Report Number: WUCS-92-45

1992-10-01

Performance Comparison of Clocked and Asynchronous Pipelines

Authors: Mark A. Franklin and Tienyo Pan

Clock (synchronous) and self-timed (asynchronous) represent the two principal methodologies associated with timing control and synchronization of digital systems. In this paper, clocked and the asynchronous instruction and arithmetic pipelines are modeled and compared. The approach, which yields the best performance is dependent on technology parameters, operating range and pipeline algorithm characteristics. Design curves are presented which permit selection of the best approach for a given application and technology environment.

Performance Comparison of Clocked and Asynchronous Pipelines

Mark A. Franklin
Tienyo Pan

WUCS-92-45

October 1992

Clocked (synchronous) and self-timed (asynchronous) represent the two principal methodologies associated with timing control and synchronization of digital systems. In this paper, clocked and the asynchronous instruction and arithmetic pipelines are modeled and compared. The approach which yields the best performance is dependent on technology parameters, operating range and pipeline algorithm characteristics. Design curves are presented which permit selection of the best approach for a given application and technology environment.

Department of Computer Science
Washington University
Campus Box 1045
One Brookings Drive
St. Louis, MO 63130-4899

This research has been sponsored in part by funding from the National Science Foundation under Grant CCR-9021041.

PERFORMANCE COMPARISON OF CLOCKED AND ASYNCHRONOUS PIPELINES

Mark A. Franklin and Tienyo Pan

Computer and Communications Research Center
Washington University
St. Louis, Missouri 63130
(314) 935-6107 jbf@wuccrc.wustl.edu

1. Introduction

Parallelism and *pipelining* represent two major approaches to obtaining processing speedup by exploiting computational concurrency. These approaches have been used at both macro levels where computational tasks may represent entire procedures, and at the micro level where tasks may correspond arithmetic operations in a vector instruction, individual instructions, or phases in the instruction execution process. Pipelining is a key element in the design of high performance vector processors, and is used extensively in contemparary RISC processors for instruction level pipelining.

This paper examines the design of pipelined systems based on two alternative control and synchronization methodologies; clocked (synchronous) and self-timed (asynchronous). While the general development presented applies to both macro and micro level pipelining, we restrict the detailed analysis to systems described at the micro (logic design) level. Our eventual goal is to develop appropriate models which will permit the designer to determine which methodology yields the best overall pipeline performance for a given application, technology and operating range.

The performance of synchronous pipelines has been considered by a number of investigators. In addition to the general work by Kogge [1], Hennessy and Patterson [2], and

Stone [3], there have been a number of studies which have considered the timing constraints, circuit level performance, and stage optimizations associated with pipeline design [4,5,6,7,8]. These studies begin with a given stage and latch design, present a set of timing constraint equations necessary to ensure correct performance, and then develop a set of technology based performance equations for the given pipeline. Since performance will vary with the number of stages used in the pipeline (due to factors such as clock skew), questions of optimal pipeline design are considered and optimal design strategies are discussed. In the Kunkel and Smith [7], and Dubey and Flynn [8] papers, instruction pipelines are considered and the problems of control and data hazards are included in the analysis.

Performance modeling of asynchronous pipelines has, in general, been less extensive and explicit than the synchronous work cited above. This is probably due, in part, to the fact that asynchronous design techniques are (currently) infrequently used in commercial designs. General work on asynchronous design issues have been considered for some time with some of the early work in this area being incorporated in the design of a set of asynchronous computer design modules [9] which could be easily used in a pipelined fashion [10]. More recently, work which has focussed on the synchronization and pipelining aspects of asynchronous systems may be found in [11] and [12]. In addition, an asynchronous RISC processor design has been implemented at Caltech [13,14] and an alternative design analyzed by Ginosar and Michell [15].

Few quantitative results have been identified which compare the performance of clocked versus asynchronous systems when these differing methodologies have been applied to the implementation of the same function. Wann and Franklin [16] consider the two approaches in the context of interconnection network design. They present performance models and associated decision curves which permit the designer to determine which approach is best for a given set of technology parameters. Kung and Gal-Ezer [17] compare asynchronous and clocked implementations of an N*N array of (systolic/wavefront) processors and conclude that for large array dimensions ($N > 25$) the asynchronous design has higher performance. Meng, Brodersen

and Messerschmitt [12] present the design of an asynchronous programmable signal processor and, using simulation techniques, show that for the design and parameters selected, the asynchronous system has about twice the throughput of the equivalent clocked system. Indications are that this performance advantange increases as feature size shrinks and chip size increases. The work presented in this paper continues in this tradition, with the goal here being to formulate a general quantatative framework for comparing the two methodologies.

2. Pipelining

An S-stage (segment) pipelined processor offers an S-fold increase in throughput over its non-pipelined counterpart only if every stage is 100% utilized. Pipelines generally do not achieve this for several reasons:

- Unbalanced Workload: Partitioning a function into S stages of equal time duration is difficult. Typically the slowest stage limits pipeline throughput.
- Control (Branching) and Data Dependencies: Control and data dependencies may cause flushing or stalling of a pipeline, thus reducing overall utilization.
- Finite Input Stream: Unless the input stream is long compared to length of the pipeline, pipeline fill and flush times can reduce utilization.
- Synchronization Overhead: Associated with each stage computation completion and transfer of results to the next stage there is a synchronization overhead. This includes delays attributable to clock skew, latching time, intentional padding (for clocked systems) and handshaking delay (for asynchronous systems). These delays have the effect of further reducing utilization and thus throughput.

If none of the above overhead constraints were present, then maximization of throughput would be achieved simply by maximizing the number of stages. When there are too many stages, however, the above problems are exacerbated and pipeline performance suffers. Thus, for each of the synchronization methodologies and technology parameters there is some optimal number of stages. The initial problem, therefore is to determine how performance varies with S for each of the design methodologies. Comparison of the two approaches can then be done in a straightforward manner.

In this paper we assume that ideal function/hardware partitioning is possible. Thus, the first constraint given above is not present. Dubey and Flynn [8] present an approach to extending the

clocked performance model to include hazards. We employ a variant of their approach and apply it to both our clocked and asynchronous analysis. The general argument in favor of using asynchronous methodologies rests on three ideas:

- Clock Skew: As clocked systems increase in size, clock skew increases and inevitably limits clock rate. The equivalent delay is not present in asynchronous systems (although other delays are present).
- Average versus Worst Case Times: The clock period in a synchronous system must be based on the worst case time for a "computational block". In asynchronous systems, however, average block delays may govern overall throughput rates. Further advantages to asynchronous designs may be possible when considering instruction pipelines where different instructions have differing average and worst case stage times.
- Design Issues: Hierarchical, modular design techniques are generally ill suited to handling global design constraints such as clock distribution. Asynchronous techniques permit one to focus on the functional and logical sequencing aspects of design and not on such global issues thus making the design task more manageable.

The models presented in this paper permit one to examine the first two ideas above in a quantitative fashion. The third idea, though intuitively appealing, is difficult to quantify and is not considered here.

Most pipelines found in commercial systems are clocked and can be modelled simply as shown as Figure 1. Each *computation block* is made up of combinational logic which performs some part of the overall function to be achieved. Following each computation block is a *latch* which, on the occurrence of a clock event, samples the outputs of the block and holds the sample at its own output until the next clock event occurs. A computation block and its latch constitute a stage or a segment of the pipeline. If t_{clk} is the clock cycle time for the pipeline, the throughput, G, for a fully utilized pipeline is $G = 1/t_{clk}$. However, if the pipeline stages are utilized on average only a fraction, u, of the time, the throughput is given by:

$$G = u/t_{clk} \tag{1}$$

Figure 2 shows the basic model of an asynchronous pipeline. At a functional level, the computation block is assumed to be the same as in the clocked case *. Now, however, the timing

* This assumption is made for simplicity of analysis. It will not hold in a number of situations where there may be no advantage, for example, in providing early completion logic when a clocked design is employed.

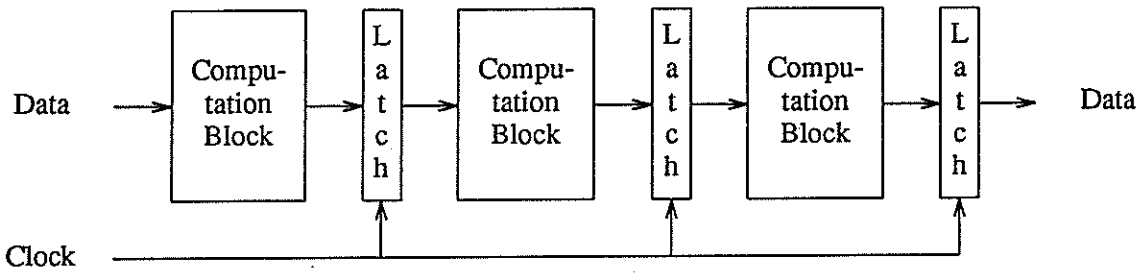

Figure 1. Basic clocked pipelined model

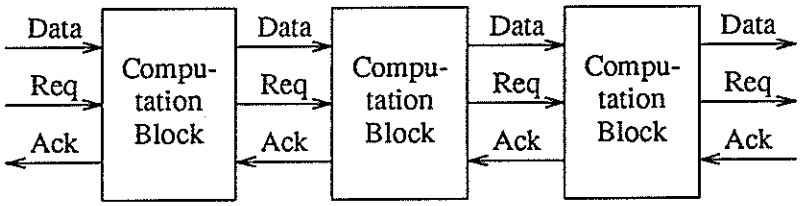

Figure 2. Basic asynchronous pipeline model

of data transitions between stages is controlled by a handshaking protocol. Computation blocks are *self-timed* so that a stage sends a *request* signal to its successor stage when computation is completed and data has been placed on the data lines. If the successor stage can accept the data an *acknowledge* signal is returned, and the job is "passed" to next stage in the pipe. Otherwise, the job waits in the current stage until the successor stage is empty. Though not shown here buffers, or multiple latches may be present between the computational blocks to help increase average computational block utilization and overall throughput. Finding the throughput of an asynchronous pipeline is difficult, because the pipeline is elastic in the sense that the time to get through a stage may be variable and dependent not only on the vacancy of succeeding stages, but on the actual data being processed in a given stage. In Section 4 stochastic models are used to find the approximate throughput for the asynchronous case.

To clarify the discussion that follows, the following terms are defined:

- **Stage Cycle Time:** The difference between the time that a job enters a stage and the time that it leaves the stage.
- **Computation Time:** The difference between the time that a job enters a computation block and the time that all of its computed outputs are stable.
- **Idle Time:** The time that a job waits, after computation has been completed, before it moves to the next stage. For a clocked pipeline, a job may have to wait until the next clock event. In the asynchronous case, the job may have to wait (i.e., is *blocked*) because its successor stage is still working on an uncompleted job.
- **Synchronization time:** Time used to synchronize two adjacent stages is called the synchronization time. For a clocked pipeline, it includes clock skew, latching time, and intentional padding delay. For an asynchronous pipeline, the synchronization time corresponds to the handshaking delay.

As shown in Table 1, for both clocked and asynchronous designs, a stage cycle time is composed of three non-overlapped components: computation time, idle time, and synchronization time. Certain design techniques can be used to overlap these components thus reducing the stage cycle time, however, such techniques are not considered here.

Table 1: Pipeline Stage Cycle and Its Elements

Methodology	Stage Cycle Time		
	Computation Time	Idle Time	Synchronization Time
Clocked	The actual amount of time used for functional computation.	Time used to wait for the next clock pulse.	Clocking (skew) and latching delay.
Asynchronous	As above.	Time used to wait for the availability of the next stage.	Handshaking delay.

The function to be pipelined is assumed to take a total of M gate levels in its nonpipelined version. When implemented in a pipeline it is taken to have N gates per stage. Thus, $M = S * N$. Gates are taken to be identical with each gate having a delay D whose minimum and maximum values are D_{min} and D_{max} respectively.

The computation time, t_{comp}, is randomly distributed with its maximum:

$$t_{comp-max} = N*D_{max} \quad (2)$$

For clocked pipelines its minimum is:

$$t_{comp-min-c} = N*D_{min} \quad (3)$$

For asynchronous pipelines, computation time may vary with the input data and early completion of the computation is possible in certain situations (e.g., ripple carry adders, comparators, etc.). To reflect the possibility of early completion the minimum computation time is defined in terms of a completion time parameter W ($0 \leq W \leq 1$) such that:

$$t_{comp-min-a} = W*t_{comp-min-c} = W*N*D_{min} \quad (4)$$

Table 2 contains a list of variable definitions.

3. Clocked Pipeline Model Development

To avoid race conditions and hazards, a given clocking scheme has associated with it a set of timing constraints on clock cycle time and pulse width. Cotten [4], Fawcett [6] and Kunkel and Smith [7] present constraints for single-phase designs. Multi-phase systems have also been considered [6,7,16]. In this development a single phase system is assumed although extentions

Table 2: Variable Definitions

Variable	Definition	
M	No. gate levels needed to implement the function	
S	No. pipeline stages	
N	No. gate levels per stage,	$N = M/S$
G	Pipeline throughput	
u	Pipeline utilization	
D_{max}	Max. prop. delay of a gate	
D_{min}	Min. prop. delay of a gate	
$t_{comp-max}$	Max. comp. time of a stage	$t_{comp-max} = N*D_{max}$
$t_{comp-min-c}$	Min. comp. time of a stage (clk. sys.)	$t_{comp-min-c} = N*D_{min}$
$t_{comp-min-a}$	Min. comp. time (asyn. sys.)	$t_{comp-min-a} = W*t_{comp-min-c}$
L	No. gate-delays for data latching	
W	Early completion parameter (asyn. sys.)	
t_{skew}	Clock skew (clk. sys.)	
t_{pad}	Padding delay (clk. sys.)	
t_{latch}	Latch delay (clk.sys.)	
t_{sync}	Synchronization delay	
k	Clock skew parameter	
p	Fraction of clk. tree levels requiring buffers	

to multi-phase systems is not difficult. Following the work cited above, three constraints are identified. The first constraint is that the clock period is longer than the maximum stage time. If $t_{latch-max}$ and t_{skew} are the maximum latching times and clock skew time, and t_{pad} is an intentional padding delay included by the designer to avoid race conditions (discussed further below) then:

$$t_{clk} \geq t_{comp-max} + t_{sync-max} = t_{comp-max} + t_{latch-max} + t_{skew} + t_{pad} \quad (5A)$$

The second constraint ensures that there is enough time for data to enter the latch properly. If $t_{clk-high}$ is the clock high pulse width, then:

$$t_{clk-high} \geq t_{latch-max} + t_{skew} \quad (6)$$

The third constraint prevents a race condition whereby the input to a stage is prevented from prematurly propagating to the next successor stage input.

$$t_{clk-high} \leq t_{comp-min-c} + t_{latch-min} \quad (7)$$

The last two constraints can be rearranged to form a constraint on the minimum computation time:

$$t_{comp-min} \geq t_{latch-max} - t_{latch-min} + t_{skew} \quad (8A)$$

If the minimum computation time is small (i.e., there are many stages, S is high, and with few gates per stage N is small) then it may be necessary to add a padding delay, t_{pad}, to the computation time to meet constraint (8A). With this intentional padding present, constraint (8A) becomes:

$$t_{comp-min} + t_{pad} \geq t_{latch-max} - t_{latch-min} + t_{skew} \quad (8B)$$

and

$$t_{pad} = \text{Max}\,[\,0,\, t_{latch-max} - t_{latch-min} + t_{skew} - t_{comp-min}\,] \quad (9)$$

Since we would like the clock period to be as short as possible the inequality in equation (5A) can be replaced with an equality. For clocked pipelines, clock cycle time is based on the worst case stage time over all stages in the pipe. The computation time part of this delay is given

in equation (2). The minimum computation time, needed for equation (9) is given in equation (3). If L is the number of equivalent gate delays associated with each latch in the pipeline, the maximum and minimum latch delays are $L*D_{max}$ and $L*D_{min}$ respectively and Equation (5A) can be rewritten as:

$$t_{clk} = D_{max} *(N + L) + t_{skew} + Max\ [0, L*(D_{max} - D_{min}) + t_{skew} - N*D_{min}] \qquad (5B)$$

3.1. Clock Skew

Four factors determine clock skew. They are:

- Differences in clock signal line lengths.
- Differences in line parameters (e.g., resistivity, etc.) that determine line time constant.
- Differences in delays through active elements inserted in the lines (e.g., clock buffers).
- Differences in threshold voltages of input latch devices.

Assuming that the number of stages, S, is a power of 2, the first source of skew can be eliminated by using a binary tree distribution of the sort shown in Figure 3. The skew associated with the remaining sources can be estimated as being some fraction (taken as .20) of the overall delay through the tree. This delay will be a function of:

- The number of tree levels ($\log_2 S$),
- The fraction of levels at which there is a driver present (p, $0 \le p \le 1$),
- Driver delay characteristics (D_{max}, D_{min}),

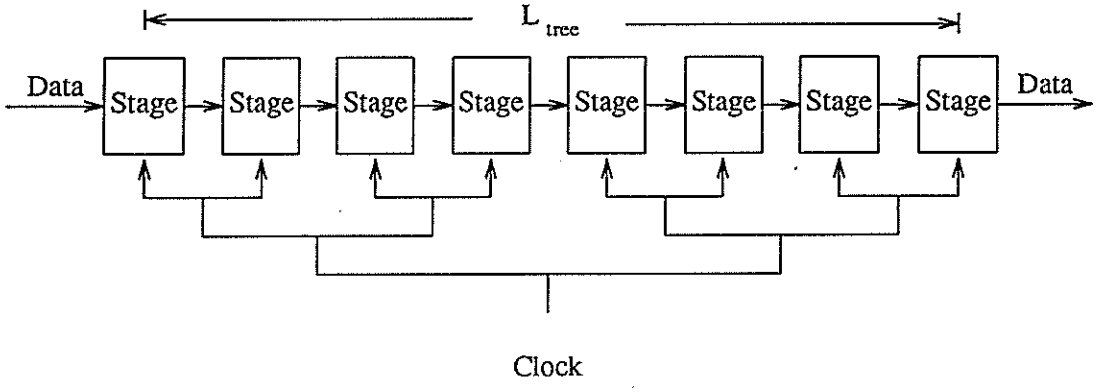

Figure 3. Clock distribution using a binary tree

- Technology factors (e.g., λ, resistivity, capacitance), and
- Implementation details (e.g., length of the tree along the leaf edge, L_{tree}).

The clock skew can be approximated by the simple model given below:

$$t_{skew} = k*(D_{max} - D_{min}) * (p*\log_2 S + 1) \quad (10)$$

To determine k, a Spice [18] program was developed and clock tree delays for various values of S were evaluated. This was done for $p = 0.5$, $D_{max} = 0.55$ns, $D_{min} = 0.45$ns, $2\lambda = 0.5\ \mu$, and $L_{tree} = 1$ cm and 2 cm. A least squares fit of equation (10) with the Spice results yielded a value of $k = 4.7$ for $L_{tree} = 1$ cm, and $k = 8.93$ for $L_{tree} = 2$ cm. For both these values of k, the model above is within 20% of the Spice results over the entire S range. Typical skew values for $S = 8$ are $t_{skew} = 1.2$ns and 2.2ns for $L_{tree} = 1$ cm, and 2 cm respectively [19]. Equation (10) can now be substituted into (5B) and the overall clock time evaluated.

3.2. Optimal Single-Phase Clocked Pipeline Design

The number of stages, S, is inversely proportional to the number of gate levels per stage, N. Thus, increasing the depth of a pipeline, decreases computation time per stage but increases clock skew and introduces intentional padding when N is small. Additionally, with more stages the effects of hazards and pipeline fill/flush times reduces performance. The growth of clock skew and padded delay, and the negative effects of hazards and fill/flush times diminishes the advantages associated with a deeper pipeline and reduced stage computation time.

The optimal number of stages, S, for a given function (e.g., number of gates levels) and technology parmeters, is found by maximizing the throughput G (equation 1). While the clock time, t_{clk}, is given in expression 5B, the utilization must still be determined. For a pipeline used principally for vector arithmetic operations, the pipeline utilization will be dependent on the vector length. In the simple situation where there is no chaining, data hazards can be ignored, and utilization will be dependent on the fill/flush times associated with the average vector length, v. In this situation, when setup times are negligible, utilization can be

expressed as:

$$u = v / (v + S - 1) \qquad (11)$$

As an example Figure 4 shows throughput as a function of number of stages for functions containing 64, 128 and 256 gate levels, vectors of average length 64, and L_{tree} values equal to 1 cm and 2 cm. As indicated, the optimum number of stages is about 16, 32 and 32 respectively (for $L_{tree} = 1$ cm) resulting in about 4 to 8 gates per stage, and clock rates of up to about 200 Mhz. This is similar to the results obtained by Kunkel and Smith for the case where vector loops are simulated [7]. As expected, the effect of a larger tree (1 to 2 cms) is to increase the clock skew and lower the throughput.

Performance analysis of instruction pipelines is more complicated. The problem is that there is little experimental data available on instruction pipeline utilization as a function of pipeline depth, and there are no simple and comprehensive theoretical models which can be used. Given this situation, an approximate utilization function (Figure 5) is developed based on

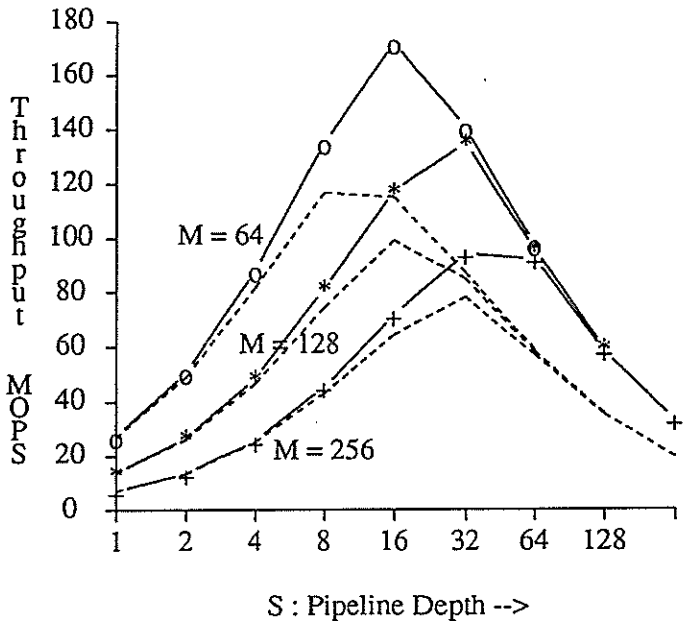

Figure 4: Throughput vs. Pipeline Depth (Clocked Arithmetic Pipeline)
(solid for $L_{tree} = 1$ cm, dashed for $L_{tree} = 2$ cm)

following notions:

- Utilization of a 1-stage processor of the DLX [2] type is 94% *.
- Simulation results indicate that the utilization of a 5-stage DLX processor (under the same assumptions as above) is 73%.
- Utilization will decrease with an increasing number of stages due to the difficulties associated with control and data hazards. That is, while hazards can be effectively controlled when the number of stages are small, this becomes increasingly difficult as S increases.
- Due to the above we take the utilization to be zero when $S \geq 32$.
- We assume that utilization decreases in a linear fashion between the three points S = 1, 5, and 32.

The resulting throughput is shown in Figure 6 which presents the variation in throughput as a function of the number of stages for M = 64,128 and 256 respectively, and for L_{tree} = 1 cm and 2 cm. Due to lower utilizations (compared with the arithmetic pipeline case), the optimum number of stages, S, is about 8 which roughly corresponds to the number being used in the

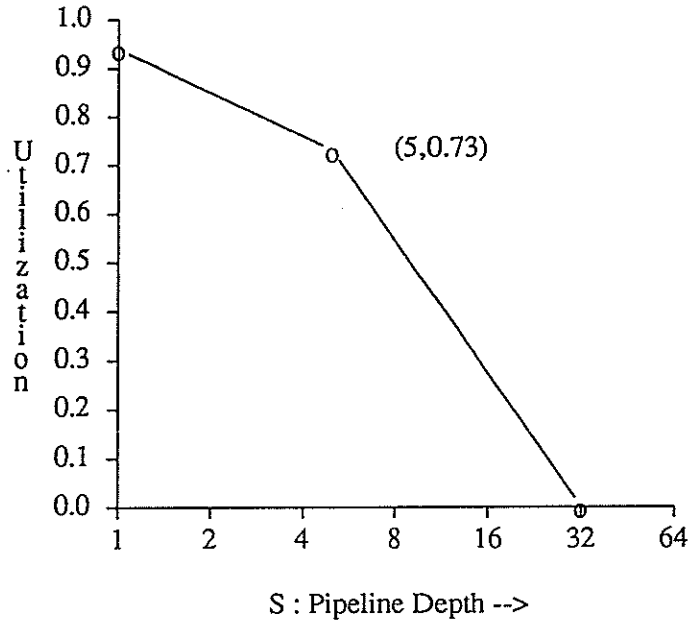

Figure 5: Instruction Pipeline Utilization

* This is under the assumption that there are no cache misses, 5% of the instructions are floating point, each non-floating point instruction takes 1 clock cycle, and each floating point instruction takes 2.3 clock cycles.

instruction pipelines of contemporary RISC processors (e.g., MIPS R4000) [20]. Clock rates between 80 and 160 Mhz would be required to achieve these throughputs. In the next section we develop equivalent expressions for the case of asynchronous pipelines.

4. Asynchronous Pipeline Model

The throughput of an asynchronous pipeline is difficult to calculate directly since one or more buffers may be present between adjacent stages and queueing delays may be introduced when a computation completes and finds the next stage busy, or the interstage buffers full. As indicated in Table 1, the stage cycle time can be expressed in terms of a computation time, an idle time, and a synchronization time.

$$\text{Stage Cycle Time} = \text{Computation time} + \text{Idle time} + \text{Synchronization time} \qquad (12)$$

To determine the components of (12), the design of the asynchronous pipeline must be considered. A number of approaches are available, however, two issues must be addressed with

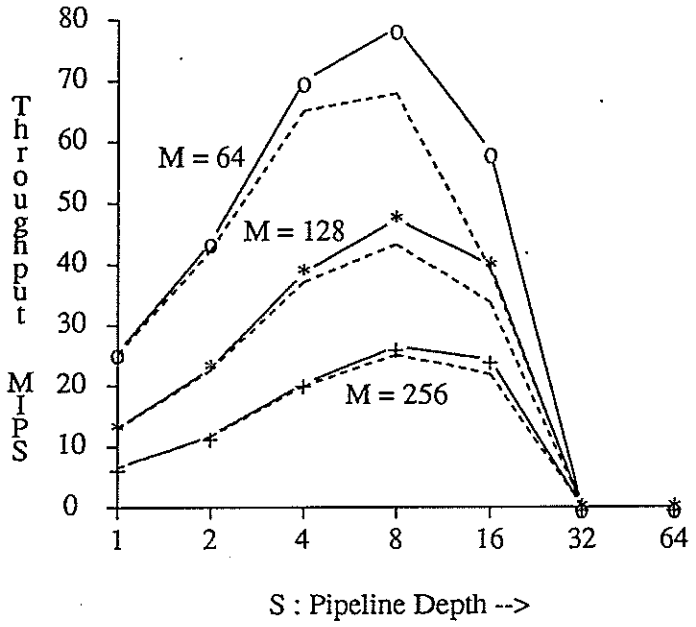

Figure 6: Throughput vs. Pipeline Depth (Clocked Instruction Pipeline)
(solid for $L_{tree} = 1$ cm, dashed for $L_{tree} = 2$ cm)

any approach. The first is the method employed in determining when the logic function at each stage has completed. This will determine the computation time. The second is the handshaking protocol employed to exchange information between adjacent stages. This will determine the synchronization time. The two designs considered in this paper are given in Table 3. The completion protocol and idle time considerations are described in this section with the handshaking protocol considered in a subsequent section.

Table 3: Asynchronous Designs

	FUNCTION COMPLETION PROTOCOL	INTERSTAGE EXCHANGE PROTOCOL
Method 1 (dr)	Double-rail encoding	4-Phase handshaking
Method 2 (de)	Parallel Delay-element	2-Phase handshaking

In double-rail designs [11,21], the computation block completion signal is formed using complementary output signals obtained from standard and complementary implementations of the logic. These two signals are fed into a Completion Detector (Figure 7) whose output is asserted when the computation block output and its complement output are either a (1, 0) or (0, 1). Assertion of the Completion Detector output indicates the computation has completed and the output of the computation block is ready and stable. The (1, 1) condition is forbidden while the (0, 0) condition is generally taken as a completion signal *reset* (i.e., completion detector signal set to 0). This serves to separate two successive computation block completions.

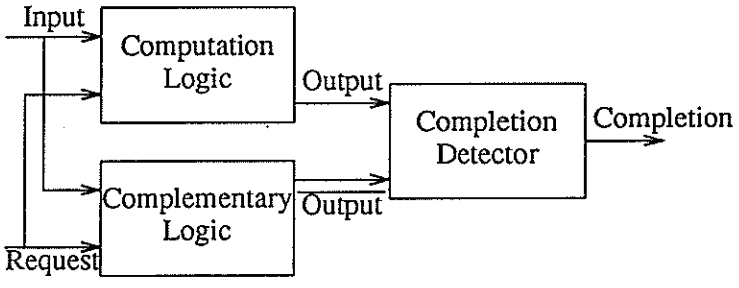

Figure 7: Double-rail Encoding Method

The principal drawback associated with double-rail encoding is the increase (approximate doubling) in the logic needed to generate both the standard output and its complement. The advantage of double-rail encoding is the speed increase associated with eliminating the clock period bound tied to $t_{comp-max}$ (equation 5A). The computation time is now close to the "real" computation block time.

The double-rail approach can also take advantage of differences in time which are related to input operands which may cause certain function evaluations to proceed faster than others. Earlier, (equation 4) the completion time parameter W ($0 \leq W \leq 1$) was introduced to reflect this possibility of early completion. The value of W for arithmetic pipelines will vary with the application. In later calculations we take $W = 0.8$.

For instruction pipelines, we can expect a large variance in stage completion times due to differing characteristics of various instructions. For example, an examination of the stage times associated with each of five instruction types associated with the DLX [2] instruction pipeline indicates that there will be instances (e.g., Execute stage time for the Branch instruction) where the stage time is effectively zero. Similar zero stage times can be found in the instruction pipelines of other processors. Thus, we take $W = 0$ for this case.

In the delay element approach, sometimes referred to as the micropipeline approach [22], a delay element is placed in parallel with the computation logic. The delay time is fixed to the maximum possible delay associated with the computation logic. Thus, when a signal exits the delay element, completion of the computation is guaranteed. The main advantage of this method over the double-rail approach is that it takes less logic (and chip area). It also does not require the separate development of the completion signal. However, since the delay is fixed at the maximum (worst case) computation delay, it is not possible to take advantage of lower delays which may occur due either to operand dependence or the frequent presence of non-worst case delays through the computation logic.

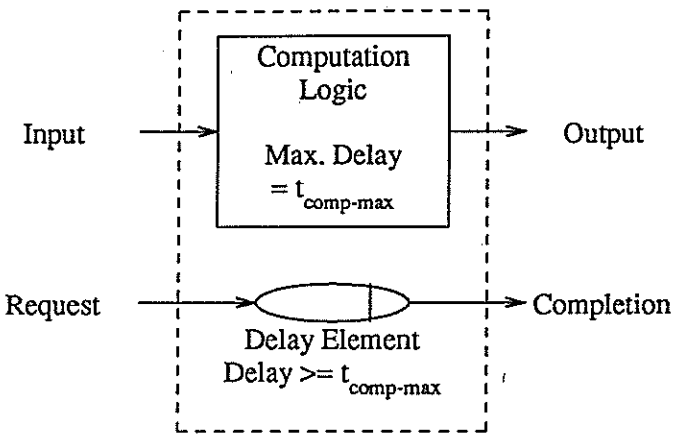

Figure 8: Delay Element Method

In the next section we assume the computation time for each stage is determined by the completion signal associated with either the double-rail (dr) or delay-element (de) approach discussed above. Since the function partitioning approach employed assigns an equal number of gates to each stage, the statistical properties of the stages are identical. This, along with some simplifying assumptions, permits us to establish bounds on the sum of the Computation Time and Idle Time.

4.1. Computation Time and Idle Time in the Asynchronous Pipeline

The expected value of the stage cycle time for a given stage i can be expressed as:

$$t_{s-i} = \mathbf{E}(t_{comp-i} + \text{Idle Time}_i + t_{sync-i}) \qquad (13)$$

Stage computation and idle times over the entire pipeline can be evaluated in terms of two extreme cases, one optimistic and one pessimistic. In the optimistic case two assumptions are made:

1. A buffer with infinite capacity is present between any two adjacent stages. Since there is always an empty buffer position, jobs leave each stage immediately after service and there is no idle time present.
2. A job is always available at the input to each stage. Thus, stages are immediately occupied after completion and departure of a job.

As a consequence of these assumptions, any synchronization time associated with a departing job may be overlapped with the computation time of the stage. At the same time, any synchronization time associated with a job entering a stage will be about half of the full synchronization time since the request part of the handshaking protocol is always high due to the constant availability of inputs. In this optimistic case the stage cycle time, t_{s-op}, over the entire pipeline of S stages can be expressed in terms of the Computation Time of a bottleneck stage (i.e., the stage whose mean time is maximum). For the double-rail case the resultant time, $t_{s-op-dr}$, is:

$$t_{s-op-dr} = \underset{i=1,..S}{\text{Max}}[E(t_{comp-i} + t_{sync}/2), E(t_{sync})] \quad (15)$$

Since the stages are identical, assuming that the computation times are uniform between $t_{comp-max}$ and $t_{comp-min-a}$:

$$t_{s-op-dr} = \text{Max}[(t_{comp-min-a} + t_{comp-max} + t_{sync})/2 , E(t_{sync})] \quad (16)$$

Substituting from equations 2 - 4 we obtain:

$$t_{s-op-dr} = \text{Max}[(NWD_{min} + ND_{max} + t_{sync})/2 , E(t_{sync})] \quad (17)$$

For the delay-element case, given the regular pipeline partitioning scheme used, the delay elements are set identically to the maximum stage computation time. Thus:

$$t_{s-op-de} = t_{comp-max} + E(t_{sync}) = ND_{max} + E(t_{sync}) \quad (18)$$

A second, pessimistic bound on the stage cycle time, t_{s-pe}, can be obtained by assuming that there is a global AND gate present which gathers completion signals from all stages. Assertion of this gate's output indicates that computation over all stages is complete. At this point the output of the AND gate enables each stage to pass its results to the succeeding stage and accept results from a prior stage thus starting the next cycle. Thus, every stage is in lock-step with the stage which has the maximum computation time. For the double-rail case, the average computation time used may be taken to be the mean of the maximum of computation times, over all blocks. This is added to the mean synchronization time which is the same for each stage.

Thus:

$$t_{s-pe-dr} = E\left[\max_{i=1...S}(t_{comp-i})\right] + E(t_{sync}) \quad (19)$$

Since all stages are uniformly distributed with identical statistical properties, the stage cycle time can be expressed as [23]:

$$t_{s-pe-dr} = (t_{comp-min-a} + St_{comp-max})/(1+S) + E(t_{sync}) \quad (20)$$

Substituting from equations 2 - 4:

$$t_{s-pe-dr} = N(WD_{min} + SD_{max})/(1+S) + E(t_{sync}) \quad (21)$$

For the delay-element case, since all the stages are identical and have the same maximum computational delay, the pessimistic case and the optimistic case are the same. That is, for the given set of assumptions, since identical delays are present, effectively no buffers are required and all the stages act in lock step. Thus, $t_{s-pe-de} = t_{s-op-de}$.

The stage cycle time, t_s, is between t_{s-op} and t_{s-pe}. For the double-rail case, if there is no buffer physically implemented in the asynchronous pipeline, then t_s will be close to the pessimistic approximation, $t_{s-pe-dr}$. On the other hand, our numerical experiments have shown that if there are buffers between all adjacent stages with capacity greater than 5, t_s is close to $t_{s-op-dr}$. Over the parameter values of interest the optimistic and pessimistic bounds are typically within about 20% of each other and the average of the two, $t_{s-avg-dr}$, is used as an approximation to t_s. As indicated above, for the delay-element case the optimistic and pessimistic cases are equal.

4.2. Synchronization Time in Asynchronous Pipelines

To evaluate the throughput we must next determine t_{sync} for the 4-phase (Figure 9) and 2-phase (Figure 10) handshaking protocols.

With 4-phase handshaking every control signal returns to zero at the end of each cycle, thus it is a natural approach in conjunction with double-rail encoding where the completion signal

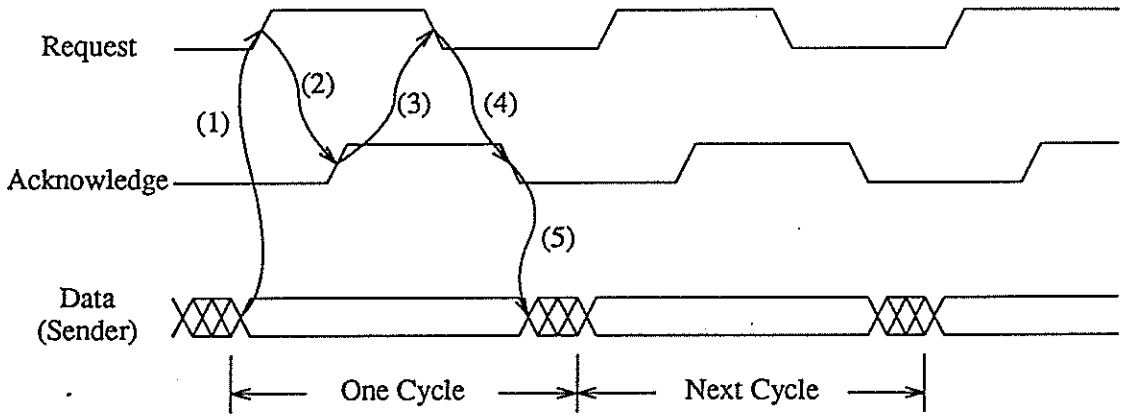

Figure 9: 4-Phase Handshaking Protocol

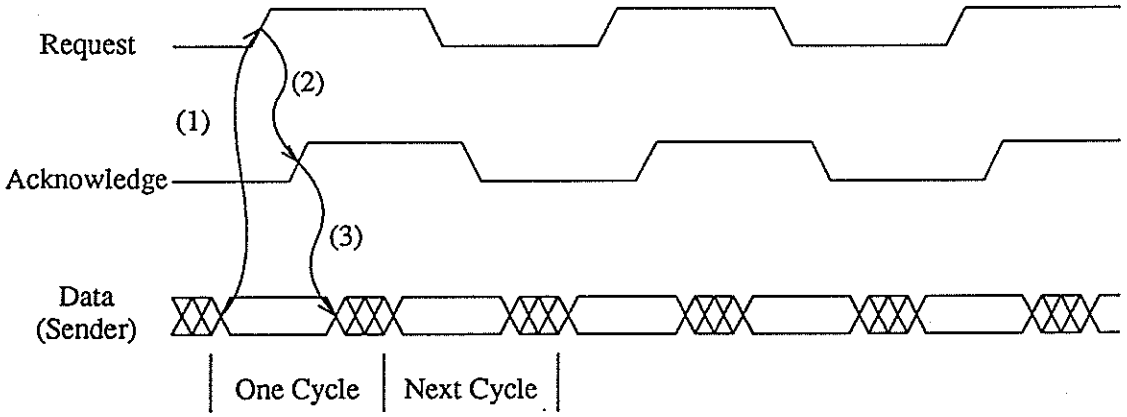

Figure 10: 2-Phase (Transition) Handshaking Protocol

also returns to zero (reset) in every cycle. However, the 4-phase handshaking has a relative high overhead since both rising and falling edges of the request and acknowledge signals exist in a cycle. The average handshaking overhead can be expressed in terms of the number of gate-delays, H, and the mean delay associated with each gate.

$$E(t_{sync}) = H(D_{max} + D_{min})/2 \qquad (22)$$

Examination of the circuitry required to implement this handshaking (and an integrated single buffer not shown in Figure 2) indicates that H is approximately 10. [19] The equivalent expression holds also for the 2-Phase protocol (also called transition signaling) except that only

two transitions are required. In addition, the logic associated with developing the double-rail completion signal is not needed. Thus, for the 2-Phase case H is approximately 4.

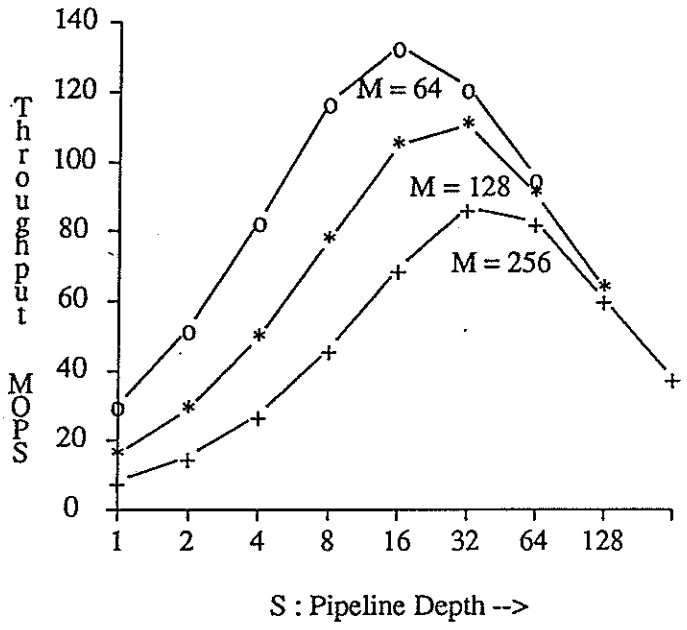

Figure 11: Throughput vs. Pipeline Depth
(Asyn. Arithmetic Pipeline: Double-Rail Design)

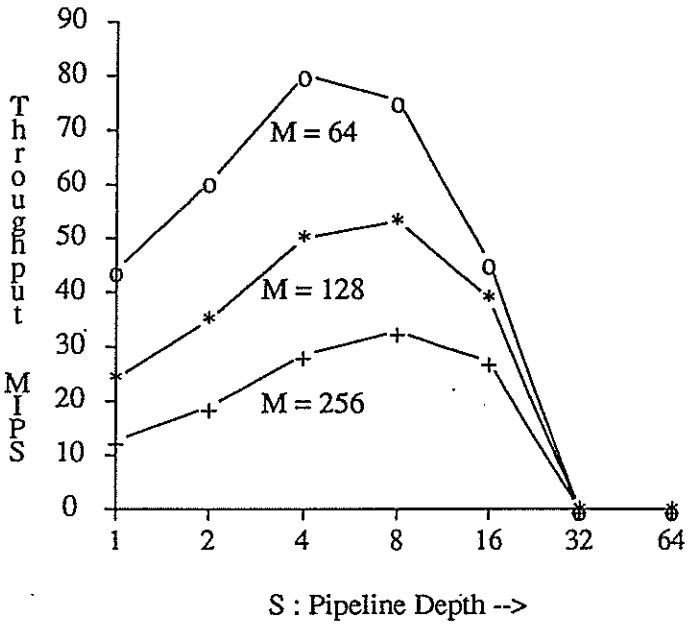

Figure 12: Throughput vs. Pipeline Depth
(Asyn. Instruction Pipeline: Double-Rail Design)

4.3. Optimal Asynchronous Pipeline Design

Unlike the clocked case, increasing the number of stages has no effect on the synchronization time, t_{sync}. As the number of stages increases, however, the number of gates per stage, N, decreases thus decreasing the computation time per stage and potentially increasing the throughput. Opposing this increase in throughput is the decrease in utilization with an increase in S. The throughput is:

$$G_{asy} = u/t_{s-avg} \tag{23}$$

where u is the utilization as given equation (11) for arithmetic pipelines, and Figure 5 for instruction pipelines. The throughput can now be evaluated by substituting for u and t_{s-avg}. The results are shown in Figures 11 through 14.

The overall shape of the throughput curves follow those obtained in the clocked case. This is to be expected since the shape is driven largely by the utilization curves which are the same in

both situations. Again the optimum value for S is lower in the instruction case than in the arithmetic pipeline case. The optimistic/pessimistic/average model approximations made in the arithmetic pipeline case are within about 12% of discrete event simulation of the system having a single buffer between stages. Simulation results for the double-rail instruction case for 5 stages is within 10% of the model results.

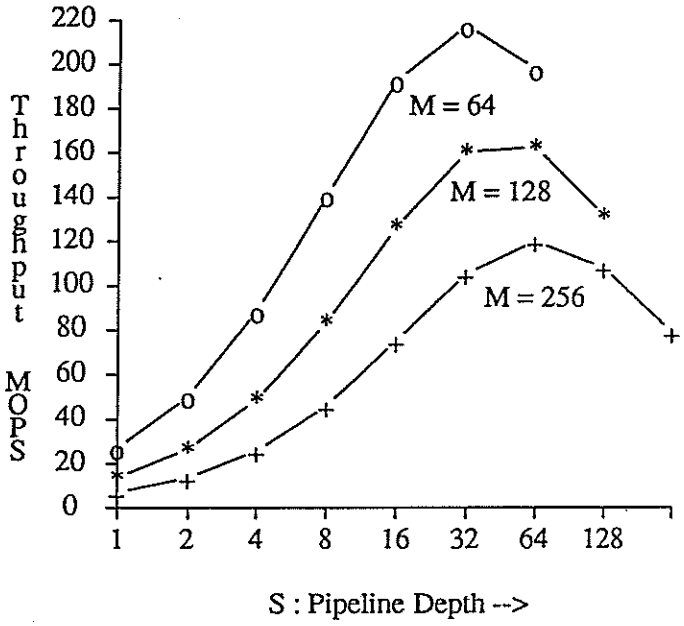

Figure 13: Throughput vs. Pipeline Depth
(Asyn. Arithmetic Pipeline: Delay-Element Design)

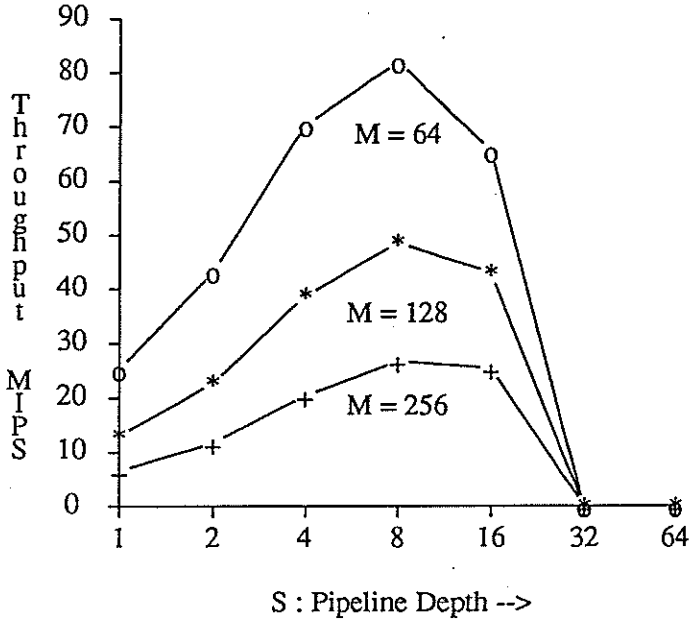

Figure 14: Throughput vs. Pipeline Depth
(Asyn. Instruction Pipeline: Delay-Element Design)

5. Summary and Conclusions

Several points can now be made concerning the relative performance of the clocked and asynchronous designs with the given model and parameter values. Tables 4 and 5 summarize the results shown in the prior figures along with information on pipeline throughput when there is no overhead present. For the clocked case this corresponds to $t_{skew} = t_{latch} = 0$ (nc table entries), while for the asynchronous case this corresponds to $t_{sync} = 0$ (ns table entries).

Table 4: Pipeline Performance Comparison (M = 64)

Method	Optimum S	Optimum Throughput	Gain (1 cm)	Gain (2 cm)
Arithmetic (64 elements)		(MOPS)		
Clocked (L_{tree} = 1cm)	16	172	1.00	1.47
Clocked (L_{tree} = 2cm)	8	117	0.68	1.00
Asynchronous: dr	16	133	0.77	1.14
Asynchronous: de	32	217	1.26	1.86
Asynchronous: ns - dr	64	1006	5.85	8.60
Asyn. ns-de/Clked. nc	64	916	5.33	7.83
Instruction		(MIPS)		
Clocked (L_{tree} = 1cm)	8	79	1.00	1.16
Clocked (L_{tree} = 2cm)	8	68	0.86	1.00
Asynchronous: dr	4	80	1.01	1.18
Asynchronous: de	8	82	1.04	1.21
Asynchronous: ns - dr	16	174	2.20	2.56
Asyn: ns-de/Clked: nc	16	125	1.58	1.84

First, we notice that the number of stages that maximizes the throughput for both the clocked and asynchronous designs are within about a factor of 2 of each other. This is due to the dominant role played by the utilization curves. Thus, the ability to effectively deal with data and control hazards may be more critical to achieving higher throughput than whether clocked or asynchronous designs are employed. This is also seen in the significantly higher throughputs associated with the arithmetic pipelines compared to the instruction pipelines.

Table 5: Pipeline Performance Comparison (M = 128)

Method	Optimum S	Optimum Throughput	Gain (1 cm)	Gain (2 cm)
Arithmetic (64 elements)		(MOPS)		
Clocked (L_{tree} = 1cm)	32	135	1.00	1.36
Clocked (L_{tree} = 2cm)	16	99	0.73	1.00
Asynchronous: dr	32	111	0.82	1.12
Asynchronous: de	64	163	1.21	1.65
Asynchronous: ns - dr	128	668	4.95	6.75
Asyn: ns-de/Clked: nc	128	609	4.51	6.15
Instruction		(MIPS)		
Clocked (L_{tree} = 1cm)	8	47	1.00	1.09
Clocked (L_{tree} = 2cm)	8	43	0.92	1.00
Asynchronous: dr	8	53	1.13	1.23
Asynchronous: de	8	49	1.04	1.14
Asynchronous: ns - dr	16	87	1.85	2.02
Asyn: ns-de/Clked: nc	16	63	1.34	1.47

Second, spanning the clock tree over larger distances ($L_{tree} = 1$ cm versus 2 cm) results in larger clock skews and lower throughputs. The throughput difference across clocked systems considered can be as much as about 50%.

Third, one of the two asynchrouous designs has higher throughput than the clocked design in all cases considered. Furthermore, at $L_{tree} = 2$ cm, all of the asynchronous designs beat the clocked design due to the increased clock skew. For instruction pipelines, however, the differences are negligible at 1 cm, and less than than 25% at 2 cm. More detailed models would be needed to verify differences at this level.

Fourth, in arithmetic pipelines, variance in computation time is taken to be moderate (W=0.8) and the double-rail asynchronous design cannot compensate for the large handshaking/latch overhead ($H = 10$) with early completion time reductions. Thus, it performs worse than the delay element design in these situations. For instruction pipelines, the double-rail design can offset this overhead and take advantage of high computation variance offered by different instruction types (W=0.0).

Finally, with the synchronization time, clock skew time and latch times set to zero, throughputs improve significantly over all designs. Note that the throughput for the clocked case with zero skew and latch times, and the asynchonous delay element case with zero synchronization time, are the same. That is, for this situation, the asynchronous pipeline acts in a lock-step fashion based on the worst case stage computational delay, and is thus equivalent to the clocked design. Over these alternative ideal designs, the double-rail design has the best performance. This is due to the fact that it alone can take advantage of early completion and non-worst case delays. Thus, the double-rail design ideally has the potential for about a doubling in throughput for the instruction pipeline, and about a factor of five improvement for the arithmetic pipeline case. To achieve this potential, design must be pursued where the handshaking overhead is significantly reduced or overlapped with part of the stage computation time.

REFERENCES

1. P. M. Kogge, *The Architecture of Pipelined Computers*, Hemisphere Publishing Corporation, New York, NY (1981).
2. J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, (Chapter 6: Pipelining)* , Morgan Kaufmann Publishers, Palo Alto, CA (1990).
3. H. S. Stone, *High-Performance Computer Architecture (Chapter 3: Pipeline Design Techniques)*, Addison-Wesley, Reading, MA (1990).
4. L. W. Cotten, "Circuit Implementation of High-Speed Pipeline Systems," *Proc. AFIPS - Fall Joint Computer Conference*, pp. 489-504 (1965).
5. T. G. Hallin and M. J. Flynn, "Pipelining of Arithmetic Functions," *IEEE Trans. Comput.* **C-21,8** pp. 880-886 (August 1972).
6. B.K. Fawcett, "Maximal Clocking Rates for Pipelined Digital Systems," *M.S. Thesis, Dept. of Elect. Engr., Univ. of Ill. Urbana-Champaign*, (1975).
7. S. R. Kunkel and J. E. Smith, "Optimal Pipelining in Supercomputers," *13th Inter. Symp. Comput. Arch.*, pp. 404-411 (June 1986).
8. P.K. Dubey and M.J. Flynn, "Optimal Pipelining," *Jrnl. of Parallel & Distributed Computing* **8** pp. 10-19 (1990).
9. W.A. Clark, et.al., "Macromodular computer systems (and several other papers on macromodules)," *Proc. AFIPS Spring Joint Compt. Conf.* 30(1967).
10. E. Sadeh and M.A. Franklin, "Monte Carlo Solution of Partial Differential Equations by Special Purpose Digital Computer," *IEEE Trans. of Computers* **C-23**(4) pp. 389-397 (April 1974).
11. T. H. Meng, *Synchronization Design for Digital Systems*, Kluwer Academic Publishers, Norwell, MA (1991).
12. T.H.-Y. Meng, R.W. Brodersen, and D.G. Messerschmitt, "Asynchronous Design for Programmable Digital Signal Processors," *IEEE Trans. on ASSP*, (April 1991).
13. A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The Design of an Asynchronous Microprocessor," *Proc. Decennial Caltech Conf. on VLSI (MIT Press)*, pp. 20-22 (March 1989).
14. A. J. Martin, S. M. Burns, T. K. Lee, D. Borkovic, and P. J. Hazewindus, "The First Asynchronous Microprocessor: The Test Results," *Comput. Arch. News* **17,4** pp. 95-110 (June 1989).
15. R. Ginosar and N. Michell, "On the Potential of Asynchronous Pipelined Processors," *Comput. Arch. News* **18,4** pp. 27-34 (Dec. 1990).
16. D. F. Wann and M. A. Franklin, "Asynchronous and Clocked Control Structures for VLSI Based Interconnection Networks," *IEEE Trans. Comput.* **C-32,3** pp. 284-293 (March 1983).
17. S.Y. Kung and R.J. Gal-Ezer, "Synchronous versus asynchronous computation in VLSI array processors ," *Proc. SPIE Int. Soc. Opt. Eng.* 341(1982).
18. L.W. Nagel, "SPICE2: A Computer Program to Simulate Semiconductor Circuits," *ERL Memo. ERL-M520, Electronics. Res. Lab., Univ. Cal., Berkeley, CA.*, (May 1975).
19. Mark A. Franklin and Tienyo Pan, "Synchronization Delay in Pipelined Systems," *Washington Univ., Comput. & Commm. Res. Ctr., Tech., Rpt. #92-08*, (Oct. 1992).
20. S. Mirapuri, M. Woodacre, and N. Vasseghi, "The Mips R4000 Processor," *IEEE Micro* **12,2** pp. 10-22 (April 1992).
21. C. Mead and L. Conway, *Chap. 7, Introduction to VLSI Systems*, Addison-Wesley, Reading, MA (1980).
22. I. E. Sutherland, "Micropipelines," *Commun. ACM* 32(6) pp. 720-738 (June 1989).
23. M.Y. Kim and A.N. Tantawi, "Asynchronous Disk Interleaving: Approximating Access Delays," *IEEE Trans. Comput.* 40(7) pp. 801-809 (July 1991).