R. M. Tomasulo

An Efficient Algorithm for Exploiting Multiple Arithmetic Units

Abstract: This paper describes the methods employed in the floating-point area of the System/360 Model 91 to exploit the existence of multiple execution units. Basic to these techniques is a simple common data busing and register tagging scheme which permits simultaneous execution of independent instructions while preserving the essential precedences inherent in the instruction stream. The common data bus improves performance by efficiently utilizing the execution units without requiring specially optimized code. Instead, the hardware, by 'looking ahead' about eight instructions, automatically optimizes the program execution on a local basis.

The application of these techniques is not limited to floating-point arithmetic or System/360 architecture. It may be used in almost any computer having multiple execution units and one or more 'accumulators.' Both of the execution units, as well as the associated storage buffers, multiple accumulators and input/output buses, are extensively checked.

Introduction

After storage access time has been satisfactorily reduced through the use of buffering and overlap techniques, even after the instruction unit has been pipelined to operate at a rate approaching one instruction per cycle,[1] there remains the need to optimize the actual performance of arithmetic operations, especially floating-point. Two familar problems confront the designer in his attempt to balance execution with issuing. First, individual operations are not fast enough* to allow simple serial execution. Second, it is difficult to achieve the fastest execution times in a universal execution unit. In other words, circuitry designed to do both multiply and add will do neither as fast as two units each limited to one kind of instruction.

The first step toward surmounting these obstacles has been presented,[2] i.e., the division of the execution function into two independent parts, a fixed-point execution area and a floating-point execution area. While this relieves the physical constraint and makes concurrent execution possible, there is another consideration. In order to secure a performance increase the program must contain an intimate mixture of fixed-point and floating-point instructions. Obviously, it is not always feasible for the programmer to arrange this and, indeed, many of the programs of greatest interest to the user consist almost wholly of floating-point instructions. The subject of this paper, then, is the method used to achieve concurrent execution of floating-point instructions in the IBM System/360 Model 91. Obviously, one begins with multiple execution units, in this case an adder and a multiplier/divider.[1]

It might appear that achieving the concurrent operation of these two units does not differ substantially from the attainment of fixed-floating overlap. However, in the latter case the architecture limits each of the instruction classes to its own set of accumulators and this guarantees independence.* In the former case there is only one set of accumulators, which implies program-specified sequences of dependent operations. Now it is no longer simply a matter of classifying each instruction as fixed-point or floating-point, a classification which is independent of previous instructions. Rather, it is a question of determining each instruction's relationship with all previous, incompleted instructions. Simply stated, the objective must be to preserve essential precedences while allowing the greatest possible overlap of independent operations.

This objective is achieved in the Model 91 through a scheme called the common data bus (CDB). It makes possible maximum concurrency with minimal effort (usually none) by the programmer or, more importantly, by the compiler. At the same time, the hardware required is small and logically simple. The CDB can function with any number of accumulators and any number of execution units. In short, it provides a hardware algorithm for the automatic, efficient exploitation of multiple execution units.

* During the planning phase, floating-point multiply was taken to be six cycles, divide as eighteen cycles and add as two cycles. A subsequent paper[3] explains how times of 3, 12, and 2 were actually achieved. This permitted the use of only one, instead of two, multipliers and one adder, pipelined to start an add cycle.

* Such dependencies as exist are handled by the store-fetch sequencing of the storage bus and the condition code control described in the following paper.[2]

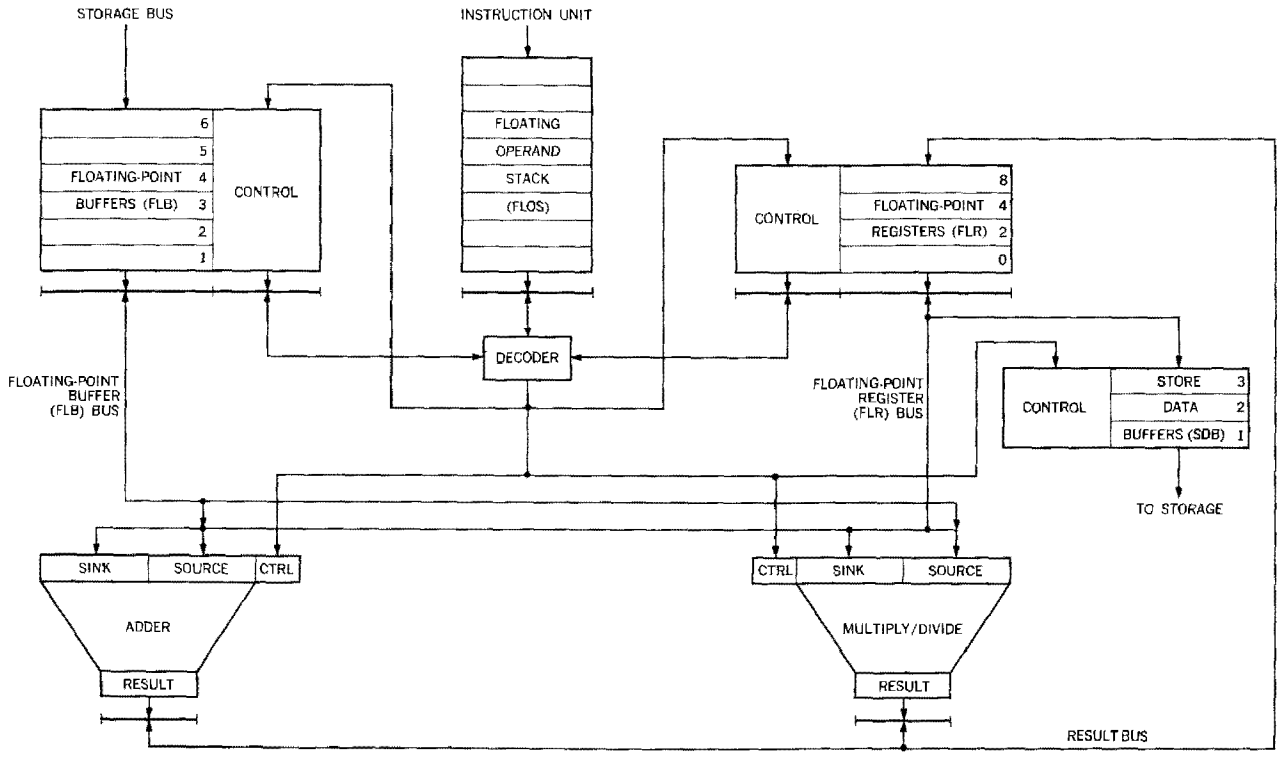


Figure 1 Data registers and transfer paths without CDB.

The next section of this paper will discuss the physical framework of registers, data paths and execution circuitry which is implied by the architecture and the overall CPU structure presented in a previous paper.[1] Within this framework one can subsequently discuss the problem of precedence, some possible solutions, and the selected solution, the CDB. In conclusion will be a summary of the results obtained.

Definitions and data paths

While the reader is assumed to be familiar with System/360 architecture and mnemonics, the terminology as modified by the context of the Model 91 organization will be reviewed here. The instruction unit, in preparing instructions for the floating-point operation stack (FLOS), maps both storage-to-register and register-to-register instructions into a pseudo-register-to-register format. In this format R1 is always one of the four floating-point registers (FLR) defined by the architecture. It is usually the *sink* of the instruction, i.e., it is the FLR whose contents are set equal to the result of the operation. Store operations are the sole exception* wherein R1 specifies the *source* of the operand to be placed in storage. A word in storage is really the sink of a store. (R1 and R2 refer to fields as defined by System/360 architecture.)

In the pseudo-register-to-register format "seen" by the FLOS the R2 field can have three different meanings. It can be an FLR as in a normal register-to-register instruction. If the program contains a storage-to-register instruction, the R2 field designates the floating-point buffer (FLB) assigned by the instruction unit to receive the storage operand. Finally, R2 can designate a store data buffer (SDB) assigned by the instruction unit to store instructions. In the first two cases R2 is the *source* of an operand; in the last case it is a sink. Thus, the instruction unit maps all of storage into the 6 floating-point buffers and 3 store data buffers so that the FLOS sees only pseudo-register-to-register operations.

The distinction between source and sink will become quite important during the discussion of precedence and should be fixed firmly in mind. All of the instructions (except store and compare) have the following form:

R1	op	R2 ⟶ R1
Register		Register Register
		or
		buffer
source		source sink

* Compares do not, of course, alter the contents of R1.

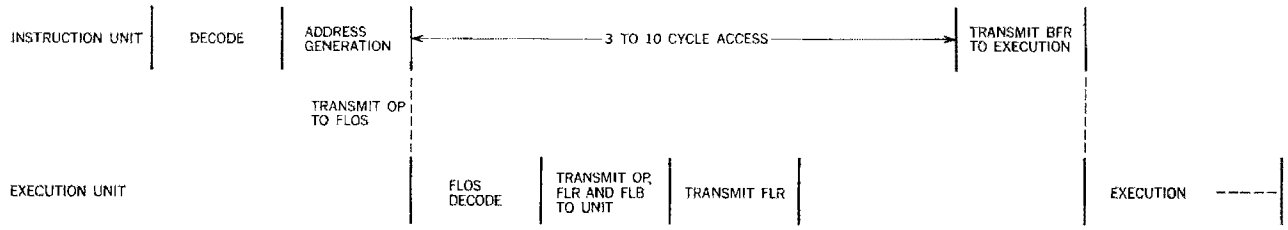


Figure 2 Timing relationship between instruction unit and FLOS decode for the processing of one instruction.

For example, the instruction AD0, 2 means "place the double-precision sum of registers 0 and 2 in register 0," i.e., R0 + R2 → R0. Note that R1 is really both a source and a sink.* Nevertheless, it will be called the sink and R2 the source in all subsequent discussion.

This definition of operations and the machine organization taken together imply a set of data registers with transfer paths among them. These are shown in Fig. 1. The major sets of registers (FLR's, FLB's, FLOS and SDB's) have already been discussed, both above and in a preceding paper.[1] Two additional registers, one sink and one source, are shown feeding each execution circuit. Initially these registers were considered to be the internal working registers required by the execution circuits and put to multiple use in a way to be described below. Later, their function was generalized under the reservation station concept and they were dissociated from their "working" function.

In actually designing a machine the data paths evolve as the design progresses. Here, however, a complete, first-pass data path will be shown to facilitate discussion. To illustrate the operation let us consider, in turn, four kinds of instructions—load of a register from storage, storage-to-register arithmetic, register-to-register arithmetic, and store. Let us first see how each can be accomplished *in vacuo*; then what difficulties arise when each is embedded in the context of a program. For simplicity double-precision (64-bit operands) will be used throughout.

Figure 2 shows the timing relationship between the instruction unit's handling of an instruction and its processing by the FLOS decode. When the FLOS decodes a load, the buffer which will receive the operand has not yet been loaded from storage.† Rather than holding the decode until the operand arrives, the FLOS sets control bits associated with the buffer which cause its content to be transmitted to the adder when it "goes full." The adder receives control information which causes it to send data to floating-point register R1, when its source register is set full by the buffer.

If the instruction is a storage-to-register arithmetic function, the storage operand is handled as in load (control bits cause it to be forwarded to the proper unit) but the floating-point register, along with the operation, is sent by the decoder to the appropriate unit. After receiving the buffer the unit will execute the operation and send the result to register R1.

In register-to-register arithmetic instructions two floating point registers are transmitted on successive cycles to the appropriate execution unit.

Stores are handled like storage-to-register arithmetic functions, except that the content of the floating-point register is sent to a store data buffer rather than to an execution unit.

Thus far, the handling of one instruction at a time has proven rather straightforward. Now consider the following "program":

Example 1

```
LD   F0  FLB1  LOAD register F0 from buffer 1
MD   F0  FLB2  MULTIPLY register F0 by buffer 2
```

The load can be handled as before, but what about the multiply? Certainly F0 and FLB2 cannot be sent to the multiplier as in the case of the isolated multiply, since FLB1 has not yet been set into F0.* This sequence illustrates the cardinal precedence principle: No floating-point register may participate in an operation if it is the sink of another, incompleted instruction. That is, a register cannot be used until its contents reflect the result of the most recent operation to use that register as its sink.

The design presented thus far has not incorporated any mechanism for dealing with this situation. Three functions must be required of any such mechanism:

(1) It must recognize the existence of a dependency.

* This economy of specification compounds the difficulties of achieving concurrency while preserving precedence, as will be seen later.

† A FULL/EMPTY control bit indicates this. The bit is set FULL by the Main Storage Control Element and EMPTY when the buffer is used. LOAD uses the adder in order to minimize the buffer outgates and the FLR ingates.

* Note that the program calls for the product of FLB1 and FLB2 to be placed in F0. This hints at the CDB concept.

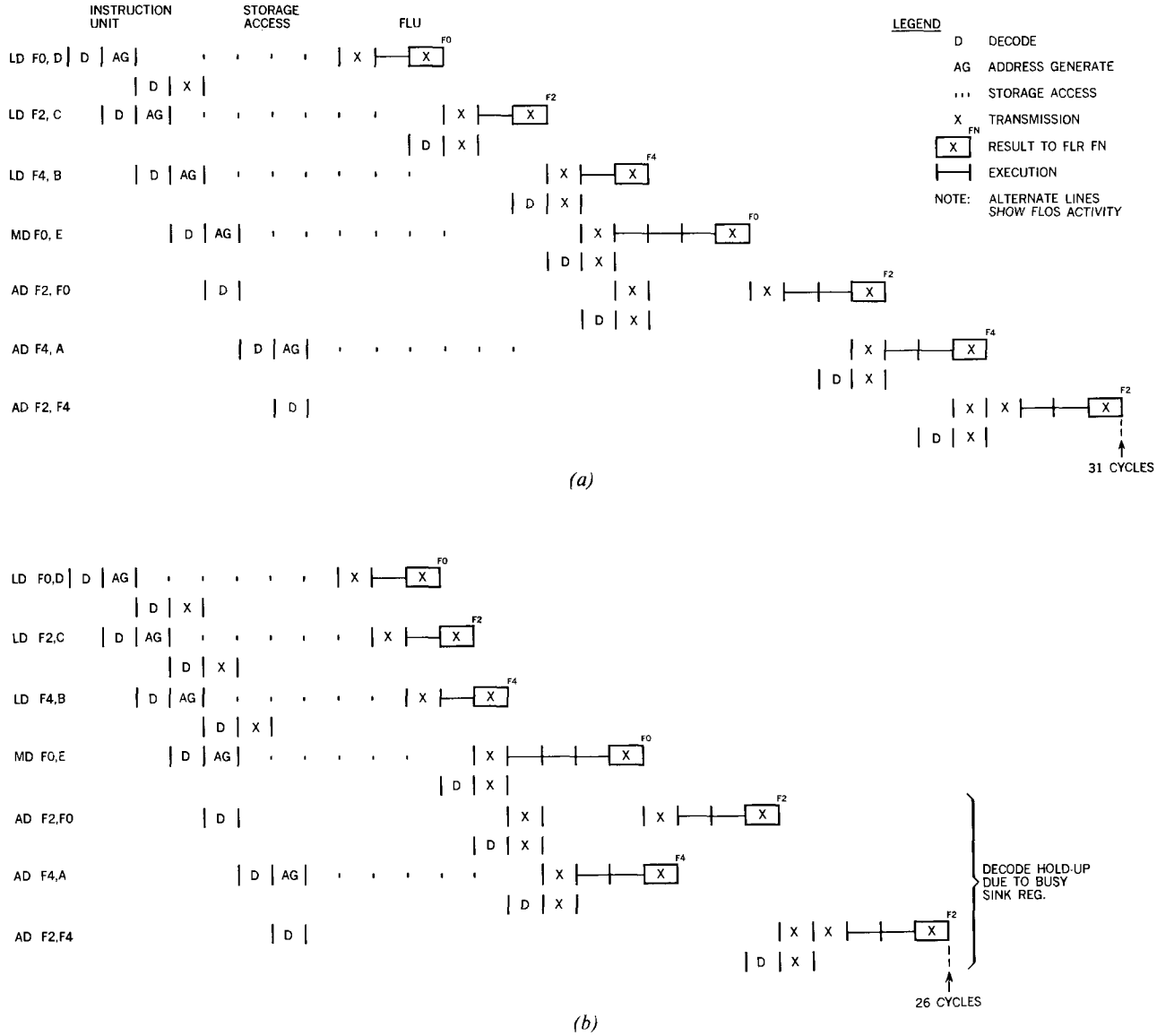


Figure 3 Timing for the instruction sequence required to perform the function A + B + C + D * E : (a) without reservation stations, (b) with reservation stations included in the register set.

(2) It must cause the correct sequencing of the dependent instructions.

(3) It must distinguish between the given sequence and such sequences as

LD F0, FLB1
MD F2, FLB2

Here it must allow the independent MD to proceed regardless of the disposition of the LD.

The first two requirements are necessary to preserve the logical integrity of the program; the third is necessary to meet the performance goal. The next section will present several alternatives for accomplishing these objectives.

Preservation of precedence

Perhaps the simplest scheme for preserving precedence is as follows. A "busy" bit is associated with each of the four floating-point registers. This bit is set when the FLOS decode issues an instruction designating the register as a sink; it is reset when the executing unit returns the result to the register. No instruction can be issued by the FLOS if the busy bit of its sink is on. If the source of a register-to-register instruction has its busy bit on, the FLOS sets

control bits associated with the source register. When a result is entered into the register, these control bits cause the register to be sent via the FLR bus to the unit waiting for it as a source.

This scheme easily meets the first two requirements. The third is met with the help of the programmer; he must use different registers to achieve overlap. For example, the expression A + B + C + D * E can be programmed as follows:

Example 2

```
LD  F0, D     F0 = D
LD  F2, C     F2 = C
LD  F4, B     F4 = B
MD  F0, E     F0 = D * E
AD  F2, F0    F2 = C + D * E
AD  F4, A     F4 = A + B
AD  F2, F4    F2 = A + B + C + D * E
```

The busy bit scheme should allow the second add and the multiply to be executed simultaneously (really, in any order) since they use different sinks. Unfortunately, the timing chart of Fig. 3a shows not only that the expected overlap does not occur but also that many cycles are lost to transmission time. The overlap fails to materialize because the first add uses the result of the multiply, and the adder must wait for that result. Cycles are lost to control because so many of the instructions use the adder. The FLOS cannot decode an instruction unless a unit is available to execute it. When an assigned unit finishes execution, it takes one cycle to transmit the fact to the FLOS so that it can decode a waiting instruction. Similarly, when the FLOS is held up because of a busy sink register, it cannot begin to decode until the result has been entered into the register.

One solution that could be considered is the addition of one or more adders. If this were done and some programs timed, however, it would become apparent that the execution circuitry would be in use only a small part of the time. Most of the lost time would occur while the adder waited for operands which are the result of previous instructions. What is required is a device to collect operands (and control information) and then engage the execution circuitry when all conditions are satisfied. But this is precisely the function of the sink and source registers in Fig. 1. Therefore, the better solution is to associate more than one set of registers (control, sink, source) with each execution unit. Each such set is called a *reservation station.** Now instruction issuing depends on the availability of the appropriate kind of reservation station. In the Model 91 there are three add and two multiply/divide reservation stations. For simplicity they are treated as if they were actual units. Thus, in the future, we will speak of Adder 1 (A1), Adder 2 (A2), etc., and M/D 1 and M/D 2.

Figure 3b shows the effect of the addition of reservation stations on the problem running time: five cycles have been eliminated. Note that the second AD now overlaps the MD and actually executes before the first AD. While the speed increase is gratifying and the busy bit method easy to implement, there remains a dependence on the programmer. Note that the expression could have been coded this way:

Example 3a

```
LD  F0, E
MD  F0, D
AD  F0, C
AD  F0, B
AD  F0, A
```

Now overlap is impossible and the program will run six cycles longer despite having two fewer instructions. Suppose however, that this program is part of a loop, as below:

Example 3b

```
LOOP 1  LD  F0, Ei
        MD  F0, Di
        AD  F0, Ci
        AD  F0, Bi
        AD  F0, Ai
        STD F0, Fi
        BXH i, -1, 0, LOOP 1 (decrease i by 1,
        branch if i > 0)
LOOP 2  LD  F0, Ei
        LD  F2, Ei + 1
        MD  F0, Di
        MD  F2, Di + 1
        AD  F0, Ci
        AD  F2, Ci + 1
        AD  F0, Bi
        AD  F2, Bi + 1
        AD  F0, Ai
        AD  F2, Ai + 1
        STD F0, Fi
        STD F2, Fi + 1
        BXH i, -2, 0, LOOP 2
```

Iteration $n + 1$ of LOOP 1 will appear to the FLOS to depend on iteration n, since the instructions in both iterations have the same sink. But it is clear that the two iterations are, in fact, independent. This example illustrates a second way in which two instruction sequences can be independent. The first way, of course, is for the two strings to have different sink registers. The second way is for the second string to begin with a load. By its definition a

* The fetch and store buffers can be considered as specialized, one-operand reservation stations. Previous systems, such as the IBM 7030, have in effect employed one "reservation station" ahead of each execution unit. The extension to several reservation stations adds to the effectiveness of the execution hardware.

load launches a new, independent string because it instructs the computer to destroy the previous contents of the specified register. Unfortunately, the busy bit scheme does not recognize this possibility. If overlap is to be achieved with this scheme, the programmer must write LOOP 2. (This technique is called *doubling* or *unravelling*. It requires twice as much storage but it runs faster by enabling two iterations to be executed simultaneously.)

Attempts were made to improve the busy bit scheme so as to handle this case. The most tempting approach is the expansion of the bit into a counter. This would appear to allow more than one instruction with a given sink to be issued. As each is issued, the FLOS increments the counter; as each is executed the counter is decremented. However, major difficulty is caused by the fact that storage operands do not return in sequence. This can cause the result of instruction $n + 1$ to be placed in a register before that of n. When n completes, it erroneously destroys the register contents.

Some of the other proposals considered would, if implemented, have been of such logical complexity as to jeopardize the achievement of a fast cycle.

The Common Data Bus

The preceding sections were intended to portray the difficulties of achieving concurrency among floating-point instructions and to show some of the steps in the evolution of a design to overcome them. It is clear, in retrospect, that the previous algorithms failed for lack of a way to uniquely identify each instruction and to use this information to sequence execution and set results into the floating-point registers. As far as action by the FLOS is concerned, the only thing unique to a particular instruction is the unit which will execute it. This, then, must form the basis of the common data bus (CDB).

Figure 4 shows the data paths required for operation of the CDB.* When Fig. 4 is compared with Fig. 1 the following changes, in addition to the reservation stations, are evident: Another output port has been added to the buffers. This port has been combined with the results from the adder and multiplier/divider; the combination is the CDB. The CDB now goes not only to the registers but also to the sink and source registers of all reservation stations, including the store data buffers but excluding the floating-point buffers. This data path will enable loads to be executed without the adder and will make the result of any operation available to all units without first going through a floating-point register.

Note that the CDB is fed by all units that can alter a register and that it feeds all units which can have a register as an operand. The control part of the CDB enumerates the units which feed the CDB. Thus the floating-point buffers 1 through 6 are assigned the numbers 1 through 6; the three adders (actually reservation stations) are numbered 10 through 12; the two multiplier/dividers are 8 and 9. Since there are eleven contributors to the CDB, a four-bit binary number suffices to enumerate them. This number is called a *tag*. A tag is associated with each of the four floating-point registers (in addition to the busy bit*), with both the source and sink registers of each of the five reservation stations and with each of the three Store Data Buffers. Thus a total of 17 four-bit tag registers has been added, as shown in Fig. 4.

Tags also appear in another context. A tag is generated by the CDB priority controls to identify the unit whose result will next appear on the CDB. Its use will be made clear shortly.

Operation of this complex is as follows. In decoding each instruction the FLOS checks the busy bit of each of the specified floating-point registers. If that bit is zero, the content of the register(s) may be sent to the selected unit via the FLR bus, just as before. Upon issuing the instruction, which requires only that a unit be available to execute it, the FLOS not only sets the busy bit of the sink register but also sets its tag to the designation of the selected unit. The source register control bits remain unchanged. As an example, take the instruction, AD F0, FLB1. After issuing this instruction to Adder 1 the control bits of F0 would be:

BB	TAG
1	1010 (A1)

So far the only change from previous methods is the setting of the tag. The significant difference occurs when the FLOS finds the busy bit on at decode time. Previously, this caused a suspension of decoding until the bit went off. Now the FLOS will issue the instruction and update the tag. In so doing it will not transmit the register contents to the selected unit but it will transmit the "old" tag. For example, suppose the previous AD was followed by a second AD. At the end of the decode of this second AD, F0's control bits would be:

BB	TAG
1	1011 (A2)

One cycle later the sink tag of the A2 reservation station would be 1010, i.e., the same as A1, the unit whose result will be required by A2.

Let us look ahead temporarily to the execution of the first AD. Some time after the start of execution but before the end,† A1 will request the CDB. Since the CDB is fed by many sources, its time-sharing is controlled by a central

* The FLB and FLR busses are retained for performance reasons. Everything could be done by a slight extension of the CDB but time would be lost due to conflicts over the common facility.

* The busy bit is no longer necessary since its function can be performed by use of an unassigned tag number. However, it is convenient to retain it.

† Since the required lead time is two cycles, the request is made at the start of execution for an add-type instruction.

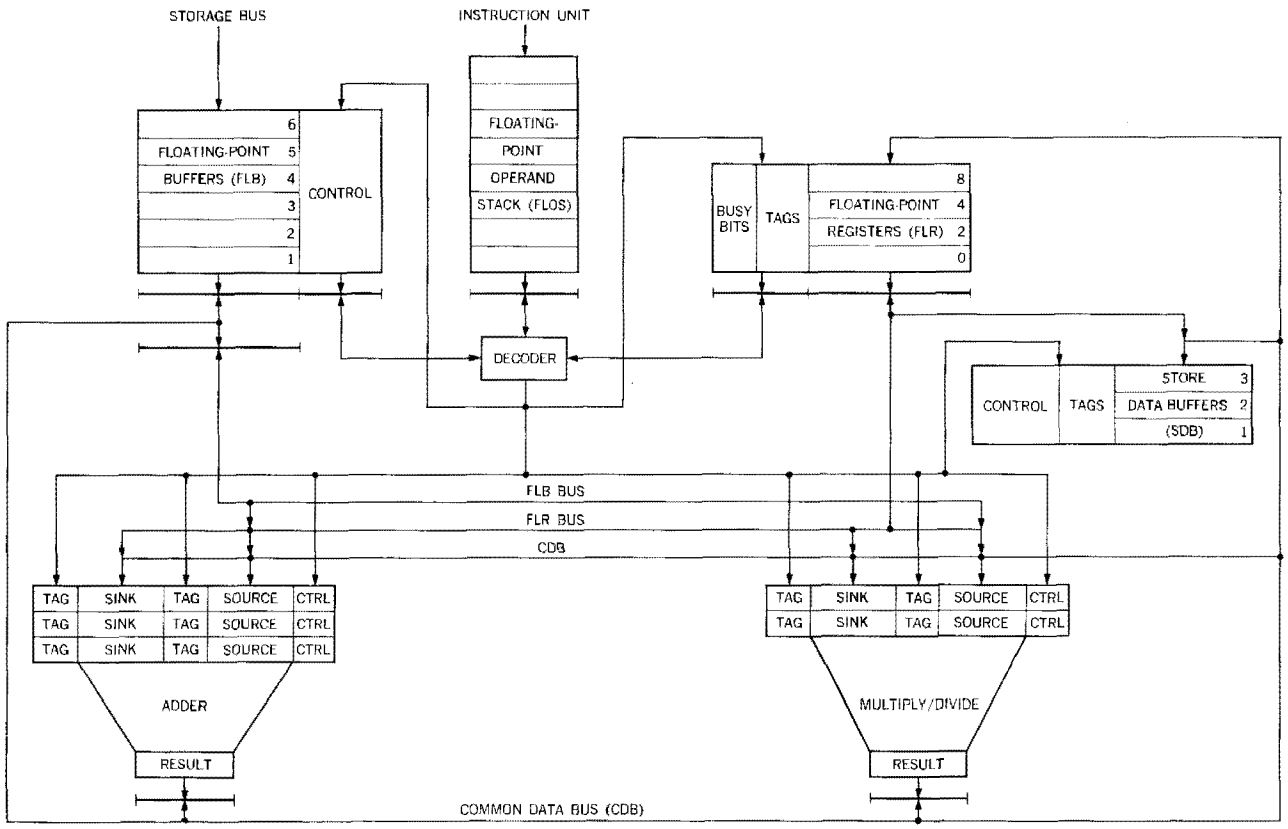


Figure 4 Data registers and transfer paths, including CDB and reservation stations.

priority circuit. If the CDB is free, the priority control signals the requesting adder, A1, to outgate its result *and* it broadcasts the tag of the requestor (1010 in this case) to all reservation stations. Each active reservation station (selected but awaiting a register operand) compares its sink and source tags to the CDB tag. If they match, the reservation station ingates the data from the CDB. In a similar manner, the CDB tag is compared with the tag of each *busy* floating-point register. All busy registers with matching tags ingate from the CDB and reset their busy bits.

Two steps toward the goal of preserving precedence have been accomplished by the foregoing. First, the second AD cannot start until the first AD finishes because it cannot receive both its operands until the result of the first AD appears on the CDB. Secondly, the result of the first AD cannot change register F0 once the second AD is issued, since the tag in F0 will not match A1. These are precisely the desired effects.

Before proceeding with more detailed considerations let us recapitulate the essence of the method. The floating-point register tag identifies the last unit whose result is destined for the register. When an instruction is issued that requires a busy register the tag is sent to the selected unit in place of the register contents. The unit continuously compares this tag with that generated by the CDB priority control. When a match is detected, the unit ingates from the CDB. The unit begins executing as soon as it has both operands. It may receive one or both operands from either the CDB or the FLR bus; the source operand for storage-to-register instructions is transmitted via the FLB bus.

As each instruction is issued the existing tag(s) is (are) transmitted to the selected unit and then the sink tag is updated. By passing tags around in this fashion, all operations having the same sink are correctly sequenced while other operations are allowed to proceed independently. Finally, the floating-point register tag controls the changing of the register itself, thereby ensuring that only the most recent instruction will change the register. This has the interesting consequence that a loop of the following kind:

Example 5

```
LOOP  LD    F0,  Ai
      AD    F0,  Bi
      STD   F0,  Ci  STORE
      BXH   i,  −1,  0,  LOOP
```

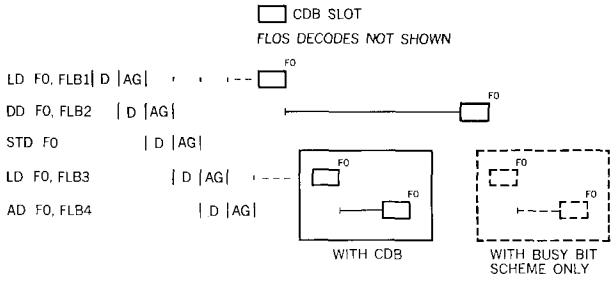


Figure 5 Timing sequence for Example 6, showing effect of CDB.

may execute indefinitely without any change in the contents of F0. Under normal conditions only the final iteration will place its result in F0.

As mentioned previously, there are two ways of starting an independent instruction string. The first is to specify a different sink register and the second is to load a register. The CDB handles the former in essentially the same way as the busy bit scheme. The load, which had been a difficult problem previously, is now very simple. Regardless of the register tag or busy bit, a load turns the busy bit on and sets the tag equal to the floating-point buffer which the instruction unit had assigned to the load. This causes subsequent instructions to sequence on the buffer rather than on whatever unit may have identified the register as its sink prior to the load. The buffer controls are set to request the CDB when the storage operand arrives. The following example and Fig. 5 show this clearly.

Example 6

```
LD      F0,  FLB1
DD      F0,  FLB2      DIVIDE
STD     F0,  A
LD      F0,  FLB3
AD      F0,  FLB4
```

Note that the add finishes before the divide. The dashed line portion of Fig. 5 shows what would happen if the busy bit scheme alone were used. Figure 6 displays the sequences followed under the two schemes. This figure graphically illustrates the bottleneck caused by using a single sink register with a busy bit scheme. Because all data must pass through this register, the program is reduced to strictly sequential execution, steps 1 through 7. With the CDB, on the other hand, the sink register hardly appears and the program is broken into two independent, concurrent sequences. This facility of the CDB obviates the need for loop doubling.

The CDB makes it possible to execute some instructions in, effectively, no time at all. In the above example the store took place during the CDB cycle following the divide. In a similar fashion a register-to-register load of a busy register is accomplished by moving the tag of the source floating-point register to the tag of the sink floating-point register. For example, in the sequence

```
AD      F0,  FLB1
LDR     F2,  F0      move F0 to F2
```

the tag of F0 will be 1010 (A1) at the time the LDR is decoded. The decoder simply sets F2's tag to 1010. Now, when the result of the AD appears on the CDB both F0 and F2 will ingate since the CDB tag of 1010 will match the tag of each register. Thus, no unit or extra time was required for the execution of the LDR.

A number of details have been omitted from this discussion in order to clarify the concept, but really only two are of operational significance. First, every unit must request the CDB two cycles before it finishes execution. (These two cycles are required for propagation of the request to the CDB controls, the establishment of priority among competing units, and propagation of a "select" signal to the chosen unit.) This limits the execution time of any instruction to a two-cycle minimum. (Of course, the faster the execution the less the need for, or gain from, concurrency.) It also adds one* cycle to the access time for loads. Because of buffering and overlap, this does not usually cause an increase in problem running time.

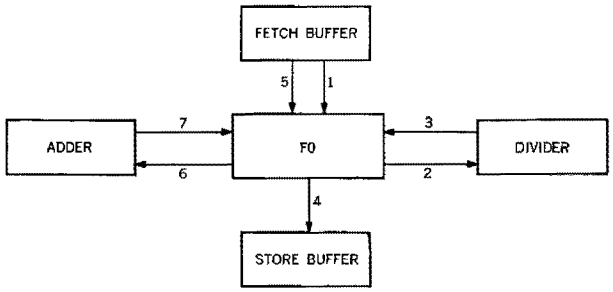
The second point is concerned with mixed precision. Because the architectural definition causes the low-order part of an FLR to be preserved during single-precision operation, an error can occur in the following kind of program:

```
LD      F0,  FLB1
AD      F0,  FLB2
AE      F0,  FLB3
```

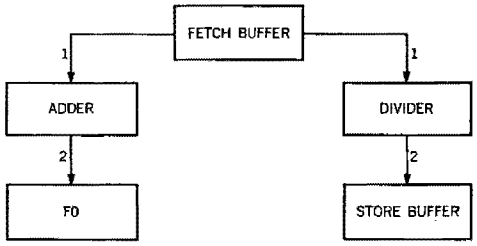
Since only the last instruction, which is single-precision, will change F0, the low order result of the double-precision AD will be lost. This is handled by associating a bit with each register to indicate whether a particular register is the sink of an outstanding single- or double-precision instruction. If this bit does not match the "length" of the instruction being decoded, the decode is suspended until the busy bit goes off. While this stratagem† solves the logic problem, it does so at the expense of performance. Unfortunately, no way has been found to avoid this. Note, however, that all-single- or all-double-precision programs run at the maximum possible speed. It is only the interface between single- and double-precision to the *same* sink register that suffers delay.

* It does not add two cycles since storage gives one cycle prenotification of the arrival of data.

† Further complications arise from the fact that single-precision multiply produces a double-precision product. This is handled separately but with the same time penalty as above.



(a)



(b)

Figure 6 Functional sequence for Example 6 (a) with busy bit controls only, (b) with CDB.

Conclusions

Two concepts of some significance to the design of high-performance computers have been presented. The first, reservation stations, is simply an expeditious method of buffering, in an environment where the transmission time between units is of consequence. Because of the disparity between storage access and circuit speeds and because of dependencies between successive operations, it is observed (given multiple execution units) that each unit spends much of its time waiting for operands. In effect, the reservation stations do the waiting for operands while the execution circuitry is free to be engaged by whichever reservation station fills first.

The second, and more important, innovation, the CDB, utilizes the reservation stations and a simple tagging scheme to preserve precedence while encouraging concurrency. In conjunction with the various kinds of buffering in the CPU, the CDB helps render the Model 91 less sensitive to programming. It should be evident, however, that the programmer still exercises substantial control over how much concurrency will occur. The two different programs for doing A + B + C + D * E illustrate this clearly.

It might appear that the CDB adds one cycle to the execution time of each operation, but in fact it does not. In practice only 30 nsec of the 60-nsec CDB interval are required to perform all of the CDB functions. The remaining time could, in this case, be used by the execution unit to achieve a shorter effective cycle. For example, if an add requires 120 nsec, then add plus the CDB time required is 150 nsec. Therefore, as far as the add is concerned, the machine cycle could be 50 nsec. Besides, even without the CDB, a similar amount of time would be required to transmit results both to the floating-point registers and back as an input to the unit generating the result.

The following program, a typical partial differential equation inner loop, illustrates the possible performance increase.

```
LOOP    MD      F0,  Ai
        AD      F0,  Bi
        LD      F2,  Ci
        SDR     F2,  F0
        MDR     F2,  F6
        AD2     F2,  Ci
        STD     F2,  Ci
        BXH     i,  -1,  0,  LOOP
```

Without the CDB one iteration of the loop would use 17 cycles, allowing 4 per MD, 3 per AD and nothing for LD or STD. With the CDB one iteration requires 11 cycles. For this kind of code the CDB improves performance by about one-third.

Acknowledgments

The author wishes to acknowledge the contributions of Messrs. D. W. Anderson and D. M. Powers, who extended the original concept, and Mr. W. D. Silkman, who implemented all of the central control logic discussed in the paper.

References

1. D. W. Anderson, F. J. Sparacio and R. M. Tomasulo, "The System/360 Model 91: Machine Philosophy and Instruction Handling," *IBM Journal* 11, 8 (1967) (this issue).
2. S. F. Anderson, J. Earle, R. E. Goldschmidt and D. M. Powers, "The System/360 Model 91 Floating-Point Execution Unit," *IBM Journal* 11, 34 (1967) (this issue).

Received September 16, 1965.