A Macro Expansion Approach to Embedded Processor Code Generation

Eero Lassila
Helsinki University of Technology
Digital Systems Laboratory
Otakaari 1, FIN-02150 Espoo, Finland
eero.lassila@hut.fi

Abstract

This paper describes an experimental prototype of a code generation tool for embedded special-purpose processors. The tool is a retargetable assembly-code-level macro expander capable of program flow analysis. The main advantage of the tool is its strong support for macro hierarchy: hierarchical macro libraries make the code (produced either by the compiler writer or by the assembly language programmer) more modular.

1. Introduction

Code generation for embedded special-purpose processors is a demanding task: typically, one has to face both high performance requirements and a rather irregular processor architecture. This paper introduces an experimental prototype of a code generation tool for special-purpose processors. The tool is a retargetable assembly-code-level macro expander. It is intended to be used both as a part of a compiler back-end and as a stand-alone macro assembler.

The novel feature of our macro expander is its *program flow analysis capability*. By being aware of the data and control flow of the underlying program, the macro expander can track which registers are in use and autonomously allocate free registers for macro temporaries. The implemented prototype tries to demonstrate lucidly the power of the flow analysis mechanism and is not yet meant for coping with real processor architectures.

The main advantage offered by our macro expander is its strong support for macro hierarchy. The *enhanced modularity* provided by libraries consisting of hierarchically used (conditional) macro definitions can make the code more easily readable, maintainable, and reusable. Still, a procedure written in the macro language retains its machine-specificity and, consequently, its efficiency. From the semantical viewpoint, the macro language can be seen as an extension of the base (assembly) language. From the syntactical viewpoint, in contrast, this is not the case: uniformity between different hierarchy levels is achieved by hiding the base language completely under the macro language.

2. Why is assembly language still in use?

Code generation for embedded special-purpose processors [25] is a difficult task for compiler writers, as well as for assembly language programmers. With many *digital signal processors (DSPs)* [21, 22] used in modern telecommunication systems, for instance, compiled code is currently ruled out by efficiency requirements. In general, such special-purpose processor architectures typically present the programmers with three inherent technical complications—which are closely related to the efficiency requirement:

- Unconventional functionality with both limitations and extensions.
- Architectural irregularity: a heterogeneous register set and an idiosyncratic instruction set.
- Fine instruction granularity (with VLIW-type instruction-level parallelism [14]).

Unconventional functionality and architectural irregularity are burdens to the compiler writer, while a fine-grained instruction set is a burden especially to the assembly language programmer. The unconventional functionality and the irregularity stem from the utmost adaptation to the narrow field of intended applications; fine instruction granularity (and hardwired control), on the other hand, seems to correlate with high execution

speed. Especially because of the irregularity, the case analysis needed in code generation eventually becomes too enormous for compilers to manage, as noticed by Wulf in 1981 [33].

We can now collect several reasons that favor assembly language programming in the case of embedded special-purpose processors:

- The exceptionally hard performance requirements: of course, this is the primary reason.
- Unavailability of high-quality high-level language compilers: in addition to the technical complications, the processor often has only a small number of potential users, which makes compiler design commercially unattractive.
- Relative insignificance of application program portability: the special-purpose processor applications are often so specialized that there is little need to consider portability.
- Relative insignificance of the programming costs in the case of such embedded computer systems that are mass-produced articles: for instance, when a mobile phone is designed, even extensive code optimization pays if it results in hardware savings.

One might suppose that the inevitable advances in integrated circuit technology and processor architecture design would soon more than compensate the performance penalty resulting from high-level language use. However, that is not the whole truth: as the processors get faster, software (or firmware) implementations become competitive in new application areas, but only if the full processing capacity can be exploited. In other words, there seems to exist a niche for assembly language programming that cannot be occupied by currently available compiler technology.

3. Flow-sensitivity in macro expansion

In computer programs, abstraction through modularity improves readability, maintainability, and reusability: the details of a module implementation become hidden from the clients of the module. Object-oriented programming is one of the popular modularity-promoting mechanisms at the high-level language level; we try to convince you that the use of relatively powerful modularity-promoting mechanisms may be justified even at the assembly language level.

Macro expansion [7, 6, 8] is a simple modularity-promoting mechanism traditionally used at the assembly language level [28]. However, conventional assembly language macros cannot be freely used hierarchically. For instance, if some macro M_1 uses a register R as a temporary data storage, the programmer has to take care when calling M_1: the macro expander is not able to issue a warning that the value possibly already stored in R will be lost during the execution of M_1 [19, pp. 15–17]. Additionally, if M_1, in turn, calls another macro, say M_2, then the original caller of M_1 must take into consideration even similar restrictions that perhaps concern M_2. In general, such implicit restrictions effectively prevent the introduction of macro hierarchy.

We claim that to support macro hierarchy a macro expander must not be a simple text string substitution engine but a more sophisticated tool aware of the control and data flow of the underlying program. Optimizing compilers [1] perform program flow analysis [18]; we apply it even to macro expansion. The usefulness of *flow-sensitivity* becomes apparent with the following C code fragment:

```
while (x < 0) {                 /* C code */
  y = x + y;
  w = liboper(z);
  x = z + w;
}
```

Let us assume that the C compiler we use can hold all the variables `x`, `y`, `z`, and `w` in registers. Furthermore, we assume that the compiler recognizes the `liboper` library function and is able to treat it as a macro, that is, to replace its call with inline assembly code. Finally, we suppose that the compiler contains a non-trivial "precompiler" that allocates the registers reserved for register variables and expands such library macros as `liboper`. Now, flow-sensitivity would allow the precompiler to observe that when this `liboper` macro call is expanded, the register allocated for `x` is free to be used as a temporary storage but the one allocated for `y` is not (`x` is rewritten by the last statement of the loop body, but `y` has to retain its value throughout the last two loop body statements).

As suggested above, the essential advantage of a flow-sensitive macro expander would be to know which registers are free at the point of each macro call. A register is *free* if it is guaranteed *not* to contain data that should *not* be overwritten, i.e. data earlier written into it and later to be read from it. Accordingly, the natural division of labor is as follows: the human programmer—e.g. a compiler writer—only chooses the optimum register class for each macro temporary; the macro expander then tries to find a free member of the chosen class. Thus, the programmer is released from a great deal of tedious bookkeeping.

If a macro expander could reach adequate flow-sensitivity, we could at least in principle construct hierarchical libraries with conditional macro definitions

as building blocks. Such libraries would, in turn, make the assembly code more easily readable, maintainable, and reusable. Indeed, in Sec. 7 we are able to present a concrete example of multilevel macro hierarchy.

Our most severe problem lies in monitoring all the versatile equipment that processors use to implement the control and data flow—unavoidably, we have to deal with indirect addressing. Note, however, that indirect addressing is often applied only to memory locations, and not to CPU registers. Furthermore, we aim at programmer-assisted mechanisms for *alias analysis* [1, Sec. 10.8].

4. Overview of the proposed tool

We propose the flow-sensitive macro expander as a tool for both compiler writers and assembly language programmers—in other words, both as a part of a compiler back-end and as a stand-alone macro assembler.

Flow-sensitive macro expansion, as well as macro expansion in general, requires two main components: the macro expander proper and a rule base consisting of macro definitions. Because the tool must be taught the target instruction set, we deem it appropriate to divide the rule base explicitly into *system macros* and *application macros*, as shown in Fig. 1. The machine instructions of the target processor are visible to the application macro programmer only through the system macros, which constitute the basis of the macro hierarchy—they are actually not genuine macros but mere placeholders for individual machine instructions. Application macro definitions can be hierarchical: in the definition of an application macro A, you may call any macro B. The task of the macro expander is to convert each application macro call in the input program into a sequence of system macro calls.

Although the macro expander is intended to be easily retargetable, we do not expect that application programs would be portable across different target hardware. As implicitly expressed above, the purpose of the system macro set is *not* to create a standard machine-independent programmer-visible layer by hiding the machine-specific details. On the contrary, the system macros exploit these very details in a controlled but still transparent fashion. There should be a natural one-to-one mapping between the system macros and the target machine instructions. For accuracy, the system macro set should be specified—and even the core of the application macro library should be written—by a single system manager (and not by application macro programmers).

Macros of any kind typically do not have any predefined semantics, contrary to the built-in elements of

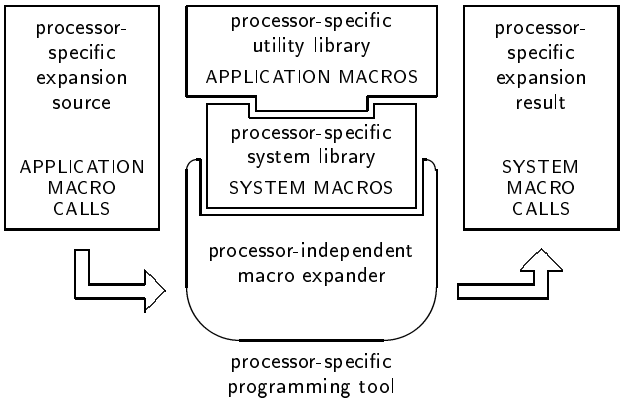

Figure 1. Operation of the proposed programming tool.

programming languages. The power of the proposed approach is in the generality resulting from this lack. For a simplified example, suppose that the target processor instruction set contains an *ADD* instruction that reads two registers and writes one register. Suppose also that the programmer uses the *ADD* instruction (i.e. the corresponding system macro) inside the definition of some application macro *MAC*. When the macro expander, then, tries to expand some call of *MAC*, all that it has to do concerning the *ADD* instance is to select three appropriate registers from the programmer-indicated classes (in particular, it must take care that the data possibly already present in the output register is not prematurely overwritten). Most importantly, it need not even know whether *ADD* actually performs an addition or something wholly different.

To sum up, the lack of predefined semantics offers the following advantages:

- The macro expander proper can be made a relatively small program. It is the rule base consisting of macro definitions that drives the expansion.
- With suitable macro definitions, the tool can be tailored to fully exploit the intricacies of the chosen target processor architecture.
- Operations not needed in the chosen application need not be supported.

5. Implications on target hardware

The flow-sensitive macro expansion is aimed at tackling the case analysis problem in code generation. Obviously, this problem is most severe when the target architecture is an irregular one—when the register set is

heterogeneous and the instruction set is idiosyncratic. Accordingly, among the presently marketed processors we spot the fixed-point DSP chips as our most promising targets (we need an additional postprocessor program for *code compaction* [17]). In particular, we focus on DSPs with limited functionality, high speed, and low price—such as the 16-bit fixed-point AT&T DSP1610 [3].

We are ready to admit that the flow-sensitive macro expansion approach can, even as its best, be feasible only for a strongly restricted class of processor architectures (see [2] for some challenging ones of the numerous intricacies typically found in DSP architectures). Therefore, this class must be given a detailed characterization. We believe that a prospective macro expander implementor should at first be content with an unrealistically narrow characterization and only after a successful prototype strive for step-by-step extensions. A significant reason for not attacking directly the commercial architectures is that even a narrow characterization may perhaps serve as a constraint on future processor design; in a somewhat similar fashion, the RISC processors [26] are designed specifically to be programmed with optimizing high-level language compilers.

Adopting additional architectural constraints of the suggested kind might pay especially in *rapid prototyping* [12, 23] of embedded computer systems. Moreover, our approach promotes customized *application-specific instruction set processors* [24] as possible implementations for microelectronic systems with a tight development schedule. For such a processor, the proposed macro expander could provide an almost ready-to-use code generation tool; with such an application, the lack of portability across different target hardware is usually insignificant.

6. Our current implementation

We have implemented and documented an experimental prototype of the flow-sensitive macro expander [20] (which is available through the Internet). This prototype can cope only with utterly simplified and fully hypothetical processor architectures. Below, Sec. 7 contains an actual macro programming example.

Currently, the essential restrictions on the target architecture include the following:

- Indirect addressing is not supported; the only possible addressing modes are direct and immediate addressing.
- Any two registers must be distinct, that is, they cannot overlap.
- VLIW-type fine-grained instruction-level parallelism is not supported.
- Functions can be neither defined nor called; neither hardware nor software stack is supported.

We are striving for eliminating these restrictions, which is admittedly a major task—or rather, a major collection of closely interdependent subtasks. The first real target we are approaching is the AT&T DSP1610. However, even when we are increasing the power of the macro expander, we absolutely want to retain the full retargetability.

7. A code generation example

Finally, we show how the implemented experimental macro expander can be run. For a more thorough discussion of this simple example, see [20, Sec. 2].

7.1. Specifying the target architecture

A rule file read by the expander should start with a target architecture specification. First, we have to introduce the data storage elements, i.e. CPU registers and memory locations. As shown in Fig. 2, there are 1024 memory locations, four auxiliary registers, and a single accumulator. Thus, there are three distinct storage classes; the expander assumes that the members of each single storage class can be used fully interchangeably.

```
STORAGE {
  M[1024];
  R[4];
  A[1];
}
```

Figure 2. Declaration of data storage elements.

Second, we have to present the assembly instructions as system macros; see Fig. 3. (Actually, we must also define the `Int`, `Lt`, `Gte`, `And`, and `Or` *forms*; for brevity, we have here omitted the form definition section of the rule file.) So there are, in all, ten system macros, among which there are one unconditional branch (`JUMP`) and three conditional branches (`BRANCH`). Most of the system macros are provided with a test (`TEST`) that constrains their use. For instance, `load` can only copy the contents of a memory location (`m`) into an auxiliary register (`r`), while `store` is able to perform the opposite data transfer. Note in particular

that the accumulator cannot be loaded directly from memory.

```
KERNEL {
  set(c > r) {
    TEST And(And(?R(r), Int(c)),
             And(Gte(c,-1024), Lt(c,1024)));
  }
  load(m > r) { TEST And(?R(r), ?M(m)); }
  store(r > m) { TEST And(?R(r), ?M(m)); }
  move(s > d) {
    TEST And(Or(?A(s), ?R(s)),
             Or(?A(d), ?R(d)));
  }
  add(a,r > a) { TEST And(?A(a), ?R(r)); }
  sub(a,r > a) { TEST And(?A(a), ?R(r)); }
  JUMP goto() [l] { }
  BRANCH eq(a) [l] { TEST ?A(a); }
  BRANCH gt(a) [l] { TEST ?A(a); }
  BRANCH lt(a) [l] { TEST ?A(a); }
}
```

Figure 3. Declaration of system macros.

7.2. Defining higher-level macros

A rule file without application macro definitions would not be very useful. In Fig. 4, you can see how a general data transfer (without storage class restrictions) is implemented. The `my_move` macro definition lists the alternative implementations in the priority order; note that the last alternative, `temp_is_needed`, is a recursive one. (The predefined pseudomacros `ALIAS` and `INIT` do not produce any code; the `INSIST` directive is here used only for breaking infinite recursion in the case of a certain macro expansion failure.)

Our second application macro interchanges the contents of two memory locations; see Fig. 5. The `my_mswap` macro requires two free storage elements, because direct memory-to-memory transfers are not supported by the system macro set. At least one of these two must be an auxiliary register, while the other one may alternatively be the accumulator. (Form '`#`' is a predefined one, like '`?`' and '`=`' shown earlier; `#R()` returns the number of free elements in storage class `R`.)

Our third and final application macro (which will not be utilized below) is a conditional branch; see Fig. 6. By including this macro definition, we want to stress the fact that the flow-sensitive macro expansion is *not* limited to straight-line code segments: control flow branches do not disrupt the data flow analysis.

```
my_move(s > d) {
  same: TEST =(s,d); { ALIAS(s > d); }
  as_set: { set(s > d); }
  INSIST Not(And(Int(s), ?R(d)));
  as_load: { load(s > d); }
  as_store: { store(s > d); }
  as_move: { move(s > d); }
  clear_acc: TEST Zero(s); USE R[r]; {
    INIT( > r); move(r > d); sub(d,r > d);
  }
  temp_is_needed: USE R[r]; {
    my_move(s > r); my_move(r > d);
  }
}
```

Figure 4. Definition of the `my_move` application macro.

```
my_mswap(m,n > n,m) {
  TEST ?M(m,n);
  two_aux_free: TEST Gte(#R(),2); USE R[r]; {
    my_move(m>r); my_move(n>m); my_move(r>n);
  }
  acc_and_aux_free: USE A[a]; {
    my_move(m>a); my_move(n>m); my_move(a>n);
  }
}
```

Figure 5. Definition of the `my_mswap` application macro.

```
BRANCH my_zero(x) [l] {
  next: TEST &(l,NEXT); { }
  const0: TEST Zero(x); { JUMP goto() [l]; }
  const: TEST Int(x); { }
  acc: TEST ?A(x); { BRANCH eq(x) [l]; }
  default: USE A[a];
      { my_move(x > a); BRANCH eq(a) [l]; }
}
```

Figure 6. Definition of the `my_zero` application macro.

7.3. Expansion of macro calls

We are now ready to generate code for a call of the my_mswap macro (defined in Fig. 5). We start the macro expander with the rule file created above, and interactively type in the following line:

```
> my_mswap(M[5],M[7] > M[7],M[5]); {R[1],R[3]}
```

Above, we specified that registers R[1] and R[3] are free at the call point. An explicit specification like this is needed here but not allowed inside a macro definition. At the lower-level macro calls that are generated during the expansion of the original call, the expander can autonomously keep track of free registers.

The expansion comprises two phases: first, a tree structure is built up; second, this structure is linearized. Fig. 7 shows the results of both these phases. (You may want to see [20, Sec. 7] for a detailed description of the macro expansion mechanism.) Note especially that when the expander is expanding the my_move(M[7] >M[5]) call (on the second level), it has to recognize that the originally free R[3] register is not free any more.

```
my_mswap(M[5],M[7] > M[7],M[5]) {
  my_move(M[5] > R[3]) {
    load(M[5] > R[3]);
  }
  my_move(M[7] > M[5]) {
    my_move(M[7] > R[1]) {
      load(M[7] > R[1]);
    }
    my_move(R[1] > M[5]) {
      store(R[1] > M[5]);
    }
  }
  my_move(R[3] > M[7]) {
    store(R[3] > M[7]);
  }
}

my_mswap(M[5],M[7] > M[7],M[5]) {R[1],R[3]} {
  load(M[5] > R[3]);
  load(M[7] > R[1]);
  store(R[1] > M[5]);
  store(R[3] > M[7]);
}
```

Figure 7. Expansion result of a my_mswap macro call.

Secondly, we want to generate code for a similar macro call that has, however, a slightly different environment. Now, in addition to register R[1], the single accumulator, A, is free, instead of register R[3]:

```
> my_mswap(M[5],M[7] > M[7],M[5]); {R[1],A}
```

This time, Fig. 8 shows only the final macro expansion result.

```
my_mswap(M[5],M[7] > M[7],M[5]) {R[1],A} {
  load(M[5] > R[1]);
  move(R[1] > A);
  load(M[7] > R[1]);
  store(R[1] > M[5]);
  move(A > R[1]);
  store(R[1] > M[7]);
}
```

Figure 8. Final expansion result of another my_mswap call.

8. Related work

Of course, current high-level programming languages are generally implemented with *compilers*. Around 1970, however, there was considerable interest in implementing high-level languages as *macro systems* [13, 6]. The advantages found were twofold:

- A high-level language implemented as a multi-level macro system could be easily ported onto various target machines by rewriting the definitions of the lowest-level macros [30, 5]. Perhaps the most representative example is Griswold's SNOBOL4 implementation [16].
- The programmer could be easily equipped with powerful macros for fully controlled machine-specific optimization. For instance, Dickman's system [11] was capable of "intraclass" register allocation according to explicit live variable information provided by the user.

Unfortunately, these two uses of macros readily appear to be in conflict with each other, for machine-specific optimizations are, by definition, not portable. Still, the present approach aims at a feasible combination: a macro expander that constitutes a retargetable tool for creating machine-specific code generators.

Today, *embedded special-purpose processor programming* is an actively studied field in which the requirements on compiler technology are especially high and not fully met yet [25, 15, 27, 4]. There are obvious similarities to the state of *system programming*

around 1970 [31, 34, 32, 29] and *microprogramming* around 1980 [10, 9]; optimizing compilers for machine-independent high-level languages have now effectively succeeded in the former case, but in the latter case (whose significance has been reduced by the RISC movement) the progress has been slower.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] G. Araujo, S. Devadas, K. Keutzer, S. Liao, S. Malik, A. Sundarsanam, S. Tjiang, and A. Wang. Challenges in code generation for embedded processors. In Marwedel and Goossens [25], pages 48–64.

[3] AT&T Microelectronics, Allentown, Pennsylvania. *DSP1610 Digital Signal Processor Information Manual*, Dec. 1992.

[4] J. C. Bier, E. E. Goei, W. H. Ho, P. D. Lapsley, M. P. O'Reilly, G. C. Sih, and E. A. Lee. Gabriel: A design environment for DSP. *IEEE Micro*, 10(5):28–45, Oct. 1990.

[5] P. J. Brown. Levels of language for portable software. *Communications of the ACM*, 15(12):1059–1062, Dec. 1972.

[6] P. J. Brown. *Macro Processors and Techniques for Portable Software*. Wiley, London, UK, 1974.

[7] M. Campbell-Kelly. *An Introduction to Macros*. Macdonald, London, UK, 1973.

[8] A. J. Cole. *Macro Processors*. Cambridge University Press, second edition, 1981.

[9] S. Dasgupta and B. D. Shriver. Developments in firmware engineering. In M. C. Yovits, editor, *Advances in Computers*, volume 24, pages 101–176. Academic Press, 1985.

[10] S. Davidson. High level microprogramming—current usage, future prospects. *ACM SIGMICRO Newsletter*, 14(4):193–200, Dec. 1983.

[11] B. N. Dickman. ETC—an extendible macro-based compiler. In *Proceedings of AFIPS 1971 Spring Joint Computer Conference*, pages 529–538, 1971.

[12] A. Dollas and J. D. S. Babcock. Rapid prototyping of microelectronic systems. In M. C. Yovits and M. Zelkowitz, editors, *Advances in Computers*, volume 40, pages 65–125. Academic Press, 1995.

[13] D. J. Farber. A survey of the systematic use of macros in systems building. *ACM SIGPLAN Notices*, 6(9):29–36, Oct. 1971.

[14] J. A. Fisher and B. R. Rau. Instruction-level parallel processing. *Science*, 253(5025):1233–1241, Sept. 1991.

[15] G. Goossens, J. Rabaey, J. Vandewalle, and H. De Man. An efficient microcode compiler for application specific DSP processors. *IEEE Transactions on Computer-Aided Design*, 9(9):925–937, Sept. 1990.

[16] R. E. Griswold. *The Macro Implementation of SNOBOL4*. W. H. Freeman, San Francisco, California, 1972.

[17] S. M. Kafka. An assembly source level global compacter for digital signal processors. In *Proceedings of 1990 International Conference on Acoustics, Speech, and Signal Processing*, volume 2, pages 1061–1064. IEEE, 1990.

[18] K. Kennedy. A survey of data flow analysis techniques. In S. S. Muchnik and N. D. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 5–54. Prentice-Hall, 1981.

[19] J. R. Larus. Assemblers, linkers, and the SPIM simulator. Appendix A of J. L. Hennessy and D. A. Patterson, *Computer Organization and Design: The Hardware/Software Interface*. Morgan Kaufmann, San Mateo, California, 1994.

[20] E. Lassila. ReFlEx—an experimental tool for special-purpose processor code generation. Technical Report B15, Helsinki University of Technology, Digital Systems Laboratory, Espoo, Finland, Mar. 1996. See ftp://saturn.hut.fi/pub/reflex/README for an electronic copy.

[21] E. A. Lee. Programmable DSP architectures (part I). *IEEE ASSP Magazine*, 5(4):4–19, Oct. 1988.

[22] E. A. Lee. Programmable DSP architectures (part II). *IEEE ASSP Magazine*, 6(1):4–14, Jan. 1989.

[23] V. K. Madisetti. *VLSI Digital Signal Processors: An Introduction to Rapid Prototyping and Design Synthesis*. Butterworth-Heinemann, Boston, Massachusetts, 1995.

[24] P. Marwedel. Code generation for embedded processors: An introduction. In Marwedel and Goossens [25], pages 14–31.

[25] P. Marwedel and G. Goossens, editors. *Code Generation for Embedded Processors*. Kluver, 1995.

[26] D. A. Patterson. Reduced instruction set computers. *Communications of the ACM*, 28(1):8–21, Jan. 1985.

[27] K. Rimey and P. N. Hilfinger. A compiler for application-specific signal processors. In *VLSI Signal Processing, III*, pages 341–351. IEEE, 1988.

[28] D. Salomon. *Assemblers and Loaders*. Ellis Horwood, Chichester, UK, 1992.

[29] W. L. van der Poel and L. A. Maarssen, editors. *Proceedings of the IFIP Working Conference on Machine Oriented Higher Level Languages*. North-Holland, Amsterdam, The Netherlands, 1974.

[30] W. M. Waite. The mobile programming system: STAGE2. *Communications of the ACM*, 13(7):415–421, July 1970.

[31] N. Wirth. PL360, a programming language for the 360 computers. *Journal of the ACM*, 15(1):37–74, Jan. 1968.

[32] W. Wulf, C. Geschke, D. Wile, and J. Apperson. Reflections on a systems programming language. *ACM SIGPLAN Notices*, 6(9):42–49, Oct. 1971.

[33] W. A. Wulf. Compilers and computer architecture. *IEEE Computer*, 14(7):41–47, July 1981.

[34] W. A. Wulf, D. B. Russel, and A. N. Habermann. BLISS: A language for systems programming. *Communications of the ACM*, 14(12):780–790, Dec. 1971.