Synthesis of Asynchronous Circuits Using Early Data Validity

N. Gupta and D.A. Edwards
Dept. of Computer Science, The University of Manchester, Oxford Road, Manchester M13 9PL, U.K.
{guptan, doug}@cs.man.ac.uk

Abstract

Interest in asynchronous circuit design is increasing due to its promise of efficient designs. The quiescent nature of asynchronous circuits allows them to remain in a stable state until necessary wire transitions trigger an event to occur. This avoids synchronizing events using a global clock tree, which can consume a large amount of energy. The need for low power and high performance circuits leads to investigation of various asynchronous design styles.

The work presented here provides an overview and novel implementation of synthesizing asynchronous circuits using an early data validity protocol. Conventional asynchronous tools synthesize circuits using a broad data validity protocol, which leads to simple circuits, but non-overlapped sequencing of consecutive operations. The early protocol requires data to be valid for a shorter period, allowing consecutive operations to overlap phases. The resulting circuits have a potential increase in performance by allowing greater concurrency and earlier execution of events.

1. Fundamentals of Asynchronous Design

The basic concept of asynchronous circuit design is the absence of a global clock to synchronize the transfer of data. Removal of the global clock eliminates the massive switching network that usually consumes a large amount of energy and dictates the speed of synchronous circuits. Other benefits of asynchronous circuits include increased modularity and lower noise and EM emission. Examples of these properties are described in [4]. Without a clock to signal when data is available, the notion of handshakes is introduced.

1.1 Handshaking

Handshakes occur between two or more circuit elements that require a synchronous transfer of data and/or control between them. Handshakes are generally implemented through the use of request and acknowledge wires and data channels between elements [16]. Between two elements, one actively initiates the handshake by sending out a request while one passively waits for this request. Once the passive unit receives the request, an acknowledgement is sent, signifying the receipt of data or transfer of control. Other similar handshake protocols exist for asynchronous circuits depending on the application and direction of data transfer [16].

Figure 1 and Figure 2 illustrate the difference between synchronous and asynchronous circuit flow where R* are synchronous registers, CL are blocks of combinational logic, and CTL blocks are elements controlling the handshaking between asynchronous stages. The clock controls when circuit components can read and write data for synchronous circuits, whereas the request and acknowledge signals control when circuit components can read and write data for asynchronous circuits.

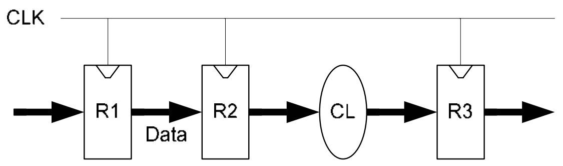

Figure 1. Synchronous circuit flow diagram

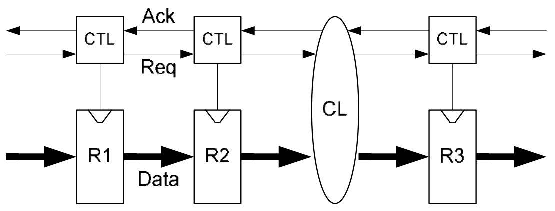

Figure 2. Asynchronous circuit flow diagram

1.2 Data Encoding

In synchronous design, data is often binary encoded where each wire represents one bit of data. For asynchronous circuits, different choices of data encoding are made based on the resulting difference in performance, area, and power characteristics. Two major encoding schemes are bundled data and delay-insensitive encodings [16].

Bundled data (also known as single-rail) encoding is illustrated in Figure 2, where request and acknowledge wires are bundled with the binary encoded data to indicate data validity. Advantages of this encoding include the ability to use the same combinational logic blocks used in synchronous circuits, and easy separation of control from datapath. Bundled data schemes can result in smaller, simpler, low power circuits and usually results in more efficient circuits than delay-insensitive circuits [16]. But this data encoding often requires explicit delay elements to be added along the control path to ensure data becomes valid before the receiving unit is signaled.

Delay-insensitive circuits encode each bit of data across multiple wires. Transitions on one or more of these wires indicate data validity, thus encoding the request signal within the data. Once all the bits of data are realized (i.e. transitions have completed for each bit of data), an acknowledgment is sent and the circuit enters a reset phase, known as the return-to-zero phase, where all the wires reset to zero. A four-phase handshake is the common term for the process of raising the request and acknowledge signals followed by resetting them. The advantage of delay-insensitive circuits is that no delay elements are needed within the circuit since the request is encoded within the data; thus the circuit will execute as fast as the logic allows. However, due to the extra wires needed to encode the data, circuits tend to be larger and extra logic is needed to realize the data bits. A common technique to realize the data bits is to use trees of Muller C-elements [16], which results in greater area and degraded performance.

Due to the nature of four-phase delay-insensitive circuits, consecutive operations cannot be overlapped. A complete four-phase handshake must occur before the next operation since the request is encoded within the data. The next request cannot be sent out before the current operation resets the data. However, for bundled data, the request signal is separate from the data, which allows for varying periods of data validity and possible overlapping of operations. The work presented here revolves around four-phase single-rail handshake circuits [13] and the corresponding data validity protocols.

2. Data Validity Protocols

Three main data validity protocols exist for single-rail circuits: early, broad, and late [13]. Figure 3 and Figure 4 illustrate the time periods when data becomes valid (clear region) and invalid (X region) for each of the three different protocols, depending on whether the circuit element is actively pushing (sending) data or actively pulling (receiving) data. The figures show the data validity for a four-phase handshake on the request and acknowledge wires, where the request wire acts as the data validity signal for push channels and the acknowledge wire acts as the data validity signal for pull channels. A detailed explanation of four-phase handshake protocols for single-rail circuits is described in [13].

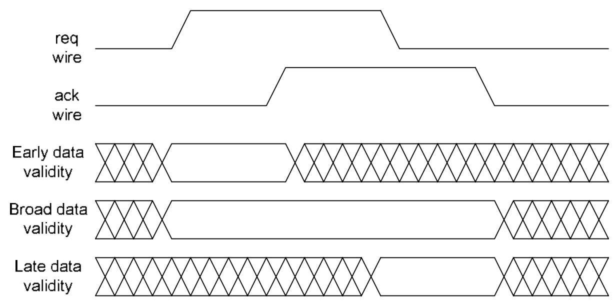

Figure 3. Single rail push channel data validity diagram

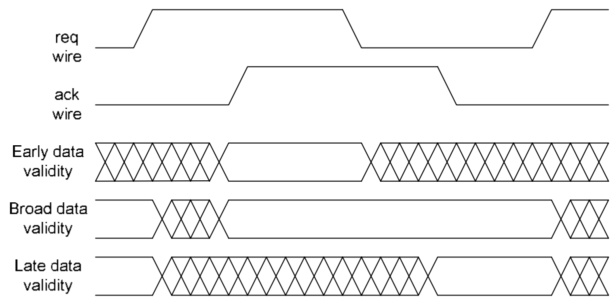

Figure 4. Single rail pull channel data validity diagram

For the early protocol, the data only remains valid for one phase of the handshake in which the data must be processed. The data is assumed to be invalid for the remainder of the handshake. For the broad protocol, the data remains valid throughout the entire handshake, and for a pull channel, remains valid until the start of the next handshake. For a late protocol, the data is assumed to only be valid during the return-to-zero phase of the handshake. Because of the inverted logic needed for implementing late pull channels, late protocols are unfavorable and are not addressed in this work.

3. Motivation

The primary motivation behind implementing an early data validity protocol is the ability to achieve greater concurrency between operations. Figure 3 and Figure 4 illustrate how the data is assumed to be invalid during the return-to-zero phase of the four-phase handshake. This means all data processing must be done before the circuit element starts the return-to-zero phase. Once the return-to-zero phase begins, data is allowed to change and can no longer be processed. Since no useful data processing occurs during the return-to-zero phase, the next operation can begin while the current operation resets. Thus the processing phase of the next operation is overlapped with the return-to-zero phase of the current operation.

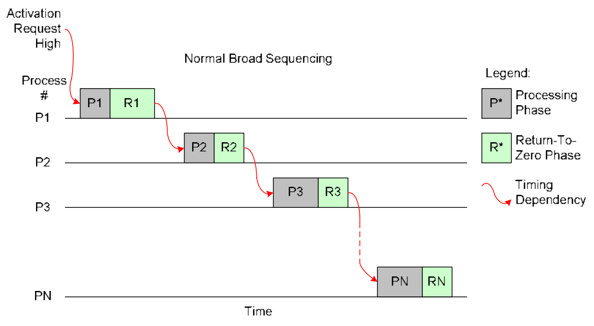

Figure 5. Normal sequencing as implemented using the broad data validity protocol

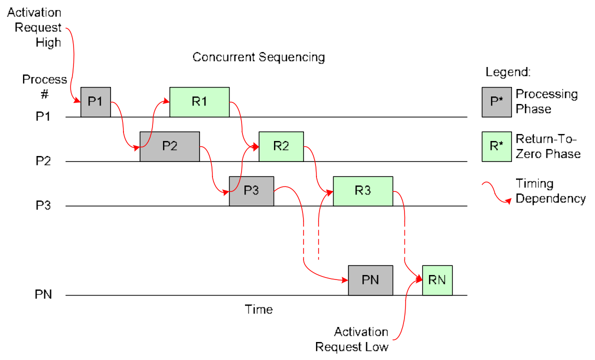

Figure 6. Concurrent sequencing as implemented using the early data validity protocol

Figure 5 illustrates the normal broad sequencing of operations and Figure 6 demonstrates the overlapping of sequential operations (or processes), where the P* blocks represent the data processing phase of a handshake and R* blocks represent the return-to-zero phase of a handshake. For the broad protocol, the data processing can be delayed until the return-to-zero phase since the data remains valid. However, this can create bottlenecks between consecutive handshakes because data is only allowed to change between handshakes. Since the circuit stalls the data validity signal until the data becomes valid, a potential bottleneck will occur if there is a large delay to set up the data for the next operation. In the early protocol, data for the next operation can be set up as soon as the processing completes for the current operation, thus allowing the data to be valid much sooner and reducing bottlenecks between handshakes.

Previous investigation into the early protocol resulted in belief that circuits would result in degraded performance compared to the broad protocol [12]. Implementations of various early control circuits are described in [12], with the conclusion that broad circuit implementations are simpler and consume less power than the equivalent early implementations. Although early circuits tend to contain extra logic to implement the protocol, this investigation did not consider that early circuits have a performance advantage over broad circuits by overlapping operations as described above. The result of overlapping operations in this way results in greater concurrency and allows the circuit to execute a sequence of operations much faster than the equivalent broad circuit.

4. Implementation

The early data validity protocol has been integrated as a back-end to the Balsa asynchronous circuit synthesis tool developed at the University of Manchester [8]. The Balsa toolkit is similar to the Philips Tangram compiler [9][15] and uses the paradigm of handshake circuits [3] to compile design descriptions, written in the Balsa language [7], into networks of interconnected handshake components. Using the set of Balsa tools and commercial CAD tools, a gate-level netlist and layout can be produced from the handshake components.

The protocol was integrated into the back-end by designing circuit implementations for each of the handshake components. A set of the available handshake components and their operation can be found in [2]. A typical handshake component waits for an activation signal to start the operation of the component. For example, the Sequencer component waits for an activation request before beginning a sequence of operations. Once the Sequencer receives this activation signal, activation request signals are sequentially sent out to start the connected processes (see Figure 7). Once the current process has finished processing, it sends an acknowledgement back to the Sequencer component. This acknowledgement signals that the current operation is complete and the next operation can occur. For the broad protocol, a complete four-phase handshake must occur between the Sequencer component and the current process before starting a handshake with the next process (see Figure 5). For the early protocol, the Sequencer component sends out the next activation request signal as soon as the current process acknowledges. The next activation request occurs in parallel with the return-to-zero phase of the current handshake, thus creating the overlap illustrated in Figure 6. The design for the Sequencer component is based on the concurrent sequencers described in [14].

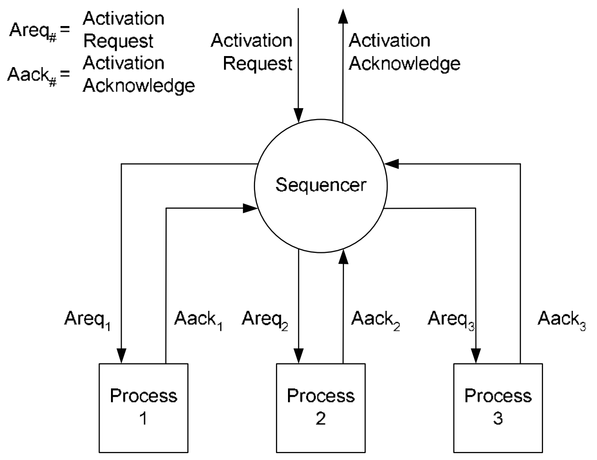

Figure 7. Handshake diagram for Sequencer component

Handshake components like CL in Figure 2 often modify the incoming data, but simply propagate the request and acknowledge signals between the input and output. To take advantage of the early protocol, special circuits were designed to decouple the return-to-zero phases of the input and output. By the time the return-to-zero phase of the input occurs, all data processing has occurred. Thus performing the return-to-zero phases of the input and output in parallel is safe.

Figure 8 and Figure 9 show the logic and timing diagram of the circuit (named Call element) designed to decouple the return-to-zero phases. Once the input begins the return-to-zero phase, the input and output return-to-zero phases operate independently, allowing the input to set up for the next operation and activate the next input request before the output finishes its handshake. However, the next input request will not propagate to the output until the output has completed the return-to-zero phase of the current operation.

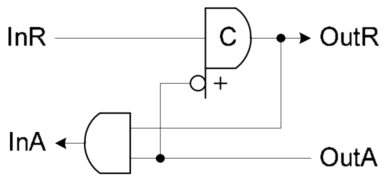

Figure 8. Logic circuit for the Call element

Similar methods to decouple return-to-zero phases have been previously explored [5][10][11], but have yielded circuits that are more complex and do not necessarily take full advantage of the early protocol. They also do not address the issue of circuit synthesis. The protocol presented here is applied to the synthesis of asynchronous circuits using handshake components. [10] provides designs for decoupled latch control circuits, but these controllers are compatible with the broad protocol, not with the early protocol. For the early protocol, as soon as the input is acknowledged, the data is allowed to change. Thus the data must be latched and the latch disabled before the input is acknowledged. Another major difference is that these latch controllers fit into a push-style micropipeline [17] style circuit, whereas the Balsa back-end provides a pull-style implementation using handshake circuits. Future work may involve investigating and integrating a micropipeline architecture into the Balsa back-end that uses decoupled latch controllers to provide a behavior similar to the early protocol while reducing the control overhead and need for added matched delays.

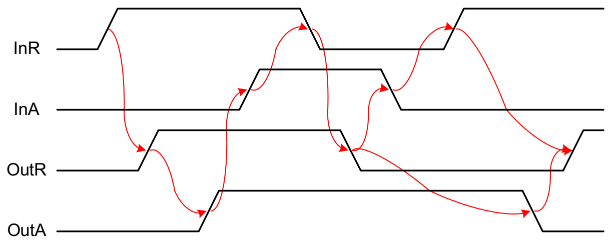

Figure 9. Timing diagram for the Call element

5. Evaluation

The early Balsa back-end has been tested on a Balsa design [1] of the Small Scale Experimental Machine (SSEM) developed at the University of Manchester and a Balsa design of a MIPS processor. The SSEM design is a 32-bit accumulator processor capable of performing arithmetic, logic, memory load, and memory store instructions. Extracted SPICE netlists of the designs were imported into Synopsys VCS/nanosim using 0.18 um technology from STMicroelectronics operating at 1.8V and 25°C. The SSEM design was tested with the Greatest Common Divisor testbench while the MIPS processor was tested with Dhrystone. Results of the simulations indicated an average of a 10% increase in performance and 40% increase in power dissipation by using the early protocol versus the broad.

The increase in power is due to a number of factors. A decrease in simulation time while executing the same logic increases the power dissipation. Also, since extra latches and latch control logic are required to control dataflow in the early components, more transitions are required. This extra logic also decreases the performance improvement of the early back-end. Delays on the request-to-acknowledge path must be added to ensure data is properly latched and latches disabled. This increases the time needed before a component can acknowledge the receipt and processing of data, whereas the broad can acknowledge the receipt of data and process the data later. Thus if the return paths are fairly short, the early protocol may not benefit as much over the broad protocol.

Although the expected increase in power dissipation appears large, the actual energy consumed is the important metric for comparison since energy consumption accounts for the increase in both performance and power. The actual increase in energy consumption is less than 25%, which is a favorable tradeoff considering the 10% performance increase. Future testing will involve running other benchmarks on the MIPS processor and simulating the design of a Java aware SPA processor [6]. These results are expected to produce similar increases in performance and energy consumption. These results will be analyzed to determine where improvements can be made in the implementation of the early handshake components.

6. Conclusions

A novel implementation of asynchronous circuit synthesis using an early data validity protocol is presented here. The fact that data does not need to remain valid throughout an entire handshake was exploited. Circuits were produced to achieve greater concurrency by overlapping the return-to-zero phase of one operation with the processing phase of the next operation. Test results show a favorable tradeoff between the increase in performance and increase in energy consumption that may be achieved using this implementation. Future tests of various circuit designs are expected to produce similar performance improvements.

7. References

[1] A. Bardsley. *Balsa: An Asynchronous Circuit Synthesis System*. Masters Thesis, Department of Computer Science, The University of Manchester, 1998.

[2] A. Bardsley. *Implementing Balsa Handshake Circuits*. Ph.D. Thesis, Department of Computer Science, The University of Manchester, 2000.

[3] C. H. van Berkel. *Handshake Circuits - An asynchronous architecture for VLSI programming*. Cambridge International Series on Parallel Computers 5. Cambridge University Press, 1993.

[4] C. H. van Berkel, M. B. Josephs, S. M. Nowick. Scanning the technology: Applications of asynchronous circuits. *Proceedings of the IEEE, 87(2)*: pp. 223-233, February 1999.

[5] A. Bystrov, D. Shang, F. Xia, A. Yakolev. Self-timed and speed independent latch circuits. In *Proceedings of the 6th UK Asynchronous Forum. University of Manchester*, July 1999.

[6] P. Capewell. *JASPA - Java Aware SPA*. Department of Computer Science, The University of Manchester, 2004. URL http://www.cs.man.ac.uk/apt/projects/processors/spa/ jaspa.html

[7] D. A. Edwards, A. Bardsley, L. Janin. *Balsa: A Tutorial Guide*, Department of Computer Science, The University of Manchester, 2003.

[8] D. A. Edwards, A. Bardsley. Balsa: An Asynchronous Hardware Synthesis Language. In *The Computer Journal, Vol. 45, No. 1*: pp. 12-18, British Computer Society, 2002.

[9] C. Farnsworth, D. A. Edwards, J. Liu, S. S. Sikand. A Hybrid Asynchronous System Design Environment. In *Proceedings of the 2nd Working Conference Asynchronous Design Methodologies*: pp. 91-98, May 1995.

[10] S. B. Furber, P. Day. Four-Phase Micropipeline Latch Control Circuits. In *IEEE Transactions on VLSI Systems, 4(2)*: pp. 247-253, June. 1996

[11] M. B. Josephs, J. T. Yantchev. CMOS Design of the Tree Arbiter Element. In *IEEE Transactions on VLSI Systems, 4(4)*: pp. 472-476, Dec. 1996.

[12] J. Liu. *Arithmetic and Control Components for an Asynchronous System*. Ph.D. thesis, Department of Computer Science, The University of Manchester, 1998.

[13] A. M. G. Peeters. *Single-Rail Handshake Circuits*. Ph.D. Thesis, Technische Universiteit Eindhoven, Eindhoven, The Netherlands, 1996.

[14] L. A. Plana. *Contributions to the Design of Asynchronous Macromodular Systems*. Ph.D. Thesis, Department of Computer Science, Columbia University, NY, US, 1998.

[15] F. D. Schalij. *Tangram Manual*. Tech. Rep. UR 008/93, Philips Electronics N.V., 1995.

[16] J. Sparso, S. Furber. *Principles of Asynchronous Circuit Design*. Kluwer Academics Publishers, 2001.

[17] I. E. Sutherland. Micropipelines. In *Communications of the ACM, vol. 32, No. 6*: pp. 720-738, June 1989.