Detection and Parallel Execution of Independent Instructions

GAROLD S. TJADEN AND MICHAEL J. FLYNN, MEMBER, IEEE

Abstract—For a single instruction stream–single data stream organization the problem of simultaneously issuing several instructions is studied.

Within a block of N instructions, there exist dependencies which must be detected and ordered so that the maximum number of instructions can be issued for execution. These dependencies include memory accessing, data registers (sink-source), operand availability, and procedural dependency (conditional branch). A simple decoding mechanism is proposed to analyze these dependencies and provide maximum instruction issuance.

A simulator has been written to provide an evaluation of the aforementioned schemes using actual programs and formats of a representative computer (IBM 7094).

The results of this simulation indicate that an 86 percent performance improvement could be realized for the appropriately modified 7094, without use of compiler techniques or programmer assistance.

Index Terms—Executably independent instructions, instruction decoding, predecode stack, stack simulator, weakly independent instructions.

INTRODUCTION

A KEY problem in the design of high-speed, single instruction stream–single data stream computers is the limitation of serial issuing of instructions.

While instructions may be executed simultaneously or out of sequence, depending on the availability of resources, they are decoded serially in strict sequence (see Anderson [11], Tomasulo [12], and Flynn [8]). Indeed, while several computers (IBM System/360 Models 85, 91, 195, and CDC 6600 and 7600) decode at a maximum serial rate, i.e., one instruction decoded per machine cycle, no system attempts a multiple simultaneous decoding of blocks of instructions. The problem has been to organize the instruction format and the decoder to readily ascertain the instructions which qualify for simultaneous execution. The conditions for this "qualification" must be simply implemented as the cost to simultaneously decode n instructions will rise exponentially with n. Coupled with this is a degeneration in decode time directly related to n.

This paper will present a simple algorithm for simultaneously decoding instructions and will evaluate the algorithm on representative existing code.

Thus, out of a block of n instructions, the m independent instructions are immediately executed (as resources are available). Upon their execution, the remaining instructions, together with the next m instructions, are decoded for independency and those qualified proceed to execution.

In actual machine environment, the detection and issuing of instructions will overlap execution. The algorithm presented here will first discuss the nonoverlap case and then the relatively straightforward extension to the overlapped case.

INSTRUCTIONS

An instruction defines an ordered pair (x_1, x_2) of source operands, a uniform binary procedure OP to be performed on the source operands whose result range is a single operand f(x_1, x_2) = x_3 OP x_2, a specification of the location of the result (sink), and a specification of the location of the next instruction.

We take care here not to make the definition too broad. If we allow n-ary operands or nonuniform procedures, we will be dealing with programs or program segments rather than individual instructions. We add an additional restriction. The specification of the source operands, the sink result, and the next instruction must be made by defining its location in storage. This location is called the effective address. Only the following procedures are considered allowable in generating an effective address.
1) **Immediate Specification**: The contents of the operand specification field is the operand itself (this specification only has meaning for source operands).

2) **Implicit Specification**: As specified by the operation, any combination of the source operands, the sink, or the next instruction location is assumed to be contained in a particular register or memory location without further specification (e.g., the use of an accumulator and/or an instruction counter).

3) **Direct Specification**: The operand location field in the instruction is the effective address of the operand or next instruction.

4) **Indexed Specification**: The operand location field is added to the contents of one or more registers to form an effective operand (or next instruction) address. Each register so used must be implicitly or directly specified.

Specifically excluded from these procedures are indirect references made to registers or memory. An instruction with a single level indirect specification of an operand is considered as a composite of two instructions, one generating an address and the other using it.

**INDEPENDENT INSTRUCTIONS**

We now must determine whether two instructions $I_1, I_2$ can be executed simultaneously or whether their order must be preserved. Intuitively, we define two instructions $I_1$ and $I_2$ to be independent if the outcomes (i.e., sink operands) are invariant with respect to the ordering of execution of $I_1$ and $I_2$ for all data sets (source operands).

While this is an accurate definition, it is not particularly useful in establishing independence. To define a useful algorithm, consider the conditions (or resources) upon which an instruction depends for its execution as its dependencies, and the conditions (or resources) which it specifies as its effects.

In general (Fig. 1) there are three types (or classes) of dependencies: 1) procedural, 2) operational, and 3) data.

A procedural dependency is a dependency in the specification of the instruction sequence (or stream). If in each case the instruction explicitly specifies the address of the next instruction, then each instruction in a sequence is dependent on its predecessor in the sequence. If, as is more usual, we assume that instructions are arranged in an "in-line" sequence, i.e., one instruction in a sequence followed by the next, then only a branch out of the sequence causes a dependency. Thus, given a sequence (ascending in time) of $m$ instructions, if $I_i$ is a branch, then for $[I_i, I_{i+1}, \ldots, I_m]$ all the instructions $I_k$, $i < k < m$, are dependent on $I_i$.

If $I_i$ is an unconditional branch, then the dependency can be quickly resolved and removed by refetching the correct sequence. If $I_i$ is a data conditional branch, we must wait until the data test is made before the dependency can be resolved.

An operational dependency arises when a resource associated with the operation specified by instruction $I_i$ is busy. Thus we have a sequence $[OP_{i1}, \ldots, OP_{in}, \ldots, OP_{im}]$ where $OP_{ij}$ refers to the $j$th resource class which is specified by instruction $I_i$. If all instructions 1 through $m$ called for an ADD, we might not have sufficient ADD type resources to service these requests. For each OP, class, resources are distributed sequentially from 1 until they are exhausted (say at $i$); following the $i$th, all instructions using resource $j$ will be dependent, i.e., all $k$ $OP_{ij}, i < k < m$. For purposes of simplicity of notation we will not consider operational dependencies any further in this paper, since their treatment is straightforward.

A data dependency arises in a sequence of $m$ instructions when an instruction $I_i$ effects (i.e., has as a sink operand) the source operands of any other instruction $I_k, 1 < k < m$.

There are two cases, corresponding to different methods of accessing operands, which must be examined.

1) **Directly Addressed Data**: Given a general instruction format of the type $I(A_1, A_2)=A_4$ (or $C(A_1)^2$ OP $C(A_2)=C(A_4)$). The contents of the data cell directly specified by $A_1$ and $A_2$ are the source operands and $A_4$ is the direct address of the result. (This is similar to the register-register formats of familiar computers.) Then, given a sequence $(I(A_1, A_2), A_3, \ldots, I(A_1, A_2), A_4, \ldots, I(A_1, A_2), A_4, I), I_1$ is data independent, and $I_i$ is independent so long as its sources, designated $A_{i1}$ and $A_{i2}$, do not correspond to (are not equal to) $A_4$ for any $j: 1 \leq j < i$ and so long as its sink $A_{i4}$ is not equal to any $A_{j1}$, $A_{j2}$ for any $j: 1 \leq j < i$.

2) **Indexed Addressed Data**: Given a format of the type $I(A_1, f(A_2, A_3))=A_4$ (or, e.g., $C(A_1)OP(C(A_2) + C(A_3))=C(A_4)$) or any format that specifies at least one of its operands indirectly (by index), several additional problems arise. Even if $A_{12} \neq A_{22}$ and $A_{31} \neq A_{32}$, it is possible that $f(A_2, A_3)=f(A_2, A_3)$. In order to handle this possibility, we will need two levels of dependency testing.

First, let $E_i=f(A_2, A_3)$, the effective address of a source or sink operand of the $i$th instruction. Then if $E_i$ specifies a

$^1$ $A_4$ is always used to refer to the sink of the instruction. It is possible for some instructions to use more than one sink resource, in which case $A_4$ is understood to refer to the set of sink resources.

$^2$ The notation $C(A_1)$ means the contents of address location $A_1$.
source and \( E_i = A_{4j}, 1 \leq j < i \), instruction \( i \) is independent on
instruction \( j \). If \( E_i \) is used to specify a result (a sink) and
\( E_i = A_{1j} \) or \( E_i = f_j(A_2, A_3) \), \( 1 \leq j < i \), instruction \( i \) is dependent
on instruction \( j \).

In practice, the data storage media are nonhomogeneous
and the directly referenced operands \( A_1 \) and \( A_4 \) are registers
distinct from main storage to which \( E \) refers. We will use
this fact in the following discussion, so as to partition the
two levels of dependency consistent with the two levels of
storage. Note that this is convenient for discussion and that
the homogeneous storage case directly follows.

Thus, we have two levels of storage; one level is directly
addressed by source address \( A_1 \) and sink address \( A_4 \); the
other level is indexed and refers to main storage.

If the effects/dependencies of two instructions \( I_i \) and
\( I_j \) are compared, and if neither \( A_1 i \), \( A_2 i \), nor \( A_3 i = A_4 j \)
and neither \( A_1 j \), \( A_2 j \), nor \( A_3 j = A_4 i \), the instructions \( i \) and
\( j \) are said to be weakly independent.

In order to convert weakly independent instructions into
independent ones, we must first determine the dependencies
and effects of the effective addresses. We distinguish the
following conditions.

1) Any instruction \( I_i \) which effects one of the component
address operands \( (A_2, A_3) \) creates a dependency. Note
that this is detected in the test for weak independence.

2) The effective address \( E_i \) may be used as a sink (hence
an effect) or a source (hence representing a dependency).
When it is used as a sink, source addresses
must be compared with \( E_i \), and when \( E_i \) is used as a
source, sink addresses must be compared; \( E_j \) is dependent
on \( E_i (i < j) \) and therefore \( I_j \) is dependent on \( I_i \)
when either

\[
E_i \text{ sink} = E_j \text{ source}
\]

or

\[
E_i \text{ source} = E_j \text{ sink}.
\]

If there is no such correspondence then \( E_j \) is independent
and the corresponding data may be fetched. Note that
dependency of addresses implies dependency of instructions
but independency of addresses does not imply independency
of instructions (there may be other dependencies).

In order that this two-level detection proceed efficiently,
it is desirable that the data be fetched from storage asyn-
chronously from the instruction execution. To facilitate
this, when an effective address is generated, one of a set of
\( S \) registers, as shown in Fig. 2, is selected (as available) and
assigned as the sink for the operand—or source in the case
of a store type instruction. A use bit is also set.

As soon as the effective address is generated and an \( S \)
register selection is made, the \( S \) register designation is
appended to the \( A_2, A_3 \) specifications in the parent instruc-
tion. When an \( S \) register is used in the execution of an
instruction, the use bit is reset and the register is available for
reassignment.

These restrictions on data dependency are, however,
overly restrictive. Certain data initializing instructions,
called open-effects instructions, depend on the availability
of a register, rather than on the contents of the register [12].
This condition arises, for example, with instructions which
load an accumulator with a word from memory in a single-
address machine. A load instruction \( I_i \) such as this will be
executably independent if it is not dependent on a previous
instruction \( I_j, 1 \leq j < i \), for putting the correct data in the
memory word and if the accumulator is not going to be
used by a previous instruction. Suppose, however, that the
processor is provided with enough extra accumulators so
that one of them is idle. The restriction that the accumulator
is idle would not apply to our load instruction, thus improv-
ing its chances of being executably independent.

Instructions which can be executably independent under
these condition are characterized by the fact that they effect
something upon which they do not depend. We call them
open-effects instructions. The load instruction effects the
accumulator, but depends on a memory address, not on the
accumulator. These open-effects instructions can be checked
for independency by determining that their effects do not
match their dependencies, and that their dependencies do
not match the effects of any instructions above them. These
instructions need not pass the test matching their effects
with the dependencies of the instructions above them.

Fig. 3 illustrates (illustration follows that of Flynn [8]) the
effect of these open-effects instructions on the program
execution.

Thus, for directly addressed data (in the sequence as
before), an instruction \( I_l \) \((A_1, A_2, A_4)\) is independent if
\( A_{4i} \neq A_{2i} \), \( A_{4i} \neq A_{1i} \) and \( A_{2i} \neq A_{4j} \) and \( A_{1i} \neq A_{4j} \) for any \( j: \)
\( l \leq j < i \), and a suitable register is available to be labeled

---

**Fig. 2. Handling weakly independent instructions.**
Assume a situation in which we are given a serial string of instructions, along with a set of data which the instructions are to manipulate. The instructions are from the repertoire of a computer organized as a single instruction stream--single data stream (SISD [8]) processor. The general organization of a processor designed to make use of independent instruction detection would include (see Fig. 4)

1) a set \( A = \{A_i\}, i = 1, 2, \ldots, n \) of data registers consisting of \( n \) different types of resources (index registers, accumulators, memory, etc.) with \( m_i \) members of each type,
2) a set \( R = \{R_k\}, k = 1, 2, \ldots, p \) of arithmetic units,
3) a control unit,
4) a predecode stack (PDS), an effective address buffer, and the set of \( S \) registers.

The instruction format of this processor has the following restriction: the dependencies and effects of each instruction \( A_i \) are explicitly presented in the op-code portion of the instruction in two fields, called the dependency field (D field) and the effect field (E field), respectively. Fig. 5 illustrates this format. To the programmer this processor appears to have only the set \( A_j \), the set of data resources, so one bit in each of the two fields is sufficient to describe a dependency or effect due to each type of resource \( A_i \). These data resource tags are presented here in this exploded (and exaggerated) fashion to illustrate the ease with which independent instructions can be decoded and detected. A practical implementation would probably pack some of these conditions into the remainder of the op-code.

The predecode stack is the one nonstandard element of this processor. Its functions are to

1) find strongly independent instructions and send the addresses of weakly independent instructions to the effective address buffer,
2) assign particular data registers \( A_{ij} \) to the instructions in the stream, and
3) hold the pointers \( S_j \) to the operands fetched upon finding an independent address.

Strongly independent instructions are found according to the discussion presented earlier. An instruction is executably independent only if it is strongly independent of all instructions above it in the PDS. It is important to note that the implementation of this algorithm involves only the simultaneous bit by bit matching, in a fixed pattern, of the \( E, D \), and address fields of the instructions in the PDS. The interval of time needed by the PDS to find the independent instructions in it (called the PDS cycle) must be kept negligible with respect to the execution time of the executably independent instructions found for any sizable improvement in instruction execution efficiency to be realized.

Data registers are assigned to the instructions according to the following algorithm.

1) An instruction is assumed to use the same register \( A_i \) as the instructions most immediately above it in the PDS unless no instruction above it uses a register \( A_i \) for a particular \( i = 1, 2, \ldots, n \) in which case the instruction is arbitrarily assigned to \( A_i \).
2) If the instruction is an independent open-effects instruction using register type \( A_i \), it is assigned to register \( A_i^b \), for \( b = (j + 1) \mod m_i \), where \( A_i \) is the register used by the preceding instruction in the PDS.
As described earlier, instructions which are found to be weakly independent in the PDS must have their effective addresses compared to determine if they are strongly independent. To make this comparison, the effective addresses of the weakly independent instructions are calculated and put into the effective address buffer (preserving their order) along with information indicating whether the effective address is a sink or a source. Each effective address is compared bit by bit with all addresses above it in the buffer. The instructions with independent effective addresses are marked strongly independent in the PDS.

The sequence of operations required to find strongly independent instructions is shown in Fig. 6. Instructions which are immediately found to be strongly independent are simultaneously sent to the decoders and the PDS is pushed up and filled with instructions from memory, to complete a “PDS cycle.” When weakly independent instructions are found to be strongly independent, this fact is noted in the PDS and the effective address buffer is ready to accept additional weakly independent instructions. The instructions found independent through the effective address buffer will be sent to the decoders during the next PDS cycle.

If the interval of time between PDS cycles (the rate at which the PDS is cycled) is such that all the strongly independent instructions are completely executed before the next cycle, the computer is a nonoverlapped computer. Overlapped computers would use the PDS by shortening the interval between PDS cycles so that instructions are found independent through the effective address buffer will be sent to the decoders during the next PDS cycle.

Using Independent Instructions in the IBM 7094

The 7094 has a single-address instruction format as shown in Fig. 7. Its organization includes three data sink-source registers: the accumulator (AC), the multiplier-quotient register (MQ), and the sense indicator register (SI). It has seven index registers, which are referenced by setting the three tag bits (T) in the instruction to the proper binary code. Indirect addressing is indicated by setting the two flag bits (F) to binary ones. The computer has over 200 machine instructions, using an op-code field of 12 bits. The address field (Y) is 15 bits long.

An example will help to show the kinds of dependencies and effects a 7094 instruction can have. The CLEAR AND ADD instruction of the 7094 (mnemonic CLA) clears the contents of the accumulator (C(AC)) and fills it with the C(Y), leaving the C(Y) unchanged. The dependencies can be listed as follows for this instruction.

1) C(T): The effective address is calculated by subtracting the contents of the index register indicated by the T field from the address given in the Y field. This effective address cannot be determined until the index register contains the proper information.

2) C(Y-C(T)): Before this instruction can be executed, memory location Y-C(T) must contain the proper information.

3) F: The flag bits being set indicate that indirect addressing should be used. The address of the data word to be placed in the accumulator is then C(Y-C(T)). Notice that an instruction using indirect addressing cannot be executed until the following three conditions are met:

a) C(T) are correct
b) C(Y-C(T)) are correct
c) C(C(Y-C(T))) are correct.

For the CLA instruction, there is only one effect, and it is C(AC). Notice that this instruction is an open-effects instruction.

A method for coding the dependencies and effects for 7094 instructions is shown in Fig. 8. The modified op-code is divided into five fields: dependency, effect, IC, F, and op-code. The dependency and effect fields each contain one bit for each of the five possible dependencies and effects that a 7094 instruction might have. Namely, C(AC), C(MQ), C(SI), C(T), C(Y). These bits are set according to the particular dependencies and effects of the instruction. For example, the CLA instruction discussed earlier would have its T and Y bits in the dependency field set, and the AC bit in its effect field set. Note that the size and arrangement of these two fields is peculiar to the organization of

3 We neglect double indexing in an effort to simplify this discussion.
the computer, and would not necessarily be the same for all computers.

The IC field shown in Fig. 8 is a 1-bit field that is set when the instruction is a transfer or conditional type instruction whose effect is or may be to alter the contents of the instruction counter (CIC). This type of instruction may or may not also have some of the bits in its dependency and effect fields set.

The F field is also a 1-bit field, and it is set if the instruction is one which may use indirect addressing. This field is necessary in the 7094 because some of its instructions cannot use indirect addressing, but may have their flag bits set.

The fifth field in the modified op-code is an op-code field which defines specifically what the instruction is to do. Note that this op-code field need not be as large as it normally would be, because much information about the instruction is contained in the other four fields of the modified op-code. It need only be large enough to distinguish the instructions having identical dependency, effect, IC, and F fields from each other.

It should be pointed out that the arrangement of the modified op-code shown in Fig. 8 does not result in the most efficient coding possible. For the instruction set of the 7094, not all of the possible bit combinations in the dependency, effect, IC, and F fields are used. Thus, the total number of bits used by these fields could be shortened, and probably would be in an actual application. For our purposes, the arrangement of Fig. 8 is sufficient and much easier with which to work.

An algorithm for determining if an instruction is independent of another instruction is presented in Appendix A of [13] and will not be given here.

An instruction in the decode stack is executably independent if it satisfies the following conditions:

1) it is independent of every instruction above it in the stack (note that the instruction at the top of the stack is thus always executably independent); or
2) the instruction passes all of the independency checks with respect to all of the previous instructions with only the following exceptions.

a) If the instruction effects the AC but does not depend on it, the instruction need not pass the test comparing its AC bit in the dependency field with the AC bit in the effect field of the previous instructions.

b) If the instruction effects the MQ but does not depend on it, the instruction need not pass the test comparing its MQ bit in the dependency field with the MQ bit in the effect field of the previous instructions.

c) If the instruction effects the SI but does not depend on it, the instruction need not pass the test comparing its SI bit in the dependency field with the SI bit in the effect field of the previous instructions.

Condition 2 is, of course, dealing with open-effects instructions.

We intuitively expect that the larger we make the decode stack the more independent instructions we would find each time we fill and check the stack. However, we should also expect that the amount of circuitry needed to implement the checking algorithm would increase exponentially with the size of the stack, because each instruction must be checked with all instructions preceding it in the stack. Even more importantly, we should expect that the amount of time required to check the stack for independent instructions would increase as the size of the stack increases, due to the increasing number of logic levels required as the fan-in, fan-out restrictions on the gates are exceeded. The results of the simulations will help decide the answers to these questions.

Simulation Results for the 7094

A computer simulator of the 7094 algorithm was written, and 31 sample programs were tested on it to obtain data to help answer the questions of how many registers should be in the decode stack, and how many extra data sink-source registers would be needed to significantly increase the efficiency of the computer. The sample programs were library programs kept on disk in the permanent program library of the IBM 7094 computing system at Bell Telephone Laboratories, Naperville, Ill. The sample programs included routines for such things as matrix manipulation, calculation of trigonometric functions, calculation of Bessel functions, sorting, and others.

The simulator assumes that all weakly independent instructions are also strongly independent. This will almost always be the case because the probability that $E_i = f(A_2, A_3) = f(A_2, A_3) = E_j$ when $A_{2i} \neq A_{2j}$ or $A_{3i} \neq A_{3j}$ is intuitively very small. The algorithm also assumes that instructions using indirect addressing are strongly independent if they are weakly independent. Although the theory does not provide for handling indirect addressing, this case could not be neglected because the 7094 code used for simulation used indirect addressing. The simulation results showed that even if the assumption concerning indirect addressing were wrong 100 percent of the time, a negligible effect in the predicted efficiency improvement can be expected.

The number of registers in the decode stack, or stack size, was the variable parameter of the simulator. Each program was tested using stacks of size two through ten. Data was recorded indicating the number of independent instructions found in the stack during each PDS cycle and the average number of independent instructions per PDS cycle in the program.
Fig. 9 is a plot of the average number of independent instructions per PDS cycle versus stack size, the average taken over all 31 of the sample programs. A total of some 5600 instructions were tested by the simulator. The sample programs were also tested on the simulator, without allowing special handling for open-effects instructions, to determine the importance of the presence of this type of instruction. Fig. 10 is a plot of the data obtained in this case. The dotted lines in Figs. 9 and 10 indicate the high and low spreads of the data.

Discussion

The plot of Fig. 9 shows that we can, indeed, expect some increase in computer efficiency through application of the algorithm presented. For example, if the algorithm is implemented to test instructions in a stack of size ten, we would expect to find 1.86 independent instructions each time the stack is checked, on the average. Assuming that the algorithm adds negligible time to the decoding process, we would then be able to execute 1.86 instructions in the same time as it would normally take to execute only one instruction. We have thus increased the efficiency (average instruction execution rate) by 86 percent. This must be discounted somewhat due to increased decoding time (see [13] for a specific illustration of this).

REFERENCES