17

REGISTER ALLOCATION IN ASSEMBLY LANGUAGE

Robert A. Wagner

Technical Report
70-69

August 1970

Computer Science Department
Upson Hall
Cornell University
Ithaca, New York 14850

REGISTER ALLOCATION IN ASSEMBLY LANGUAGE

by

R. A. Wagner*

Abstract

This paper describes a scheme for using the facilities of a macro assembler to aid in allocating program variables to local-memory registers. The scheme allows the programmer to write the entire program before making any register-allocation decisions. The scheme requires that the programmer make explicit his assumptions about register ordering and usage, thus improving documentation.

Key words and phrases: register allocation, register assignment, symbolic register names, macro assembler, variable allocation, scalar variable equavalence, graph coloring.

CR categories: 4.21, 5.32

*Computer Science Department
Upson Hall
Cornell University
Ithaca, New York 14850

REGISTER ALLOCATION IN ASSEMBLY LANGUAGE

R. A. Wagner

Many modern computers (IBM 360; RCA Spectra 70; DEC PDP-10) have multiple "accumulators" or "general registers". Allocation of quantities of various kinds to these registers is somewhat harder than the corresponding problem for a machine with either one or several hundred accumulators. The accumulators play a central role in scientific and logical computations, because nearly every instruction executed obtains one or more operands from an accumulator, and/or stores its result there. In the presence of one accumulator, no "allocation" problem arises, since nearly every calculation must use the accumulator, preventing any one quantity from residing there over several instructions. Similarly, if several hundred accumulators are available, each accumulator can be allocated to hold a single variable more or less permanently. In the presence of a relatively limited number of such registers, variables must be moved between accumulators and main memory during the course of the program, while opportunities sometimes arise for leaving certain variables in accumulators over several instructions.

The present paper describes a technique for allocating variables to registers. The technique could be expanded into a scheme which automatically determines the optimum assignment of variables to registers. For practical reasons we have stopped short of this, preferring to supply a design tool for use by the assembly language programmer. (The practical considerations include the fact that optimum assignment would require a great deal of time, and also the detail that, on the IBM 360 where this scheme has been used, instructions which access a variable in a general register have different forms from corresponding instructions which access a variable residing in main memory.)

The design technique relies on the existence of a reasonably powerful macro assembler. The basis for the method is simple. One identifies a small segment of code which is logically isolated from the remainder of some large program. Within this segment of code occur several variables which, when referenced, should reside in general registers. Assign a distinct symbolic name to each such variable. These names will ultimately be EQU'd to register numbers. (As "register aliases" they should be chosen distinct from other types of names. We have used one and two character names as register aliases, while other names were more than two characters in length.)

The code is written as if each distinct name represented a distinct register. This scheme differs from other "symbolic register" schemes by explicitly assuming no order relation among the names; that is, name A might be assigned register 3 while name B is assigned register 10. Any requirement which imposes an ordering on the names must be explicitly stated, in the form of a macro-instruction.

Order requirements arise, on the IBM 360, when certain instructions are used. For example, use of the STM (Store Multiple) instruction implies that the contents of a certain set of registers, addressed in sequence, will be stored. This in turn implies that a certain collection of variables must reside in a sequentially addressed set of registers. By making this assumption explicit, we can require that, say I, K, and Q be allocated to sequentially addressed registers, while leaving the choice as to which registers until later.

Other kinds of orderings are sometimes important. One such is required because certain IBM 360 instructions operate on even/odd register pairs. Thus, there is need to refer to two registers symbolically, and be sure that they are the even and odd components of a register pair. Also, it is occasionally necessary to require that two particular symbolic names ultimately refer to the same physical register. This occurs primarily when parameters held in registers are passed from one subroutine to another.

A macro, RGCNS, (Register Constraints) has been developed to ensure that all explicitly stated order constraints are obeyed by the actual register assignments made. The variable field of this macro accepts a list of register aliases. In the absence of further instructions, each register alias occurring in a given RGCNS statement is assigned to a distinct register. Order constraints can be imposed, by prefixing a register alias in the list by one of two special characters:

period (.) -- meaning that the prefixed name is to be assigned a register in numeric sequence with the name immediately before it in the list; and slash (/) -- meaning that the prefixed name is to be the odd member of an even/odd register pair, the even member to be the immediately preceeding name. Thus, the characters . and / act as binary operators between two adjacent names in the list, imposing the relation "next," and "next and odd" on the second of the two names. The characters . and / are termed "order operators."

Examples:

RGCNS X,Y,.Z

Requires that X , Y and Z all be assigned distinct registers, with Z = Y + 1 .

RGCNS A,/X,Z

Requires that A , X and Z be distinct, with X = A + 1 and X odd.

The RGCNS statements act as abbreviations for sets of relations which must hold between register aliases. The register aliases are viewed as names of single-valued variables, each of which is assigned a register address as value. The "value" assigned to a register alias is assigned globally, remaining unchanged throughout the program. "Value" here refers to the location of a variable, and not to its contents.

Each RGCNS statement is an abbreviation for a set of relations on pairs of register aliases. This set of relations or constraints can be profitably divided into two distinct types of constraints. The types are called "distinct-register constraints" and "order constraints" respectively. A distinct register constraint is a relation of the form:

(1) $X \neq Y$

where X and Y are register aliases. An RGCNS statement is an abbreviation for one constraint of the form (1) for each pair of register aliases occurring in the RGCNS variable field. An order constraint is a relation of either form (2) or (3):

(2) $X = Y + 1$

(3) $X \bmod 2 = 1$

An RGCNS statement abbreviates one order constraint of type (2) for each occurrence of an order operator (. or /). Also, one type (3) constraint must be understood for each occurrence of the operator (/).

Using the System

This system for register allocation has been used in designing two fairly large (> 1000 instructions) programs, and four smaller ones. In the course of this experience, the system itself has been refined, and a number of heuristics which aid in its use have been developed. It should be noted that the system as presently implemented is a design aid, not a completely automatic tool.

The major problems which arise while using the register allocation scheme are two in number. First, there is the problem of describing the register constraints correctly. Second, after register constraints have been established, a valid assignment of register aliases to registers must be made. These problems must be solved "iteratively," since if it should develop that, for certain given constraints, no assignment which satisfies them is possible, the constraints must be revised. Occasionally, this revision necessitates changes in the instructions of the program.

Register constraints can be derived as the program is written. Whenever the programmer finds a new order-assumption necessary, he simply records the assumption. Unfortunately, this approach, used alone, makes revision of constraints and program difficult. We have found it useful to be able to derive register constraints directly from the assembly-language instructions making up the program. This analysis is presently done by hand, with knowledge of the programs flow chart.

Methods are available for deriving sets consisting solely of distinct-register constraints directly from the program. The formal derivation of these type (1) constraints follows closely a solution to the problem of packing scalar variables into least storage space. Furthermore, pure sets of type (1) constraints can be "solved," yielding an acceptable assignment of register aliases to register members by a process of graph node coloring.

RGCNS statements actually represent a mixed set of constraints of all three types. All the constraints must be satisfied simultaneously by any valid assignment of register addresses to register aliases. Unfortunately, no formal technique for the satisfaction of such mixed sets of constraints is known. This problem is akin to the packing problem for arrays, which has as yet no completely satisfactory solution. A discussion of several methods for the partial solution of this packing problem can be found in [3].

In practice, the order constraints imposed on register aliases by actual programs have proven easy to satisfy. Once the existence of a physically realizable solution to the distinct-register constraints has been demonstrated, the order-constraints can generally be satisfied by a variety of ad hoc methods. In particular, the program is often tolerant of slight changes in certain order constraints. For example, we have found it possible to change the order of the variables stored by STM instructions without changing much of the program.

Equivalencing Scalar Variables

The problem of deriving distinct-register constraints from a program is equivalent to deciding which sets of scalar variables can share the same storage location. Lavrov [2] has studied this problem, and our method for deriving constraints is based on his technique, as described in [3]. The algorithm assumes that no register-address fields are modified during the program's execution.

Intuitively, two variables can share the same storage location if they are never "busy" simultaneously. Occasionally, a single variable is actually "busy" over disjoint portions of the flowchart, holding different values. Such a variable can be "split," by renaming it differently in each such portion of the flowchart. Variables (register aliases) which are simultaneously busy must appear together in some RGCNS statement, to expose the fact of their incompatibility. A variable which is splittable need not be so split, however. If splitting is not performed, the program which results is still valid, but the assignment of register aliases to registers may use more registers than necessary.

An abstraction of a program's flowchart is necessary in deciding when variables are simultaneously busy. The abstraction is composed of nodes, representing program instructions, and arcs, representing flow-of-control. Nodes may be labelled:

- <u>use V</u> : the current value of V is used in a calculation performed here;
- <u>set V</u> : the value of V is changed here;
- <u>call S</u>: closed subroutine S is called (invoked) here.

Certain instructions may have to be represented as a sequence of more than one node. We require that:

1) Each node have a single label;

2) The order of the nodes in a sequence correspond to the order in which variables are <u>used</u> or <u>set</u> by the instruction;

3) There be exactly one <u>use</u> or <u>set</u> node for each distinct register-address field in the program. (If an instruction uses a single register-address field to specify that a register is both <u>used</u> and <u>set</u>, the <u>set</u> node should be eliminated.)

These restriction ensure a precise correspondence between the abstract flowchart nodes which reference variables and the address fields in the instruction in the program.

A <u>busy path for a variable V</u> consists of all paths in the flowchart from a <u>set V</u> node to a <u>use V</u> node, except those paths passing through a <u>set V</u> node.

Two busy paths for V <u>unite</u> when both paths include the same <u>use V</u> or <u>set V</u> node; two busy paths are said to <u>connect</u> if they unite, or if there is a third path connected to both of them.

The union of all busy paths for V is termed the <u>busy graph</u> of V.

The "connected" relation is an equivalence relation, and thus partitions a busy graph into mutually un-connected subgraphs B_i . A distinct name V_i can be substituted for V throughout each B_i without affecting the program's meaning. Thus variable V can be split into variables V_i .

Two busy paths are said to <u>touch</u> just when some arc or node of the flowchart occurs in both paths. If P and Q are touching busy paths for variable V and W respectively, and $V \neq W$, then V and W are said to be <u>simultaneously busy</u>.

Busy paths can be computed by propagating a "busy path flag" backward through the flowchart. A busy path flag for V is generated for each <u>use V</u> node of the flowchart. A busy path flag for V reaching a <u>set V</u> node is not propagated backwards through the <u>set V</u> node. The partition of V's busy graph can be derived by examining the set of busy path flags which reach each <u>set V</u> node. If two busy path flags for V reach the same <u>set V</u> node, the paths they mark unite.

<u>Closed Subroutines</u>

A closed subroutine physically occurs only once in the program. Any references to V within it cannot reference register V_1 during one call on the subroutine, and register V_2 during another call. Thus, in propagating busy path flags for V, a copy of the subroutine flowchart must be substituted for each <u>call</u> on that subroutine. However, each <u>use V</u> node contained in the subroutine gives rise to the same busy path flag in all copies; similarly, all copies of the same <u>set V</u> node must be treated as the same node.

Distinct-Register Constraints

The previous section has described a technique for deriving distinct-register constraints from a program. Each pair of simultaneously-busy register aliases gives rise to a distinct-register constraint on that pair of register aliases. The present section relates the solution of a set of distinct-register constraints to a classic problem in graph theory.

Satisfying a set of distinct-register constraints is closely related to the problem of coloring the nodes of a certain graph. Think of each register alias as a node of a graph. Connect every pair of aliases which appear in the same distinct-register constraint by an arc of the graph. A graph derived in this way will be termed an "assignment graph." The arcs represent "not equal" relations. To satisfy the set of distinct-register constraints, it is necessary to assign integers to each node of the graph in such a way that no two nodes which are connected by an arc are assigned the same integer. Each integer represents the address of some register. An assignment which satisfies the set of distinct-register constraints is an "acceptable coloring" for the assignment graph.

Temporarily, the physical constraint on the number of registers available will be ignored. Instead, an assignment will be sought which minimizes the number of registers needed to satisfy all distinct-register constraints. If the number of register used in the minimal assignment exceeds the number available, the program's structure must be changed, to remove or weaken some register constraints.

Graph Coloring Techniques

For the particular assignment graphs which occur in practice, a specialized graph-coloring technique has proven efficient. Observation shows that assignment graphs usually consist of several connected subgraphs, which are themselves not interconnected. Furthermore, it usually appears that the complement graph of each such connected subgraph contains fewer arcs than the "true" subgraph. (The complement-graph G' of a graph G consists of all nodes of G, and contains an arc joining nodes i and j if and only if nodes i and j are not connected by an arc in G.) These facts prompted us to derive methods for coloring the complement graph directly.

Suppose that the assignment graph G has been divided into connected subgraphs $G_1, \ldots, G_n$, where G_i and G_j have no nodes in common, and no arcs in G extend between G_i and G_j. Each G_i may then be colored independently. Let us consider the coloring of one such subgraph, G_i. Let its complement be G_i'.

We say that two nodes j and k of G_i' may be "coalesced" into a node ℓ, if j and k are joined by an arc in G_i'. The result of the coalescence operation is a new graph, $C_{jk}(G_i')$, consisting of all the nodes and arcs of G_i, except those involving nodes j and k. In $C_{jk}(G_i')$, nodes j and k are replaced by the single node ℓ. An arc from ℓ to another node m of $G_{jk}(G_i')$ appears, if and only if arcs (j,m) and (k,m) appeared in G_i'. The "coalescence" operation has, in effect, "equated" nodes j and k. Any coloring of the condensed graph $C_{jk}(G_i')$ induces a coloring of G_i' in which nodes j and k are assigned the same color. Since, before the condensation, j and k were joined in the complement graph G_i' by an arc, every valid coloring of $C_{jk}(G_i')$ induces a valid coloring of G_i' and hence of G_i. The coloring of G_i' need not be minimal, however, even if the coloring of $C_{jk}(G_i')$ is minimal.

In one case, we can guarantee that a minimal coloring of $C_{jk}(G_i')$ yields a minimal coloring for G_i'. Let $S(x)$ denote the set of all nodes connected directly to node x by arcs in G_i'. Suppose we arbitrarily place x in $S(x)$ as well. If $S(j) \supset S(k)$, then a minimal coloring of $C_{jk}(G_i')$ induces a minimal coloring of G_i'. For, let $D_j(G_i')$ be the graph derived from G_i' by deleting node j (and all arcs impinging on node j) from G_i'. Let $N(G)$ be the number of colors needed in coloring graph G minimally. It should be clear that $N(D_j(G_i')) = N(C_{jk}(G_i'))$, since $S(\ell) = S(j) \cap S(k) = S(k)$. Furthermore, $N(D_j(G_i')) \leq N(G_i')$, because a minimal coloring of G is also a coloring of $D_j(G)$ for any deleted node j of G. Also, $N(C_{jk}(G_i')) \geq N(G_i')$, because a minimal coloring of $C_{jk}(G_i')$ induces a coloring of G_i'. (In the induced coloring, nodes j and k are colored alike.)

But the relations

$$N(D_j(G_i')) \leq N(G_i') \leq N(C_{jk}(G_i'))$$

and $\quad N(D_j(G_i')) = N(C_{jk}(G_i'))$

imply $\quad N(G_i') = N(C_{jk}(G_i'))$.

Hence, if $S(j) \supset S(k)$ then a minimal coloring of $C_{jk}(G_i')$ induces a minimal coloring of G_i'. A complete description of this algorithm can be found in [4].

Current Status and Conclusions

The current implementation of the register allocation scheme makes no attempt to either derive register constraints, or to attempt to satisfy a set of such constraints automatically. Instead, the programmer is expected to describe both constraints and an acceptable assignment of register aliases to registers. (Another macro, 'RGASN n,(ℓ)' is used to announce that the list of aliases ℓ are to be assigned to register number n.) The RGCNS macros act passively, checking that the assignment provided satisfies all constraints.

This register allocation technique is basically a design tool. It permits the programmer to delay certain decisions about details until the entire program is constructed. The delay permits all constraints on the decision to become apparent, and thus creates a rational basis for the decision. The technique has proven quite successful in the design of parts of the PL/C compiler [5], and of other programs.

REFERENCES

1. Yershov, A. P. ALPHA - An automatic programming system of high efficiency. JACM, vol 13, number 1 (Jan 1966) pp. 17-24 (See Editor's Note, p. 24).

2. Lavrov, S. S. Economy of memory in closed operator schemes. Zh. Vychist. Mat. Fiziki 1: 687-701, 1961.

3. Bovet, D. P. Memory allocation in computer systems. Ph.D. thesis, Dept of Engineering, Univ. of California, Los Angeles, California.

4. Wagner, R. A. An algorithm for coloring the nodes of graphs. TR 70- , Computer Science Dept., Cornell Univ., Ithaca, NY, 1970.

5. Conway, R. W., Morgan, H. L., Wagner, R. A. and Wilcox, T. R. PL/C - A high-performance subset of PL/I. TR 70-55, Computer Science Dept., Cornell Univ., Ithaca, NY, 1970.