (grouping)	0.393	8.90
nFU (gruping + unfolding)	0.408	13.33

Table II: Energy results for nanoSpa using the nFU

Results show that the first-read-unfold technique described in section VI-B1 is a key factor for the performance gain in the nFU, contributing more than 50% of the speed-up. Table III shows transistor-level simulation results of first-read-unfolded loops with different data widths. The simulated loop was a simple read-then-write to a variable. These figures give an estimated upper bound for the performance gain that can be obtained and show that for datapath widths greater than 3 bits, the speed-up achieved by RTZ overlapping is greater than the overhead of the merge required in the unfolded control tree of figure 13.

width (bits)	1	2	3	4	8	16	32	64
speedup (%)	-11.8	-2.5	-1.0	5.0	7.2	9.0	8.8	11.4

Table III: Influence of data widths in first-read-unfold of read-write unbounded repetitions

VIII. Conclusions and future work

The work presented in this paper demonstrates the feasibility of describing a synthesisable quasi-delay insensitive result forwarding unit in Balsa and obtaining a significant performance increase. Compared to a hand-optimised, full-custom design, the synthesised nFU achieves close to 50% the maximum possible performance increase. This work also introduces a new optimised way of sequencing unbounded repetitions of read-write operations that allows safe overlapping on the RTZ phases and produces non-trivial increases in performance. The work also highlights some of the performance issues that arise from the use of a synthesisable forwarding unit in Balsa, namely the lack of efficient ways of describing and synthesising associative arrays (CAM) and the problem of deadlock-safe concurrent writes and reads in dual-rail variables to perform speculative reading. These problems are currently being analysed together with some peep-hole optimisations that the nFU design has highlighted, including new language constructs. The nFU makes extensive use of arrayed variables and arrayed channels for storing and broadcasting data. At the moment, the authors are looking into the generated structures to find ways of implementing those as optimised handshake modules that could be described with new constructs. From the ongoing analysis of the nFU some peep-hole optimisations such as 4-phase broad semi-decoupled transferrers (different from those presented in [19]) and removal of redundant *FalseVariable* handshake components are currently being investigated. Future work will also include extending the pipeline depth of nanoSpa to decouple the memory stage and explore the effects of new optimisations and components.

References

[1] J. Hennessy and D. Patterson, *Computer Architecture: a Quantitative Approach (2nd edition)*. Morgan Kaufmann, 1996.

[2] J. Sparsø and S. Furber, Eds., *Principles of Asynchronous Circuit Design: A Systems Perspective*. Kluwer Academic Publishers, 2001.

[3] D. A. Gilbert and J. D. Garside, “A result forwarding mechanism for asynchronous pipelined systems,” in *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 1997, pp. 2–11.

[4] T. Verhoeff, “Delay-insensitive codes—an overview,” *Distributed Computing*, vol. 3, no. 1, pp. 1–8, 1988.

[5] D. E. Muller, “Asynchronous logics and application to information processing,” in *Symposium on the Application of Switching Theory to Space Technology*. Stanford University Press, 1962, pp. 289–297.

[6] A. J. Martin, “The limitations to delay-insensitivity in asynchronous circuits,” in *Sixth MIT Conference on Advanced Research in VLSI*, W. J. Dally, Ed. MIT Press, 1990, pp. 263–278.

[7] K. van Berkel, *Handshake Circuits: an Asynchronous Architecture for VLSI Programming*. Cambridge University Press, 1993.

[8] *http://www.handshakesolutions.com/Technology/Haste*.

[9] A. Bardsley, “Implementing Balsa handshake circuits,” Ph.D. dissertation, Department of Computer Science, University of Manchester, 2000.

[10] L. A. Plana, P. A. Riocreux, W. Bainbridge, A. Bardsley, J. D. Garside, and S. Temple, “SPA – a synthesisable Amulet core for smartcard applications,” in *Proc. International Symposium on Asynchronous Circuits and Systems*. IEEE Computer Society Press, Apr. 2002, pp. 201–210.

[11] Q. Zhang and G. Theodoropoulos, “Towards an asynchronous MIPS processor,” in *Cryptographic Hardware and Embedded Systems (CHES 2003)*, ser. Lecture Notes in Computer Science, vol. 2779. Springer-Verlag, 2003, pp. 137–150.

[12] L. A. Plana, S. Taylor, and D. Edwards, “Attacking control overhead to improve synthesised asynchronous circuit performance,” in *Proc. International Conf. Computer Design (ICCD)*. IEEE Computer Society Press, Oct. 2005, pp. 703–710.

[13] L. Plana, D. Edwards, S. Taylor, L. Tarazona, and A. Bardsley, “Performance-driven syntax directed synthesis of asynchronous processors,” in *Proc. International Conference on Compiles, Architecture & Synthesis for Embedded Systems*, Sept. 2007, pp. 43–47.

[14] *http://intranet.cs.man.ac.uk/apt/projects/processors/amulet/*.

[15] A. J. Martin, A. Lines, R. Manohar, M. Nyström, P. Pénzes, R. Southworth, and U. Cummings, “The design of an asynchronous MIPS R3000 microprocessor,” in *Advanced Research in VLSI*, Sept. 1997, pp. 164–181.

[16] R. F. Sproull, I. E. Sutherland, and C. E. Molnar, “The counterflow pipeline processor architecture,” *IEEE Design & Test of Computers*, vol. 11, no. 3, pp. 48–59, Fall 1994.

[17] S. B. Furber, J. D. Garside, and D. A. Gilbert, “AMULET3: A high-performance self-timed ARM microprocessor,” in *Proc. International Conf. Computer Design (ICCD)*, Oct. 1998.

[18] D. A. Gilbert, “Dependency and exception handling in an asynchronous microprocessor,” Ph.D. dissertation, Department of Computer Science, University of Manchester, 1997.

[19] A. Peeters and K. van Berkel, “Single-rail handshake circuits,” in *Proc. Working Conf. on Asynchronous Design Methodologies*, May 1995, pp. 53–62.