Efficient code generation for ASIPs with different word sizes

Andrea G. M. Cilio Henk Corporaal

Coputer Architecture and Digital Techinques group, Faculty of Electrical Engineering,
Delft University of Technology,
P.O. Box 5046, 2628 CD Delft, The Netherlands
A.Cilio@et.tudelft.nl H.Corporaal@et.tudelft.nl

Keywords: Embedded System design, Retargetable code generation, SUIF

Abstract

We propose a complete methodology for extending our automatic ASIP (Architecture Specific Instruction set Processor) synthesis framework to a much wider target architecture space. In this new architecture space the width of the integer data word and of any hardware resource data path is user-definable and application specific. This methodology, developed on the basis of a retargetable C compiler (the SUIF compiler), requires the definition of a new intermediate format and a complete set of non-ambiguous rules determining the bit precision of each integer operation depending on the size of its operands. We also assess the implementation requirements and the complexity increases of the scheduler and the hardware template. Examples illustrate the methodology.

1 Introduction

As the integration scale becomes larger, the complexity of embedded systems and the algorithms they implement increases. This complexity poses new challenges in the design and places greater stress on the reduction of design time and cost; assembly programming and hand optimization become impractical. High level languages like C become a typical choice for designing software for embedded systems [3].

Much research on compilation of C applications for embedded systems as part of a software-hardware co-design framework focuses on architectures with strong resource limitations, that drastically reduce the complexity of code generation. A typical example of these target architectures is the Texas Instruments' TMS320C25, considered in [2][9]. This DSP is accumulator based and has no general purpose registers. Such restrictions greatly simplify the code generation and instruction scheduling task, because the life times of memory and accumulators are easily estimated. In practice, adjacency of operations determines where spills and reloads are required. Our solution is opposite: we start with an architecture template well suited for general purpose applications, the MOVE template [5], and we enlarge it to meet the requirements of embedded system applications. The major restrictions to the solution space of the current MOVE Framework are due to the C language, used to specify the application. Like other common high level languages, C and its type system [1] favor portability and are suitable for general purpose processors (GPPs), for which a number of characteristics apply: only a single data word size (referred to as "natural") is fully supported by means of a complete set of machine opcodes; a minimum addressable unit is present, which is a sub-multiple of the natural data size. Other word sizes, usually smaller than the natural one, may be partially supported by a limited set of opcodes.

Our approach is to augment the C language semantics in order to independently express the number of bits of each variable. By specifying the word size and, implicitly, the bit precision of each operation, we can support code generation for architectures with datapath width(s) defined by the designer. In this paper we focus on the code generation issue, but it is evident that the architecture template must be correspondingly extended to support user-definable datapath widths of the hardware resources. The processor generator must generate functional units, register files, busses, etc., with parameterized width.

The main motivation for our approach is the design efficiency. For several applications, substantial area savings can be obtained if the optimal width can be specified for each hardware resource. In many cases, the area of combinatorial circuits increases at least linearly with the width; in some cases, for example in multipliers, the dependence is even quadratic, and each area saving due to width reduction has a critical effect on the total area occupation. Moreover, in several combinatorial circuits the maximum delay and the datapath width have a proportional relation. A third argument for using optimal datapath width is the power reduction: a narrower datapath means less gates and therefore a potential decrease of power consumption.

This paper describes an efficient code generation methodology for architectures supporting arbitrary word sizes. The remainder is structured as follows. Section 2 contains a definition of the problems and the requirements of the proposed methodology. Section 3 presents the code generation trajectory and analyzes the actions performed in each compilation step. Section 4 contains a few significant hand-coded examples to illustrate the proposed methodology. Finally section 5 concludes this paper and lists its contributions.

2 Specification methodology

In this section we consider the issue of how to specify within the source code the size of each data word and the bit precision of each operation and how to pass this information through the compilation phases. Efficiency is a fundamental requirement: with the proposed solution the code using user-defined word sizes is as efficiently handled as the code that uses the standard base types.

2.1 Language and compiler restrictions

The major problems that we face when expressing different word sizes come from the C type system. The ANSI standard defines four integer base types, and suggests that the `int` type be the natural word size of the target architecture. The standard also poses lower bounds and conditions to the sizes of base types [8] and rules of automatic type conversion.

Several C compiler implementations for general purpose architectures pose further restrictions to the intermediate format of the code and the type system in order to fulfill efficiency and quality requirements. For example, the current MOVE Framework is based on an efficient C compiler for GPPs: the GNU C compiler [12]. Its intermediate code representation, called Register Transfer Language, allows for a restricted number of word sizes to be used for operands. Moreover, these word sizes must be a multiple of a minimum size, the byte, that is also the minimum addressable unit of memory. All such restrictions, reasonable when generating code for typical GPPs, severely limit the efficiency of software compiled for ASIPs.

The newly proposed solution is based on the SUIF [6] compiler and takes advantage of its flexible intermediate representation, that allows arbitrary sizes for integer types. The SUIF compiler is also extremely convenient because its high level intermediate representation hides most machine details, a vital quality for retargetable compilation. SUIF (Stanford University Intermediate Format) supports annotations and allows for accessing and manipulating symbol table entries, making it easy to change the type of whatever object.

2.2 Type size specification

The proposed methodology for code generation is to express new integer types that represent user-defined word sizes. We have devised four possible approaches:

1. Front-end extension.
2. Annotation of the base C types.
3. Use of C++ classes and operator overloading.
4. Annotation of variable declarations.

The first two seem the most obvious. However, the first one lacks flexibility because it requires extensive modifications to any new version of the front-end that we intend to adapt. The second approach requires the definition and the subsequent annotation of a new type using `typedef`, e.g.:

```
typedef int int24;
#pragma typesize int24 24
```

This solution does not work because base C types are not distinguished from `typedef`'d alias types; therefore there is no way to bind a type annotation to the desired alias of the type. Another approach, that has been tried on the GNU compiler, is to define a set of classes implementing the new integer types and to use operator overloading. Each class represents an integer type with a defined size, e.g.:

```
class int24 {
private:  int data;
public:
      int24(){};
      int24(int n){data=n;};
      operator int20(){return *this;};
      operator int(){return data;};
      operator+(const &int24 w);
      ...  }
```

The preliminary results [4] showed that this solution is impractical, due to the amount of overhead generated by any class object, when compared to base types [10]. The final, and proposed approach is to annotate `int` variable declarations with the desired word size, when this is different from the standard size. The annotation is passed by means of a `#pragma` directive. See example 1.

Example 1 Annotation of a variable declaration
Let us consider the following variable declaration

```
unsigned int a;
#pragma bitsize a 24
```

`a` *represents a 24 bit unsigned integer. A* `bitsize` *annotation will be attached to the symbol table that contains* `a`.

□□

The SUIF front-end uniquely binds an annotation referring to a variable name with the correct symbol table entry. A pass immediately following the front-end must adapt the intermediate code by effectively introducing the new types and the necessary conversion instructions. After this step, the intermediate code can be further processed by all machine-independent analysis and optimization steps within SUIF.

2.3 Aspects of type size specification through annotations

Annotating the variable declaration is not free from drawbacks: specifying the word size associated to declared variables is not enough to emulate the semantics of a base type in the source code. Function prototypes and type casting are not associated with any variable declaration, therefore they require a specific approach, as shown in this section.

Implicit conversion
Implicit conversions in integer mixed type expressions follow the simple rule that the source operands are promoted to the width of the largest one and the result is converted to the destination size, if different. We maintain the same convention. In addition, the C standard requires that every integer base type shorter than `int` be converted to the `int` size before executing the operation. This is reasonable for GPPs, for which `int` is the type of the natural word. In ASIPs, however, functional units may support arbitrary word sizes and this additional rule is not necessary. Therefore, we allow SUIF instructions of the intermediate representation to have operands with sizes other than 32 bit.

Explicit conversion (type casting)
A type casting is unrelated to any declared symbol. Annotating variable declarations is thus not sufficient to express the number of bits of a type casting. Our solution is to adopt a "function-like" syntax, as the one allowed in C++. For this purpose, a set of functions with reserved names is declared in a special header file (see example 2. These functions are used as placeholders for type casting information: their (reserved) names deliver the required size information. A pass on the intermediate code substitutes these SUIF call instructions with appropriate conversion instructions, equivalent to those introduced for the built-in type casts. Note that type casting from a user-defined size to 32 bits must be specified with the new notation. In fact, for the front-end a type casting like `(int)a`, where `a` has an annotated type is redundant, and removes it, as it considers `a` already to be a variable of type `int`.

Example 2 Type casting to a 24 bit, signed integer
Let us considering the following piece of C code

```
x = sub-expr1 + int24(sub-expr2);
```

the intermediate result of sub-expression 2 is sign extended (or truncated) to 24 bits. The function `int24` *is declared in a specific header file for arbitrary word size support. Its name is reserved.*

□□

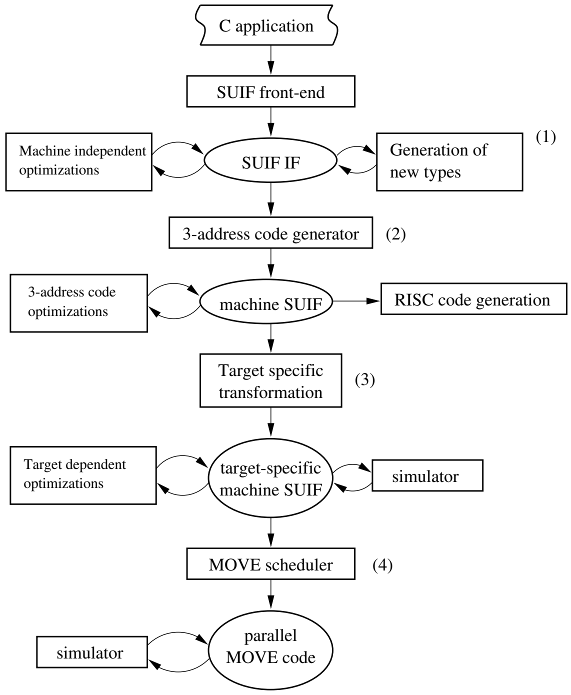

Figure 1: Code generation trajectory.

Type checking of function prototypes
Prototype checking for the new integer types is implemented by means of a dedicated annotation. This annotation contains the name of the function it refers to and a list of numbers, associated by order with the function arguments. Each value indicates the size of one integer argument. The values specified for non-integer arguments are ignored. The value zero means that the built-in size of `int` is maintained. This latter convention allows to use portable types, for which the actual size depends on the target architecture (see example 4).

3 Code generation trajectory

Figure 1 shows the passes necessary to generate code for a MOVE processor starting from a C application. The four numbered passes are either introduced or modified in order to support user-defined word sizes. Of course, the simulators must also be adapted to deliver the correct results accordingly to the word sizes used in the code. In the following sections we describe the aspects of these four passes specifically related to our methodology.

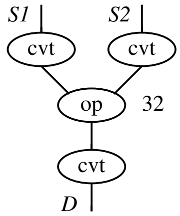

Figure 2: SUIF instruction tree.

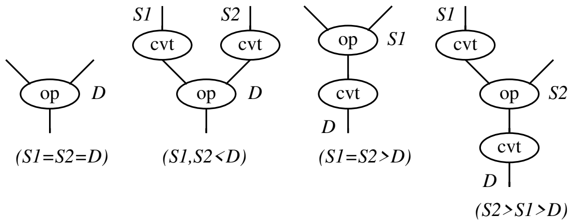

Figure 3: Rules for conversion pruning.

3.1 Transformations on intermediate code

A new pass is run immediately after the SUIF front-end to collect the information expressed by the annotations and to adapt the intermediate code. In particular: (1) it generates the new integer types for the annotated variable declarations and propagates the word size information along the integer expressions; (2) it performs a comparison between the annotated function prototypes and their annotated formal arguments; (3) it replaces type casting function calls with the appropriate conversion instructions.

Most SUIF instructions require that all their operands be of the same type. The front-end introduces conversion instructions `cvt` to adapt operands of different sizes. In this pass, we must enforce the same condition on the user-defined types by inserting conversions to 32 bits between source operands and integer operations, and conversions from 32 bit destination to $\max\{S_i\}$ bits, where S_i is the size of i-th source operand.

Since we no longer require that all integer operations be performed on 32 bits, after introducing the new types we remove all unnecessary type conversions, both on user-defined types and integer base types. A simple one-pass, rule-based algorithm is sufficient to achieve this result. Figure 3 shows the effect of the transformation rules described below on a SUIF instruction tree as the one shown in figure 2. Let us have a three-address operation Op, with result size Op_{size}. Let us call cvt_{S1} and cvt_{S2} the `cvt` operations producing the source operands and cvt_D the operation reading the destination (if present). We define the *effective size* S_1, S_2 and D of Op's operands as follows: if a source (destination) operand is produced (read) by a `cvt` operation, then its effective size is the input (output) of this `cvt` operation); otherwise it is its own size, equivalent to Op_{size}. Depending on the relation among the effective operand sizes, one of the following four rules is applied. This rules conservatively promote the operation precision to the largest operand size.

1. $(S_1 = S_2 = D) \Rightarrow$ remove $cvt_{S1}, cvt_{S2}, cvt_D$
   $Op_{size} \leftarrow D$
2. $(S_1, S_2 < D) \Rightarrow$ remove cvt_D
   $cvt_{S1,size} \leftarrow D; cvt_{S2,size} \leftarrow D$
   $Op_{size} \leftarrow D$
3. $(S_1 = S_2 > D) \Rightarrow$ remove cvt_{S1}, cvt_{S2}
   $Op_{size} \leftarrow S_1$
4. $(S_1 \neq S_2 > D) \Rightarrow$ remove cvt_S
   $cvt_{S',size} \leftarrow S$
   $Op_{size} \leftarrow S$
   where $S = \max\{S_1, S_2\}$ and $S' = \min\{S_1, S_2\}$

Given a minimum addressable unit U of the memory, if a smaller data word internally used by the processor is not exactly a sub-multiple of U, then the problem of aligning such words in memory operations arises. The solution is to provide different opcodes of memory operations involving sub-multiples of U (like the typical load-byte, store-byte instructions) and to align a small word to the minimum sub-multiple that can contain it.

3.2 Three-address code generation

At this point the high-level SUIF code is instrumented enough to represent our enlarged application space and can be translated into machine-dependent SUIF code[11]. During code generation instructions are selected from the instruction repertoire supported by the MOVE Framework [7] and several low-level details are exposed, like the calling conventions, the organization of run-time storage and the target specific instruction set restrictions. Many optimization opportunities are also exposed at this stage. A three-address instruction resembles a RISC instruction, except that for each operand its size is indicated.

The mapping rule from SUIF instructions is straightforward: (1) for the destination operand size we take the result type of the SUIF instruction, (2) the same size is taken for all the source operands, except those that are the result of a `cvt` instruction, which take the size of the `cvt` source operand. Example 3 illustrates this rule.

Example 3 Three address SUIF instruction
The instructions generated for the instruction trees in figure 3 are, from left to right:

(1) `add r3<D>,r1<D>,r2<D>`
(2) `add r3<D>,r1<S1>,r2<S2>`
(3) `add r3<S1>,r1<S1>,r2<S1>`
(4) `add r3<S2>,r2<S1>,r2<S2>`

for example, instruction (2) sums the S1 least significant bits of register `r1` *and S2 least significant bits of* `r2` *and stores the result in the D least significant bits of* `r3`*. Note that the* `cvt` *operation on the result (cases 3 and 4) will be taken into account when applying the same rule to the successor operation(s).*

□□

3.3 Target-specific code transformations

We distinguish two levels of support for a word size: direct execution using hardware resources of the appropriate width or execution on a larger datapath, using sign or zero extensions when necessary to preserve the correct operation semantics. During three-address code generation all word sizes are supposed to be supported directly. The subsequent pass adjusts the three-address code to take into account the width of the hardware resources of the current target ASIP. For a fast design space exploration it is important to split this pass from three-address code generation, because the former is highly dependent from the target configuration and needs to be run for every solution point. Conversely, the three-address code generation pass is only partially target-dependent and needs to be re-run when major changes, like support of new instructions, are added to or removed from the target processor.

Let us consider a three-address instruction in the following form and suppose that the necessary hardware resources with a m bit datapath are present:

```
opc r3<m>,r1<n>,r2<n>
```

The operation precision is determined by the width of the source and destination operands, through a set of rules:

1. If $n = m$ the operation precision is m bits.

2. If $n > m$ the precision is m; the source operands are truncated and the instruction is changed into:

   ```
   opc r3<m>,r1<m>,r2<m>
   ```

3. If $n < m$ there are two possibilities. If a functional unit is available with the required operand registers we can perform a perfect mapping to such unit, therefore we leave the instruction unchanged. Otherwise, we force the source operands to size m by changing the instruction as in case (2). The transport path will automatically sign extend the operands to m bits.

The extension of the last two rules to the case in which source operands have not the same size is straightforward. If the destination size is not supported, then the operation has to be mapped to a larger functional unit. For most operations, among which `add`, `sub`, `mul`, `shl` and all the logic operations, the exceeding bits of the result can be simply ignored, if the destination size does not exceed the operand sizes. Other operations, like `shr` and `div`, always require a conservative sign or zero extension of their operands from the original size to the actual size.

After this set of rules is applied, the scheduler is guaranteed that any operand size specified in the three-address code is available.

source	dest	size	source	dest	size	source	dest	size

Figure 4: Internal representation of a sequential MOVE machine instruction.

3.4 Scheduling

The intermediate three-address code is internally split into three move data objects before starting scheduling, as shown in figure 4. Each move structure contains a size field that represents the minimum width of the resource required for the destination. When more than one of the required hardware resource is available for an operation, the narrowest resource is not always the optimal choice: it may be profitable to select a wider transport path and a wider FU, depending on what resources are free during the scheduling process, rather than postponing the operation until the cheapest resources are available. The resource binding in this case is therefore deferred until the operation is going to be scheduled.

4 Hand coded examples

In this section we present three examples which illustrate how the proposed methodology generates code for arbitrary word sizes. Example 4 shows how additional prototype information is specified for user-defined types; example 5 illustrates the entire code generation trajectory for a small expression including user-defined word sizes; finally, example 6 shows how sign extensions are introduced into the three-address code in order to execute operations with word sizes not directly supported by the hardware.

Example 4 A function prototype specification
Let us consider the following piece of code

```
int foo(int, int, double, int);
#pragma prototype foo 0 20 20 -1 0
...
int foo(int a, int b, double c, int
d) {
#pragma bitsize a 20
#pragma bitsize b 20
...
}
```

The `prototype` *annotation specifies that the first and the second argument have a width of 20 bits and that the function definition should really match a 20 bit type. The value -1 referring to the floating point argument is ignored. The values 0 for the return type* *and for the last argument specify that the width of* `int` *defined by the compiler for the MOVE architecture should be taken (that is, 32 bits). In the function body, immediately after the function header, the* `bitsize` *annotations for the formal arguments with user-defined sizes must be inserted. These sizes must match those specified in the prototype annotation of the same function, to enforce the function prototype semantics of the standard types. After a successful checking, the annotations are removed and the compilation process can proceed.)*

□□

Example 5 Implicit type conversion and code generation

Let us consider the following piece of code

```
short a;
int b,c;
#pragma bitsize b 20
...
c = b*(a+b);
```

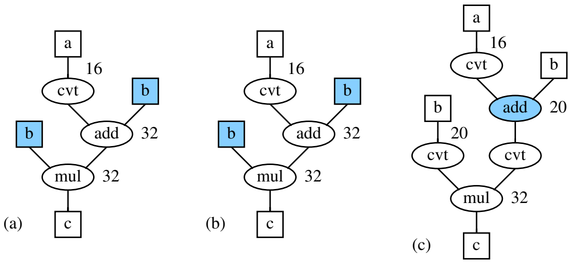

Figure 5: Example of transformations on a SUIF instruction tree. (a) Original intermediate representation. (b) Code resulting after the introduction of `cvt` instructions for the annotated types. (c) Code after conversion pruning.

The SUIF front-end translates the integer expression into the instruction tree in figure 5(a). The first step updates the type of `b` *(shaded boxes) and introduces the necessary conversion instructions, as shown in figure 5(b). The second step prunes the unnecessary* `cvt` *instructions, using the rules explained in subsection 3.1 and changes the* `add` *bit precision to 20 bits (shaded oval). The result, shown in figure 5(c), is translated into three-address, machine-dependent SUIF code:*

```
add rt<20>,ra<16>,rb<20>
mul rc<32>,rb<20>,rt<20>
```

where `ra`, `rb`, `rc` *and* `rt` *are symbolic registers into which variables and temporary values are allocated. Before scheduling, a pass adjusts the three-address code in order to correctly execute it on the target processor. For example, if the target machine contains a 20 bit datapath but not a 16 bit datapath,* `ra` *will be sign extended from 16 to 20 bits by an* `sx` *instruction before the operation. If a multiplier unit taking 20 bit source operands and returning 32 bit products is available, the* `mul` *instruction stays unchanged, otherwise the source operands* `rb` *and* `rt` *are promoted to the destination size. This implies that the transport path will automatically sign extend the 20 bit words during the move. The code we obtain, assuming neither resource available, is:*

```
sx ra<20>,16,ra<20>
add rt<20>,ra<20>,rb<20>
mul rc<32>,rb<32>,rt<32>
```

If the required multiply unit with 20 bit operands and a wider one are available at the same time, the choice is postponed until the scheduling pass. The instruction to be scheduled is first internally translated into MOVE sequential representation, as shown in figure 6, and then, depending on which multiply unit is available at that point of the scheduling process, the moves are assigned to the 32 bit or the 20 bit multiplier. Note that assigning the operation to a 32 bit unit yet does not require inserting an explicit sign extension, if the transport path automatically performs it.

□□

ra	add_o	16	rb	add_t	20	add_r	rt	20
rb	mul_o	20	rt	mul_t	20	mul_r	rc	32

Figure 6: Example of sequential MOVE code.

Example 6 Code generation for word sizes not supported

Let us suppose that the following expression involve 12 bit signed operands, but only a 16 bit data path is available.

```
d = (a+b)/c;
```

The three-address code generation pass produces:

```
add rt<12>,ra<12>,rb<12>
div rd<12>,rt<12>,rc<12>
```

The subsequent transformation pass introduces sign extensions for the `div` *operands, so that their value is in the range of 12 bits.*

```
add rt<16>,ra<16>,rb<16>
sx  rt<16>,12,rt<16>
sx  ru<16>,12,rc<16>
div rd<16>,ru<16>,rt<16>
```

Note that if a `mul` *operation was used instead of* `div`*, these sign extension would be unnecessary, because the 12 least significant bits of the result are only affected by the 12 least significant bits of the source operands.*

□□

5 Conclusions

In this paper we proposed an original methodology to extend the C language and its compilation trajectory to integrate low-level specifications for efficient ASIP synthesis in a high level language. In particular, we annotate integer variable declarations to specify the word sizes of data and we introduce a new pass on the intermediate representation to update the operation precision and the type conversions to the new, enlarged type system. Machine code generation and instruction scheduling are adapted to take into account the required word sizes. The future work will focus on further implementation of the proposed methodology, research on the field of retargetable compilation, and code optimization for this much vaster target architecture space.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesly Series in Computer Science. Addison-Wesly Publishing Company, Reading, Massachusetts, 1985.

[2] Guido Araujo, Sharad Malik, and Mike Tien-Chien Lee. Using Register-Transfer Paths in Code Generation for Heterogeneous Memory-Register Architectures. In *33th Design Automation Conference*, 1996.

[3] Raul Camposano and Jörg Wilberg. "embedded system design". *Design Automation for Embedded Systems*, 1(1):5–50, 1996.

[4] Andrea G. M. Cilio. Comparative Analysis between Automatic Design Methodology and Manual of an Embedded System for MPEG Audio Decoding. Master's thesis, Technical University of Delft, July 1996.

[5] Hendrik Corporaal. *Transport Triggered Architectures – Design and Evaluation*. PhD thesis, Technical University of Delft, August 1995.

[6] Stanford Compiler Group. *The SUIF Library*. Stanford University, 1994.

[7] Jan Hoogerbrugge. *Code Generation for Transport Triggered Architectures*. PhD thesis, Technical University of Delft, February 1996.

[8] Brian W. Kernighan and Dennis M. Ritchie. *The C programming language*. Prentice Hall, 2nd edition, 1988.

[9] Stan Yi-Huang Liao. *Code Generation and Optimization for Embedded Digital Signal Processors*. PhD thesis, Massachusetts Institute of Technology, June 1996.

[10] Stanley B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[11] Michael D. Smith. Extending SUIF for Machine-dependent Optimizations. In *SUIF Workshop*, January 1996.

[12] Richard M. Stallman. *Using and Porting GNU CC*. Free Software Foundation, 675 Massachusetts Avenue, Cambridge, MA 02139 USA, September 1994. version 2.6.