DECOUPLED ACCESS/EXECUTE COMPUTER ARCHITECTURES

James E. Smith

Department of Electrical and Computer Engineering
University of Wisconsin-Madison, Madison, Wisconsin 53706

Abstract

An architecture for improving computer performance is presented and discussed. The main feature of the architecture is a high degree of decoupling between operand access and execution. This results in an implementation which has two separate instruction streams that communicate via queues. A similar architecture has been previously proposed for array processors, but in that context the software is called on to do most of the coordination and synchronization between the instruction streams. This paper emphasizes implementation features that remove this burden from the programmer. Performance comparisons with a conventional scalar architecture are given, and these show that considerable performance gains are possible.

Single instruction stream versions, both physical and conceptual, are discussed with the primary goal of minimizing the differences with conventional architectures. This would allow known compilation and programming techniques to be used. Finally, the problem of deadlock in such a system is discussed, and one possible solution is given.

1. Introduction

It has long been known that a practical impediment to scalar computer performance is that any straightforward instruction decoding/issuing scheme has some bottleneck through which instructions pass at the maximum rate of one per clock period [1]. Furthermore, modern organizations additionally constrain instructions to issue in program sequence. Some potential instruction overlap is lost because later instructions that could issue may be be held up behind an earlier instruction being blocked due to conflicts. For example, studies by Foster and Riseman [2] and Tjaden and Flynn [3] have shown that average speedups of 1.7 to almost 1.9 times are possible by issuing instructions out of order or by allowing multiple instructions to issue at once. Sophisticated issue methods used in the CDC 6600 [4] and IBM 360/91 [5] were intended to achieve some of this performance gain, but these complex issue methods have been abandoned by their manufacturers, no doubt in large part because any performance improvement was more than offset by additional hardware design, debugging, and maintenance problems.

A second critical constraint on performance is time required for processor-memory communication. Current trends, both in hardware and software, tend to aggravate the memory communication problem. In hardware, the trend toward higher levels of integration has the effect of increasing the performance impact of all forms of inter-chip communication, including processor-memory communication. At the architectural level, the trend is toward elaborate virtual memory and protection methods. These tend to slow memory communication because of the required address translation and protection checks. The use of multiprocessors often means that individual processors must contend for memory resources. In addition, interconnection structures add delay both due to their size and additional contention. Cache memory becomes a less effective solution in multiprocessor systems due to the problem of maintaining coherence. At the software level, facilities for defining elaborate data types and structures are being developed. This causes an increase in the number of operations needed to check types, compute indices, etc. all of which adds to increased delay when accessing data. All of the above point to the need for processors that can diminish the effects of increased memory communication time.

This paper discussed a new type of processor architecture which separates its processing into two parts: access to memory to fetch operands and store results, and operand execution to produce the results. By architecturally decoupling data access from execution, it is possible to construct implementations that provide much of the performance improvement offered by complex issuing methods, but without significant design complexity. In addition, it can allow considerable memory communication delay to be hidden.

The architecture proposed here represents an evolutionary step, since a similar, but more restricted, separation of tasks appeared as early as STRETCH [6], and has been employed to some degree in several high performance processors, including those from IBM, Amdahl, CDC and CRAY. Recently, an array processor, the CSPI MAP 200 [7] has pushed the degree of access and execution decoupling beyond that in any of the mainframe computers mentioned above. The architecture of the MAP 200, is, of course, directed largely toward vector or array type calculations. In

0149-7111/82/0000/0112$00.75 © 1982 IEEE

addition it has a relatively "bare bones" architecture, as do other array processors, that places a great deal of responsibility for resource scheduling and interlocking on software. The benefits of a highly decoupled access/execute architecture go beyond array processor applications, however. The author was independently studying a virtually identical decoupling method in the context of high performance mainframe computers when he became aware of the MAP 200. As a result of the viewpoint taken in this study, the methods discussed here reflect a philosophy of reducing programmer responsibility (and compiler complexity) while achieving improved performance.

This paper begins with an overview of decoupled access/execute architectures. Then some specific implementation issues are discussed. These are handling of stores, conditional branches, and queues. All three of these are handled in new ways that remove the burden of synchronization and interlocking from software and place it in the hardware. Next, results of a performance analysis of the 14 Lawrence Livermore Loops [8] is given. This is followed by a discussion of ways that the two instruction streams of a decoupled access/execute architecture can be merged while retaining most, if not all, the performance improvement. Finally, a brief discussion of deadlock, its causes, detection and prevention is given.

2. Architecture Overview

In its simplest form, a decoupled access/execute (DAE) architecture is separated into two major functional units, each with its own instruction stream (Fig. 1). These are the Access Processor or A-processor and the Execute Processor or E-processor. Each unit has its own distinct set of registers, in the A-processor these are denoted as registers A0, A1, ..., in the E-processor they are X0, X1, ...

The two processors execute separate programs with similar structure, but which perform two different functions. The A-processor performs all operations necessary for transferring data to and from main memory. That is, it does all address computation and performs all memory read and write requests. It would also contain the operand cache, if the system has one. Data fetched from memory is either used internally in the A-processor, or is placed in a FIFO queue and is sent to the E-processor. This is the Access to Execute Queue, or AEQ.The E-processor removes operands from the AEQ as it needs them and places any results into a second FIFO queue, the Execute to Access Queue or EAQ.

The A-processor issues memory stores as soon as it computes the store address; it does not wait until the store data is received via the EAQ. Store addresses awaiting data are held internally in the Write Address Queue or WAQ. As data arrives at the A-processor via the EAQ, it is paired with the first address in the WAQ and is

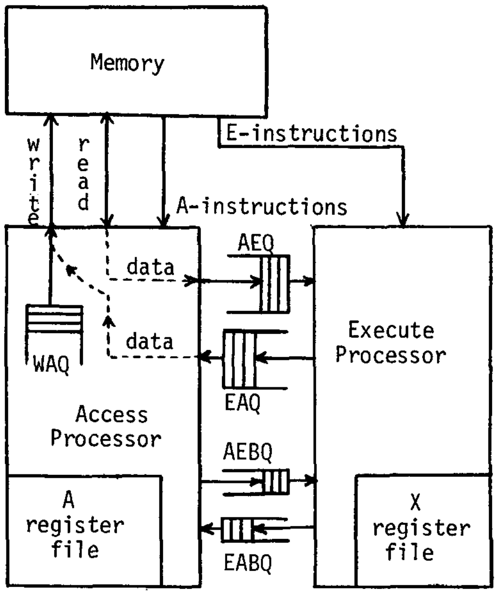

Fig. 1. Conceptual DAE Architecture

sent to memory. This pairing takes place automatically as the data becomes available. It should be noted that in [7] there is a third functional unit separate from the A- and E-processors that handles this write data/address pairing as one of its tasks.

The EAQ can also be used to pass data to the A-processor that is not stored into memory, but which is used for address calculation, for example. In this case, an instruction in the A-processor that reads from the EAQ must wait for the WAQ to be empty before it issues. Upon issuing it reads and removes the first element from the EAQ. In some instances it might be desirable to perform duplicate calculations in the two processors to avoid having the A-processor wait for results from the E-processor.

When producing software for a DAE architecture, the E- and A-processor programs have to be carefully coordinated so that data is placed into and taken out of the two data transmission queues in correct sequence. Each group of instructions is constrained to issue in sequence, but the two sequences may "slip" with respect to each other. In many cases, the accessing stream rushes ahead of the execute stream resulting in significantly less memory fetch delay.

Examples and preliminary performance comparisons given here are made with respect to a simplified CRAY-1-like scalar architecture. The CRAY-1 was chosen because:

1) The emphasis here is on high performance processors; the CRAY-1 represents the state-of-art in high performance scalar architecture and implementation.
2) The CRAY-1 has an instruction set that to some extent separates operand access and execution; this makes it easier to define and produce code for a comparable DAE architecture.

3) The CRAY-1 is a very straightforward design and instruction timings are predictable and relatively easy to calculate.

Example 1: Fig. 2a is one of the 14 Lawrence Livermore Loops (HYDRO EXCERPT) orginally written to benchmark scalar performance [8]. Fig. 2b is a "compilation" onto a stylized CRAY-1-like architecture. The scalar registers are labelled X0, X1, ... and there is only one set of scalar registers (instead of S and T registers in the CRAY-1). The address registers are labelled A0, A1, A2, ..., and there are no B registers. In Fig. 2, registers X0, X1, A0, and A1 are not used since they will later be given special meaning. For this reason the conditional branch (JAM) is assumed to use register A7 rather than A0. The compiled code is very similar to CRAY Assembly Language with arrows inserted for readability. Actual CRAY FORTRAN compiler output (with the vectorizer turned off) was used as a guide, so that the level of optimization and scheduling is what can be expected from a state-of-the-art optimizing compiler. For example, the addition of Q in the loop has been optimized away because Q = 0.0. Register allocation and handling of loop and index variables have been changed slightly to accomodate later examples.

Fig. 2c contains the A and E-programs for the straight-line section of code making up the loop. An example with branch instructions is deferred until branch instructions have been discussed. Performance comparisons are deferred still later until queue implementations have been discussed.

```
     q = 0.0
     Do 1  k = 1, 400
1    x(k) = q + y(k) * (r * z(k+10) + t * z(k+11))
```

Fig. 2a. Lawrence Livermore Loop 1 (HYDRO EXCERPT)

```
        A7 ← -400          . negative loop count
        A2 ← 0             . initialize index
        A3 ← 1             . index increment
        X2 ← r             . load loop invariants
        X5 ← t             . into registers
loop:   X3 ← z + 10, A2    . load z(k+10)
        X7 ← z + 11, A2    . load z(k+11)
        X4 ← X2 *f X3      . r*z(k+10)-flt. mult.
        X3 ← X5 *f X7      . t * z(k+11)
        X7 ←  y, A2        . load y(k)
        X6 ← X3 +f X4      . r*z(x+10)+t*z(k+11))
        X4 ← X7 *f X6      . y(k) * (above)
        A7 ← A7 + 1        . increment loop counter
        x, A2 ← X4         . store into x(k)
        A2 ← A2 + A3       . increment index
        JAM  loop          . Branch if A7 < 0
```

Fig. 2b. Compilation onto CRAY-1-like architecture

```
      Access              Execute
        .
        .
        .
AEQ ← z + 10, A2       X4 ← X2 *f AEQ
AEQ ← z + 11, A2       X3 ← X5 *f AEQ
AEQ ← y, A2            X6 ← X3 +f X4
A7 ← A7 + 1           EAQ ← AEQ *f X6
x, A2 ← EAQ                    .
A2  ← A2+ A3                   .
        .                      .
        .
        .
```

Fig. 2c. Access and execute programs for straight-line section of loop

3. Handling Memory Stores

As mentioned earlier, memory addresses for stores may be computed well in advance of when the data is available. These addresses are held in the WAQ, and as store data is passed over the EAQ, it is removed by the A-processor and lined up with its address in the WAQ before being sent to memory. The issuing of stores before data is available is an important factor in improving performance, because it allows load instructions to be issued without waiting for previous store instructions.

A problem that arises, however, is that a load instruction might use the same memory location (address) as a previously issued, but not yet completed, store. The solution in [7] is to provide the programmer with interlocks to hold stores from issuing until data is available when there is any danger of a load bypassing a store to the same location.

An alternative, but slightly more expensive, solution that relieves the programmer (or compiler) of inserting interlocks is to do an associative compare of each newly issued load address with all the addresses in the WAQ. If there is a match, then the load should be held (and all subsequent loads should be held, possibly by blocking their issue) until the match condition goes away. This associative compare would be a limiting factor on the size of the WAQ, but a size of 8 - 16 addresses seems feasible. Study of the performance impact of the WAQ length is being undertaken.

4. Conditional Branch Instructions

In order for the A- and E-processors to track each other, they must be able to coordinate conditional jumps or branches. It is proposed that FIFO queues also be used for this purpose. These are the E to A Branch Queue (EABQ) and A to E Branch Queue (AEBQ) in Fig. 1.

Either processor could conceivably have the data necessary to decide a conditional branch.

Consequently, each processor has a set of conditional branch instructions that use data residing within it. There is also a "Branch From Queue" (BFQ) instruction that is conditional on the branch outcome at the head of the branch queue coming from the opposite processor. When a processor determines a conditional branch outcome, it places it on the tail of the branch queue to the opposite processor. Thus conditional branches appear in the two processors as complementary pairs. If a conditional branch in the A-processor uses its own internal data, the conditional branch in the E-processor is a BFQ, and vice versa.

For performance reasons, it is desirable for the A-processor to determine as many of the conditional branches as possible. This reduces dependency on the E-processor and allows the A-processor to run ahead. Furthermore, if the A-processor is running ahead of the E-processor, branch outcomes in the AEBQ can be used by the instruction fetch hardware in the E-processor to reduce or eliminate instruction fetch delays related to conditional branches, i.e. it is as if the E-processor observes unconditonal branches rather than conditional ones. Often, as when a loop counter is also used as an array index, it happens naturally that the A-processor determines conditional branches.

5. Queue Architecture and Implementation

Thus far, the E- and A-processors have communicated via EAQ and AEQ which are explicit architectural elements. An architecturally cleaner alternative is to make some of the general purpose registers the queue heads and tails. For example, in the A-processor, A0 could be designated the head of the EAQ, and A1 the tail of the AEQ. Similarly, in the E-processor, X0 could be the head of the AEQ, and X1 the tail of the EAQ. In this way, no special instructions or addressing modes are needed to access the queues, they can be referred to just as registers are.

It might be convenient to give a processor access to the top two (or more) elements of a queue, as when the top two elements of a queue are both operands for an add or multiply. In this case, one could designate separate registers for each position in the queue to be accessed. For example, X0 and X1 could be the first two elements in the AEQ, with X2 being the tail of the EAQ. The instruction X2 ← X0 + X1 adds the first two members of the AEQ and returns the result on the EAQ. In the remainder of this paper, however, we give access only to the element at the head of a queue and use the register assignments as given in the previous paragraph.

The use of registers as queue heads and tails also suggests a convenient and efficient implementation. In Fig. 3 the AEQ is shown implemented as a standard circular buffer held in a register file. A "head counter" points to the element of the register file that is at the head of the queue. This counter controls the selection of head element.

There is a multiplexer that selects elements of the regular X register file, with the output of the AEQ file feeding the X0 input to the multiplexer. The space for X0 in the regular register file is not used; any read from X0 gets the element from the top of the queue and increments the head counter (modulo the AEQ size).

The "tail counter" points to the first open slot at the tail of the AEQ. A write by the A-processor into A1 also causes the data to be written into the tail position of the AEQ and increments the tail counter (modulo the AEQ size). When the head counter = tail counter + 1 (modulo the AEQ size) then the AEQ is considered to be full and no more data can be written until data is removed and the head counter is incremented. When the head counter = tail counter, then the queue is empty.

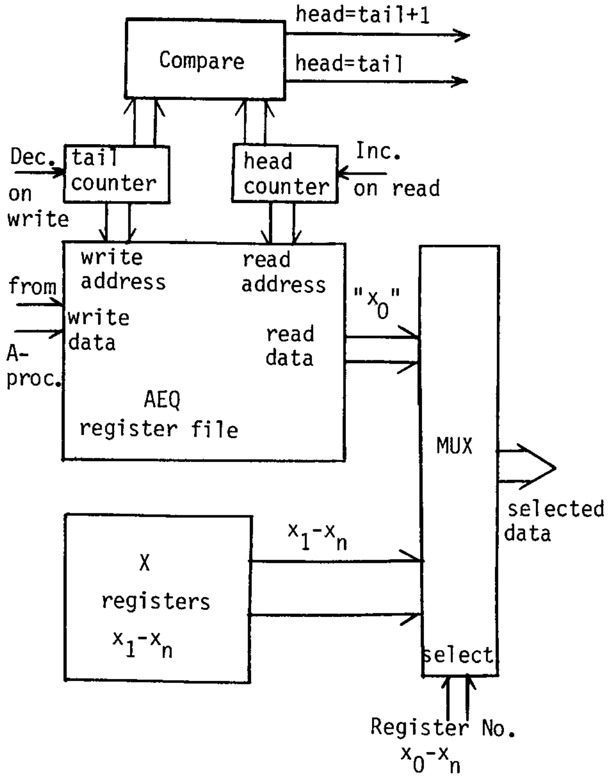

Fig. 3. An AEQ implementation

The design for the EAQ can be virtually identical to that for the AEQ. With this implementation, the EAQ fits quite naturally into the A-processor and the AEQ fits equally well into the E-processor. It appears that the time needed to access a queue, either for reading or writing, need not be any longer than to access a register file as demonstrated by the above implementation.

By using the above implementation with registers as queue heads and tails, certain register writes need to be flagged as illegal. Any write by the E processor into X0 would be illegal, and any write by the A-processor into A0 would be illegal. A more complicated structure could conceivably be used to allow these writes, but there appears to be little advantage to doing so.

The A-processor is allowed to read from A1, and the E-processor is allowed to read from X1. These registers hold the most recent elements entered into the AEQ and EAQ, respectively (these are actually duplicates since separate copies reside in the AEQ or EAQ register file). This read access is useful if a computed result is to be sent to the opposite processor, and is also needed for subsequent computation in the originating processor.

This method of using registers as queue heads and tails also simplifies testing queues for full and empty conditions. In a typical pipelined processor, e.g., the CRAY-1, a set of flip-flops, one for each register, is used to coordinate the reading and writing of registers so that register contents are read and updated in correct sequence. When an instruction issues that changes a register's contents, the corresponding flip-flop is set to designate the register as being busy. Any subsequent instruction using the register as an input or output operand encounters the busy bit and is blocked from issuing. After an instruction that modifies a register completes, the busy bit is cleared, and any instruction blocked by the bit is allowed to issue.

The queues can be checked for empty/full status by exactly the same busy bits. If the AEQ is empty, for example, the X0 busy bit is set so that any instruction needing an operand from the AEQ is blocked. Similarly, if the AEQ is full, then the A1 busy bit is set so that any instruction needing to place data into the queue is blocked. A similar implementation for the EAQ should also be used.

6. Performance Improvement

In this section, estimates of possible performance improvement with DAE architectures are made. These are based on a simplified CRAY-1 model first discussed in Example 1. To get single instruction stream estimates, the 14 Lawrence Livermore Loops were first hand compiled onto the stylized CRAY-1 scalar architecture. The actual object code generated by the CRAY FORTRAN compiler was used as a guide so that the level of code optimization and scheduling is realistic. The simplified architecture has only one level of scalar registers (A and X) and it was assumed that the number of scalar registers is not a limitation, although very seldom are more than 8 of each type needed.

Execution times were estimated in clock periods, using the number of clock periods required by the CRAY-1 for each operation, e.g., a load from memory is 11 clock periods, a floating add is 6 and a floating multiply is 7 [9]. It was assumed that there are no memory bank conflicts, and that loads can issue on consecutive clock periods. Branch instructions are assumed to require 5 clock periods for a taken branch, and 2 clock periods for a not taken one--i.e. optimum conditions are assumed.

The DAE hand compilation was extracted directly from the simplified CRAY-1 compilation. No further optimization or scheduling was done. Here, the same CRAY-1 execution times were used to estimate program execution time. Each of the two instruction streams was assumed to issue in program sequence, just as the simplified CRAY-1 was. The registers A0, X0, A1, X1 are used as queue heads and tails as discussed earlier.

As shown in secton 5 the time needed to communicate through a queue should be no longer than to communicate through a register. This was assumed in making the time estimates given below.

Fig. 4a shows the timings for the HYDRO EXCERPT loop. The simplifed CRAY-1 takes 39 clock periods for each pass through the loop (34 clock periods to get through the loop, plus 5 more for the taken branch at the bottom).

Fig. 4b show the timings in clock periods for the Access and Execute programs in the DAE version. In this program, the A-processor decides all the conditional branches and computes all addressing information itself. This means the A-processor is never delayed by the E-processor. The timings given assume complete independence between the two loops, although initially the E-program will be held up waiting for its first set of operands.

```
Issue Time

    0     loop:     X3 ← z + 10, A2
    1               X7 ← z + 11, A2
   11               X4 ← X2 *f X3
   12               X3 ← X5 *f X7
   13               X7 ← y, A2
   19               X6 ← X3 +f X4
   25               X4 ← X7 *f X6
   26               A7 ← A7 + 1
   32               x, A2 ← X4
   33               A2 ← A2 + A3
   34               JAM loop
```

Fig. 4a. Timing estimate for HYDRO EXCERPT loop on simplified CRAY-1

```
Issue Time              Access

    0     loop:          A1← z + 10, A2
    1                    A1 ← z + 11, A2
    2                    A1 ← y, A2
    3                    A7 ← A7 + 1
    4                    x, A2 ← A0
    5                    A2 ← A2 + A3
    6                    JAM loop
```

```
Issue Time              Execute

    0     loop:          X4 ← X2 *f X0
    1                    X3 ← X5 *f X0
    8                    X6 ← X3 +f X4
   14                    X1 ← X0 *f X6
   15                    BFQ loop
```

Fig. 4b. Timing estimate for HYDRO EXCERPT loop on DAE as architecture

The A-processor can make each pass through its loop in 11 clock periods (including the 5 for the taken branch). The E-processor takes 20 clock periods and would lag behind the A-processor. Nevertheless, after the first two passes through its loop (where there is a wait by the E-processor for AEQ) the computation proceeds at the rate of 20 clock periods per iteration--nearly twice the speed of the single stream version. This improvement is due entirely to the decoupling of access from execution.

Loop	Simplified CRAY-1 Time	DAE Arch. Time	Speedup
1	39	20	1.95
2	53	27	1.96
3	27	13	2.08
4	31	13	2.38
5	65	38	1.71
6	59	39	1.51
7	71	55	1.29
8	178	112	1.59
9	94	60	1.57
10	87	55	1.58
11	25	19	1.32
12	25	10	2.50
13	132	120	1.10
14	147	105	1.40
		Average	1.71

Fig. 5. Performance estimates for the 14 Lawrence Livermore Loops. All times are in clock periods per loop iteration.

All fourteen of the original Lawrence Livermore Loops were analyzed as just described. The results are given in Fig. 5. The speedup is computed by dividing the simplified CRAY-1 clock periods by the DAE architecture clock periods. The average speedup is just over 1.7 with some speedups as high as 2.5. By using two processors a speedup of greater than two is achieved because the issue logic in a pipelined processor typically spends more time waiting to issue instructions than actually issuing them. By using a DAE architecture the amount of waiting can be reduced considerably. If strictly serial processors were used then the maximum speedup would be two.

7. Single Instruction Stream DAE Architectures

While the dual instruction stream DAE architecture is conceptually simple and leads to straightforward implementations it does suffer some disadvantages. For the most part these are due to the human element--the programmer and/or compiler writer must deal with two interacting instruction streams. The programmer problem can be overcome if a high level language is used. This forces the work onto the compiler, however, and new techniques would probably need to be developed.

A disadvantage of secondary importance is that two separate instruction fetch and decode units are needed, one for each instruction stream. This might also require two ports into main memory for instructions rather than one. This hardware cost problem can probably be partially alleviated by using the same design for both instruction fetch/decode units.

In this section we briefly outline solutions to the above problems that

1) Physically merge the two instruction streams into one, or

2) Conceptually merge the two instruction streams for the purpose of programming and compilation, but leave them physically separate for execution.

The simplest way to physically achieve a single instruction stream is to "interleave" the instructions from the two streams. Let a_1, a_2, ..., a_n be the sequence of instructions in the A-program and let e_1, e_2, ..., e_m be the sequence of instructions in the E-program. An interleaving consists of combining the two sequences into one so that:

1) if a_i precedes a_j in the original A-program then a_i precedes a_j in the interleaved sequence,

2) if e_i precedes e_j in the original E-program then e_i precedes e_j in the interleaved sequence,

3) if a_k and e_ℓ are corresponding branch instructions in the two sequences, i.e., a conditional branch and the corresponding

branch from queue or two corresponding unconditional branches, then a single branch instruction is placed in the interleaved sequence which satisfies the precedence constraints 1) and 2) for both a_k and e_ℓ.

As the two sequences are interleaved, a bit can be added to each nonbranch instruction, say as part of the opcode, to indicate the stream to which it originally belonged. After instructions are fetched from memory and decoded, the bit can be used to guide instructions to the correct processor for execution. Queues in front of the processors can be used to hold the decoded instructions so that the processors retain the freedom to "slip" with respect to each other. With this scheme, only one program counter is required, and the BFQ instructions are no longer needed.

It should be noted, however, that this approach reintroduces the one instruction per clock period bottleneck in the instruction fetch/decode pipeline. These would in some instances result in reduced performance.

Example 2: An interleaving of the HYDRO EXCERPT program is shown in Fig. 6. The processor to which each instruction belongs is noted in parentheses. This particular interleaving places an instruction sending data via a queue before the instruction in the other processor that receives the data.

```
loop:  A1 ← z + 10, A2 (A)
       A1 ← z + 11, A2 (A)
       X4 ← X2 *f X0   (E)
       X3 ← X5 *f X0   (E)
       X6 ← X3 *f X4   (E)
       A1 ← y, A2      (A)
       X1 ← X0*f X6    (E)
       A7 ← A7 + 1     (A)
       x, A2 ← A0      (A)
       A2 ← A2 +A3     (A)
       JAM loop
```

Fig. 6. An interleaved instruction stream.

The simple interleaving of the two instruction streams does little to alleviate programming and readability problems. The program in the example above is rather confusing when one is used to thinking of conventional architecture. These problems can be partially overcome by inserting "noise" instructions into the listing. These noise instructions do not result executable code, but make explicit the implicit data transfers done via the queues. That is, the "instruction" A0 ← X1 can be used to denote the transfer of information via the EAQ. This "instruction" would be inserted after the instruction in the E-processor that places data into X1 and before the instruction in the A-processor that uses the data. This added "instruction" would be used only for programming or, in the case of a compiler, for bookkeeping purposes. It would not actually lead to any machine code.

Example 3. Fig. 7 shows an interleaving of the HYDRO EXCERPT with the A0 ← X1 and X0 ← A1 "noise instructions" inserted.

From the above example, it can be seen that we are very close to a conventional architecture which uses different registers for addressing and functional unit execution, i.e., the CDC and CRAY architectures. The only difference is that "copies" from X to A and A to X registers are restricted to take place among A0, A1, X0, and X1, and all memory loads and stores must take place via A registers.

```
loop:      A1 ← z + 10, A2     (A)
           X0  ← A1            (noise)
           X4 ← X2 *f X0       (E)
           A1 ← z + 11, A2     (A)
           X0 ← A1             (noise)
           X3 ← X5 *f X0       (E)
           X6 ← X3 +f X4       (E)
           A1 ← y, A2          (A)
           X0 ← A1             (noise)
           X1 ← X0 *f X6       (E)
           A0 ← X1             (noise)
           x, A2 ← A0          (A)
           A2 ← A2 + A3        (A)
           JAM loop
```

Fig. 7. An interleaved instruction stream with noise instructions inserted to enhance readability.

The architecture is now so similar to conventional architectures that many standard compiler techniques can probably be used. Then after compilation, "noise" instructions can be removed. If one physical instruction stream is to be used, the instructions can easily be "marked" with the processor they belong to.

If two instruction streams are to be used, then the compiler can pull apart the two instruction streams, with BFQ instructions being inserted.

The above discussion is by no means the last word on compilation for DAE architectures. As mentioned earlier, for performance reasons, dependency of the A-processor or E-processor results should be reduced so that the A-processor can run ahead of the E-processor. This can often be achieved by duplicating calculations in both processors. For high performance, a compiler would have to have this capability. Furthermore, the compiler would have other optimization and scheduling problems that differ from those encountered in a conventional architecture. It is clear that these and other research problems remain in the area of compilation for DAE architecture.

8. Deadlock

In a DAE architecture, deadlock can occur if both the AEQ and EAQ are full and both processors are blocked by the full queues, or if both queues are empty and both processors are blocked by the empty queues. An example of this is shown in Fig. 8. Here, the queues have once again been made explicit to make the problem clearer. Deadlock detection and prevention are both important problems. Deadlock can be detected by simply determining when instruction issue is being blocked in both processors due to full or empty queues. This should be flagged as a program error, and the program should be purged.

Access	Execute
A4 ← EAQ	X3 ← AEQ
AEQ ← A5	EAQ ← X2

Fig. 8. A solution which leads to deadlock: An attempted transfer from A5 to X3 and from X2 to A4.

Deadlock prevention is more complicated, and it is beyond the scope of this paper to go into detail. Rather, a sufficient condition for deadlock-free operation is informally given, and a way of achieving this sufficient condition is given.

Consider the dynamic instruction streams as they flow through the processors. For each data transfer through the EAQ or AEQ, there is an instruction in one processor that sends the data item, and an instruction in the other processor that receives the data item. The instruction that sends data item i is called "SEND i," and the instruction that receives data item i is called "RECEIVE i."

An interleaving of instructions (Section 7) is defined to be proper if the instruction causing SEND i precedes the instruction causing RECEIVE i for all data transfers i. The interleaving shown in Fig. 6 is a proper interleaving. Furthermore, a proper interleaving is needed when using the method of inserting noise instructions to improve readability as shown in Fig. 7.

It can be shown that if the A- and E-program can be properly interleaved then deadlock cannot occur. Again, it is beyond the scope of this paper to develop the formalism needed for a rigorous proof.

The program in Fig. 7 represents a proper interleaving for our HYDRO EXCERPT compilation, so the program must be deadlock-free. Turning to Fig. 8, it can be seen that it is impossible to properly interleave the A- and E-programs. To be proper, EAQ ← X2 must precede A4 ← EAQ and AEQ ← A5 must precede X3 ← AEQ. This can not be done since the definition of an interleaving requires that A4 ← EAQ must preceed AEQ ← A5 and X3 ← AEQ must preceed EAQ ← X2. Therefore the sufficient condition given above is not satisfied.

9. Conclusions

It has been shown that DAE Architectures can be implemented in ways that minimize programmer involvement. It has also been shown that considerable performance improvement is possible, while using straightforward instruction issue methods that are currently in use today. Furthermore, the improvement is achieved using code that is optimized roughly at the level of current compilers.

DAE architectures are relatively new, and many variations are possible. Multiple queues, additional processors, vector versions, and VLSI implementations are a few examples that deserve further study.

Acknowledgement

The author would like to thank David Anderson for his assistance in obtaining CRAY-1 object listings for the Livermore Loops.

References

[1] Flynn, M. J., "Very High-Speed Computing Systems," Proceedings of the IEEE, Vol 54, No. 12, pp. 1901-1909, December 1966.

[2] Riseman, E. M. and C. C. Foster, "Percolation of Code to Enhance Parallel Dispatching and Execution," IEEE Trans. on Computers, Vol. C-21, No. 12, pp. 1411-1415, December 1972.

[3] Tjaden, G. S. and M. J. Flynn, "Detection and Parallel Execution of Independent Instructions," IEEE Trans. on Computers, Vol. C-19, No. 10, pp. 889-895, October 1970.

[4] Thornton, J. E., Design of a Computer - The Control Data 6600, Scott, Foresman and Co., Glenview, IL, 1970.

[5] Anderson, D. W., F. J. Sparacio, and R. M. Tomasulo, "The IBM, System/360 Model 91: Machine Philosophy and Instruction Handling," IBM Journal of Research and Development, pp. 8-24, January 1967

[6] Bucholz, W., ed., Planning a Computer System, McGraw-Hill, New York, 1962.

[7] Cohler, E. U. and J. E. Storer, "Functionally Parallel Architecture for Array Processors," Computer, Vol. 14, No. 9, pp. 28-36, September 1981.

[8] McMahon, F. H., "FORTRAN CPU Performance Analysis," Lawrence Livermore Laboratories, 1972.

[9] CRAY-1 Computer Systems, Hardware Reference Manual, Cray Research, Inc., Chippewa Falls, WI, 1979.