A Comparison of Asynchronous and Synchronous Digital Design

Sohum Datta
Electrical Engineering and Computer Science
UC Berkeley, Berkeley, CA – 94704
Email: sohumdatta@berkeley.edu

***Abstract*—This report briefly summarizes the main principles of asynchronous digital design, and describes a set of experiments for a class project in a Spring 2017 Course: Advanced Digital IC Design offered at UC Berkeley. The main objective is to compare synchronous and asynchronous realizations of a simple five-stage RISC-V in-order pipeline.**

I begin with a broad introduction to asynchronous (*clock-less*) circuits. The main arguments and counter-arguments for using such circuits are summarized. A collection of basic gates, signal interfacing conventions and pipeline building blocks are provided with explanations of their functions. I then give an outline of an experiment to test synchronous and variations of asynchronous implementations of the integer pipeline. A previous work with similar setup and possible improvements is also described.

This document is a concise survey on the topic (Mid-term report on Phase 1 of the project). Therefore, a greater emphasis has been given on design principles. I wanted this report to be a self-contained primer on asynchronous circuits for a reader with a conventional digital background. The experiment was planned to reinforce the basic principles and gain a first-hand experience of designing asynchronous systems.

I. Introduction

Most digital circuits today are *synchronous* – they have a global oscillating signal (the *clock*) routed to all data-holding components in the system. The clock defines the notion of time in the system: the input for next processing is captured only in a small interval around a specified transition. Therefore, the notion of time is discrete. Since it is far easier to generate and broadcast a common clock than modulate its period at different parts of the chip, the main design decision is to arrange logic such that the delay of all stages are bounded.

Asynchronous circuits do not work on the principle of discrete time. An operation begins just when it is ready – no later or earlier. Instead of using a global *clock* to synchronize dependent operations, local signals are used to indicate to a dependent logic when a previous operation is complete. An asynchronous system can also be modelled as a perfectly *clock-gated* synchronous system where a sequential element is clocked only when its participation is required in the logic.

II. Asynchronous versus Synchronous Design

In synchronous systems, clock delays extend the computation time (and power consumption) of a logic task that does not require global maximum latency. Several engineering issues common in synchronous systems, such as clock skew and jitter, do not affect functionality of asynchronous systems [1].

There are three **main advantages** of asynchronous over synchronous designs:

- **Lower power consumption** Since it does not activate unused units, asynchronous circuits consume power proportional to the amount of computation required for the given task (function and input set). Provided that the leakage power can be kept to a minimum, the power consumption of the circuit depends on the number of LOW to $HIGH$ transistions of signals. It can be shown that any circuit dissipates a minimum amount of *switching power* proportional to the entropy of a formal description of its logic, and that asynchronous circuits can achieve this lower bound within a constant factor [2].

- **Smaller Latency** Due to similar reasons, asynchronous circuits exhibit *average case* behaviour rather than the worst case for their synchronous counterparts. The worst-case functionality guarantee in synchronous systems results in a far higher mean computing latency for random inputs. For instance, the average case latency of an N-bit Ripple-Carry Adder (RCA) is $O(logN)$ but worst case latency is $O(N)$. The worst case behaviour of a Carry-Lookahead Adder (CLA) is $O(logN)$. Therefore, an asynchronous RCA has similar asymptotic time complexity as a synchronous CLA, but both have similar asymptotic logic complexity of $O(N)$ [3].

- **Better Composability and Modularity** As chips include more transistors due to scaling (doubled approximately 20 months from 1971 to 2009), it gets exponentially harder to coherently arrange them and design functioning VLSI systems. On chip process variations makes it worse. Design complexity and cost rises and it gets harder to argue about the correct operation of many concurrent and interacting modules. The Intel Pentium *FDIV* bug [4] and *F00F* bug [5] are some examples of design errors that were discovered a long time after the product was first marketed. Asynchronous circuits provide a more *modular* design paradigm due to *abstraction*: the exact values of component delays are not important as long as they respond to a handshake convention. Then, the design of the VLSI system *decouples* into the selection of a network of components for correctness and (physical

or device) optimization of components for performance.

There are following **main disadvantages** of asyncrhonous circuits [1]:

- **CAD Complexity** Unlike the synchronous design environment, asynchronous synthesis tools aren't easily available and widely supported by the industry. The presence of feedback loops required for correct functioning of its sequential elements complicate the distinction of combinational and sequential components, loop and timing check analysis.

- **Robustness** Discretizing time in synchronous systems reduces exposure of the data stored in sequential elements to electric noise. Since a large enough disturbance is unlikely to happen in the very small interval where the data is latched in, synchronous systems are very robust to electrical noise. However, asynchronous systems will register *any* large and sufficiently long disturbance incorrectly as valid data.

- **Limits to Power Efficiency** For correct functioning of asynchronous systems under noisy inputs, elaborate handshaking protocols are usually applied that make glitch contaminatio unlikely. This results in a far higher logical complexity and power consumption of relatively small systems.

- **Limits to Delay-Insensitivity** We would ideally want all timing information to be abstracted out of asynchronous correctness models. However, there are very few circuits which are truly independent of their component delays [6]. A possible compromise is to design *self-timed* circuits – those that were described as regions of similar delay (negligible wire propagation time) and *channels* of signal transport between them [7]. However, circuits based on these formalisms proved to be very hard to design and optimize [1]. Most asynchronous systems today assume *Quasi-Delay Insensitivity* (QDI): gates are allowed to have arbitrary delays provided that some relative timing constraints are satisfied for each signal that propagates out to a multiple fanouts [8].

III. The Building Blocks

A. Clocks and Handshakes

An easy distinction of asynchronous circuits versus its synchronous counterparts can be understood by the way control and data travels down a feed-forward pipeline. In Fig. 1 the rising edge of CLK travels together with the data carried down the pipeline. The asynchronous implementation (Fig. 2) replaces CLK with *handshake* control units (CTLs) which activate their corresponding latches when ready to receive data. The *control link* is the bundle connecting adjacent CTLs.

An asynchronous link has two control lines in opposite directions: request (REQ) from the sender of the data to the receiver, and acknowledge (ACK) from receiver to sender.

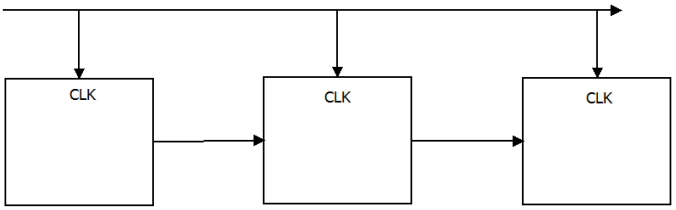

Fig. 1. Data and Control in a synchronous pipeline

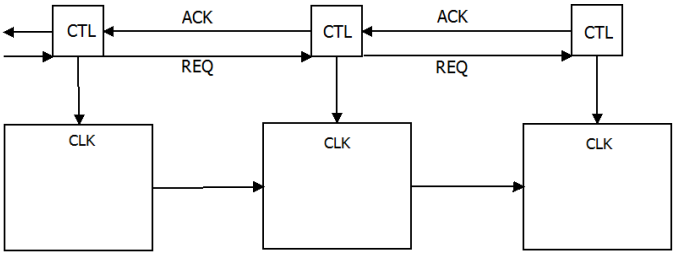

Fig. 2. Data and Control in an asynchronous pipeline

Handshake protocols are a predefined sequence of transitions on these lines. The sequence is trigerred only when an *initial condition* of the state of the two control lines is true for a *sufficient* gate delay. The data transfer can occur *only during* a progression of this sequence. Note that the roles of these control lines depend on the direction of data transfer: in Fig. 2 if data were to travel from right to left, REQ and ACK would be the acknowledge and request controls respectively.

A common distinction between handshake protocols is the encoding of data. *Bundled* handshakes encode data in the usual manner, but use control link to tag data with request and acknowledge messages. Hence, the control signals form an *extended data* that is bundled with the original. Bundled data protocols depend on *delay matching* such that the order of signals transmitted by the sender is preserved when it reaches the receiver. The *4-phase* bundled protocol (Fig. 3) requires four transitions: (1) sender places data and raises REQ (2) the receiver latches the data in and raises ACK to communicate the receipt (3) the sender returns REQ to zero after detecting a HIGH ACK (4) the receiver returns ACK to zero after

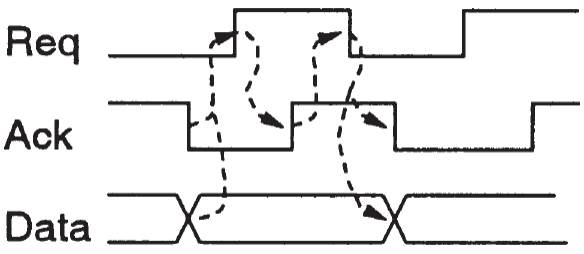

Fig. 3. Signal diagram for 4-phased bundled handshake.

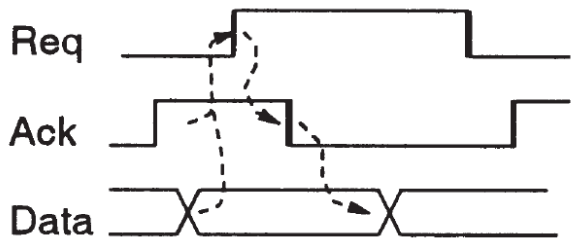

Fig. 4. Signal diagram for 2-phased bundled handshake.

detectinga LOW REQ.
The 4-phase Bundled protocol always returns its control signals to their default value of LOW. Since both control lines have to transition twice per transaction, a possible improvement would be to remove the transitions (3) and (4). This reduces transmit time and power consumption per transaction. The resultant *2-phase* bundled protocol (Fig. 4) encodes an event (*receipt* or *acknowledgement*) by signal transition rather than signal value.
Preserving data arrival order is hard to implement in large asynchronous systems [1], hence a more robust encoding is often preferred for critical links: separate wires transition for transmitting data 0 and data 1, called the *dual-rail* protocol. Dual-rail links are very robust as the two sides can communicate regardless of wire delay mismatches. However, they are also very expensive and result in a larger power envelop. I will only use the two bundled protocols in the experiments.

B. The Muller C Element

The Muller C Element is the basic gate of asynchronous circuits. Based on the work by David Muller in the Illiac II computer [9], the gate (Fig. 5) works on the *indication principle*. When the output y transitions from $HIGH$ to LOW, we are certain that the last both the inputs have transitioned from $HIGH$ to LOW and remain LOW. Similarly, when y transitions from LOW to $HIGH$, we are sure that the both a and b had also gone from LOW to $HIGH$. From the previous section on handshakes, it can be easily seen that the indication principle is central to the control of handshakes. The next subsection describes how to implement circuits based on this gate.

C. The Muller Pipeline

The Muller pipeline is a method of carrying the control request and acknowledge signals down a pipeline chain (the CTLs in Fig. 2). Fig. 6 depicts three consecutive control links of such a pipeline. To see how this works, consider the case where a datum is traveling from stage $(i-1)$ to stage $(i+1)$ and the stable state of both REQ and ACK is LOW (pipeline empty). The signal $C[i]$ represents the output of the corresponding C gate, which is fed into the ACK to the previous stage, the REQ to the next stage, and the **Enable**

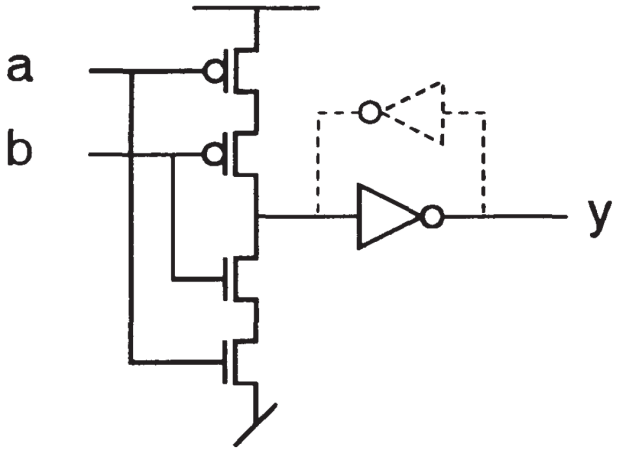

Fig. 5. The Muller C Element.

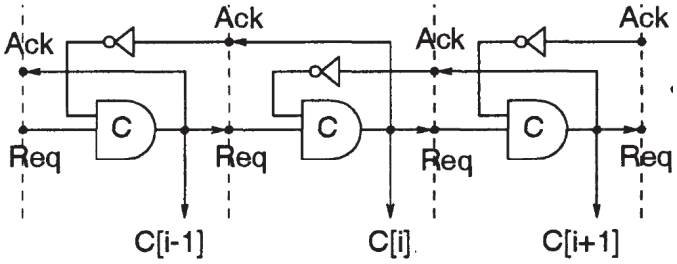

Fig. 6. The Muller Pipeline control.

signal of the corresponding latching element. Henceforth, $C[i]$ will stand for the signal $C[i]$ and the corresponding Muller C gate – the meaning will be clear from the context.
When the data first appears in stage $(i-1)$, the REQ line of $C[i-1]$ gets raised. Since the feedback from $C[i]$ through an inverter is already HIGH, $C[i-1]$ flips to HIGH, sends the ACK signal to the left and latches in the data. Finally, the REQ line to $C[i]$ is raised.
Again, $C[i]$ has a HIGH feedback from $C[i+1]$ via an inverter. When REQ reaches $C[i]$ is raised to HIGH. This would cause the feedback from $C[i]$ to $C[i-1]$ to fall to LOW and therefore $C[i-1]$ to low as well (we assume no new datum coming in, hence REQ to $C[i-1]$ was LOW once the ACK to the left was raised). Hence, the data has moved from $C[i-1]$ to $C[i]$ without changing anything in the rest of the pipeline. Hence, by induction we can conclude the datum will pass without holds through the pipeline.
The Muller pipeline is the primary structure used to implement pipeline control of asynchronous systems [8]. Since the REQ of a stage passes through an inverter into the C gate of the previous stage, and the C gate only flips if both their inputs agree, $C[i]$ and $C[i+1]$ cannot be both HIGH at any instant of time for all stages i (assuming the pipeline started not-full i.e. atleast one $C[i]$ was LOW and the entire pipeline was stable). Since two adjacent being HIGH simultaneously would mean

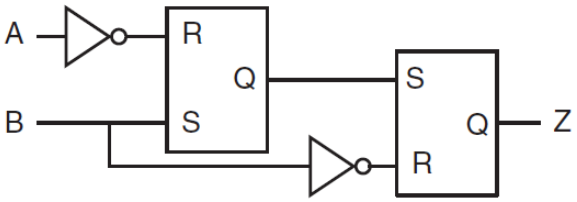

Fig. 7. A Latch-based implementation of the Muller C Element.

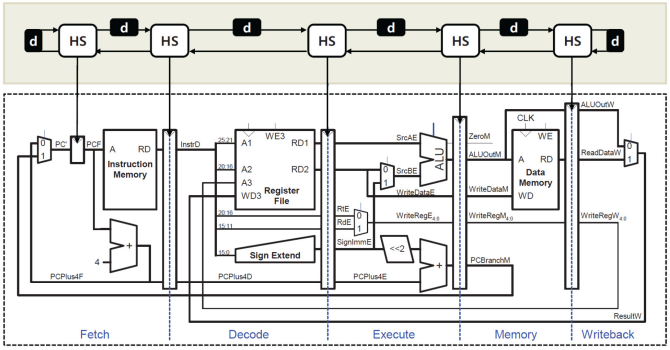

Fig. 8. The Integer Pipeline with Asynchronous Control Logic.

transferring data at the same instant, this means the pipeline can transfer data at most every alternate stages.

Note that the circuit controls throttling automatically. If the right-most consumer is not ready to take in the data, the last stage control will be HIGH and pull down the feedback to LOW for the previous stage. This would make the second last stage returning a zero acknowledgement to its left. Hence, by the induction principle, the stall will eventually reach the left of the pipeline.

This circuit is functionally symmetrical since it can transport data in *both* directions without any modifications; only the REQ and ACK signals will exchange meanings. The phases of handshakes does not matter for correctness. Finally, this circuit is completely delay-insensitive: it does not depend on relative wire or gate delays.

IV. Experimental Setup for Asynchronous Design

The experiment consists of comparing a basic inorder RISC-V 32I Integer pipeline with asynchronous and synchronous control logic, and test for power consumption, area overhead and tolerance to change in frequency. The synchronous pipeline will be available from EECS251 FRGA Lab Design; the goal of this project will be to successfully synthesize and test an asynchronous version on a Xilinx Virtex 5 FPGA. The experiment will be conducted on an ML505 Developement Board, the standard setup for EECS251 FPGA Lab.

For the experiment to succeed, the C element must be realized using synchronous components on the Virtex FPGA. A latch based equivalent design is shown in (Fig. 7) [10]. The benefits of this implementation is many-fold: the standard synchronous CAD tools provided by Xilinx will not read it as a combinational loop and correctly infer latches. Extra user-defined constraints will be required to ensure that the SLICEs of the allocated Latches are not routed far apart for the C Element to function properly. The latch-based implementation is know to be about 77% worse in power consumption and 29% worse in delay than the CMOS C Element.

After an extensive search, I was unable to find substantial material on synthesizing asynchronous systems on commercial synchronous FPGAs. Most systems used commercial Asynchronous FPGAs with SLICEs containing Muller C Gates and David Cell [11].

However a recent work [12] (see Fig. 8) was successful in synthesizing a FIFO and a MIPS pipeline on a Xilinx Virtex 5 FPGA. Although their pipeline was tested using externally supplied input waveforms, they haven't reported power measurements by harnessing tests of varying activities. The most important goal of the next Phase of my project would be to synthesize and test a C Element and an Asynchronous Muller pipeline (a simple FIFO). These smaller experiments will ensure that I have the right tools for testing the more complex pipeline in the later phase.

The ultimate plan will be to implement asynchronous RISC-V pipelines with 2-phase (micropipelined) and 4-phase handshaking protocols. The differences in FPGA resources allocated and power consumed will be recorded. Finally, I plan to run the EECS251 FPGA Lab test suite on the asynchronous pipeline through inputs sourced from another FPGA board, and measure the power for all three designs.

V. Conclusion

A very brief introduction to the design of asynchronous logic was given. The most important concepts required to realize a processor pipeline was provided in significant depth. Finally, the plan for the experiment was described and the major goals outlined.

Acknowledgment

The author would like to thank Prof. Borivjoe Nikolic and Prof. Elad Alon, instructors of EE241B and EECS151/251 offerings in Spring 2017 at UC Berkeley respectively, for their kind support and comments. Special thanks is also due to Vignesh Iyer, a Graduate Student Instructor of EECS151/251 for helping me with the setup and encouraging me to try it as an extension of FPGA Lab.

References

[1] J. Spars and S. Furber, *Principles of Asynchronous Circuit Design: A Systems Perspective*, 1st ed. Springer Publishing Company, 2010.

[2] Jose A. Tierno, Rajit Manohar, and Alain J. Martin, *The Energy and Entropy of VLSI Computations*, In Proceedings of the 2nd International Symposium on Advanced Research in Asynchronous Circuits and Systems (ASYNC '96), IEEE Computer Society, Washington, DC, USA, 1996.

[3] F.-C. Cheng, S. H. Unger, M. Theobald, and W.-C. Cho, *Delay-Insensitive Carry-Lookahead Adders*, In Proceedings of the Tenth International Conference on VLSI Design: VLSI in Multimedia Applications (VLSID '97). IEEE Computer Society, Washington, DC, USA, 1997

[4] Tom R. Halfhill, *An error in a lookup table created the infamous bug in Intel's latest processor*, BYTE, March 1995.
[5] Collins, Robert R., *The Pentium F00F Bug*, Dr. Dobb's Journal, May 1, 1998.
[6] Martin, Alain J., *The Limitations to Delay-Insensitivity in Asynchronous Circuits*, Conference on Advanced Research in VLSI , Springer 1990, New York.
[7] Charles L. Seitz, *System Timing. Chapter 7* in *Introduction to VLSI Systems* by Carver Mead and Lynn Conway, Addison-Wesley, 1979.
[8] Martin, Alain J., *Synthesis of Asynchronous VLSI Circuits*, Caltech Computer Science Technical Report, CS-TR-93-28, 1993.
[9] D. E. Muller, *Theory of asynchronous circuits*, Report no. 66, Digital Computer Laboratory, University of Illinois at Urbana-Champaign, 1955.
[10] J. Murphy, *Design of latch-based C-element*, Electronics Letters, September 2012.
[11] Teifel, John, and Rajit Manohar, *Highly pipelined asynchronous FPGAs* Proceedings of the 2004 ACM/SIGDA 12th international symposium on Field programmable gate arrays. ACM, 2004.
[12] Lee SJ., Lee DY., Ko YW., Lee JG, *Asynchronous Circuit Design on an FPGA: MIPS Processor Case Study* In: Lee G., Howard D., lzak D., Hong Y.S. (eds) Convergence and Hybrid Information Technology. ICHIT 2012. Communications in Computer and Information Science, vol 310. Springer, Berlin, Heidelberg (2012)