ARAS: Asynchronous RISC Architecture Simulator[1]

Chia-Hsing Chien, Mark A. Franklin, Tienyo Pan[2] , and Prithvi Prabhu

Computer and Communications Research Center
Washington University
St. Louis, Missouri, 63130-4899
U.S.A.

Abstract

In this paper, an asynchronous pipeline instruction simulator, ARAS is presented. With this simulator, one can design selected instruction pipelines and check their performance. Performance measurements of the pipeline configuration are obtained by simulating the execution of benchmark programs on the machine architectures developed. Depending on the simulation results obtained by using ARAS, the pipeline configuration can be altered to improve its performance. Thus, one can explore the design space of asynchronous pipeline architectures.

1 Introduction

This paper presents a graphic simulation tool, ARAS (Asynchronous RISC Architecture Simulator), which has been developed to ease in the modeling, visualization and performance evaluation of asynchronous instruction pipelines. The paper has three objectives:

- To present the essential elements of the ARAS modeling tool.
- To show how ARAS can model instruction pipelines.
- To demonstrate how ARAS may be used to obtain important performance data which can then guide the design of asynchronous instruction pipelines.

Early RISC machines often employed a single, 4 or 5-stage pipeline [9, 6]. In more advanced machines, performance has been improved by increasing the pipeline depth (i.e., *superpipelined*) and by employing multiple pipelines (i.e., *superscalar*). Superpipelined techniques divide each instruction into finer segments thus reducing the cycle time. Superscalar techniques, on the other hand, issue several instructions into parallel pipelines and thus increase the average number of instructions being processed per cycle. In this paper, instruction pipelines are modeled using the ARAS simulation tool which permits visualization of pipeline operation and the collection of pipeline performance data.

Most commercial machines are currently clocked, however, asynchronous design has attracted more attention in recent years as clock rates and power levels have increased. Although asynchronous methodology currently requires more chip area, generally entails higher design complexity, and does not have a large base of available design automation tools, there are certain potential advantages. Among these are having performance governed by mean versus worst case function delays, eliminating limitations associated with clock skew (although introducing other limitations), and having potentially lower power levels. The performance advantages associated with asynchronous systems are discussed in more detail in [3]. While the use of asynchronous modules in the design of processors goes back to the late 1960s with work at Washington University (St. Louis) [2], it wasn't until 1988 that an entire asynchronous microprocessor was designed at the California Institute of Technology [10]. Later an asynchronous version of the ARM processor, AMULET1, was completed at University of Manchester (UK) [5]. Currently, SUN Microsystems is developing an asynchronous microprocessor called Counterflow Pipeline Processor (CFPP) [11] and it appears that several other institutions and companies are also investigating the design of asynchronous machines [8].

Figure 1 shows a typical 5-stage instruction pipeline with several buffers between each stage. Analytic modeling of such a system is not easily accomplished for the following reasons:

[1]This research has been funded in part by NSF Grant CCR-9021041 and ARPA Contract DABT-93-C0057.

[2]Currently with ITRI: Industrial Technology Research Institute, Taiwan, Republic of China.

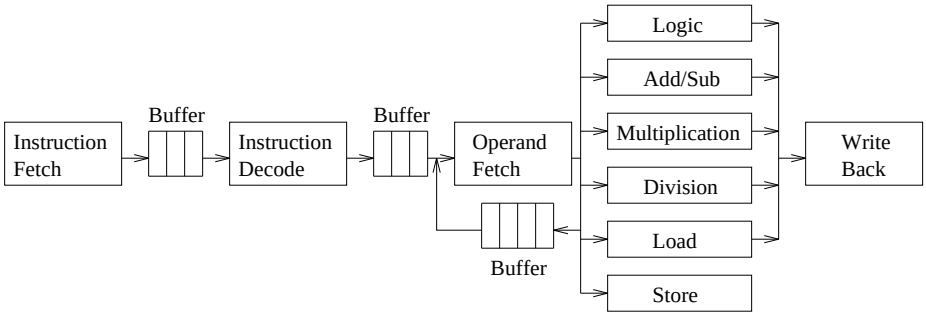

Figure 1: A 5-stage pipeline with multiple buffers

- The service rates associated with each stage are not easily quantifiable since they reflect the processing requirements of each instruction type.
- To achieve a realistic pipeline arrival distribution it is best to model the arrival process with real workload instruction traces. Such traces generally can not be modeled analytically in a tractable manner.
- The basic queueing system associated with a realistic instruction pipeline model has a host of logical exception conditions associated with eliminating hazards, result forwarding, etc. These are not easily captured in analytic models.

In this paper, Section 2 illustrates how the ARAS display helps in visualization of the pipeline's operation and also discusses the data collection facilities. In Section 3, the basic ARAS operation is briefly considered. Section 4 considers the problems associated with insuring that the ARAS pipeline presented by the user can be implemented and indicates the constraints associated with pipeline construction. Section 5 illustrates three uses of ARAS. The first is a pipeline design experiment, the second and third consider the effects of adder design and handshaking delays on pipeline performance. Conclusions and suggestions for future research and modifications to ARAS are presented in Section 6.

2 Using ARAS

ARAS simulates the instruction pipeline of a processor executing a (benchmark) program and then evaluates processor performance. Figure 2 is an example of an ARAS display. Construction of the display and driving program is considered later.

Each rectangle in the display represents a block of micro-operations associated with a stage in a processor's instruction pipeline. Lines between blocks represent paths that instructions may take during execution. Figure 2 illustrates a 4-stage instruction pipeline. The pipeline begins with two parallel instruction fetch blocks, both of which connect to a shared instruction decode block. The two fetch blocks represent the possible parallel fetching of instructions. The third stage has two different, but parallel blocks to handle branch and operand fetch operations respectively. In certain situations this will permit further instruction parallelism. Finally, three different but parallel blocks are present. This permits two ALU register based instructions to proceed in parallel while a memory access instruction (e.g., Load) is also taking place.

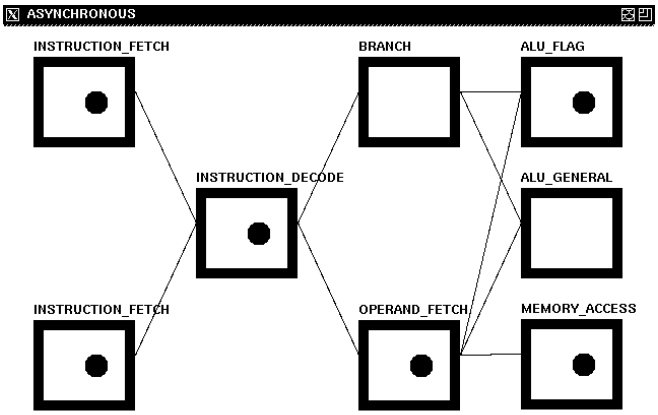

Figure 2: ARAS Display

Visualization of pipeline operation and dynamics employs the following main techniques:

- The execution and movement of instructions through the pipeline corresponds to the movement of dots from one block to another. The presence of a dot in a particular block indicates that an instruction is being processed in that block; otherwise, the block is empty.
- At the time an instruction moves from one block

to another, the line between the blocks involved momentarily thickens.

- The color of the dot (not shown in the black and white figure) corresponds to the instruction type being executed (e.g., arithmetic, logic, branch, etc.).
- The border of each block is color coded to reflect changes in block status. A block may be idle, busy, or blocked. Changes in border color reflect changes in block status.

Since there is no clock in this system, movement of dots between blocks is governed by the availability of blocks and input instructions. As indicated, a dot (instruction) may be blocked in a particular block if the successor block that is required by the instruction is busy processing another instruction. Thus, the movement of the dots, the temporary thickening of block interconnect lines, and the color changes (dot or border) together, allow a designer to visualize the progress of instructions through the asynchronous pipeline. Instructions flow from left to right, finishing (in this example) on completion of the fourth stage (or sometimes earlier when a branch is encountered).

In addition to the dynamic visual display of asynchronous instruction processing, a designer can gather both global system and local block performance results. Global system performance includes: system throughput, the number of processed instructions, and the simulation time required for execution of the processed instructions. The local performance of any given block can also be obtained. These include: the number of instructions processed, throughput, and the percent of idle, working and blocked time associated with the selected block.

Using these global and local results, a designer can attempt to improve pipeline performance by modifying and restructuring the pipeline. For example, pipeline performance is generally improved if each stage takes roughly equal time. Thus, the designer can examine the effect of moving various micro-operations between blocks (done by editing a configuration data file), rerunning ARAS, and comparing the various results. Technical report [1] provides procedures to be followed for altering the pipeline configuration data files.

3 Basic ARAS Operation

3.1 Discrete-event simulation

The core of ARAS is a standard trace-driven discrete-event simulator. After ARAS has accessed the configuration file for a particular pipeline, a block

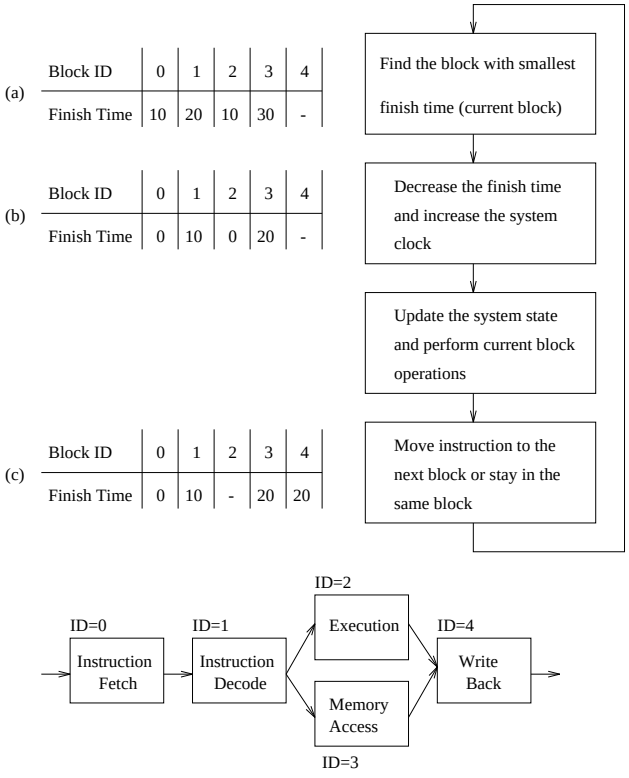

Figure 3: The steps to process an event in ARAS

table is created. This table is used to schedule events (e.g., the completion of an instruction's usage of a block) generated by the pipeline's operation. As indicated earilier, a block can be in one of three possible states: idle, busy, or blocked. A block in an idle state is available to process an instruction. The busy state denotes that the block is currently processing a portion of the instruction's operation. The blocked state occurs when an instruction is prevented from moving into the next block in the pipeline because its successor block is busy or because there exists a data dependency which is unresolved. An event occurs in ARAS principally when a block completes its processing of an instruction and changes state from busy to either idle, blocked, or busy (for the next instruction). Once an event has been processed, other blocks may change their states triggering yet other events.

Event scheduling is depicted in Figure 3 where the configuration of an example pipeline is shown. The block table lists the blocks using their unique identification numbers and indicates the finish time associated with each block. Whenever an instruction enters a block, a finish time for the block is calculated based on the instruction type and operation(s) to be performed by the block. Block table (a) of Figure 3

shows the allocated finish times at a point in the simulation. All the blocks have finish times except for block 4, which is idle.

ARAS proceeds through the simulation by selecting the busy block with the smallest finish time (scanning from right to left). The finish time of the selected block (the "current block") is added to the global simulation clock and is also used to decrement the finish time of all other busy blocks in the table. Next ARAS performs the operations associated with processing of the instruction in the "current block" (e.g., updating registers and tables, changing block states, etc.). It now checks to see if the "current block" instruction can be moved to the next block it requires. With every movement, all other blocks are checked to see if the event permits the further movement of blocked instructions.

The procedures described above continue, with a new block now being selected as the "current block." The simulation ends when the entire trace program has been processed.

3.2 Driving ARAS

The instruction traces used to drive ARAS are obtained by collecting execution information for programs running on a SPARC computer. The program to be traced is first executed on a SPARC in single step mode under control of the debugger and an instruction trace is collected. The execution information collected is sent through an interpreter which puts it into a form useful to ARAS. It is placed in a part of ARAS memory corresponding to the source segment. Data and stack segments are also created for ARAS.

Standard benchmark traces are being developed for users interested in experimenting with various pipeline designs. For users interested in developing their own traces, details on how to use the interpreter programs are described in technical report [1].

4 Pipeline Design Constraints

There are some basic constraints on the use of blocks in the design of instruction pipelines. This section discusses these constraints, an associated set of rules, and the way ARAS handles them.

The ARAS instruction set is derived from the SPARC instruction set. Each instruction can be divided into a number of micro-operations (initially based on DLX micro-operations [6]), some of which may be common for all or a group of instructions. Prior to defining pipeline blocks, the instruction state diagram which indicates sequencing of micro-operations must be constructed. A simple example is shown in Figure 4 where certain ALU and Jump instructions are illustrated [6].

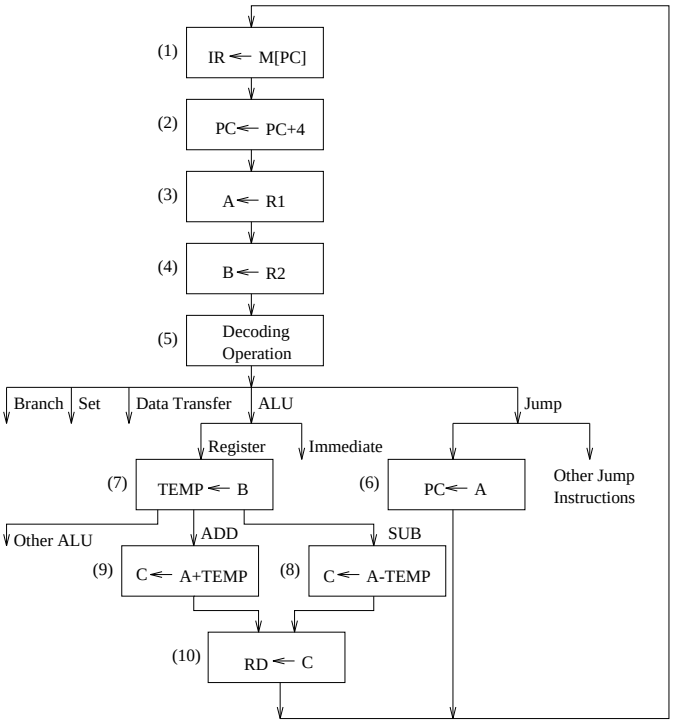

Figure 4: A Typical Finite State Diagram

In general, a pipeline block may consist of a single micro-operation, or a group of sequential micro-operations as defined in the state diagram. While the entire set of micro-operations available to the user does not correspond to the actual micro-operations associated with any given SPARC implementation, the basic micro-operations needed for execution of any SPARC instruction are present. Currently, due to implementation limitations, ARAS does not handle arbitrary assignments of micro-operations to blocks. However, a set of 40 predefined pipeline blocks (with their associated micro-operations) are available to the user to explore a variety of pipeline designs. Work is under way to extend this capability and to develop a more flexible micro-operation to block assignment mechanism.

Figure 4 shows three instructions (ADD R1, R2, Rd: SUB R1, R2, Rd: and JR R1) and the path they must take through the finite state diagram in order to be executed correctly. Figure 5 shows how each individual state represents a micro-operation and how these micro-operations can grouped together to form blocks which constitute stages in the pipeline.

As a simple example, consider the implementation of the Instruction Fetch (IF) operation. One approach is to define a single IF block which performs both the

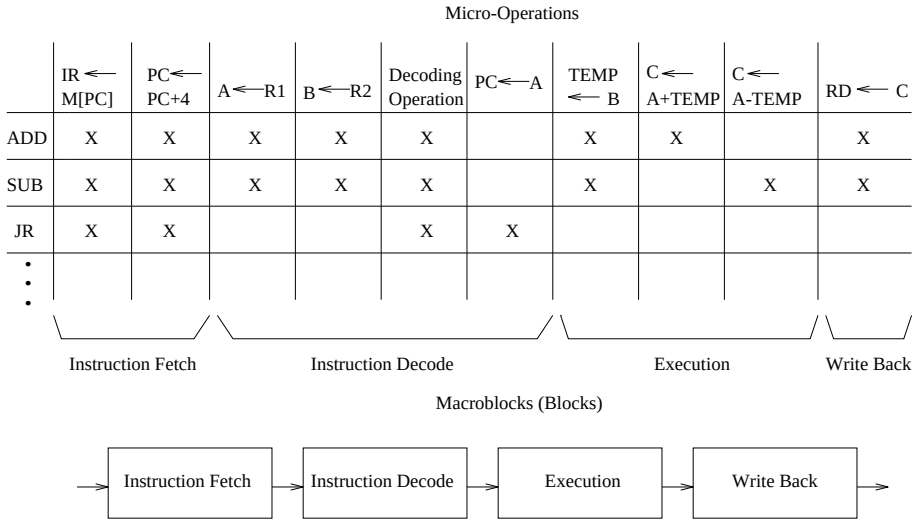

Figure 5: Synthesis of Micro-Operations and Macroblocks

instruction fetch (micro-operation 1 in Figure 4) and the program counter (PC) update (micro-operation 2) micro-operations. Another approach is to have two separate sequential IF blocks, the first performing the instruction fetch, and the second handling the PC update. Block types are available to explore both approaches.

In designing a working pipelined machine, general rules apply:

- The micro-operations constituting an instruction should be executed in the proper sequence, and this should be true for every instruction and possible instruction sequence.
- A pipelined machine should be able to handle control hazards which occur due to branching.
- A pipelined machine should be able to handle resource hazards that may arise when multiple instructions request the same execution resource (e.g., adder).
- A pipelined machine should be able to handle data hazards and ensure that instructions are executed with their proper operands.

The first rule is implemented by the user in defining blocks and pipelines that maintain the operation sequence defined by each instruction's state diagram. For example, suppose that the write back micro-operation (RD←C) is placed prior to the execution micro-operations performing addition and subtraction, (C←A+TEMP and C←A-TEMP). This clearly would result in incorrect instruction execution since the C register would be written prior to being updated. ARAS users are provided with a table (similar to Figure 5) which gives the sequence of micro-operations to be followed. The table also indicates which micro-operations can be performed in parallel (provided there are no resource hazards).

The second rule is implemented by requiring that all instructions are in-order up to and including the block or blocks which include instruction decoding, branch address calculation, and branch condition evaluation. Out-of-order execution is permitted after these micro-operations have been performed. If a branch is detected when the condition evaluation takes place, only instructions earlier in the pipeline need to be flushed. This ensures that there will be no control hazards since the instructions which have entered the pipeline after the branch will not have changed any register states before they are flushed.

The third rule is enforced by the handshaking protocol inherent in the asynchronous design. That is, an instruction requesting resources (other blocks) that are busy will be automatically stalled in the current block until an acknowledgement is received.

ARAS handles data hazards by using a standard scoreboarding technique [6]. This includes checking the register reservation table at the start of the operand fetch micro-operation to ensure that it has not been reserved by a prior instruction, and stalling at this block if the register has been reserved. In addition, an instruction which will write to a register, must reserve that register when it is in the operand fetch

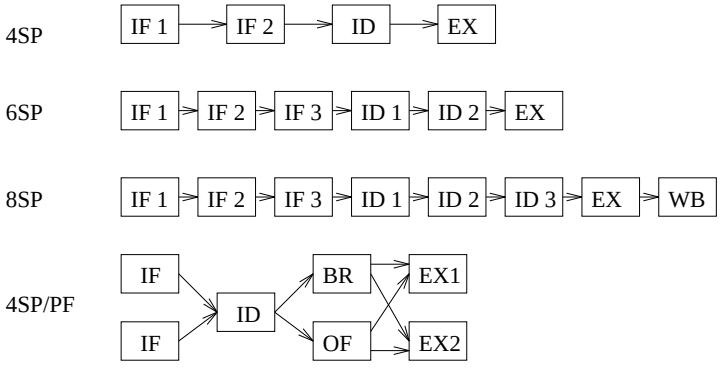

Figure 6: Pipeline Configurations Analyzed Using ARAS

block. Later, when the write back micro-operation has completed, this same instruction must cancel the register reservation. This ensures that register operations are all in-order.

5 Performance Evaluation and Design

This section illustrates three uses of ARAS. First, the architecture of a pipelined machine is developed by using ARAS. Second, the effect of alternative module (in this case adder) designs on pipeline performance is presented. Third, the influence of a particular design parameter (handshaking delay) on performance is considered.

5.1 Design an Asynchronous Pipeline with ARAS

Consider an initial pipeline which has three basic stages, an Instruction Fetch stage, an Instruction Decode stage and an Execute stage. Assume that Write Back operations may be performed in the Execute stage. The Execute stage also includes the Memory Access micro-operations.

One approach to developing higher performance machines is to attempt to lengthen the instruction pipeline. Various design alternatives are available and the issue is to find a design which yields the highest instruction throughput. Preliminary executions indicated that in a three stage pipeline, the Instruction Fetch stage was a bottleneck. It was therefore divided into two stages[3] and the four stage pipeline of Figure 6 was examined.

[3]We are not concerned here whether or not this can be achieved in an actual hardware implementation. If the results show significant improvement, then such an implementation can be explored.

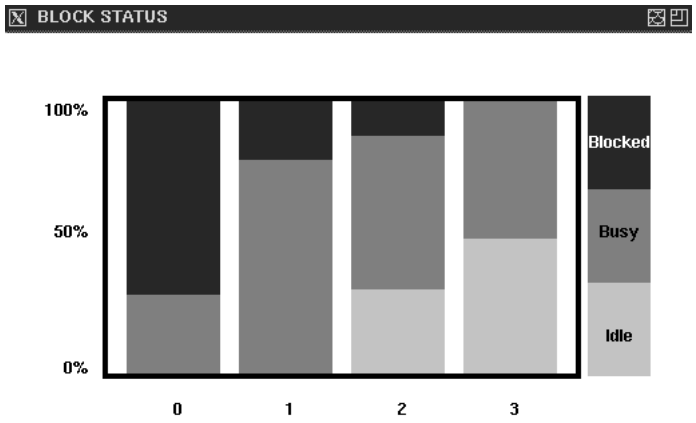

Figure 7: Block Status Display

A set of benchmark programs (see Table 1) was executed on the four stage pipelined machine and simulation results were then used to identify blocks in this pipeline which were performance bottlenecks.

Block status information can be obtained both visually (using the ARAS display) and by the ARAS output at the end of the simulation. Figure 7 shows the visualization of the block status in the ARAS display. With this display option the user can get information about both the dynamic and the final block status. The figure shows the final block status for a 4-stage pipeline machine for one of the benchmark programs.

The micro-operations for those stages identified as performance bottlenecks were redistributed over a greater number of blocks. This resulted in more blocks (increasing the pipeline depth) with a fewer number of micro-operations per block (reducing block delay) and enabled us to find configurations which yielded

Table 1: Benchmark Programs and Instruction Counts

Program Name	Description	Instruction Type and Counts			
		ALU	Branch	Memory	Total
Qsort	quick sort of 30 randomly generated numbers	11833	3508	10893	26234
Sieve	calculation of prime numbers up to 100	20462	8393	3235	32090
Dsim	discrete event simulation (first 121 iterations)	22321	5715	12442	40478
Percent		55.3%	17.8%	26.9%	100%

Table 2: Simulation Results for the 4SP Pipeline

Block State	IF1	IF2	ID	EX
Idle	0	5.4%	7.6%	40.2%
Busy	61.2%	68.2%	90.9%	59.8%
Blocked	38.8%	26.4%	1.4%	0

Table 3: Simulation Results (MIPS)

Model	Qsort	Sieve	Dsim	Harmonic Mean
4SP	57.1	71.2	63.4	63.8
6SP	65.7	76.8	70.5	71.0
8SP	62.8	74.7	67.4	68.2
4SP/PF	69.2	85.3	76.1	76.8

higher throughput. There are a large number of ways in which the blocks may be divided, however, using simple heuristic approaches in conjunction with ARAS it was not difficult to analyze various configurations.

Table 1 gives details about the three benchmark programs used: their instruction counts and the total number of ALU, branch and memory access instructions for each program. Qsort is a program which uses the quick sort algorithm to sort thirty numbers. Sieve is a program which uses Erastothenes' sieve method to obtain all prime numbers less than hundred. Dsim is a discrete event simulator which performs a hundred and twenty one iterations. Table 2 shows the percentage of idle, busy and blocked time for the 4-Stage Pipeline (4SP) shown in Figure 6. The benchmark programs discussed above were used to obtain this data. The simulation results (Table 2) show that the ID stage was busy for 90.94% of the time indicating that it is a performance bottleneck. This is reinforced by the high percentage of time that the IF stages are blocked. The user of ARAS could also see this through the visualization of the pipeline, noting that the ID block was almost always busy. Note also that the IF1 stage is never idle since instructions are always available to be processed.

One approach to reducing this bottleneck is to divide the micro-operations associated with the ID block into two blocks, ID1 and ID2. In addition, since the IF blocks are busier than the EX block, further increase in performance may result from dividing the micro-operations associated with the instruction fetch stage into three blocks, IF1, IF2 and IF3. Simulation results (Table 3) for this 6-Stage Pipeline (6SP) show an 11% performance improvement over the four stage case.

An 8-Stage Pipeline (8SP) was then constructed with the ID micro-operations now divided across three blocks, and the EX block divided into an EX and a separate WB (Write Back) block. The results of that simulation are shown in Table 4. Performance for this case, however, while higher than the 4-stage pipeline, was lower than the 6-stage pipeline. This results from a combination of factors. First, as the number of pipeline stages increases, the average cycle time per stage decreases. Handshaking delays (discussed later) thus become a larger percentage of the stage cycle time and establish a lower bound on this time. In addition, with longer pipelines, the role of hazards increases since both the probabilities of a hazard and the pipeline stall penalties associated with a hazard increase. The combination of these factors results in a lower throughput [3].

Another approach to increasing performance, in addition to dividing the blocks, is to alter the pipeline configuration to take advantage of instruction level parallelism. An example of this is shown in the 4SP/PF case of Figure 6 which illustrates a simple superscalar configuration. In this case, branch instructions flow through the BR (Branch) block, while other instructions proceed through the OF (Operand Fetch) block. Since there are multiple IF and EX blocks, instruction level parallelism is present thus potentially

Table 4: Simulation Results for the 8SP Pipeline

Block State	IF1	IF2	IF3	ID1	ID2	ID3	EX	WB
Idle	0	4.6%	10.8%	14.4%	26.2%	24.3%	48.8%	66.1%
Busy	55.4%	66.9%	61.1%	55.2%	33.1%	72.3%	50.8%	33.9%
Blocked	44.6%	28.5%	28.1%	30.4%	40.7%	3.4%	0.4%	0

Table 5: Simulation Results for the 4SP/PF Pipeline

Block State	IF(Up)	IF(Down)	ID	BR	OF	EX1	EX2
Idle	0	0	12.2%	84.9%	28.4%	60.6%	69.0%
Busy	67.4%	74.1%	64.8%	12.7%	69.7%	39.4%	31.0%
Blocked	32.6%	25.9%	23.1%	2.4%	1.9%	0	0

improving performance. Note that while the output of the BR block is shown as entering the EX blocks, since branch instructions will have zero execute time, they will essentially exit from the system after leaving BR. As shown in Table 3 this configuration yields the highest performance.

5.2 Influence of a Functional Module: Adder

In the current version of ARAS, several functional modules are simulated in greater detail and provide the information of operation time to the main pipeline simulation. For example, a cache memory simulator is present to determine the times associated with each memory access. In this simulator, separate set associated caches are present for instructions and data.

Another module which is simulated in greater detail is the 32-bit 2's complement integer adder. Three different adder designs are available: asynchronous Ripple-Carry Adder (RCA), asynchronous carry-SELect adder (SEL), and Conditional-Sum Adder (CSA). The structures of these adders are described in [4]. Users can observe the change in performance resulting from the use of different adders in a particular configuration. The simulation results for previous configurations using different adders are shown in Table 6.

Since the CSA has a tree structure and takes a constant amount of time ($O(\log_2 n)$) to complete addition, it is considered to be one of the fastest adders (for clocked design) [7, 12] and achieves the highest throughput for all configurations. However, the complexity of the CSA circuit is higher than that of most other adder circuits. Of the remaining adders, the use of the SEL(8) results in the best throughput. The SEL(8) has 8 blocks, each of which performs an addition of two 4-bit numbers. If the SEL(16) (containing 16 blocks, each of which performs an addition of 2-bit numbers), is used, the performance is not as good as in the SEL(8) case. This is due to the presence multiplexers, which alter the output of each block depending on the carry of previous block. The multiplexer delay becomes more significant when the number of partitions is increased. Therefore, the overall throughput is decreased. The configuration using the RCA has the worst throughput among all adders since the long carry chain present increases the addition delay.

Figures 8 and 9 show the distribution of the addition times for two of the adders, RCA and SEL(8). These were obtained using the Qsort benchmark program referred to in Table 1. The ARAS display enables the user to view the distribution of addition times either dynamically or at the end of the simulation. Addition times are dependent on the operand distribution and the adder used, and not on the pipeline configuration. The distributions indicate that long carry chains occur more often then would be expected with uniformly distributed operands. This appears to be due to additions which occur during effective address calculations (e.g., stack operations which occur at high memory addresses) and can influence the performance of pipeline configurations and choice of adders.

With these sort of simulation results, used in conjunction with VLSI implementation requirements, users can decide which adder design is most appropriate for their design environment. For example, given that a CSA is much larger than a SEL(8) and increases the pipeline performance by only 3%, an SEL(8) de-

Table 6: Simulation Results by Using Different Adders (MIPS: H. Mean for all Benchmarks)

Adder	RCA	SEL(2)	SEL(4)	SEL(8)	SEL(16)	CSA
4SP	54.6	58.5	62.4	63.4	60.8	66.2
6SP	60.4	64.9	69.5	70.7	67.9	73.3
8SP	60.0	63.3	67.2	68.2	65.8	73.3
4SP/PF	64.4	69.0	73.6	74.9	71.9	79.3

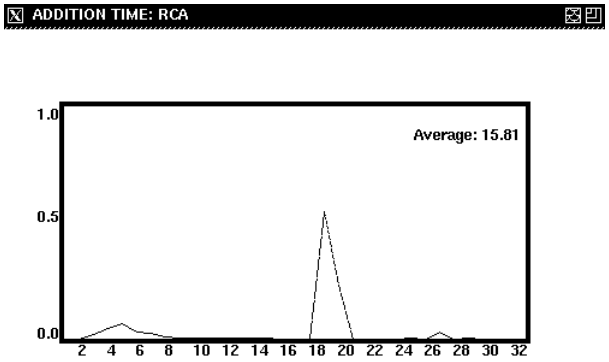

Figure 8: Addition Time for RCA

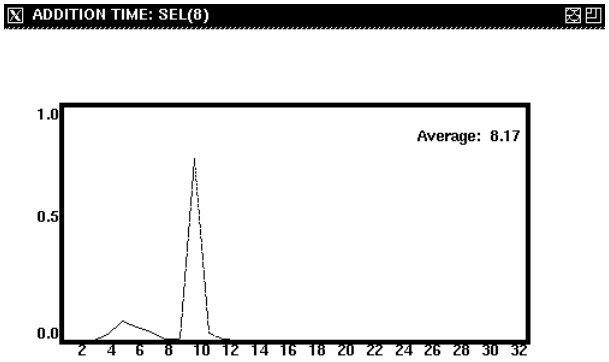

Figure 9: Addition Time for SEL(8)

sign might be the desired choice. Similar studies can be performed by focusing on other functional modules, such as the influence of the number of sets in cache memory or the size of instruction and data cache memories.

5.3 Technology and Design Parameters

The prior sections have focused on high level pipeline architecture considerations and on an individual functional modules. However, underlying the asynchronous simulation is a set of assumptions concerning technology parameters and implementations. ARAS allows one to explore the effects of many of these parameters on performance in conjunction with configuration considerations.

Table 7: Operation Delays

Type of operation	Delay
Handshaking	3 ns
Cache access	10 ns
Main memory access	100 ns
Instruction decode	5 ns
Simple ALU operation	3 ns
Add or Sub (each bit of carry chain)	1 ns
Register access and update	3 ns

The performance results associated with Table 3 are based on the parameter values of Table 7. The optimum configuration may depend on these parameters. For example, if handshaking delays are reduced, then the optimum configuration may have more stages. These synchronization overheads are important factors which affect the performance of asynchronous pipelines (especially in the case of the longer pipelines). To obtain an idea of the sensitivity to handshaking delays with a given configuration consider the results of Table 8. These results indicate that a 16% performance gain can be achieved in a 6-stage pipeline if the handshaking delays are reduced from 3 to 1.5 ns. If techniques for overlapping handshaking with other operations are can be developed which lead to an effective zero handshaking delay, then a 38% improvement can be achieved over the 3 ns case. The results also show that with lower handshaking delays, the 8-stage pipeline yields higher performance than the 6-stage pipeline.

Thus, ARAS reflects the change in the optimal depth of the pipelines with the change in handshaking delays. Keeping other factors constant, if handshaking delays are reduced, deeper pipelines will not necessarily perform better than shallower pipelines. This is due to data dependencies and is seen in the compari-

Table 8: Simulation Results (MIPS: H. Mean for all Benchmarks)

Delay	4SP	6SP	8SP	4SP/PF
0.0 ns	81.3	97.7	104.1	100.1
1.5 ns	71.5	82.5	83.1	87.0
3.0 ns	63.8	71.0	68.2	76.8
4.5 ns	57.7	62.1	58.0	67.1

son of 6- and 8-stage pipelines for handshaking delays of 3 ns. Since ARAS uses program traces, such factors are taken into account. ARAS also indicates that for handshaking delays greater than 1.5 ns, the superscalar configuration presented outperforms all the single pipeline configurations since this configuration is less sensitive to changes in handshaking delays.

6 Conclusions and Future Research

An asynchronous RISC architecture simulator, ARAS, was presented in this paper. A brief description of simulator operation and of the user interface was given. General rules for building a working instruction pipeline were explained and examples of how ARAS can be used to explore the performance of alternative pipeline configurations, functional module performance and parameter values was given. ARAS can be used to simulate a pipelined machine, visualize its operation and obtain performance data. ARAS simulation times depend on the size of benchmark programs being used. For the set of programs given in Table 1, ARAS took an average of about 30 seconds on a Sun SPARC-20 workstation for simulation and presentation of the final results. If intermediate results are required (such as the status of registers after each instruction etc.) or if the visualization is used then longer times will result.

This paper reports on a first version of ARAS. Current work includes making ARAS more user friendly and flexible in the block and configurations specfication areas. Other enhancements include permitting ARAS to simultaneously simulate both clocked and asynchronous versions of the same pipeline configuration. The performance data presented in this paper used program traces which did not include system calls (supervisor mode instructions). Work is now being done remove this limitation and to integrate the SPECmark set of programs into ARAS.

References

[1] C-H Chien and P. Prabhu. ARAS: Asynchronous RISC Architecture Simulator. Technical Report WUCCRC-94-18, Washington Univ., St. Louis, MO, 1994.

[2] W.A. Clark. Macromodular Computer Syhstems. In *Proc. AFIPS - Spring Joint Computer Conference*, pages 489–504, April 1967.

[3] M.A. Franklin and T. Pan. Clocked and Asynchronous Instruction Pipelines. In *Proc. 26th ACM/IEEE Symp. on Microarchitecture*, pages 177–184, Austin, TX, December 1993.

[4] M.A. Franklin and T. Pan. Performance Comparison of Asynchronous Adders. In *Proc. Symp. on Advanced Research in Asynchronous Circuits and Systems*, Salt Lake City, Utah, November 1994.

[5] S.B. Furber, P. Day, J.D. Garside, N.C. Paver, and J.V. Woods. A Micropipelined ARM. In *Int'l Conf. on Very Large Scale Integration (VLSI'93)*, September 1993.

[6] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers, Palo Alto, CA, 1990.

[7] K. Hwang. *Computer Arithmetic: Principles, Architecture, and Design*. Wiley, 1979.

[8] IEEE Computer Society. *Proceedings of Int'l Symposium on Advanced Research in Asynchronous Circuits and Systems*, Salt Lake City, UT, November 1994.

[9] N.P. Jouppi and D.W. Wall. Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines. In *ASPLOS-III Proceedings*, pages 272–282, April 1989.

[10] A.J. Martin, S.M. Burns, T.K. Lee, D. Borkovic, and P.J. Hazewindus. The Design of an Asynchronous Microprocessor. In *Proc. Decennial Caltech Conf. on VLSI*, pages 20–22. The MIT Press, March 1989.

[11] R.F. Sproull, I.E. Sutherland, and C.E. Molnar. Counterflow Pipeline Processor Architecture. *IEEE Design and Test of Computers*, Fall 1994.

[12] S. Waser and M. Flynn. *Introduction to Arithmetic for Digital System Designers*. CBS College Pub., New York, 1982.