Proof of correctness of high-performance 3–1 interlock collapsing ALUs

by J. E. Phillips
S. Vassiliadis

A 32-bit 3–1 interlock collapsing ALU, proposed to allow the execution of two interlocked ALU-type instructions in one machine cycle using an instruction-level parallel machine implementation, is shown to produce results equivalent to a serial execution of the instructions using a 2–1 ALU. The equivalence is shown by deriving tables which represent all possible requirements for the serial execution of the instructions followed by the generalization of the table to represent sets of instructions rather than the individual instructions themselves. Consequently, the equivalence of the 3–1 interlock collapsing ALU operations with these generalized requirements of the serial execution of the instructions is shown. The correctness of a proposed high-speed interlock collapsing ALU is thereby demonstrated.

1. Introduction

The requirement for a 3–1 high-speed fixed-point ALU arises from the existence of execution interlocks [1], the demand for high-performance computation in the form of multiple-issuance/execution machines* [2–12], and the need to avoid increasing execution cycle time.

To clarify, consider the instruction sequence

AR R1,R2,

SR R3,R1,

where AR is an add instruction that adds the contents of register R1, denoted as (R1), to (R2) and writes the results to R1, and SR is a subtract instruction subtracting (R1) from (R3) and writing the results to R3. This instruction sequence can be handled with no loss of performance in a serial machine by forwarding the results from the ALU to the inputs of the ALU. The same sequence, however, causes an underutilization of the second ALU in a multiple-issuance machine that can issue two independent instructions in a single cycle. This underutilization results in no performance gains for the multiple-issuance machine when compared with the execution of the pair using a serial machine.

Several solutions have been proposed to relieve such data dependency hazards for floating-point [10], vector [4], and fixed-point (integer) [5, 6] units. However, as discussed in [13, 14], solutions for the fixed-point units can have detrimental effects: requiring increased numbers of opcodes, incompatibility with existing implementations, and potential increases in the cycle time of the machine due to the concatenation of two ALUs. As a result,

Vassiliadis et al.* [13] establish the necessity of a 3–1 high-speed fixed-point ALU and propose a device that can be used to execute, in a single machine cycle, two instructions exhibiting execution interlocks. Additionally, it has been suggested [13, 15] that the device can compute the correct results in a multiplicity of notations and instruction sets including two's-complement and unsigned notation and all logical, arithmetic, and register transfer instructions present in most architectures. Furthermore, it has been suggested that the device can accommodate RISC and CISC architectures, and that its implementation requires only one more stage than a 2–1 binary adder [13, 16–18] designed with the use of equations reported in [19] that result in high-speed adders [19, 20] using commonly available technologies.

While it has been suggested that a device can be designed that will perform 3–1 interlock collapsing ALU (ICALU) operations correctly [13], and while equations have been proposed that will not result in an implementation which increases the machine cycle time [13], a proof of equivalence between the equations and the serial execution of two ALU operations incorporated in the 3–1 ICALU has not been included. Given that studies have shown that the inclusion of a high-speed 3–1 ICALU, allowing the concurrent execution of interlocked instructions, produces substantial improvements in the parallel execution of instructions [11, 12, 21–24], and given the lack of proof of correctness of the high-speed 3–1 ICALU, it is of interest, for completeness, to prove the correctness of the new device.

In proving the correctness of the high-speed 3–1 ICALU, we proceed as follows. In Section 2, we provide a background for the execution of the 3–1 ICALU by discussing the results from the serial execution of interlocked ALU instructions and provide preliminaries required for the proof of correctness of the 3–1 ICALU. In Section 3 we prove the correctness of the high-speed 3–1 ICALU by showing the equivalence of the results to those of the serial execution using a 2–1 ALU. Finally, in Section 4, we present some concluding remarks.

2. Background, preliminaries, and instruction considerations

A 3–1 ICALU is an ALU with three inputs and one output, which can execute in a single machine cycle two fixed-point instructions, with the operation of the second instruction dependent upon the computation of the first instruction. A high-speed implementation, as provided in [13] and shown in **Figure 1**, was developed for the ESA/370™ architecture [25], which was chosen to represent a worst-case scenario for investigating the feasibility of the device in a general-purpose architecture,

*S. Vassiliadis, "Compound Instruction Set Machines," private communication, 1989.

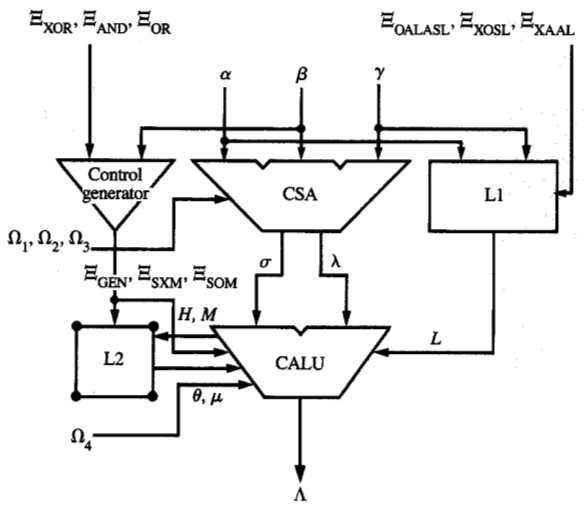

Figure 1

Data flow for high-speed implementation of ICALU.

because such an architecture possesses an instruction set with rich, and potentially complex, ALU operations. Clearly the feasibility of such a device, as suggested in [13, 16], also implies that the incorporation of an interlock collapsing device implementing only a subset of the instruction set, e.g., arithmetic operations, is also feasible. It should also be noted that the device can be incorporated to perform other interlocked operations, such as address generation interlocks and branch determination interlocks. Furthermore, the possibility of including such a device in a RISC machine is also implied. The speed of the device is attributed to the parallel computation of the result, as can be seen from Figure 1, in which L2 calculates controls in parallel with the execution of the CALU, with the operation of the device specified by the following equations [13]:

$$\Lambda_i = \mu_i \phi_{i+1} + \theta_i \overline{\phi_{i+1}} + \Xi_{SOMi},$$

$$\theta_i = H_i \Xi_{GENi} + \bar{H}_i \Xi_{SXMi},$$

$$\mu_i = M_i \Xi_{GENi} + \bar{M}_i \Xi_{SXMi},$$

$$G^{\dagger n}_{i+1} = G^{*n}_{i+1} + \Xi_{SOMi},$$

$$L_i = \alpha_i \Omega_{OALASL} + \gamma_i \Omega_{OALASL} + \bar{\alpha}_i \gamma_i \Omega_{XOSL} + \alpha_i \bar{\gamma}_i \Omega_{XAAL},$$

$$\Xi_{GENi} = \Xi_{ADD} + \Xi_{OR} + \beta_i \Xi_{AND} + \bar{\beta}_i \Xi_{XOR},$$

$$\Xi_{SXMi} = \beta_i \Xi_{XOR},$$

$$\Xi_{SOMi} = \beta_i \Xi_{OR},$$

$$\sigma_i = \alpha_i \forall \beta_i \forall \gamma_i,$$

$$\lambda_i = \Omega_1 \alpha_i \beta_i + \Omega_2 \alpha_i \gamma_i + \Omega_1 \beta_i \gamma_i + \Omega_3 \beta_{i-1},$$

Table 1 ESA/370 RR-format loads, logicals, arithmetics, and compares—function and operand representation, and description.

Instruction	*Operation*	*h*
AR R1,R2	$(R1) \leftarrow (R1) + (R2) + h$	0
SR R1,R2	$(R1) \leftarrow (R1) + (\overline{R2}) + h$	1
LPR R1,R2;	$R2<0$; $(R1) \leftarrow 0 + (\overline{R2}) + h$	1
	$R2 \geq 0$; $(R1) \leftarrow 0 + (R2) + h$	0
LNR R1,R2	$R2 \geq 0$; $(R1) \leftarrow 0 + (\overline{R2}) + h$	1
	$R2<0$; $(R1) \leftarrow 0 + (R2) + h$	0
LR R1,R2	$(R1) \leftarrow 0 + (R2) + h$	0
LTR R1,R2	$(R1) \leftarrow 0 + (R2) + h$	0
LCR R1,R2	$(R1) \leftarrow 0 + (\overline{R2}) + h$	1
ALR R1,R2	$(R1) \leftarrow (R1) + (R2) + h$	0
SLR R1,R2	$(R1) \leftarrow (R1) + (\overline{R2}) + h$	1
CR R1,R2	$X \leftarrow (R1) + (\overline{R2}) + h$	1
CLR R1,R2	$X \leftarrow (R1) + (\overline{R2}) + h$	1
NR R1,R2	$(R1) \leftarrow (R1) \wedge (R2)$	0
OR R1,R2	$(R1) \leftarrow (R1) \vee (R2)$	0
XR R1,R2	$(R1) \leftarrow (R1) \forall (R2)$	0

Table 2 Basic ALU operations to execute single ALU instruction.

Instruction type	*Representation*	*Operation*	*h*
Addition	Add R1,R2	$(R1) + (R2) + h$	0
Subtraction	Sub R1,R2	$(R1) + (\overline{R2}) + h$	1
Logical	LOP R1,R2	$(R1)\ LOP\ (R2)$	0

$$H_i = (\Omega_4 \sigma_i + L_i) \forall \lambda_{i+1},$$

$$G_i = (\Omega_4 \sigma_i + L_i) \lambda_{i+1},$$

$$T_i = \Omega_4 \sigma_i + L_i + \lambda_{i+1},$$

where Ω_1, Ω_2, Ω_3, Ω_4, Ω_{OALASL}, Ω_{XAAL}, Ω_{XOSL}, Ξ_{ADD}, Ξ_{OR}, and Ξ_{XOR} are control signals, ϕ_i, $M_i = H_{i-1} \forall T_i$, G_i^{*n}, and T_i^n are parameters for recursive CLA equations as defined in [19], and α_i, β_i, and γ_i are inputs to the 3–1 ICALU.

The operation for the high-speed 3–1 ICALU was determined by considering all of the operations that would be required by sequences of two ALU operations interlocked with one another [14]. To determine the operations required of a 3–1 ICALU for this instruction set, one need only consider the RR-format ALU instructions, since all of the ALU operations have a corresponding RR-format instruction. Other ALU instructions differ from the RR-format instructions in the source of the operands on which the ALU is to execute rather than in the operation to be performed. The RR-format ALU instructions are summarized in **Table 1**. In this table, the instruction mnemonic is given, along with register designations for the operands (denoted by R1 and R2) in the first column. In the second column, the operation is provided with $(R1)$ representing the contents of register R1, $(\overline{R1})$ representing the one's complement of the contents of R1, h representing a "hot one" supplied to the adder, + indicating binary addition, $\wedge$ representing bitwise AND, $\vee$ representing bitwise OR, and $\forall$ representing bitwise EXCLUSIVE-OR. The contents of the registers, for example $(R1)$, are 32-bit signed or unsigned numbers, with the bits numbered in ascending order from the most significant bit (MSB) to the least significant bit (LSB). Since status is discussed elsewhere [13, 15, 26] and not considered in this paper, no distinction is made between signed and unsigned numbers other than for the instructions LOAD POSITIVE REGISTER (LPR) and LOAD NEGATIVE REGISTER (LNR), for which the operation to be performed depends upon the sign of the operand. For each of these instructions, one of two operations must be performed, depending on the sign of the operand. The two possible operations are specified by the two rows associated with each of these instructions in Table 1 under the Operation column. The condition leading to the operation is also shown in that column. Finally, compare operations (CR and CLR) are used to set ALU status (considered in [13]) rather than to provide ALU results. Therefore, their results are not written to any facility and thus are denoted by showing the destination of the operation as X.

From Table 1 it can be observed that if the capability is provided to zero the ALU input corresponding to operand $R1$, then, with respect to the result of an operation, the ALU operations can be reduced to three instruction types. These instruction types are summarized in **Table 2**. The addition instruction type covers the instructions AR; LPR, when $R2 \geq 0$; LNR, when $R2 < 0$; LR; LTR; and ALR. The subtraction instruction type covers the instructions SR; LPR, when $R2 < 0$; LNR, when $R2 \geq 0$; LCR; SLR; CR; and CLR. The logical instruction type covers NR, OR, and XR, with LOP representing either of these logical operations. To determine the operations required by an ALU designed to execute, in a single cycle, interlocks between ALU instructions, all combinations of the three instruction types and all potential interlock situations must be considered. **Table 3** compiles the results from such considerations for the instruction types shown in Table 2.

In Table 3, the operations to be performed on the register operands are shown in the second column, in which addition or subtraction operations are represented as ADD and logical operations are represented as LOP. The third column of the table shows the functions to be performed on the 3–1 ICALU inputs to produce the desired result of column two. The routing of operands to allow the proper operation to be executed is shown in the

Table 3 Operand routings to produce desired operations for LOGICAL-ADD operations.

Row	*Desired operation*	*ALU operation*	*OP1*	*OP2*	α	β	γ
1	(R1 LOP R2) ADD R4	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	+,−	R2	R4	R1
2	R3 ADD (R1 LOP R2)	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	+	R2	R3	R1
3		−(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	+	R2	R3	R1
4	(R1 LOP R1) ADD R4	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	+,−	R2(=R1)	R4	R1
5	R3 ADD (R1 LOP R1)	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	+	R2(=R1)	R3	R1
6		−(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	+	R2(=R1)	R3	R1
7	(R1 LOP R2) ADD (R1 LOP R2)	(γ OP1 α) OP2 (γ OP1α)	$\wedge,\vee,\oplus$	+,−	See note		
8	(R1 ADD R2) LOP R4	(γ OP1 α) OP2 β	+,−	$\wedge,\vee,\oplus$	R2	R4	R1
9	R3 LOP (R1 ADD R2)	(γ OP1 α) OP2 β	+,−	$\wedge,\vee,\oplus$	R2	R3	R1
10	(R1 ADD R1) LOP R4	(γ OP1 α) OP2 β	+,−	$\wedge,\vee,\oplus$	R2(=R1)	R4	R1
11	R3 LOP (R1 ADD R1)	(γ OP1 α) OP2 β	+,−	$\wedge,\vee,\oplus$	R2(=R1)	R3	R1
12	(R1 ADD R2) LOP (R1 ADD R2)	(γ OP1 α) OP2 (γ OP1 α)	+,−	$\wedge,\vee,\oplus$	See note		
13	(R1 LOP R2) LOP R4	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	$\wedge,\vee,\oplus$	R2	R4	R1
14	R3 LOP (R1 LOP R2)	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	$\wedge,\vee,\oplus$	R2	R3	R1
15	(R1 LOP R1) LOP R4	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	$\wedge,\vee,\oplus$	R2(=R1)	R4	R1
16	R3 LOP (R1 LOP R1)	(γ OP1 α) OP2 β	$\wedge,\vee,\oplus$	$\wedge,\vee,\oplus$	R2(=R1)	R3	R1
17	(R1 LOP R2) LOP (R1 LOP R2)	(γ OP1 α) OP2 (γ OP1 α)	$\wedge,\vee,\oplus$	$\wedge,\vee,\oplus$	See note		
18	(R1 ADD R2) ADD R4	β OP2 (γ OP1 α)	+,−	+	R2	R4	R1
19		−β OP2 (γ OP1 α)	+,−	+	R2	R4	R1
20	R3 ADD (R1 ADD R2)	β OP2 (γ OP1 α)	+,−	+,−	R2	R3	R1
21	(R1 ADD R1) ADD R4	β OP2 (γ OP1 α)	+,−	+	R2(=R1)	R4	R1
22		−β OP2 (γ OP1 α)	+,−	+	R2(=R1)	R4	R1
23	R3 ADD (R1 ADD R1)	β OP2 (γ OP1 α)	+,−	+,−	R2(=R1)	R3	R1
24	(R1 ADD R2) ADD (R1 ADD R2)	(γ OP1 α) OP2 (γ OP1 α)	+,−	+,−	See note		

Note: These operations, which occur when the register specifications for R1, R3, and R4 are the same, were not implemented in the high-speed 3–1 ICALU; therefore, their operand routings were not included.

last three columns. The following example, in which the data dependencies for a subtraction-followed-by-addition instruction type (i.e., SR R1,R2; AR R3,R4) are considered, provides insight into the considerations that must be made to produce Table 3. The combination of data dependencies that can arise from this sequence are R1 = R3; R1 = R4; and R1 = R3 = R4. Furthermore, there may be no data dependencies, a condition denoted here as "independent." Other data dependencies such as R2 = R3 and R2 = R4 are considered in this paper to be "independent," because they do not influence the ALU operation and are trivial to resolve in a hardware implementation. For each of these instruction and data dependency sequences, the SR instruction produces

$$(R1) \leftarrow (R1) - (R2).$$

The results for the addition operation, however, differ for the different interlock situations:

R1 = R3: $(R1) \leftarrow (R1) - (R2) + (R4)$,

R1 = R4: $(R3) \leftarrow (R3) + (R1) - (R2)$,

R1 = R3 = R4: $(R1) \leftarrow (R1) - (R2) + (R1) - (R2)$,

and in the case of "independent" instructions, the results are

Independent: $(R3) \leftarrow (R3) + (R4)$.

These instruction sequences, contained in rows 18, 20, and 24 of Table 3, require the operations

Independent: $(R3) \leftarrow (R3) + (R4)$; $h2 = 0$,

R1 = R3; $(R1) \leftarrow (R1) + (\overline{R2}) + (R4)$; $h1 = 1$, $h2 = 0$,

R1 = R4: $(R3) \leftarrow (R3) + (R1) + (\overline{R2})$; $h1 = 1$, $h2 = 0$,

R1 = R3 = R4: $(R1) \leftarrow (R1) + (\overline{R2}) + (R1) + (\overline{R2})$; $h1 = h2 = 1$,

represented as two's-complement operations for which $h1$ and $h2$ designate the provision of "hot ones" to the first and second serial operations, respectively.

From Table 3, it can be observed that, strictly speaking, a 4–1 ICALU is required to execute all execution interlocks that can occur between two ALU-type instructions. In addition, interlocked instruction sequences, in which the second instruction is LPR or LNR, can

Table 4 Concatenated 2–1 ALU operations supported in 3–1 ICALU.

Row	*OP1*	*OP2*	*Dependency*	*ALU1 operation*	*ALU2 operation*	*Result*	*h1*	*h2*
1	Add	Add	$R3 = R1; R4 = R1$	$\delta = a + g + h1$	$\Lambda = \delta + b + h2$	$\Lambda = a + g + b + h1 + h2$	0	0
2	Add	Sub	$R3 = R1$	$\delta = a + g + h1$	$\Lambda = \delta + \overline{b} + h2$	$\Lambda = a + g + \overline{b} + h1 + h2$	0	1
3	Add	Sub	$R4 = R1$	$\delta = a + g + h1$	$\Lambda = \overline{\delta} + b + h2$	$\Lambda = \overline{a + g + h1} + b + h2$	0	1
4	Add	Log	$R3 = R1; R4 = R1$	$\delta = a + g + h1$	$\Lambda = \delta\ LOP2\ b$	$\Lambda = (a + g + h1)\ LOP2\ b$	0	0
5	Sub	Add	$R3 = R1; R4 = R1$	$\delta = a + \overline{g} + h1$	$\Lambda = \delta + b + h2$	$\Lambda = a + \overline{g} + b + h1 + h2$	1	0
6	Sub	Sub	$R3 = R1$	$\delta = a + \overline{g} + h1$	$\Lambda = \delta + \overline{b} + h2$	$\Lambda = a + \overline{g} + \overline{b} + h1 + h2$	1	1
7	Sub	Sub	$R4 = R1$	$\delta = a + \overline{g} + h1$	$\Lambda = \overline{\delta} + \overline{b} + h2$	$\Lambda = (\overline{a + \overline{g} + h1}) + \overline{b} + h2$	1	1
8	Sub	Log	$R3 = R1; R4 = R1$	$\delta = a + \overline{g} + h1$	$\Lambda = \delta\ LOP2\ b$	$\Lambda = (a + \overline{g} + h1)\ LOP2\ b$	1	0
9	Log	Add	$R3 = R1; R4 = R1$	$\delta = a\ LOP1\ g$	$\Lambda = \delta + b + h2$	$\Lambda = (a\ LOP1\ g) + b + h2$	0	0
10	Log	Sub	$R3 = R1$	$\delta = a\ LOP1\ g$	$\Lambda = \delta + \overline{b} + h2$	$\Lambda = (a\ LOP1\ g) + \overline{b} + h2$	0	1
11	Log	Sub	$R4 = R1$	$\delta = a\ LOP1\ g$	$\Lambda = \overline{\delta} + b + h2$	$\Lambda = (\overline{a\ LOP1\ g}) + b + h2$	0	1
11	Log	Log		$\delta = a\ LOP1\ g$	$\Lambda = \delta\ LOP2\ b$	$\Lambda = (a\ LOP1\ g)\ LOP2\ b$	0	0

require two concatenated ALUs for their execution, since the first instruction's result is required before the second ALU's operation to execute the LPR or LNR can be determined. Since execution interlocks between instruction sequences that require either a 4–1 ICALU or two concatenated ALUs for their execution were infrequent in the instruction traces we considered, it was decided to require that the instruction issue logic serialize the issue of these sequences [13]. This results in the serial execution of these sequences, which can be performed in two cycles using a conventional 2–1 ALU. By enforcing this requirement, the potential delay problems and complexity associated with a 4–1 ICALU or two concatenated ALUs can be avoided. With these assumptions, the design of a 3–1 ICALU is sufficient for executing most interlocked ALU instructions in a single cycle. All operations of Table 3 except for those contained in rows 7, 12, 17, and 24 can be supported by the 3–1 ICALU.

The operations of the 3–1 ICALU shown in Table 3 must provide results identical to those for two concatenated 2–1 ALUs, denoted as ALU1 and ALU2, executing the instruction pair in a serial fashion. **Table 4** summarizes the serial execution for all combinations of the three instruction types and data dependencies that are supported by the 3–1 ICALU. In this table, the first and second operations (OP1 and OP2) are given in the second and third columns of the table. In addition, the results (δ) from the operation of the first ALU (ALU1), results (Λ) from the execution of the second ALU (ALU2), and the overall result specified as the operation performed on the three operands (a, g, and b) supplied to the two 2–1 ALUs are shown in the fifth, sixth, and seventh columns, respectively. Finally, the hot ones, $h1$ and $h2$, supplied to ALU1 and ALU2, respectively, are provided in the last two columns. **Table 5** shows the setup of operand values (denoted as A, Γ, and B) to the three inputs (denoted as α, γ, and β) of the 3–1 ICALU, as dictated by the operations of Table 4, to execute pairs of add/sub-type operations for various supported data dependencies. In this table, λ_i represents the carry from a 3–2 CSA used in designing the 3–1 ICALU, and ϕ_i represents the carries produced in the 2–1 adder used in the design of the 3–1 ICALU, with all the carries being produced from bit position i into $i - 1$. The values, shown in the table, are for $i = 32$, which represents the hot ones that must be supplied to the adders as boundary conditions to produce the appropriate result. The correctness of the 3–1 ICALU is proven by showing the equivalence between its results and the results of the serial execution using a 2–1 ALU, as provided in Table 4.

3. Proof of equivalence of high-speed 3–1 ICALU with serial execution by a 2–1 ALU

In this section, the data flow of the high-performance 3–1 ICALU, depicted in Figure 1 and described by the Boolean equations in Section 2, is shown to produce results equivalent to the serial execution of a pair of interlocked ALU instructions by a 2–1 ALU. Equivalence must be shown for each pair of instruction types and possible data dependencies as provided by Table 4.

The setup of the high-speed 3–1 ICALU to implement the required operations is shown in **Table 6**. In Table 6, the second column specifies the operation dictated by the first instruction, with the third column specifying the operation dictated by the second instruction. Because a 3–1 adder is known to produce the correct results [13] for addition as well as subtraction operations with the proper inversion of operands and supply of hot ones, only add operations are shown in Table 6, except for those conditions in which a subtraction requires unique functions to be supported by logic other than that executing a 3–1 addition. For example, for the operations AND followed by add, an AND of A with Γ must be performed by the

Table 5 ALU setup to execute collapsing operations where OP1 and OP2 are add/sub-type operations.

Row	*OP1*	*OP2*	*Dep*	α_i	β_i	γ_i	λ_{32}	ϕ_{32}	δ_i^*
1	Add	Add	R3 = R1; R4 = R1	A_i	B_i	Γ_i	0	0	δ_i
2	Add	Sub	R3 = R1	A_i	$\overline{B}_i$	Γ_i	1	0	δ_i
3	Add	Sub	R4 = R1	$\overline{A}_i$	B_i	$\overline{\Gamma}_i$	1	1	$\overline{\delta}_i$
4	Sub	Add	R3 = R1; R4 = R1	A_i	B_i	$\overline{\Gamma}_i$	1	0	δ_i
5	Sub	Sub	R3 = R1	A_i	$\overline{B}_i$	$\overline{\Gamma}_i$	1	1	δ_i
6	Sub	Sub	R4 = R1	$\overline{A}_i$	B_i	Γ_i	1	0	$\overline{\delta}_i$

Table 6 ALU setup to execute collapsing operation categories.

Row	*OP1*	*Subsequent operation*	α_i	β_i	γ_i	λ_{32}	ϕ_{32}	Ξ_{AND}	Ξ_{OR}	Ξ_{XOR}	Ξ_{ADD}	Ω_1	Ω_2	Ω_3	Ω_4	Ω_{OALASL}	Ω_{XOSL}	Ω_{XAAL}
1	Add	Add	A_i	B_i	Γ_i	0	0	0	0	0	1	1	1	0	1	0	0	0
2	AND	Add	A_i	B_i	$\overline{\Gamma}_i$	0	0	0	0	0	1	0	0	1	0	0	0	1
3	OR	Add	A_i	B_i	Γ_i	0	0	0	0	0	1	0	0	1	0	1	0	0
4	XOR	Add	A_i	B_i	Γ_i	0	0	0	0	0	1	0	0	1	0	0	1	1
5	AND	Sub	$\overline{A}_i$	B_i	$\overline{\Gamma}_i$	0	1	0	0	0	1	0	0	1	0	1	0	0
6	OR	Sub	A_i	B_i	$\overline{\Gamma}_i$	0	1	0	0	0	1	0	0	1	0	0	1	0
7	XOR	Sub	$\overline{A}_i$	B_i	Γ_i	0	1	0	0	0	1	0	0	1	0	0	1	1
8	Add	AND	A_i	B_i	Γ_i	0	0	1	0	0	0	0	1	0	0	0	1	1
9	Add	OR	A_i	B_i	Γ_i	0	0	0	1	0	0	0	1	0	0	0	1	1
10	Add	XOR	A_i	B_i	Γ_i	0	0	0	0	1	0	0	1	0	0	0	1	1
11	AND	AND	A_i	B_i	$\overline{\Gamma}_i$	0	0	1	0	0	0	0	0	0	0	0	0	1
12	AND	OR	A_i	B_i	$\overline{\Gamma}_i$	0	0	0	1	0	0	0	0	0	0	0	0	1
13	AND	XOR	A_i	B_i	$\overline{\Gamma}_i$	0	0	0	0	1	0	0	0	0	0	0	0	1
14	OR	AND	A_i	B_i	Γ_i	0	0	1	0	0	0	0	0	0	0	1	0	0
15	OR	OR	A_i	B_i	Γ_i	0	0	0	1	0	0	0	0	0	0	1	0	0
16	OR	XOR	A_i	B_i	Γ_i	0	0	0	0	1	0	0	0	0	0	1	0	0
17	XOR	AND	A_i	B_i	Γ_i	0	0	1	0	0	0	0	0	0	0	0	1	1
18	XOR	OR	A_i	B_i	Γ_i	0	0	0	1	0	0	0	0	0	0	0	1	1
19	XOR	XOR	A_i	B_i	Γ_i	0	0	0	0	1	0	0	0	0	0	0	1	1

logic block, L1, while for the operations AND followed by sub, a NAND of A with Γ must be performed by L1. These unique situations must be shown to be realizable. In the following, each row of Table 6 is shown to be executed by the high-speed 3–1 ICALU implementation. First, row 1 is shown to produce a 3–1 addition. Next, rows 2–4 are shown to produce the expected logical operation followed by an addition. Subsequently, rows 5–7 (logical followed by subtraction), rows 8–10 (add followed by logical), and rows 11–19 (logical followed by logical) are considered in three separate groups.

- *Add followed by add*

In assuming that the ICALU is presented with the correct inputs and the hot ones as dictated by Table 5 for the add category, and realizing that the operation of the 3–1 ICALU for the "independent" data dependency case can be achieved by a 3–1 addition with one of the operands zeroed, to prove that add followed by add is performed by the ICALU, we need to prove that the device essentially performs a 3–1 binary addition on its inputs. With substitution, and assuming that the proper inputs are denoted by α_i, β_i, and γ_i for bit position i, i.e., the ALU setup of Table 6, row 1, the expressions for the fast implementation of the 3–1 ICALU give

$$\Xi_{\mathrm{GENi}} = 1 + 0 + B_i 0 + \overline{B}_i 0 = 1,$$

$$\Xi_{\mathrm{SXMi}} = B_i 0 = 0,$$

$$\Xi_{\mathrm{SOMi}} = \overline{B}_i 0 = 0,$$

$$\sigma_i = A_i \forall B_i \forall \Gamma_i,$$

$$\lambda_i = 1A_iB_i + 1A_i\Gamma_i + 1B_i\Gamma_i + 0B_{i-1}$$
$$= A_iB_i + A_i\Gamma_i + B_i\Gamma_i .$$

The previous equations for σ_i and λ_i determine the output of a 3–2 binary addition. It must be proven that the remaining equations determining the behavior of the device compute a 2–1 binary addition. Given that

$$L_i = A_i0 + \Gamma_i0 + \overline{A}_i\Gamma_i0 + A_i\overline{\Gamma}_i0 = 0,$$

the remaining equations perform

$$H_i = (1\sigma_i + 0) \forall \lambda_{i+1} = \sigma_i \forall \lambda_{i+1},$$
$$G_i = (1\sigma_i + 0)\lambda_{i+1} = \sigma_i\lambda_{i+1},$$
$$T_i = 1\sigma_i + 0 + \lambda_{i+1} = \sigma_i + \lambda_{i+1},$$
$$G_{i+1}^{\dagger n} = G_{i+1}^{*n} + 0 = G_{i+1}^{*n},$$
$$\theta_i = H_i1 + \overline{H}_i0 = H_i,$$
$$\mu_i = M_i1 + \overline{M}_i0 = M_i,$$
$$\Lambda_i = \mu_i\phi_{i+1} + \theta_i\overline{\phi_{i+1}} + 0 = M_i\phi_{i+1} + H_i\phi_{i+1}.$$

Furthermore, given that H_i, G_i, T_i, $G_i^{\dagger n} = G_i^*$, and Λ_i are the equations proven to determine a binary addition [19], the proposed device indeed computes a 3–1 binary addition as required by the add-followed-by-add category of interlocked ALU instructions.

- *Logical followed by add*

For these operations, a bitwise logical AND, OR, or XOR must be performed between A and Γ. This result should then be added to a third operand B. The logical operation is performed by the L1 block of Figure 1 as specified by the expression for L in the defining equations. We first demonstrate that, for the setup in rows 2–4 of Table 6, L produces bitwise logical AND, OR, and XOR, respectively. For the setup of the ICALU device as shown in row 2,

$$L = A_i0 + \overline{\Gamma}_i0 + \overline{A_i\Gamma_i}0 + A_i\overline{\overline{\Gamma}}_i1 = A_i\Gamma_i,$$

which is the bitwise logical AND, as desired. For the setup of row 3 of Table 6,

$$L = A_i1 + \Gamma_i1 + \overline{A}_i\Gamma_i0 + A_i\overline{\Gamma}_i0 = A_i + \Gamma_i,$$

which is the bitwise logical OR, as desired. Finally, for the setup of row 4 of Table 6,

$$L = A_i0 + \Gamma_i0 + \overline{A}_i\Gamma_i1 + A_i\overline{\Gamma}_i1 = \overline{A}_i\Gamma_i + A_i\overline{\Gamma}_i = A_i \forall \Gamma_i,$$

which is the bitwise EXCLUSIVE-OR, as desired. In the following, LOP is used to represent one of bitwise AND, OR, or EXCLUSIVE-OR.

Other than the setup to produce the appropriate logical operation between A and Γ, the setup of Table 6 is identical for rows 2–4. Using this setup in the equations defining the fast ICALU implementation produces

$$\Xi_{GENi} = 1 + 0 + B_i0 + \overline{B}_i0 = 1,$$
$$\Xi_{SXMi} = B_i0 = 0,$$
$$\Xi_{SOMi} = B_i0 = 0,$$
$$\sigma_i = \text{don't care},$$
$$\lambda_i = 0A_iB_i + 0A_i\Gamma_i + 0B_i\Gamma_i + 1B_{i-1} = B_{i-1},$$
$$L_i = A_i\ LOP1\ \Gamma_i,$$
$$H_i = (0\sigma_i + A_i\ LOP1\ \Gamma_i) \forall B_i = (A_i\ LOP1\ \Gamma_i) \forall B_i,$$
$$G_i = [0\sigma_i + (A_i\ LOP1\ \Gamma_i)]B_i = (A_i\ LOP1\ \Gamma_i)B_i,$$
$$T_i = 0\sigma_i + (A_i\ LOP1\ \Gamma_i) + B_i = (A_i\ LOP1\ \Gamma_i) + B_i,$$
$$G_{i+1}^{\dagger n} = G_{i+1}^{*n} + 0 = G_{i+1}^{*n},$$
$$\theta_i = H_i1 + \overline{H}_i0 = H_i,$$
$$\mu_i = M_i1 + \overline{M}_i0 = M_i,$$
$$\Lambda_i = \mu_i\phi_{i+1} + \theta_i\overline{\phi_{i+1}} + 0 = M_i\phi_{i+1} + H_i\phi_{i+1},$$

which is the expression for the sum from a 2–1 CLA using the recursive equations described in [19], with inputs being the logical operation A *LOP*1 Γ and B. Therefore, the ICALU produces the result (A *LOP*1 Γ) + B, where + represents addition.

- *Logical followed by sub*

These operations are identical to those for "logical followed by add," as just discussed, with the exception that the bitwise logical operations NAND, NOR, and XNOR must be produced by the logic block L1 to support the subtraction of the logical operation from the third operand, B. Therefore, all that is required is to demonstrate that L1 can produce these operations. For the setup of row 5 of Table 6,

$$L = \overline{A}_i1 + \overline{\Gamma}_i1 + 0A_i\overline{\Gamma}_i + 0\overline{A}_i\Gamma_i = \overline{A}_i + \overline{\Gamma}_i = \overline{A_i\Gamma_i},$$

which is the bitwise logical NAND, as desired. For the setup of row 6 of Table 6,

$$L = A_i0 + \overline{\Gamma}_i0 + \overline{A_i\Gamma_i}1 + A_i\Gamma_i0 = \overline{A_i\Gamma_i} = \overline{A_i + \Gamma_i},$$

which is the bitwise logical NOR, as desired. Finally, for the setup of row 7 of Table 6,

$$L = \overline{A}_i0 + \Gamma_i0 + A_i\Gamma_i1 + \overline{A_i\Gamma_i}1 = A_i\Gamma_i + \overline{A_i\Gamma_i} = \overline{A_i \forall \Gamma_i},$$

which is the bitwise EXCLUSIVE-NOR, as desired. Therefore, the 2–1 CLALU is set up to subtract A *LOP*1 Γ from the middle operand as desired.

- *Add followed by logical*

For these operations, which encompass rows 8–10 of Table 6, the setup of the 3–1 ICALU is identical except for

the controls Ξ_{AND}, Ξ_{OR}, and Ξ_{XOR}, which specify one of the three possible bitwise logical operations to be performed between the sum of A and Γ and the third operand B. Therefore, the common expressions for these operations are

$$L = A_i 0 + \Gamma_i 0 + \overline{A}_i\Gamma_i 1 + A_i\overline{\Gamma}_i 1 = \overline{A}_i\Gamma_i + A_i\overline{\Gamma}_i = A_i \forall \Gamma_i,$$

σ_i = don't care,

$$\lambda_i = 0A_iB_i + 1A_i\Gamma_i + 0B_i\Gamma_i + 0B_{i-1} = A_i\Gamma_i,$$

$$H_i = (0\sigma_i + A_i \forall \Gamma_i) \forall (A_{i+1}\Gamma_{i+1}) = (A_i \forall \Gamma_i) \forall (A_{i+1}\Gamma_{i+1}),$$

$$G_i = (0\sigma_i + A_i \forall \Gamma_i)(A_{i+1}\Gamma_{i+1}) = (A_i \forall \Gamma_i)(A_{i+1}\Gamma_{i+1}),$$

$$T_i = 0\sigma_i + (A_i \forall \Gamma_i) + (A_{i+1}\Gamma_{i+1}) = (A_i \forall \Gamma_i) + (A_{i+1}\Gamma_{i+1}).$$

First consider the operation when the logical operation is a bitwise logical AND. For this case, the setup of row 8 of Table 6 yields

$$\Xi_{GENi} = 0 + 0 + B_i 1 + \overline{B}_i 0 = B_i,$$

$$\Xi_{SXMi} = B_i 0 = 0,$$

$$\Xi_{SOMi} = B_i 0 = 0,$$

$$\mu_i = M_iB_i + \overline{M}_i 0 = M_iB_i,$$

$$\theta_i = H_iB_i + \overline{H}_i 0 = H_iB_i,$$

$$\Lambda_i = M_iB_i\phi_{i+1} + H_iB_i\overline{\phi_{i+1}} + 0 = B_i(M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}).$$

However, the last expression in parentheses is the recursive expression for the 2–1 addition of two values. The values in this case are $A_i \forall \Gamma_i$ and $A_{i+1}\Gamma_{i+1}$. But $A_i \forall \Gamma_i$ represents the sum from a 3–2 CSA with a third input of zero, and $A_{i+1}\Gamma_{i+1}$ represents the carry, λ_{i+1}, from such a 3–2 CSA. From the results above, $\lambda_{i+1} \forall \phi_{i+1} = \kappa_{i+1}$ for this case, so that

$$\Lambda_i = B_i[(A_i \forall \Gamma_i) \forall \kappa_{i+1}] = B_i(A + \Gamma)_i,$$

which is the bitwise AND between the third input B and the sum between the first two inputs A + Γ, as desired. Next consider the bitwise logical OR operation. From row 9 of Table 6,

$$\Gamma_{GENi} = 0 + 1 + B_i 0 + \overline{B}_i 0 = 1,$$

$$\Xi_{SXMi} = B_i 0 = 0,$$

$$\Xi_{SOMi} = B_i 1 = B_i,$$

$$G^{\dagger n}_{i+1} = G^{*n}_{i+1} + B_i,$$

$$\mu_i = M_i 1 + \overline{M}_i 0 = M_i,$$

$$\theta_i = H_i 1 + \overline{H}_i 0 = H_i,$$

$$\Lambda_i = M_i\phi_{i+1} + H_i\overline{\phi_{i+1}} + B_i.$$

However, by the same arguments as for the logical operation being a bitwise AND, $M_i\phi_{i+1} + H_i\overline{\phi}_{i+1}$ represents the sum of A and Γ. Therefore,

$$\Lambda_i = (A_i \forall \Gamma_i \forall \kappa_{i+1}) + B_i,$$

where + represents bitwise logical OR, so that Λ is the bitwise logical OR between B and the sum A + Γ, where + represents addition, as desired. Finally, consider the bitwise logical EXCLUSIVE-OR operation. From row 10 of Table 6,

$$\Xi_{GENi} = 0 + 0 + B_i 0 + \overline{B}_i 1 = \overline{B}_i,$$

$$\Xi_{SXMi} = B_i 1 = B_i,$$

$$\Xi_{SOMi} = B_i 0 = 0,$$

$$G^{\dagger n}_{i+1} = G^{*n}_{i+1} + 0 = G^{*n}_{i+1},$$

$$\mu_i = M_i\overline{B}_i + \overline{M}_iB_i,$$

$$\theta_i = H_i\overline{B}_i + \overline{H}_iB_i,$$

$$\begin{aligned}
\Lambda_i &= (M_i\overline{B}_i + \overline{M}_iB_i)\phi_{i+1} + (H_i\overline{B}_i + \overline{H}_iB_i)\overline{\phi_{i+1}} + 0,\\
&= \overline{B}_i(M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}) + B_i(\overline{M}_i\phi_{i+1} + \overline{H_i\phi_{i+1}}),\\
&= \overline{B}_i(M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}) + B_i(\overline{M_iH_i}\phi_{i+1}\\
&\quad + \overline{M}_i\phi_{i+1} + \overline{M_iH_i\phi_{i+1}} + \overline{H_i\phi_{i+1}}),\\
&= \overline{B}_i(M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}) + B_i(\overline{M_iH_i} + \overline{M}_i\phi_{i+1} + \overline{H_i\phi_{i+1}}),\\
&= \overline{B}_i(M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}) + B_i[(\overline{M}_i + \overline{\phi_{i+1}})(\overline{H}_i + \phi_{i+1})],\\
&= \overline{B}_i(M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}) + B_i[\overline{M_i\phi_{i+1} + H_i\phi_{i+1}}],\\
&= \overline{B}_i(A + \Gamma)_i + B_i\overline{(A + \Gamma)_i},\\
&= B_i \forall (A + \Gamma)_i,
\end{aligned}$$

where + has the mixed usage of bitwise logical OR or addition, depending on the context, and where the final expression is the EXCLUSIVE-OR between the third input operand and the sum between the first two input operands as desired

- *Logical followed by logical*

For the add-followed-by-logical operations, it was shown that, for the specifications of Ξ_{AND}, Ξ_{OR}, and Ξ_{XOR} of rows 8–10 of Table 6, the desired logical operation between the sum of a 2–1 addition between the two input operands A and Γ and a third input operand B is executed. The logical-followed-by-logical operations are produced if the 2–1 addition can be set up to add the output of the logical block, L1, to zero. It has already been shown, for the logical-followed-by-add case, that for the operand setup given in rows 11–19 of Table 6 and with the setups for Ω_{OALASL}, Ω_{XOSL}, and Ω_{XAAL} specified in those rows, the desired logical operation is produced by L1 for OP1. It remains to be shown that the expressions defining the fast 3–1 ICALU implementation produce a 2–1 addition among the output, L1, and zero. Since $\Omega_4 = 0$ in these rows, the

value for σ_i is a don't care, because it is ANDed with Ω_4. Furthermore, $\lambda_i = 0$, since Ω_1, Ω_2, and Ω_3 are zero. Therefore,

$$H_i = [0\sigma_i + (A_i\ LOP1\ \Gamma_i)]\ \forall\ 0 = (A_i\ LOP1\ \Gamma_i),$$

$$G_i = [0\sigma_i + (A_i\ LOP1\ \Gamma_i)]0 = 0,$$

$$T_i = 0\sigma_i + (A_i\ LOP1\ \Gamma_i) + 0 = (A_i\ LOP1\ \Gamma_i) = H_i,$$

$$\Lambda_i = B_i\ LOP2\ (M_i\phi_{i+1} + H_i\overline{\phi_{i+1}}).$$

However, since $\phi_{32} = 0$ and $\lambda_{32} = 0$, and since $G_i = 0$, $\phi_i = 0$ for all $0 \le i \le 31$; therefore,

$$\Lambda_i = B_i\ LOP2\ (M_i 0 + H_i 1) = B_i\ LOP2\ (A_i\ LOP1\ \Gamma_i),$$

which is the desired result. Therefore, the 3–1 ICALU produces a result equivalent to the serial execution of two interlocked ALU instructions for all cases of instruction pairings and data dependencies of Table 4.

4. Concluding remarks

A 32-bit 3–1 ICALU, proposed by [13], to allow the execution of two interlocked ALU-type instructions in one cycle using an instruction-level parallel machine implementation has been proven to produce results equivalent to a serial execution of the instructions using a 2–1 ALU. The proof was shown by deriving tables which represent all possible requirements for the serial execution of the instructions followed by the generalization of the table to represent sets of instructions rather than the individual instructions themselves. Consequently, we have proven the equivalence of the 3–1 ICALU operations proposed in [13] with these generalized requirements of the serial execution of the instructions, thereby demonstrating the correctness of the proposed design of a high-speed ICALU presented in [13]. The fast implementation of the 3–1 ICALU can be implemented with only a 3–2 CSA logic stage of additional delay when compared with a 2–1 ALU, suggesting that incorporation of this implementation of the device may produce no impact on the cycle time of the machine.

References

1. P. M. Kogge, *The Architecture of Pipelined Computers*, McGraw-Hill Book Co., Inc., New York, 1981.
2. R. M. Tomasulo, "An Efficient Algorithm for Exploiting Multiple Arithmetic Units," *IBM J. Res. Develop.* **11,** 25–33 (January 1967).
3. R. D. Acosta, J. Kjelstrup, and H. C. Torng, "An Instruction Issuing Approach to Enhancing Performance in Multiple Functional Unit Processors," *IEEE Trans. Computers* **35,** 815–828 (September 1986).
4. A. Padegs, B. B. Moore, R. M. Smith, and W. Buchholz, "The IBM System/370 Vector Architecture: Design Considerations," *IEEE Trans. Computers* **37,** 509–520 (May 1988).
5. W. A. Wulf, "The WM Computer Architecture," *Computer Architecture News* **16,** 70–84 (March 1988).
6. W. A. Wulf and C. Y. Hitchcock III, "Apparatus for Reading To and Writing From Memory Streams of Data While Concurrently Executing a Plurality of Data Processing Operations," U.S. Patent 4,819,155, April 1989.
7. N. P. Jouppi, "The Nonuniform Distribution of Instruction-Level and Machine Parallelism and Its Effect on Performance," *IEEE Trans. Computers* **38,** 1645–1658 (December 1989).
8. N. P. Jouppi and D. W. Wall, "Available Instruction-Level Parallelism for Superscalar and Superpipelined Machines," *Proceedings of ASPLOS III*, ACM, 1989, pp. 272–282.
9. H. S. Warren, Jr., "Instruction Scheduling for the IBM RISC System/6000 Processor," *IBM J. Res. Develop.* **34,** 85–92 (January 1990).
10. R. R. Oehler and R. D. Groves, "IBM RISC System/6000 Processor Architecture," *IBM J. Res. Develop.* **34,** 23–36 (January 1990).
11. S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, "SCISM: A Scalable Compound Instruction Set Machine," *Technical Report TR01.C683*, IBM Glendale Laboratory, Endicott, NY, October 1992.
12. S. Vassiliadis, B. Blaner, and R. J. Eickemeyer, "On the Attributes of the SCISM Organization," *Computer Architecture News* **20,** 44–53 (September 1992).
13. S. Vassiliadis, J. Phillips, and B. Blaner, "Interlock Collapsing ALUs," *IEEE Trans. Computers*, 1992, accepted for publication.
14. S. Vassiliadis, J. Phillips, and B. Blaner, "ICU Design Considerations," *Technical Report TR01.C114*, IBM Glendale Laboratory, Endicott, NY, October 1991.
15. J. Phillips and S. Vassiliadis, "Condition Code and Overflow Determination for 3-Operand SCISM ALUS," *Technical Report TR01.C207*, IBM Glendale Laboratory, Endicott, NY, December 1991.
16. J. Phillips and S. Vassiliadis, "High Performance 3–1 Interlock Collapsing ALU," *IEEE Trans. Computers*, submitted for publication, 1992.
17. S. Vassiliadis and J. Phillips, "Interlock Collapsing SCISM ALU Design," *Technical Report TR01.C115*, IBM Glendale Laboratory, Endicott, NY, October 1991.
18. S. Vassiliadis and J. Phillips, "3–1 Arithmetic Logic Unit for RISC Architectures," *Technical Report TR01.C605*, IBM Glendale Laboratory, Endicott, NY, September 1992.
19. S. Vassiliadis, "Recursive Equations for Hardwired Binary Adders," *Int. J. Electron.* **67,** 201–213 (August 1989).
20. S. Vassiliadis, "A Comparison Between Adders with New Defined Carries and Traditional Schemes for Addition," *Int. J. Electron.* **64,** 617–626 (April 1988).
21. N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Instruction-Level Parallelism for Execution Interlock Collapsing," *Computer Architecture News* **20,** 38–43 (September 1992).
22. N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Interlock Collapsing ALU for Increased Instruction-Level Parallelism," *Conference Proceedings, Annual International Symposium on Microarchitecture*, Portland, OR, December 1992, pp. 149–157.
23. N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Execution Interlock Collapsing Under Restricted Memory Models," *Proceedings of ISCIS VII International Conference on Computer and Information Sciences*, Antalia, Turkey, November 1992, pp. 181–187.
24. N. Malik, R. J. Eickemeyer, and S. Vassiliadis, "Architectural Effects on Dual Instruction Issue with Interlock Collapsing ALUs," presented at the Twelfth Annual IEEE International Phoenix Conference on Computers and Communications, March 1993.

25. *ESA/370 Principles of Operation*, Order No. SA22-7200-0, 1989; available through IBM branch offices.
26. J. Phillips and S. Vassiliadis, "Early SCISM ALU Status Determination," *Technical Report TR01.C205*, IBM Glendale Laboratory, Endicott, NY, December 1991.

Received June 25, 1992; accepted for publication November 11, 1992

James E. Phillips *IBM Advanced Workstation Systems, 11400 Burnet Road, Austin, Texas 78758 (PHILJE at AUSVM6, phillips@vnet.ibm.com).* In 1974 Mr. Phillips received a B.S. degree in nuclear engineering from North Carolina State University, after which he worked in the field of radiation safety. He received the M.S.E.E. and the B.S.E.E. degrees from the University of Tennessee, Knoxville, in 1984 and 1982, respectively. Mr. Phillips is currently employed at IBM Austin, where his assignments include computer engineering and architecture, and the high-level design and implementation of new computer systems. Prior to his current assignment, he worked in the Glendale Laboratories, IBM Endicott, New York, contributing to the research on the SCISM processors. His research interests include parallel and pipelined architectures, VLSI, and computer arithmetic. Since joining IBM, Mr. Phillips has received the First Invention Filed Award, the second level of the Invention Achievement Award, and the first level of the Publication Achievement Award. He has nine patents currently on file.

Stamatis Vassiliadis *IBM Enterprise Systems, P.O. Box 950, Poughkeepsie, New York 12602 (STAMATIS at GLDVM2, stamatis@gdlvm2.vnet.ibm.com).* Dr. Vassiliadis received the Dr. Eng. degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1978. He is currently a Senior Engineer at the IBM Mid-Hudson Valley Laboratory, Poughkeepsie, and previously worked at the Glendale Laboratories, IBM Endicott, New York. His work assignments include the development of new computer organizations and architectures, high-level design and technical leadership in the implementation of new computer systems, and advanced research in a variety of computer-related topics. Previous work included participation in the design of the IBM 9370 Model 60 computer system. Since joining IBM he has received a number of awards, including ten levels of the Publication Achievement Award, 13 levels of the Invention Achievement Award, and an Outstanding Innovation Award for engineering/scientific hardware design in 1989. In 1990 he was awarded the most patents in IBM. His research interests include computer architecture, hardware design and functional testing of computer systems, parallel processors, computer arithmetic, EDFI for hardware implementations, neural networks, fuzzy logic and systems, and software engineering. Dr. Vassiliadis has been an Adjunct Professor in the School of Electrical Engineering, College of Engineering, Cornell University, Ithaca, New York, and in the Electrical Engineering Department at the Thomas J. Watson School of Engineering and Applied Science, State University of New York (S.U.N.Y.), Binghamton, New York. He is currently a Visiting Professor in the School of Electrical Engineering, College of Engineering, Cornell University, Ithaca, New York.