Digital Asynchronous VLSI Final Project

Douglas Ellwanger, Scott McClure, Christopher Stone, and Jonathan Tse

I. Introduction

As described in [4], currently, the majority of VLSI circuits are synchronous. In other words, the timing characteristics of each individual block of logic are known and predicable within a margin of error. With this timing information, logic blocks in a synchronous system can be controlled by a global clock, where each tick of the clock represents the start of a computation, and each tock represents the completion of a computation. This method offers a number of benefits, the most promient of which being global knowledge of completion. Since each tick-tock cycle is no shorter than the longest computation time of each logic block, the global clock is a de-facto indicator of completion of any portion of the circuit. Additionally, synchronous systems offer a measure of resistance to noise input. By only evaluating the results of a computation at a tock signal from the clock, we can allow the output of a logic block to fluctuate between each tick and tock with no ill effects.

Of course, there are always tradeoffs. The frequency of the clock signal is dependent on the physical characteristics of the circuit, because the period of the clock cannot be less than the longest computation time of any logic block. Thermal effects, changes in geometry, and changes in relative size of components can change the computation time for a logic block, so the selection process for the clock frequency must take all of these effects into account. Additionally, the clock signal itself is typically a high-frequency square waveform which must be routed to all parts of a circuit in order to be effective. Due to the effects of parasitic capacitance, as circuit complexity and size increase, so does the amount of current required to drive the clock signal. The high frequency and increased current result in increased power dissapation regardless of whether or not there is any real computation going on at all.

A solution to the downsides of synchronous circuits is to remove the clock entirely and design an asynchronous circuit. The responsibility for detecting computation completion now falls to each individual logic block. While this results in increased complexity at the logic block level, along with the requiste additional delay, each logic block is now delay-insensitive [4]. Rather than waiting for a clock signal to signify completion, each logic block can begin computation as soon as its inputs are valid. This also means that the logic block with the longest computation time will no longer determine the computation time of the entire system, which means that asynchronous systems often follow average-case completion time scenarios as opposed to the worst-case of a synchronous system [4].

One downside to asynchronous computation is newfound susceptiblity to noise. In a clocked system, the inputs and outputs are evaluated on the falling and rising edges of the clock signal; whatever happens in between edges can be transient and noisy, so long as the signals have resolved on each clock edge. As a result, asynchronous systems need to have some resistance to these types of signals, where the voltage is either a transient or somewhere between logic levels. The asychronous design methodologies outlined in [2], [3] assume that voltage changes are monotonic, so adopting a two-wire protocol solves most of the issues. In a two-wire protocol, there are two separate logic lines, one to represent the true and the other to represent false. Each bit value can then be represented by raising the appropiate logic line high, while keeping the other low, and bits can be distinguished by lowering both logic lines to give a neutral state. A further assumption is that at no time should both lines be high.

This additional complexity of the two-wire protocol and completion detection mechanism requires a handshaking protocol. Each logic block needs knowledge of whether or not the previous block has completed. In order accomplish this, every communcation between blocks has a handshaking protocol. A typical 4-phase handshake protocol between two blocks A and B, where A precedes B in the logic circuit, essentially looks like this[1]:

1) B raises the handshaking line between A and B to indicate that it is ready to receive data
2) A raises its output lines to the correct values that it has computed
3) B acknowledges the receipt of data by lowering the handshaking line
4) In response to the lowering of the handshaking line, A returns its outputs to the neutral state

A. Project Overview

Our final project for ENGR3430: Digital VLSI is an exploration of asynchronous circuit design. In order to accomplish our goal, we designed several types of 32-bit wide full adders, both synchronous and asynchronous. By comparing the performance of synchronous and asynchronous adders over a wide range of input data and environmental conditions–temperature, supply voltage fluctuations, etc–we hope to qualitiatively determine which is more robust and efficient. We've chosen three different adder topologies–carry ripple, carry lookahead, and Kogge-Stone–and implemented each using both synchronous and asynchronous logic, making sure to use static CMOS for both logic types.

[1]The 4-Phase handshaking protocol was taken from [2], [3]

II. Adders

A. Carry Ripple

The carry ripple adder is the simplest possible adder to implement. It is completely naive and greedy in computing sums, as each carry out signal is reliant on the carry in signal preceding it. The asynchronous variant of the carry ripple does make minor improvements, resulting in a the average-case computation time [2], [3], [1].

A synchronous ripple-carry full adder is very simple, involving only five 2-input gates: 3 NAND and 2 XOR, for a total of 32 transistors. An asynchronous ripple-carry full adder is significantly more complex, as described in [2], [3], [1]. In our implementation, taken from [2], [3], [1].

The asynchronous ripple carry is broken up in to four distinct pieces. Two of the pieces handle the sum true and false states, and the other two handle the carry out true and false. Part of the reason for the increased transistor count is the two-wire bus; every bit now needs two transistors. Some of this cost is made up for in the pull-up stack. Looking at the adder implmentation in [2], [3], [1] we see that the pull-up stack is either just the carry bits or the si signal, which represents the acknowledge signal from the logic after the entire 32-bit full adder chain. Instead of having to invert the pull-down logic, the pull-up logic is only the logic that returns the outputs to the neutral state. It is worth nothing that the output is the logical not of the desired output. Normally this would be fixed with a simple inverter, but in this case, we need to help guard against transients so we introduce some stateholding elements by way of a feedback inverter.

Note that the implementation of the asynchronous carry-ripple requires a control signal for si, which can be tied to the validity of the output of the adder, i.e. is the output neutral or a real value. This can be accomplished with a NOR and an inverter.

B. Carry Lookahead

The carry lookahead represents the next step of advancement beyond the carry ripple adder. Operating under the assumption that all inputs to the entire 32-bit adder arrive simultaneously, the carry lookahead has logic that enables it to make decisions regarding the carry out for a particular adder cell without knowledge of the carry in. For example, if both inputs are 1, then the carry out will definitely be a 1, regardless of what the carry in is. On the other hand, if the inputs satisfy an XOR (i.e. only one of them is true) then the carry in will be propogated to the carry out. Looking at an adder cell somewhere in the middle of the 32-bit adder chain, it is now possible to compute the carry out without knowing about the carry in, thus reducing the overall execution time, since adder cells far away from the initial carry in can compute without having to wait.

In our implementation, the carry lookahead logic happens in blocks of four adder cells to allow for easy propogation across blocks of four adder cells. Four cells is the optimum number since it allows for a significant jump across a large number of adder cells, while remaining simple enough to implement in transistors. Since the actual carry propogate time is shortened due to both the capability to "jump over" adder cells as well as the generated carry out signals–the ones independant of carry in–the carry lookahead adder represents a significant leap in efficiency over the carry ripple adder.

C. Kogge-Stone

The Kogge-Stone adder has the shortest computation time of static CMOS adders. It gains this speed through use of many layers of generate propogate logic. Each layer of generate and propogate logic forwards to the next layer, but significantly more towards the most significant bit of the adder. This topology offers the same adder cell skipping advantages of the carry lookahead adder, but can skip more than four cells due to the layer topology. The layer topology also keeps the logic complexity down, at the cost of a massive amount of wiring.

Appendix A: Cellview Breakdown

All the cellviews are called <addername>nopad

Dellwanger

All cellviews are under the library "async."

- 1bitmux4 – 1-bit 4to1 mux
- 8bitmux4 – 8-bit 4to1 mux
- addercell – asynchronous carry ripple adder cell without the staticizer
- async-addercell – asynchronous carry ripple adder cell with the staticizer
- biginverter – big inverter
- blackeven – synchronous Kogge-Stone black cell, even row
- blackodd – synchronous Kogge-Stone black cell, odd row
- equals32 – cell that tests two 32-bit inputs for equality
- grayeven – synchronous Kogge-Stone gray cell, even row
- grayodd – synchronous Kogge-Stone gray cell, odd row
- grayoddpolyout – synchronous Kogge-Stone gray cell, poly for outputs
- koggenopad – synchronous Kogge-Stone layout with test bench, but no pads
- ks12connector – synchronous Kogge-Stone interconnect between row 1 and row 2
- ks12connectornop – synchrounous Kogge-Stone interconnect between row 1 and row 2, no poly
- ks23connector – synchronous Kogge-Stone interconnect between row 2 and row 3
- ks23connectornop – synchronous Kogge-Stone interconnect between row 2 and row 3, no poly
- ks34connector – synchrounous Kogge-Stone interconnect between row 3 and row 4
- ks34connectornop – synchrounous Kogge-Stone interconnect between row 3 and row 4, no poly
- ks45connector – synchronous Kogge-Stone interconnect between row 4 and row 5

- ks5connector – synchronous Kogge-Stone connect from row 5 out
- ksadder – nothing
- ksadder16 – synchronous Kogge-Stone 16-bit adder
- ksadder32 – synchronous Kogge-Stone 32-bit adder
- ksadder32_split – synchronous Kogge-Stone 32-bit adder split down the middle for ease of layout of chip
- ksadder4 – synchronous Kogge-Stone 4-bit adder
- nand2small – 2-input NAND, small size
- nequals4 – 4-input NEQ
- nor2nocont – 2-input NOR, no poly contacts
- pgcell – propogate generate cell for synchronous Kogge-Stone
- pgconnector – connector for the propogate generate cell of the synchronous Kogge-Stone
- pgconnectorlast – connector for the last propogate generate cell of the synchronous Kogge-Stone
- shiftreg – 2-bit shift register
- skinnyinverter – a skinny inverter
- staticizer – inverter with quarter strength feedback for state holding
- sum – sum stage of the synchronous Kogge-Stone
- xor2short – 2-input XOR short size
- xor2shortinv – 2-input XOR short size with an inverter

Cstone

All cellviews are under the library “final.” We got lazy and only put the useful ones below.

- decoder32good – 32-bit decoder
- look_ahead_32 – 32-bit carry lookahead adder

Smcclure

All cellviews are under the library “async,” and are all pretty much self-explanatory for a 32-bit asynchronous Kogge-Stone.

Jtse

Jon did pretty much all of his work in other people’s folders, or copied individual cells back and forth.

References

[1] Martin, A. J. “Asynchronous Datapaths and the Design of an Asynchronous Adder.” California Institute of Technology, 1991

[2] Martin, A. J. “Synthesis of Asynchronous VLSI Circuits.” 1991

[3] Manohar, R. “Asynchronous VLSI Systems.” 2003

[4] Martin, A. J. “Tomorrow’s Digital Hardware will be Asynchronous and Verifed.” California Institute of Technology, 1992