Instruction Set Design and Optimizations for Address Computation in DSP Architectures

Guido Araujo[1], Ashok Sudarsanam[2] and Sharad Malik[2]

Department of Electrical Engineering
Princeton University, Princeton, New Jersey 08544, USA
{guido,ashok,sharad}@ee.princeton.edu

Abstract

In this paper we investigate the problem of code generation for address computation for DSP processors. This work is divided into four parts. First, we propose a branch instruction design which can guarantee minimum overhead for programs that make use of implicit indirect addressing. Second, we give a formulation and propose a solution for the problem of allocating address registers (ARs) for array accesses within loop constructs. Third, we describe retargetable approaches for auto-increment (decrement) optimizations of pointer variables, and loop induction variables. Finally, we use a graph coloring technique to allocate physical ARs to the virtual ARs used in the previous phases. The results show that the combination of the above techniques considerably improves the final code quality for benchmark DSP programs.

1. Introduction

Computing the address of an operand is a task frequently performed in accessing data-streams such as those found in DSP algorithms. Traditionally, data is stored in arrays and accesses are made through array indexing. Alternatively, pointers can be used to directly address data. Computing the address of an array element involves adding an *offset* to the *base* address of the array. This computation can be determined implicitly by the compiler in the case of array indexing, or explicitly by the programmer when the access occur by means of pointer variables.

Due to hard performance constraints in this application domain, it is not surprising that DSP designers have added hardware features which enable fast address computation. These features are present in almost every commercial DSP processor. They are, in general, based on an Address Generation Unit (AGU) which contains a number of address registers and an arithmetic unit that can perform basic arithmetic operations such as increment/decrement. The major goal of a compiler when optimizing address computation should be to guarantee that this architectural feature is effectively used by the source program.

[1] Work partially supported by Brazilian Council for Research and Development (CNPq) under Proc.204033/87.0 and Institute of Computing, UNICAMP, Brazil.

[2] Supported by NSF Award MIP 9457396.

2. Addressing Mode Design

In order to make effective use of indirect addressing, some DSP processors (e.g. TMS320C25 processor) use implicit indirect addressing mode instructions. Consider for example the following Instruction Set Architecture (ISA) model.

```
(a) I <AR,ARP>
(b) B <ARP>
(c) S <ARP>
```

In instructions of type (a) I represents the instruction opcode, AR is the address register used by the instruction to access one of its operands, and ARP the value of the next selected AR after the instruction is finished. In instructions of type (b) B represents a conditional branch and ARP again is the next AR. In order to give full flexibility to the programmer, an instruction to explicitly set the next selected AR, like instruction (c), is usually provided. An additional $S < ARP >$ is used when the programmer needs to explicitly set ARP pointing to the AR required by a particular instruction. In the next section we propose an algorithm which guarantees that only a single $S < ARP >$ is required for any program which uses implicit indirect addressing instructions, provided that the target ISA satisfies a specific implementation for the branch instruction.

2.1. Minimizing the ARP Overhead

Let $B < ARP >$ be a conditional branch instruction in the target processor ISA. Assume that *cond* and *label* are respectively the condition tested by the branch instruction and

the address of the instruction executed next if *cond* holds true. Assume also that the branch instruction execution follows the machine algorithm below, where PC is the architecture program counter.

```
Branch (cond)
begin
   if (cond) then
      PC ← label;
      ARP ← AR;
   else
      PC ← PC + 1;
   endif;
end
```

Theorem 1 (Branch Theorem) *If the Branch algorithm above is used to implement the conditional branch instruction in the target architecture, then a single instruction $S < ARP >$ will be required for any program which uses implicit indirect addressing mode instructions.*

Proof. Let $I_i < AR_i, ARP_i >$, $I_j < AR_j, ARP_j >$ and $I_k < AR_k, ARP_k >$ be three indirect addressing mode instructions and $B < AR_l >$ a conditional branch instruction in the target architecture. Now we must consider two cases:

(a) Assume that I_i and I_j are instructions in the same *basic block*[2] such that I_j follows I_i in program order, and there exist no other indirect addressing instructions in between them (Fig. 1(a)). In this case by making $ARP_i = AR_j$ one can satisfy the AR requirement of instruction I_j after the execution of instruction I_i.

(b) Assume now that I_i, I_j and I_k are instructions in different basic blocks, namely B_i, B_j and B_k, such that B_j and B_k are the successors of B_i as in Fig. 1(b). Notice that all instructions in basic block B_j (B_k), but the first one, will have their AR requirements satisfied by instructions which precede them within the basic block. In this case we have only to consider the first indirect addressing instruction in basic block B_j (B_k), which we assume to be I_j (I_k). Without loss of generality consider also that I_i is the last indirect addressing instruction in basic block B_i. Now suppose that the branch instruction at the end of basic block B_i was implemented using the algorithm *Branch*. In this case, if the branch is taken, by making $ARP_l = AR_k$, then the instruction I_k will have its address register requirement satisfied. On the other hand, if the branch falls through, then according to the algorithm *Branch* the ARP will not be updated. Hence, by making $ARP_i = AR_j$ one can also satisfy the address register requirement of instruction I_j. Here we restrict ourselves to two-way branch instructions and structured programs only.

[2] A *basic block* is a sequence of consecutive instructions in which the flow of control enters at the beginning of the sequence and leaves at the end without halt or deviation except at the end.

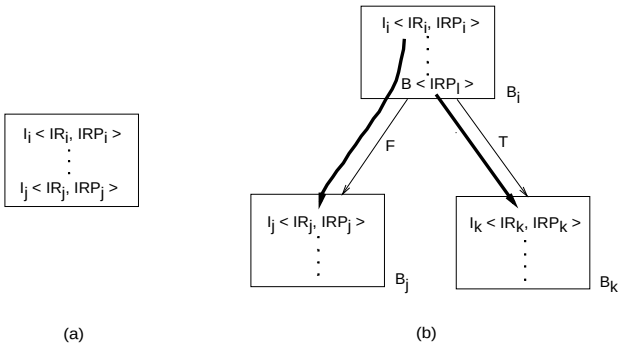


Figure 1. *Branch* **algorithm and ARP setup**

All other situations (**if-then-else**, **for**, **while** statements and their composition) not included in cases (a) and (b) above can be reduced to those two cases by applying data-flow analysis in the program Control-Flow Graph (CFG). Finally notice that only one $S < ARP >$ instruction will be required in order to satisfy the AR requirement of the first indirect addressing instruction in the first basic block of the program, or in case this does not exist, to set the AR for the first indirect addressing instruction that can be reached from the beginning of the program, by following only fall-through paths at each conditional branch. □

From the proof of Theorem 1, one can derive a simple linear time algorithm to determine the ARP field of each indirect access and branch instruction.

Example 1 Consider, for example, the program fragment shown in Fig. 2(a). Notice that pointer variables a, b and c are used to copy variables x, y and z into memory. The CFG corresponding to this program is shown if Fig. 2(b). In this program segment there exist three basic blocks B_1, B_2 and B_3. Assume that address registers AR_1, AR_2 and AR_3 are respectively allocated to variables a, b and c. Let instruction $*a$ be the last indirect access in basic block B_1. Also let instruction $*b$ ($*c$) be the first indirect access instruction in basic block B_2 (B_3). The ARP has to be properly set before instructions $*b$ and $*c$ are executed in order to guarantee that the correct address register is used when required. This can be done by making instruction $*a$ set its ARP to point to AR_3 and the branch instruction ARP to point to AR_2. Notice that if the branch falls-through, then the ARP is not updated by the branch instruction and at the start of basic block B_3 it is correctly pointing to AR_3. Similarly if the branch is taken, the ARP is updated to point to AR_2 just before instruction $*b$ is executed.

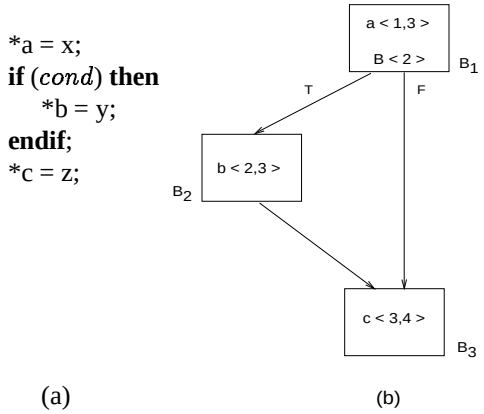


Figure 2. (a) Program example; (b) and its corresponding CFG

Although the idea behind this approach is extremely simple, the TMS320C25 DSP, which uses implicit indirect addressing mode instructions, fails to make effective use of it. In the TMS320C25 the value of the next ARP is always loaded, when specified in a branch instruction, regardless of the result of the branch condition [1]. The improvements that could have been achieved from the proposed instruction design would certainly pay-off. Consider for example the program *adpcm*, a large speech compression algorithm from the DSPstone benchmark suite [2]. When the TI compiler generates assembly code for this program it produces 100 instructions ouf of 2170, whose only purpose is to update the contents of the ARP. Some of these instructions (47) can be eliminated by using the approach described in part (a) of Theorem 1. Unfortunately a large number of these (53) are due exclusively to the design choice of the branch instruction. It may be possible that the implementation of the proposed *Branch* instruction impacts the processor cycle time. This does not seem to be the case though, based on the information available in [1].

3. Array Index Allocation

In this section we formulate the problem and propose a solution for the task of allocating virtual ARs to array accesses which are part of the body of loop statements. The goal here is to take advantage of the auto-increment (decrement) properties of the AGU such as to perform efficient access to array elements.

Assume a code generation approach in which indexed array elements within loops are not decomposed into its atomic operations. Assume a single loop construct where induction variable i is linearly updated by the integer quantity s (*loop step*), $s \neq 0$, and for which the loop boundaries are statically defined integer quantities. Consider, for example, the loop of Fig. 3 where $s = 1$ and the loop *trip-count* is N. The majority of DSP programs use well-defined loop constructs like the one just described.

Definition 1 *Let $access(m) = n$ be the function which maps an instance of array element m into n, where $n = 1, 2, \ldots$ is the order of the array element in the code sequence resulting after the instructions are scheduled. We say that n is an access of array element m.*

Definition 2 *Let n_1 and n_2 be array accesses. Access n_1 (n_2) is said smaller (larger) than n_2 (n_1), denoted by $n_1 < n_2$ ($n_1 > n_2$), if and only if n_1 (n_2) precedes n_2 (n_1) in schedule order.*

Example 2 In the loop of Fig. 3(b) each time an array element is used we associate a number in parenthesis corresponding to the order the element is accessed in the program. This number is the access of that array element. For example $access(a[i+1]) = 2$.

Consider that the array indexes within the loop are affine functions of the type $k * i + p$, where k and p are integer quantities. Assume in the following analysis that $k = 1$. This assumption is not a serious restriction, since the indexes of the majority of array accesses in DSP programs are affine functions of this type. When mutidimensional array elements are present, array accesses can be usually reduced to this simple case with the help of induction variable elimination algorithms [3]. Observe that the goal here is to allocate the minimum number of ARs such as to address all the array accesses within the loop. In this case it is desirable to maximize the number of accesses that can share a single AR. In order to identify the possibility of sharing between two access we define the concept of *indexing distance*.

Definition 3 *Let n_1 and n_2 be array accesses. Let $index(n)$ be a function which takes access n and returns the index associated with that access. The indexing distance between accesses n_1 and n_2 is the positive quantity:*

$$d(n_1, n_2) = \begin{cases} |index(n_2) - index(n_1)| & \text{if } n_1 < n_2 \\ |index(n_2) - index(n_1) + s| & \text{if } n_1 > n_2. \end{cases}$$

Example 3 Consider for example the array accesses of Fig. 3(a). In this case, as in the majority of loops in DSP programs, step $s = 1$. The indexing distance $d(1,4) = |i-(i-2)| = 2$, implies that no auto-increment (decrement) operation can be used to update the address register allocated to access 1, such that it ends up pointing to the data requested by access 4. On the other hand since $d(4,1) = 1$ an auto-decrement operation can be used to redirect the address register associated to access 4 such that it points to access 1. Notice that this will occur when access 1 is reached from access 4 across consecutive loop iterations.

Definition 4 *An indexing graph (IG) is a directed graph where each node corresponds to an array access, and there exist an edge (n_1, n_2) if and only if $d(n_1, n_2) \leq |s|$.*

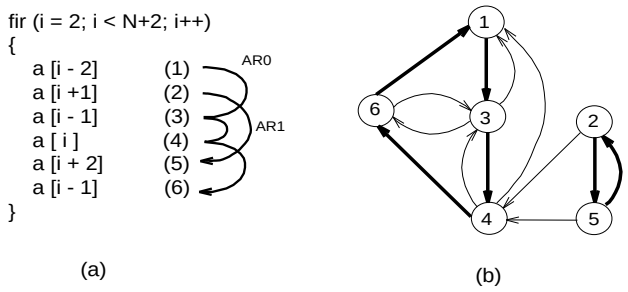


Figure 3. (a) Typical loop construct in DSP programs; (b) Corresponding IG

There exists an edge (n_1, n_2) in the IG when AGU operations can be used to update the index register associated to access n_1, such that it points to the data associated to the access n_2.

Example 4 The IG of Fig. 3(b) was built from the array accesses patterns in the body of the loop of Fig. 3(a). Observe that some edges in the IG, e.g. $(3, 4)$, captures the possibility for auto-increment (decrement) between array accesses in scheduling order. Other edges, e.g. $(6, 1)$, identify auto-increment (decrement) operations which can be performed across loop iterations.

3.1. The Array Index Allocation Problem

It is a consensus among DSP programmers that array accesses ought to be transformed into pointer operations. The main reason for that is the inability of compilers to perform efficient allocation of ARs in the presence of array accesses. Although researchers have been addressing this issue [4], transformation into pointers is still considered the technique of the choice.

Array Index Allocation is the problem of allocating virtual address registers to array accesses within loops such that the total number of virtual address registers is minimized. The importance of this problem comes from the fact that the majority of array accesses in DSP algorithms occur within finite loops, which have linearly updated induction variables and for which the boundaries can be statically defined at compiling time. We assume here an architectural model, as that in Sec. 1, where an Address Generation Unit (AGU) is available with auto-increment (decrement) operations. The following approach is not restricted for the case of increment (decrement) though. Notice that the definition of indexing distance also accommodates non-unitary AGU operations.

Definition 5 *A path $n_i \rightarrow n_j$ in the IG is a sequence of distinct arrays accesses $(n_i, n_{i+1}, \ldots, n_j)$, such that $n_k < n_{k+1}$, $i \leq k \leq j-1$, where $i, j = 1, 2, \ldots$.*

Definition 6 *A cycle in the IG is a sequence of nodes $(n_i, n_{i+1}, \ldots, n_j, n_i)$ such that subsequence $(n_i, n_{i+1}, \ldots, n_j)$ forms a path in the IG, where $i, j = 1, 2, \ldots$.*

A path in the IG corresponds to the allocation of the same AR to a sequence of array accesses. Similarly, a cycle indicates that the same AR can be used not only for accesses in program order, but also by one more access in the next loop iteration. The problem of minimizing the number of virtual address registers given an IG can be formulated as a graph optimization problem as follows:

(IG Covering) *Given an IG determine the disjoint path/cycle cover of the graph which minimizes the total number of paths and cycles. Assume for the purpose of this problem that a node is a degenerated cycle of zero length.*

Notice that not all cycles are allowed in the cover above. According to Definition 6 a cycle can only contain a single backward edge, i.e. an edge from n_j to n_i where $n_j > n_i$. The reason for this is that cycles should reflect auto-increment (decrement) operations across a single iteration and not across multiple iterations. Each path and cycle in the resulting cover corresponds to an address register. The formulation of IG Covering does not consider the cost of the instruction to reset the AR at the *tail* of a path, such that it can be used by the access at *head* of the path.

The problem above is similar to the minimum disjoint cycle cover of a graph (MDCC). The number of disjoint cycles which cover the nodes of a graph is know as the *Hamiltonian cycle index*. Determining the minimum Hamiltonian cycle index of a graph has been shown to be NP-complete [5]. Cycles in a cover for the MDCC problem, unlike cycles for the IG Covering, can contain more than one backward edge. Although we have no proof at this point, we believe that IG Covering is NP-hard.

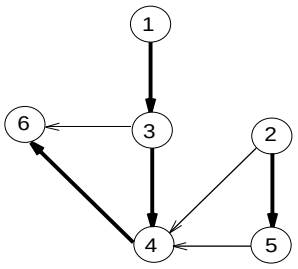


Figure 4. Solving the MDPC for an acyclic IG

Example 5 Consider the IG showed in Fig. 3(b). Covering the IG in that case produces a two cycle cover which is represented in bold on Fig. 3(b). Each cycle correspond to a virtual address register ($AR0$ and $AR1$).

Given that IG Covering is possibly NP-hard we have been studying heuristics to tackle this problem. The most obvious one is to formulate the problem such that auto-increment (decrement) operations across loop iterations are

not permited as in Fig. 4. As a result of that the IG becomes acyclic, and the original problem is reduced to the one of determining the minimum node-disjoint path covering of the graph (MDPC), for which there exist a $O(n)$ solution [6], where n is the number of nodes in the IG.

Example 6 Solving the MDPC for the acyclic IG of Fig. 4 results in paths $(1, 3, 4, 6)$ and $(2, 5)$. Cycles can still be identified in this case by computing the indexing distance between the tail and the head of a path. For example, since $d(6, 1) = 0 \leq 1$ ($d(5, 2) = 0 \leq 1$) then virtual register $AR0$ ($AR1$) can be used, at the tail of its corresponding path, to point to the data accessed at the head of the path. In this case the heuristic approach produces the same result as the exact solution in Example 5.

4. Auto-Increment Optimization

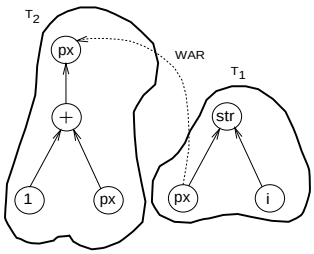
Using pointer variables to access data-stream elements is a common operation in DSP programs. Consider for example the program fragment in Fig. 5 extracted from the DSPstone benchmark kernel *fir.c*. In this program pointer variables px and ph are used to initialize the contents of an array. Let us consider the expression DAG generated

```
(1)   for ( i = 0; i <= LENGTH; i++)
(2)   {
(3)       *px++ = i;
(4)       *ph++ = i;
(5)   }
```

Figure 5. Part of the fir.c DSPstone benchmark kernel

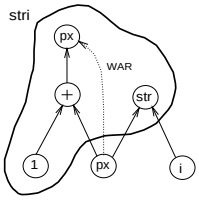
from statement (3). By using the tree-based code generation approach in [7], one can dismantle this DAG into expression trees using two different approaches (Fig. 6(a) and (b)). In Fig. 6(a)(b) nodes labeled px are used to represent operations $read\ px$ and $write\ px$, node str takes the value contained at memory position i and stores it at the memory position pointed by px, and the a Write After Read (WAR) constraint edge is used to enforce the original post-increment behavior in the source program.

In the first approach (Fig. 6(a)) tree T_1 contains the operations used to perform the increment of px and T_2 those required for copying variable i. The total cost of pseudo-assembly code corresponding to trees T_1 and T_2 (Fig. 6(b)) was 6 instructions. In another approach operation str and increment p_x are matched by a new instruction $stri$ (store and increment) as it is shown in Fig. 6(b). The total cost of the resulting pseudo-assembly code Fig. 6(b) was 4 instructions. Observe that if all auto-increment (decrement) addressing instruction can be compacted this way, then no overhead will exist due to address computation. We have introduced patterns which enable matching of auto-increment (decrement) operations in the expression DAG. These patterns are specified using primitive *Intermediate Representation* (IR) operations resulting in improved retargetability. The results show that this optimization considerably improves the code quality of benchmark DSP programs.



T_1:
$AR_1 = px$
$R = i$
$*IR_1 = R$
T_2:
$AR_1 = px$
$AR_1 = AR_1 + 1$
$px = AR_1$

(a)



T_1:
$AR_1 = px$
$R = i$
$*IR_1 + + = R$
$px = AR_1$

(b)

Figure 6. (a) Dismantling the expression DAG into trees; (b) Pattern matching auto-increment operation

5. Loop Induction Variable Optimization

Address registers may be used as general-purpose registers, although in a very limited context. In this optimization, an address register may be allocated to hold the loop induction variable, thus obviating the need to access this variable in memory. The TMS320C25 ISA features a *BANZ* instruction (*Branch on Auxiliary Register Not Zero*) which was specifically designed to improve the efficiency of loops. When this instruction is used it becomes possible to test and modify the loop induction variable using just one instruction. We have implemented this optimization by restructuring loops at the control-flow-graph level, a machine-independent representation of the program. A virtual address register was allocated to each induction variable in a loop whenever this was possible. The results of this optimization in Sec. 7 show a considerable improvement in code quality due to this optimization.

6. Global Address Register Allocation

In this section we use a register coloring technique to allocate physical address registers to the virtual address registers used in Sections 3, 4 and 5. Our goal here is to

perform physical allocation for address registers only after all optimizations for address computation have been performed. Local address register allocation has been considered in [8]. Unfortunately not much consideration has been given to global address register allocation, particularly for DSP architectures. Our approach builds an interference graph where each node represents a virtual AR and an edge (v_1, v_2) between virtual ARs v_1 and v_2 indicates that virtual ARs v_1 and v_2 have intersecting lifetimes, and should therefore be allocated to different physical address registers. We currently use Chaitin's algorithm [9] to perform graph coloring of the interference graph, where the number of colors used corresponds to the number of available physical address registers. Addressing operations in DSP programs are largely used to perform sequential accesses. Hence it is reasonable to believe that the majority of data accesses through address registers will occur within loop constructs. In this case a better approach is to use the Callahan-Koblenz algorithm [10]. In their work, a register allocation technique is proposed based on hierarchical coloring, which gives priority to allocate inner loop variables. This approach is not currently implemented.

7. Experimental Results

We have used the approach described in the previous sections to compile a set of kernel programs from the DSPstone benchmark suite. The results are listed in Table 1. The metric used to measure the code quality is code size, since it reflects an important goal of compiling for DSPs. Column *Unopt.* in Table 1 shows the number of assembly instructions after compiling the program using no address register optimization techniques. In column *Inc/Dec* one can find the final assembly code size when the approach described in Sec. 4 is used. The average improvement was 13% and in all but three cases the improvement was larger or equal to 10%. In all programs the address computation associated with the auto-increment (decrement) operation was compacted into datapath instructions. In column *ARs* we have listed the number of address registers used by the address register allocator. Notice that the target architecture (TMS320C25) has 8 ARs and that at most 7 ARs were used in all kernel programs. It is possible though that for larger application programs 8 ARs are not enough and spilling operations have to be performed. Column *Loop Var.* in Table 1 shows the size of the final code after loop induction variable allocation is performed. The average improvement with respect to the previously optimized code was 12%. The solution proposed in Sec. 3 is currently being implemented and experimental results are not yet available.

Program	Unop.	Inc/Dec		Loop Var.		ARs
fir	104	89	14%	82	8%	4
convolution	73	61	16%	51	16%	3
matrix	160	153	4%	128	16%	5
matrix_1x3	56	51	9%	41	20%	3
dot_product	56	50	11%	45	10%	4
n_real_updates	113	89	21%	79	11%	4
fir2dim	294	246	16%	184	25%	7
complex_update	86	83	4%	83	0%	4
n_complex_updates	191	142	26%	137	4%	5
iir_N_biquad	160	144	10%	131	9%	6

Table 1. Experiments with Address Register Optimizations

8. Conclusions

In this paper we have addressed the problem of improving the code quality for address computation in DSP programs. The main contributions of this paper are: (a) an efficient branch instruction design for implicit indirect addressing mode instructions; (b) a formulation and solution for the array index allocation problem. We are currently in the process of investigating better solutions for this problem.

References

[1] Texas Instruments, Inc. *Digital Signal Processing Applications with the TMS320 Family*, 1990.

[2] V. Zivojnovic, J.M. Velarde, and C. Sclåager. DSPstone, a DSP benchmarking methodology. Technical report, Aachen University of Thecnology, August 1994.

[3] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers, Principles, Techniques and Tools*. Addison Wesley, Boston, 1988.

[4] C. Liem, P. Paulin, and A. Jerraya. Address calculation for retargetable compilation and exploration of instruction-set architectures. In *Proc. 33rd Design Automation Conference*, pages 597–600, June 1996.

[5] M.R. Garey and D.S. Johnson. *Computers and Intractability*. W. H. Freeman and Company, New York, 1979.

[6] F.T. Boesch and J.F. Gimpel. Covering the points of a digraph with point-disjoint paths and its application to code optimization. *Journal of the ACM*, 24(2):192–198, April 1977.

[7] G. Araujo, S. Malik, and M. Lee. Using register-transfer paths in code generation for heterogeneous memory-register architectures. In *Proc. 33rd Design Automation Conference*, pages 591–596, June 1996.

[8] K. Kennedy. *Design and Optimization of Compilers*. Prentice-Hall, 1972. R. Rustin, editor.

[9] G. Chaitin. Register allocation and spilling via graph coloring. In *Proc. of the ACM SIGPLAN'82 Symposium on Compiler Construction*, pages 98–105, June 1982.

[10] D. Callahan and B Koblenz. Register allocation via hierarchical graph coloring. In *Proc. of the ACM SIGPLAN'91 Conference on Programming Language Design and Implementation*, pages 192–202, June 1991.