Report Number: WUCS-96-05

1996-01-01

Simulation of Asynchronous Instruction Pipelines

Chia-Hsing Chien and Mark A. Franklin

This paper presents the ARAS simulator with which asynchronous instruction pipelines can be modelled, simulated and displayed. ARAS allows one to construct instruction pipelines by preparing various configuration files. Using these files and a number of benchmark programs, performance of the instruction pipelines can be obtained. The performance of asynchronous instruction pipelines can also be compared to synchronous case. Thus, one can decide the optimal design for instruction pipelines in asynchornous or synchronous cases and explore the deisng space of asynchronous instruction pipeline architectures.

... Read complete abstract on page 2.

Simulation of Asynchronous Instruction Pipelines

Chia-Hsing Chien and Mark A. Franklin

Complete Abstract:

This paper presents the ARAS simulator with which asynchronous instruction pipelines can be modelled, simulated and displayed. ARAS allows one to construct instruction pipelines by preparing various configuration files. Using these files and a number of benchmark programs, performance of the instruction pipelines can be obtained. The performance of asynchronous instruction pipelines can also be compared to synchronous case. Thus, one can decide the optimal design for instruction pipelines in asynchornous or synchronous cases and explore the deisng space of asynchronous instruction pipeline architectures.

Simulation of Asynchronous Instruction Pipelines

Chia-Hsing Chien and Mark A. Franklin

WUCS-96-05

February 1996

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

This research has been funded in part by ARPA Contract DABT-93-C0057.

For Submission as a Regular Paper
to the
1996 Summer Computer Simulation Conference

Simulation of Asynchronous Instruction Pipelines *

Chia-Hsing Chien
justin@wuccrc.wustl.edu
(314)935-8562

Mark A. Franklin
jbf@wuccrc.wustl.edu
(314)935-6107

FAX:(314)935-7302

Computer and Communications Research Center
Campus Box 1115
Washington University
St. Louis, Missouri 63130-4899

Key Words

asynchronous, instruction pipeline, event simulation, performance visualization

Abstract

This paper presents the ARAS simulator with which asynchronous instruction pipelines can be modelled, simulated and displayed. ARAS allows one to construct instruction pipelines by preparing various configuration files. Using these files and a number of benchmark programs, performance of the instruction pipelines can be obtained. The performance of asynchronous instruction pipelines can also be compared to synchronous case. Thus, one can decide the optimal design for instruction pipelines in asynchronous or synchronous cases and explore the design space of asynchronous instruction pipeline architectures.

*This research has been funded in part by ARPA Contract DABT-93-C0057

Simulation of Asynchronous Instruction Pipelines

Chia-Hsing Chien Mark A. Franklin
Washington University
St. Louis, Missouri

January 31, 1996

1 Introduction

This paper presents ARAS, an Asynchronous RISC Architecture Simulator which allows for easy simulation of asynchronous instruction pipelines. The objectives of this paper include the illustration of the ARAS simulation architecture, the specification of the simulation models, the visualization of simulation results, and the discussion of particular results of interest.

To achieve higher performance most contemporary computers use instruction pipelining techniques. By employing pipelining techniques, a computer can overlap the execution of several instructions and obtain higher throughput. For example, Figure 1 shows the high level view of the DLX [13, 18] instruction pipeline. The standard DLX instruction pipeline consists of five stages; Instruction Fetch (IF), Instruction Decode (ID), Execution (EX), Memory Access (MA), and Write Back (WB). An instruction, when executing, passes through each of the stages in a sequential fashion (although not all instruction types will use all stages).

There are a number of factors which influence the efficiency of instruction pipelines. For example, the stages will have higher utilization and the pipeline be more efficient if the workload associated with each stage is roughly the same (i.e., the stages are balanced). Other efficiency factors include reducing synchronization overhead and, where possible, exploiting instruction-level parallelism by having multiple parallel pipelines. Various RISC processors incorporate these and other techniques for improving overall instruction throughput and their effect on the performance of instruction pipelines has been studied in [4, 5, 6, 8, 12, 13, 14, 15, 20].

Although the idea of asynchronous design has been explored since 1950's, most digital systems are currently clocked. However, as clock rates have increased, problems of clock skew control and chip power levels associated with today's CMOS based microprocessors have become increasingly

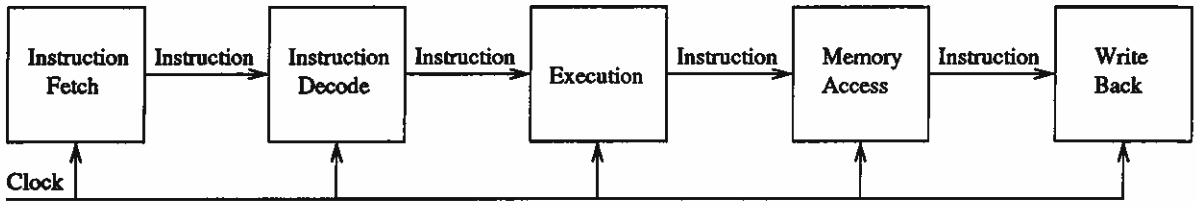

Figure 1: Synchronous (Clocked) DLX Instruction Pipeline

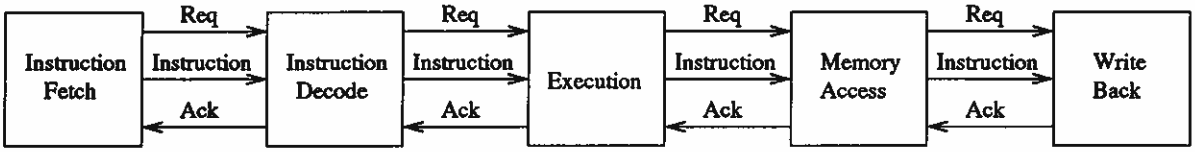

Figure 2: Asynchronous DLX Instruction Pipeline

difficult to overcome. Researchers have therefore been studying the benefits resulting from using asynchronous design techniques in microprocessors. With such techniques there is no clock skew, there is the potential for lower power levels, and there is the possibility of an increase in the overall instruction throughput [9, 10].

Consider the synchronous (clocked) DLX instruction pipeline of Figure 1. After an instruction has finished its operation in the current block, it enters its successor block under the control of a global clock. The clock frequency is selected so that every instruction is able to finish its operation in the current block and is ready to move to its successor block within a clock cycle time. Thus, at one level, clock rate in a synchronous system is governed by the longest operation time among the blocks.

In the asynchronous DLX instruction pipeline shown in Figure 2, however, instruction progress through the pipeline is controlled by a handshaking protocol. After an instruction has finished its operation, the current block ($block_i$) sends a request signal (Req) to its successor block ($block_{i+1}$). If the $block_{i+1}$ can accept the instruction, an acknowledgement signal (Ack) is sent back to $block_i$. The instruction then proceeds from the $block_i$ to $block_{i+1}$ for further processing. If $block_{i+1}$ is busy, the instruction waits in the current block until the successor $block_{i+1}$ can accept a new instruction.

The use of asynchronous modules in the design of processors goes back to the 1960's with work at Washington University in St. Louis [3]; however, an entire asynchronous microprocessor was not

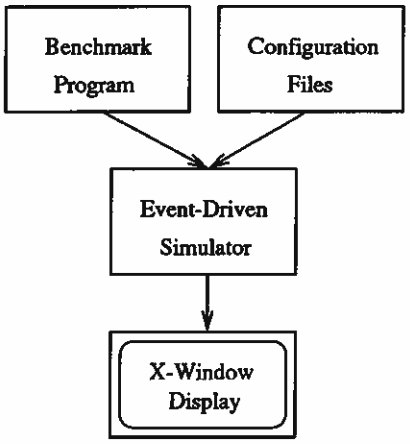

Figure 3: An Overview of the Simulation

developed until 1988 at the California Institute of Technology [16]. Later, an asynchronous version of the ARM processor, AMULET1, was built at University of Manchester (UK) [11]. Currently, SUN Microsystems Inc. is developing an asynchronous microprocessor called the CounterFlow Pipeline Processor (CFPP) [19]. The performance advantages associated with asynchronous design are discussed in more detail in [9, 10]

At the Second Working Conference on Asynchronous Design Methodologies (1995), several additional asynchronous machine designs were proposed although the performance of these machines is still not clear [17, 1, 7]. The simulation tool ARAS is designed to help evaluated the performance of alternative asynchronous architectures. Thus, evaluations can be obtained prior to implementing the real machine.

Section 2 below presents an overview of the ARAS simulator and how it is used. Section 3 discusses the data structures associated with defining ARAS pipelines. This is followed by Section 4, which considers an example instruction pipeline executing in asynchronous and synchronous modes and shows simulation results for several performance metrics. The final section, Section 5, presents conclusions.

2 ARAS: An Overview

Figure 3 shows the overall structure of the ARAS simulator. The pipeline structure is specified by a set of configuration files (to be discussed later) developed by the user. A standard event-driven

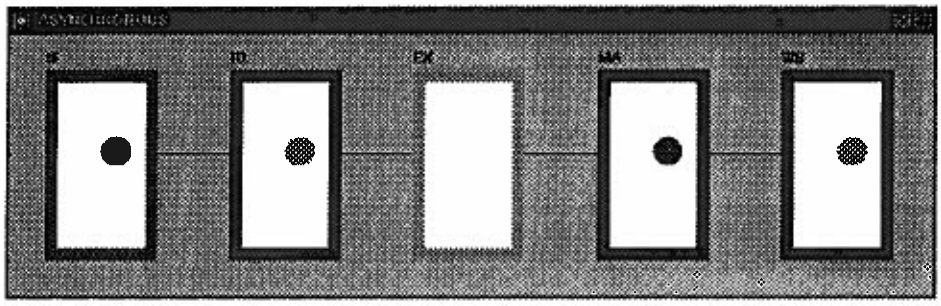

Figure 4: ARAS Display

simulation approach is used with the simulation being driven by a benchmark program assembled into SPARC assembly language. The benchmark programs can be selected either from a library, or can be user supplied. As the simulation progresses results are displayed dynamically using the X-Window system and overall performance statistics are also presented.

As an example, consider the five stage DLX [13, 18] instruction pipeline shown in Figure 4. Each rectangle in the display represents a pipeline block or stage. Each block executes a set of associated micro-operations which have been set by the user to reflect operations which should take place in that stage. Lines between blocks represent paths that instructions may take during execution. The present of a dot in a particular block indicates that an instruction is being processed in that block (e.g. IF, ID, MA, WB), otherwise the block is empty (e.g. EX). The movement of instructions through the pipeline corresponds to the movement of dots from one block to another and can be seen as a dynamic visualization of instruction processing. At the time an instruction moves from one block to another, the lines between the blocks involved momentarily thicken. Though not shown in this black and white representation, dots within the blocks are color-coded to reflect each instruction type. In addition, the border of each block is color-coded to reflect changes in block status. A block may be idle, busy, or blocked.

As the simulation progresses, the user can thus easily follow the progress of instructions through the pipeline. In addition to the dynamic visual display, a designer can gather global system and local block performance, including system throughput, execution time required for processed instructions, number of processed instructions on a block or system level, throughput, and percent of idle, busy, and blocked time associated with individual blocks. These results can be used to improve

Table 1: Content of Operation Delays File (Unit: ns)

Operation	Function	Time / Value
HAND_SHAKING_ACK	Acknowledgement Signal	1.0
HAND_SHAKING_REQ	Request Signal	1.0
BUFFER_DELAY	Buffer Delay	0.5
REG_ACCESS	Register Access	3.0
MEMORY_HIT	Memory Access If In Cache (a hit)	5.0
ASYN_MEMORY_MISS	Memory Access If Not In Cache (a miss)	20.0
ADD_BASE	Addition Base for 1-bit of Addition	1.0
INS_DECODE	Instruction Decode	3.0
SIMPLAE_OP	Simple Operations (e.g. Arithmetic Operations)	5.0
RCA	Ripple-Carry Adder	FALSE
SEL	Carry-Select Adder	TRUE
NUMBER_BLKS	Block Number for SEL	8
FW	Data Forwarding	FALSE
PURGE	Instruction Purging (e.g. Branch Instructions)	FALSE
CLOCK_MODE	Clock Mode Simulation	FALSE
CLOCK_TIME	Clock Time When CLock Mode Is Selected	20.0
CLKS_MEMORY_MISS	Memory Access When a Miss Occurs in Clock Mode	40.0
TIME_STEP	Delay Factor for Display	50000

performance by redesigning the instruction pipeline.

The core of the simulator implements a standard discrete-event simulation algorithm. After the simulator receives the benchmark program and the configuration files (see Section 3) the simulator schedules events for each block following the operation times associated with the instructions in the given block (Table 1) and the requirements of an asynchronous handshaking protocol. For example, the delays associated with the two-phase request-acknowledge protocol used in the asynchronous mode are shown at the top of Table 1.

When the instruction finishes the operations of a particular block, the simulator decides whether this instruction may proceed to the next block depending on whether or not this next block is busy. To make sure every model (more complex pipeline configurations than shown in the example can be constructed) can be simulated properly, a standard score-boarding technique [13] is used to prevent data hazards. The handshaking protocol automatically prevents resource hazards. While the simulator schedules the instruction and its micro-operations in the proper sequence, out-of-

order execution is permitted after the Instruction Decode block. To enable this, the simulator has been designed to handle control hazards resulting from branching instructions.

The configuration files include three files specifying the structure of simulated pipeline models and a file (Table 1) denoting operation delays and other parameters of the pipeline. ARAS also supports two asynchronous adders [10], the Ripple-Carry Adder (RCA) and the carry-SELect adder (SEL), with the parameter NUM_BLKS used to specify the number of the blocks used to implement SEL. The ARAS simulator can also simulate a clocked instruction pipeline if CLOCK_MODE is selected. In this case, the clock rate is determined by the parameter CLOCK_TIME. Details of the parameters in the operation delays file can be found in [2].

With ARAS, benchmark programs can be written in any programming language if these programs can be compiled as SPARC assembly code. However, before the benchmark programs are fed into ARAS, they need to be translated from SPARC assembly codes into the ARAS format. This is done with an interpreter which is part of the ARAS system.

Currently, there are two benchmark programs from SPECint92 (**Espresso** and **Compress**) that are available as standard benchmark inputs. In addition, there are several other benchmark programs which are available which have been developed locally. In this paper we consider only the two SPECint92 benchmark programs.

3 Pipeline Structures

As shown on the left side of Figure 5, there are three steps needed in specifying an instruction pipeline configuration. Corresponding to these steps is the development of associated configuration files. While we discuss these steps in a sequential fashion, the designer must keep in mind all three steps simultaneously since they interact with each other in determining the final pipeline design.

The designer first has in mind a rough pipeline idea which consists of a set of blocks, their functions, and their interconnections. At the lowest level, the functions available for inclusion in a blocks operation are specified in terms of a set of available micro-operations (see Table 2). In addition, each SPARC assembly instruction will require the completion of a certain set of micro-operations for its proper execution. This association of SPARC assembly instructions and micro-

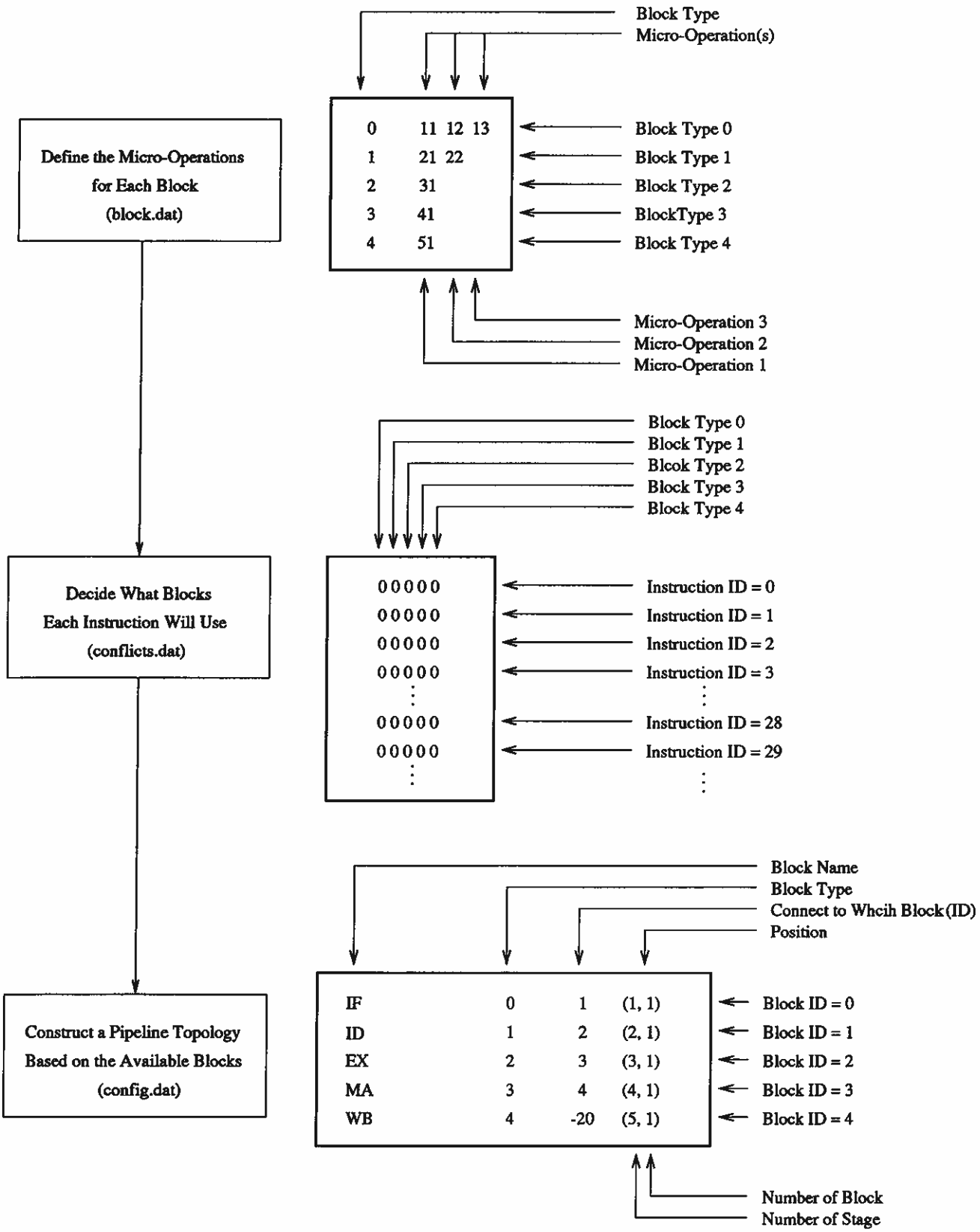

Figure 5: Generation of Instruction Pipeline Configurations and Files

Table 2: Micro-Operation Identification Numbers and Operations

Micro-Operation ID	Operation
11	Write PC value to instruction memory address register
12	PC update
13	Read instruction and store it in instruction register
21	Instruction decode
22	Operand fetch and branch process
31	Execution for ALU instructions
41	Memory access for memory instructions
51	Write back to register

operations is shown in Table 4.

Knowing the SPARC instruction set and associated micro-operations, the designer first determines those micro-operations which will be included in each pipeline block. This is defined in the file block.dat which is shown symbolically in Figure 5 (top right). In this example, five blocks are specified (0, 1, ..., 4) with block 1, for example, executing the micro-operations 21 (Instruction Decode) and 22 (Operand fetch and branch processing).

The second step involves specifying which instructions will be processed by each block (file conflicts.dat). Each entry in conflicts.dat corresponds to a SPARC assembly instruction, with the entry (0 or 1) determining whether the instruction will require the use of the associated block. A zero entry indicates instruction use of the designated block while a one entry indicates the block will not be used.

For the DLX example, all the instructions require processing in all blocks (although for some instructions the delay through the block may be negligible) thus all entries in Figure 5 are zero. There are other pipeline structures, however, where this may not be true. Consider, the superscalar architecture shown in Figure 6 where two instructions may be fetched from memory and subsequently executed in parallel. By permitting both parallel and out-of-order execution of instructions (after decode), increased performance may be achieved. Consider, for example, the third stage of the pipeline where there are three parallel block types. A load or store instruction type will use the memory access block while an arithmetic type instruction will use one of the two other blocks.

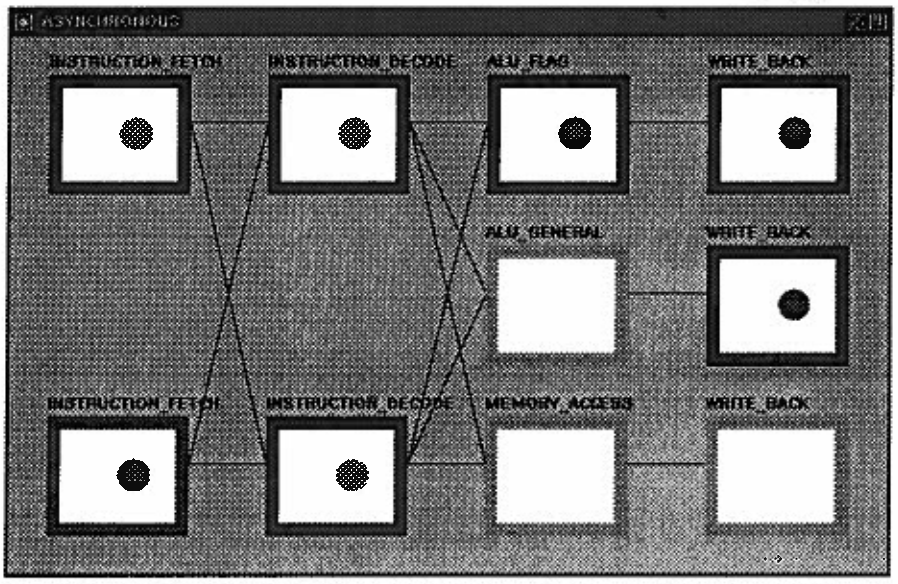

Figure 6: A Superscalar Instruction Pipeline

Both instructions may executed in parallel at this stage and the associated instruction dependent information will be stored in the conflicts.dat file.

The third step concerns describing the interconnection topology associated with the pipeline (i.e., the pipeline architecture). This is done by specifying the contents of the file config.dat. Each entry in config.dat (bottom right of Figure 5) associates a block name with a block type (defined in block.dat), determines connections to other block(s), determines at what stage (numbering from left to right) the block should be located in the pipeline display, and determines the number of parallel blocks of this type that should be present. For example, block 1 is labelled as the ID block, is connected to block 2, is in the second position in the pipeline, and only a single block of this type is present at that stage. Note that a -20 in column 3 of config.dat indicates that the associated block is the last stage of the instruction pipeline.

With the completion of these three steps and specification of the associated files, definition of the pipeline architecture is complete. The simulator uses these files to determine the proper sequence of events to create in response to the instructions provided by the benchmark programs. In the next section, an example will be presented which demonstrates operation of an ARAS simulation and illustrates some typical results.

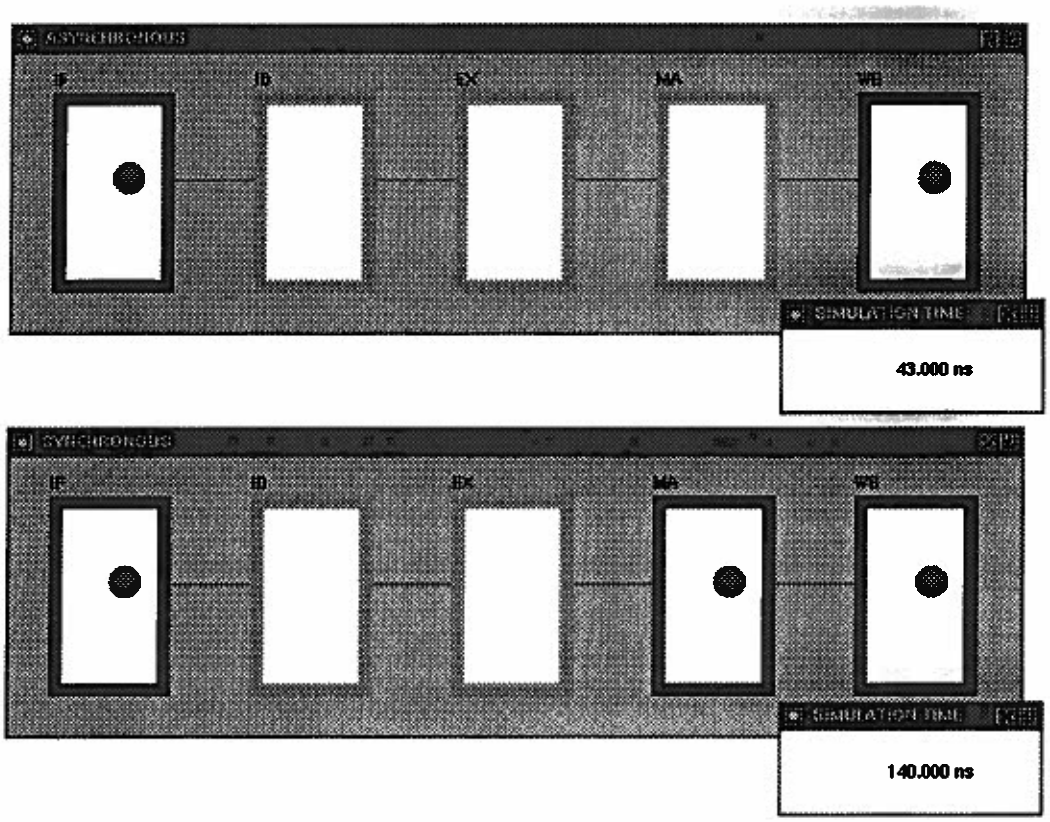

Figure 7: Display of Simulation

4 ARAS Simulation of Clocked & Asynchronous Pipelines

In this section, the DLX instruction pipeline discussed earlier is simulated using asynchronous and synchronous timing methodologies. Operation delays used in the simulation are shown in Table 1. The clock rate has been selected to be longest operation time (assuming cache hit) among all blocks in the pipeline. In this case, the ID block has the longest operation time (20 ns). Note that when an instruction is not in the cache, a miss penalty is applied with the penalty being an extra clock period time.

Figure 7 shows the simulation at the point when the first instruction of the benchmark is just about to leave the last WB stage. The upper window is for asynchronous case and the lower window is for the synchronous case. Notice that the asynchronous case requires 43 ns, but the synchronous case needs 140 ns. This difference is due to a number of factors. In both cases, for example, there is an initial memory cache miss associated with the IF stage. For the asynchronous case, however,

Table 3: Simulation Results (MIPS)

Simulation Case	Espresso	Compress	Simulation Case	Espresso	Compres
DLX (Asynchronous)	55.8	58.8	6-Stage (Asynchronous)	55.6	59.0
DLX (Synchronous)	29.0	26.8	6-Stage (Synchronous)	33.3	31.1

the miss penalty is 20 ns, while for the synchronous case it is 40 ns (adjusted so actions take place on a clock tick). In addition, the first instruction for this benchmark is a branch and requires no processing in the EX, MA and WB stages. The asynchronous design can take advantage of this while the synchronous design, in our model, requires a minimum of 20 ns delay for each stage.

After executing the benchmark programs, the overall simulation results are obtained and shown in Table 3. For the 5-stage asynchronous DLX instruction pipeline the mean throughput for the two benchmark programs is 57 MIPS while for the synchronous case it is 28 MIPS. These differences reflect a host of factors. One is the average versus worst case performance advantage associated with asynchronous designs. Another is the relatively small asynchronous handshaking delays which have been selected for simulation.

In addition to obtaining the overall system performance, a designer can also observe the performance of each block. In the above example, the utilization of each block was found (in the asynchronous case) to be IF = 97%, ID = 61%, EX = 28%, MA = 22% and WB = 23%. IF has the highest utilization as expected since there are always instructions available to be fetched.

Taking the next highest utilized block, ID, it might be interesting to examine the effects dividing its associated micro-operations across two sequential blocks thus constructing a 6-stage pipeline. In this case, dividing ID into two sequential blocks results in the maximum block time being reduced to 17 ns from 20 ns. The clock time for the synchronous implementation can now be somewhat reduced. Note that basic functions are associated with micro-operations, and micro-operations are not divisible. In this case, ID is composed of two the micro-operations 21 and 22 (see Table 2), with operation 21 taking 3 ns, and 22 taking 17 ns. Thus, the ID block cannot be divided evenly. This is a common architecture problem which is encountered when trying to balance stages in a pipeline.

The two timing methodologies can now be compared for this 6-stage case. As indicated in Table 3, the synchronous MIPS rate is 32 (up from 28) and while the asynchronous MIPS rate remains about the same. The increase in the synchronous case is due entirely to the increase in clock rate. Note that the maximum MIPS rate that can be achieved is about 62 MIPS due to the limitations associated with instruction fetching. This slight change in the pipeline thus has little effect on the asynchronous rate which is already at about 57 MIPS.

5 Conclusions

In this paper, the ARAS simulator is presented and discussed. ARAS allows for the simulation of both asynchronous and clocked instruction pipelines. The details of ARAS and its use are explained. This includes the basic steps needed to develop and specify a pipeline topology, and the development of driving benchmark programs. An example 5-stage pipeline is examined in more detail and the results of using ARAS on this pipeline are presented. The 5-stage pipeline is modified by dividing one of the stages into two stages. The restrictions associated with such a stage division are discussed, and the performance of a resulting 6-stage pipeline are presented. It is shown that, for the example considered, there is little performance gain associated with moving from a 5 to 6 stage pipeline. Overall the ARAS system permits users to develop and explore alternative pipeline architectures and determine their relative performance in both a clocked and asynchronous environment.

References

[1] D.K. Arvind, R.D. Mullins, and V.E.F. Rebello. Micronets: A Model for Decentralising Control in Asynchronous Processor Architectures. In *2nd Working Conference on Asynchronous Design Methodologies*, London, England, May 1995.

[2] Chia-Hsing Chien. ARAS: Asynchronous RISC Architecture Simulator. Technical Report WUCCRC-95-04, Washington University, St. Louis, MO, August 1995.

[3] W.A. Clark. Macromodular Computer Systems. In *Proc. Spring Joint Comput. Conf., AFIP*, 1967.

[4] L.W. Cotten. Circuit Implementation of High-Speed Pipeline Systems. In *Proc. AFIPS - Fall Joint Computer Conference*, pages 489–504, 1965.

[5] P.K. Dubey and M.J. Flynn. Optimal Pipelining. *J. of Parallel and Distributed Computing*, pages 10–19, January 1990.

[6] P.K. Dubey, G.B. Adams III, and M.J. Flynn. Evaluating Performance Tradeoffs Between Fine-Grained and Coarse-Grained Alternatives. *IEEE Trans. on Parallel and Distributed Systems*, pages 17–27, January 1995.

[7] C.J. Elston, D.B. Christianson, P.A. Findlay, and G.B. Steve. Hades-Towards the Asynchronous Superscalar Processor. In *2nd Working Conference on Asynchronous Design Methodologies*, London, England, May 1995.

[8] Fawcett. Maximal Clocking Rates for Pipelined Digital Systems. Master's thesis, EE, UI-UC, 1975.

[9] 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.

[10] 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.

[11] 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.

[12] T.G. Hallin and M.J. Flynn. Pipelining of Arithmetic Functions. *IEEE Trans. Computers*, pages 880–886, August 1972.

[13] J.L. Hennessy and D.A. Patterson. *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Palo Alto, CA, 1995.

[14] P.M. Kogge. *The Architecture of Pipelined Computers.* Hemisphere Publishing Corporation, New York, NY, 1981.

[15] S.R. Kunkel and J.E. Smith. Optimal Pipelining in Supercomputers. In *13th Inter. Symp. Comput. Arch.*, pages 404–411, Tokyo, Japan, June 1986.

[16] 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.

[17] S.V. Morton, S.S. Appleton, and M.J. Liebelt. ECSTAC: A Fast Asynchronous Microprocessor. In *2nd Working Conference on Asynchronous Design Methodologies*, London, England, May 1995.

[18] Philip M. Sailer and David R. Kaeli. *The DLX Instruction Set Architecture Handbook.* Morgan Kaufmann Publishers, Palo Alto, CA, 1996.

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

[20] H.S. Stone. *High-Performance Computer Architecture.* Addison-Wesley, Reading, MA, 2nd edition, 1990.

Table 4: Instruction Set and Micro-Operations

Micro-Instructions Group 1	11 12 13 21 22 31 51
Micro-Instructions Group 2	11 12 13 21 22
Micro-Instructions Group 3	11 12 13 21 22 31 51
Micro-Instructions Group 4	11 12 13 21 22 41 51
Micro-Instructions Group 5	11 12 13 21 22 31 41 51

Ins ID	Ins Name	Micro Operations	Ins ID	Ins Name	Micro Operations	Ins ID	Ins Name	Micro Operations
0	add	Group 1	36	sth	Group 1	72	bvs,a	Group 2
1	addcc	Group 1	37	std	Group 1	73	bvc	Group 2
2	addx	Group 1	38	stb	Group 1	74	bvc,a	Group 2
3	addxc	Group 1	39	ba	Group 2	75	ta	Group 2
4	taddcc	Group 1	40	ba,a	Group 2	76	tn	Group 2
5	taddcctv	Group 1	41	bn	Group 2	77	te	Group 2
6	sub	Group 1	42	bn,a	Group 2	78	tne	Group 2
7	subcc	Group 1	43	be	Group 2	79	tl	Group 2
8	subx	Group 1	44	be,a	Group 2	80	tle	Group 2
9	subxcc	Group 1	45	bne	Group 2	81	tge	Group 2
10	tsubcc	Group 1	46	bne,a	Group 2	82	tg	Group 2
11	tsubcctv	Group 1	47	bl	Group 2	83	tlu	Group 2
12	mulscc	Group 1	48	bl,a	Group 2	84	tleu	Group 2
13	and	Group 1	49	ble	Group 2	85	tgeu	Group 2
14	andcc	Group 1	50	ble,a	Group 2	86	tgu	Group 2
15	andn	Group 1	51	bge	Group 2	87	tpos	Group 2
16	andncc	Group 1	52	bge,a	Group 2	88	tneg	Group 2
17	or	Group 1	53	bg	Group 2	89	tcs	Group 2
18	orcc	Group 1	54	bg,a	Group 2	90	tcc	Group 2
19	orn	Group 1	55	blu	Group 2	91	tvs	Group 2
20	orncc	Group 1	56	blu,a	Group 2	92	tvc	Group 2
21	xor	Group 1	57	bleu	Group 2	93	wry	Group 3
22	xorcc	Group 1	58	bleu,a	Group 2	94	save	Group 4
23	xnor	Group 1	59	bgeu	Group 2	95	restore	Group 4
24	xnorcc	Group 1	60	bgeu,a	Group 2	96	mov	Group 5
25	sll	Group 1	61	bgu	Group 2	97	sethi	Group 3
26	srl	Group 1	62	bgu,a	Group 2	98	rd	Group 3
27	sra	Group 1	63	bpos	Group 2	99	cmp	Group 5
28	ld	Group 1	64	bpos,a	Group 2	100	jmpl	Group 2
29	ldub	Group 1	65	bneg	Group 2	101	call	Group 5
30	ldsb	Group 1	66	bneg,a	Group 2	102	jmp	Group 5
31	lduh	Group 1	67	bcc	Group 2	103	clr	Group 5
32	ldsh	Group 1	68	bcc,a	Group 2	104	ret	Group 5
33	ldd	Group 1	69	bcs	Group 2	105	nop	Group 2
34	swap	Group 1	70	bcs,a	Group 2			
35	st	Group 1	71	bvs	Group 2			