Chapter One
Introduction

1.1. Asynchronous Design

Because of the advances in semiconductor technology, VLSI circuit density and speed have increased dramatically over the years. In a synchronous design, one of the limitations on system speed is clock skew, which is the phase difference of the clock signals at different locations of the system. As the feature size decreases, wire delays and loadings become proportionally more significant, and the clock skew problem becomes worse. Clock skew can be minimized with proper distribution, but clock balancing cannot be done until the chip layout is complete and loadings characterized. More robust circuits can be designed to be less sensitive to clock skew, but most likely at the expense of speed and chip area. Furthermore, the maximum clock frequency has to accommodate the critical paths in the system, even though some stages may be able to operate at a much faster rate.

Instead of having a global clock for synchronization, each block in an asynchronous system decides when it is ready to accept a new set of data to be processed. Various handshake protocols may be used to coordinate the communication, and they will be discussed in the next chapter. A properly designed self-timed system can be made very modular. For example, if an application requires a faster adder, a ripple-carry adder may be directly replaced with a carry-lookahead one. Since the circuit is self-timed, the rest of the system does not need to be modified; it will just run faster when add operations are performed. Conversely, any module may be substituted with a slower version to save power, and there is no clock to be adjusted to compensate

for the increase in critical path.

The following examples illustrate some advantages of asynchronous systems. The torus routing chip developed at Caltech [Dall86] is a self-timed circuit for multiprocessor interconnections. Due to an oversight in the critical path, the first silicon could be run only at 4 MHz rather than the expected 20 MHz. However, the chip still functioned correctly because of the self-timed design. In the DEC PDP-6 computer, an asynchronous adder was used to take advantage of the higher average performance [Bell78]. It was observed by von Neumann and others that the average number of carries is $\log_2(\textit{width of data})$. Therefore, for the 36-bit word size of PDP-6, the average performance was increased by a factor of 7 because the number of carries is 5.2 on average rather than the worst case of 36. In a recent RISC architecture, the 100 MHz HP PA-RISC CPU employs a self-timed floating point coprocessor due to speed and area considerations [Wils92]. The multiplier can compute a double-precision result in 20 ns and consumes only 0.07 cm^2 chip area.

However, by not having a global clock, additional handshake circuits must be added to handle inter-block communication, and self-timed circuits themselves are more complex, in general. These timing and area overheads for handshaking must be considered when choosing between a synchronous or an asynchronous path. Also, designing circuits for proper handshake sequencing is not a trivial task. Several approaches have been made to synthesize hardware from high-level specifications [Burn87, Meng89, Moln85].

The difficulty with metastability may seem to be a major drawback for asynchronous systems. However, as pointed out in [Meng89], asynchronous design does *not* introduce more undecidable timing problem. For latching data, the clock in a synchronous system is made slow enough so that data arrives before the clock edge. In

a properly designed, fully asynchronous system, the latching signal is *guaranteed* to arrive after the data, based on the handshake methodology. An arbiter, on the other hand, is inherently metastable because more than one request may become active simultaneously [Meng89, Seit80]. Asynchronous systems may be more dependent on arbiters due to the nature of unpredictable timing. However, if fair mutual exclusion is to be incorporated in a synchronous design, then metastable circuits are usually necessary.

1.2. Fault Tolerance

In applications where reliability is a major concern, fault tolerant features should be designed into a system to minimize down time. Since transient faults occur much more frequently than permanent faults [Cast82], fast recovery from transient faults is a key to improve system performance. In environments where a high rate of transient faults is expected (due to radiation, noise, etc.), software-driven corrections may be too slow and impractical, and hardware fault tolerant features must be added to assist recovery. The RH32 radiation-hard, 5-chip computer under development by TRW, McDonnell Douglas, and United Technologies is an example of fault-tolerant processor [TRW92]. Various degrees of fault tolerance may be achieved by running the CPU stand-alone, with other chips in the family, or as a processor/checker pair with two CPUs. Many faults can be detected and corrected under 2 μs (25 MHz clock) without any software intervention.

To achieve a high degree of fault tolerance, it is often necessary to detect errors as soon as they occur so that corrupted data is not spread throughout the system. One way is to check the data in series by delaying execution until verification is complete. However, performance is reduced because either the cycle time or the number of

pipeline stages must be increased. To minimize performance degradation, checkers can operate in parallel with the functional units. The next pipeline stage can start execution while data verification is carried out, and the checking result will be available some time later. This largely solves the problem with delay through the checker, but fault recovery becomes more complicated because all computations on the corrupted data must also be discarded. *Micro rollback* is a technique that utilizes concurrent verification, and the system is rolled back several cycles in response to a delayed error signal [Tami88, Tami90a]. Micro rollback will be briefly described in the next chapter, and for complete discussion, please see the listed references.

1.3. Scope of Thesis

As the title of this thesis suggests, AMPIRE is an asynchronous fault-tolerant microprocessor. The DLX architecture [Henn90] is the basis for the design because it is well-known, and its load-store RISC architecture allows ease of implementation in an academic environment. Asynchronous processors have been built and tested [Jaco90, Mart89], and a fault-tolerant processor capable of micro rollback has been designed and implemented [Tami90b]. Therefore, the original goal of this work was to design an asynchronous processor that supports micro rollback. Due to unexpected complexities that will be discussed in Chapter 7, AMPIRE rolls back to instruction boundaries only, not sub-instruction level as in micro rollback. However, many fault-tolerant ideas are still based on the research results of micro rollback.

In the next chapter, previously published work related to self-timed design and micro rollback will be reviewed. The AMPIRE architecture and some design decisions will be discussed in Chapter 3. In order to check if the design is logically correct, a Verilog model of the processor has been written and simulated. The behavioral model

will be described in Chapter 4, and simulation results will be shown in Chapter 5. Since this processor is not physically implemented, circuit diagrams will be presented in Chapter 6 to justify some high-level constructs used in the behavioral model. Chapter 7 will then discuss some issues found in the design process, and what future work can be done as extensions of this research. The full Verilog code and the source code for AMPIRE assembler are listed in the appendices.