

# Trade-Offs in HW/SW Codesign

**Wolfram Hardt, Raul Camposano**  
**University of Paderborn**  
**FB 17 Computer Science**  
**e-mail: hardt@uni-paderborn.de**  
**Germany**

## Abstract

HW/SW codesign is becoming an increasingly more interesting research field because most practical systems consist of both HW and SW. In this paper we explore a bottom up HW/SW codesign strategy to investigate trade-offs in time behavior and area. A comparison of hardware and software implementations of low level modules is given. A first prototype implementation extracts time and area criteria analyzing the software code generated.

## 1.0 Introduction

At present, hardware and software are mostly developed independently. As a consequence there is little opportunity to optimize both hardware and software together. Moreover, it is also difficult to reason about a complete system (i.e. simulation, verification). These problems have been emphasized recently by several trends such as increasing complexity and increasing “personalization” of systems in software. Thus, HW/SW codesign is a research area of growing importance. One basic approach can be characterized by identifying and implementing software parts which consume high computing resources (usually time) in hardware [Henk92]. The dual approach seeks to identify complex system parts which are good candidates to be implemented in software [Gupt92].

Our goal is to develop methods and criteria for the partitioning of such systems into a hardware part and a software part. We chose a strategy that distinguishes several abstract levels. The main levels considered here are hardware (typically a computer and special purpose hardware), machine language (assembler), programming language (C, C++), application modules (functions, procedures, methods) and whole applications (e.g. a compiler). These levels are examined bottom-up. At each level trade-offs in performance and complexity are examined by moving software primitives (different at each level, e.g. instructions, programming language constructs, functions) into hardware (Figure 1).

Notice that we ignore explicit support software such as the operating system or communication subsystems; this would complicate the approach too much and goes beyond our initial goals. This paper presents results of HW/SW trade-offs at each of these levels and attempts to develop initial guidelines for partitioning. The rest of the paper is organized as follows. Our HW/SW codesign environment is described next. Chapter 3.0 introduces a base for HW/SW comparison and chapter 4.0 points out trade-offs found transforming elements of the different levels into hardware. The paper ends stating some conclusions

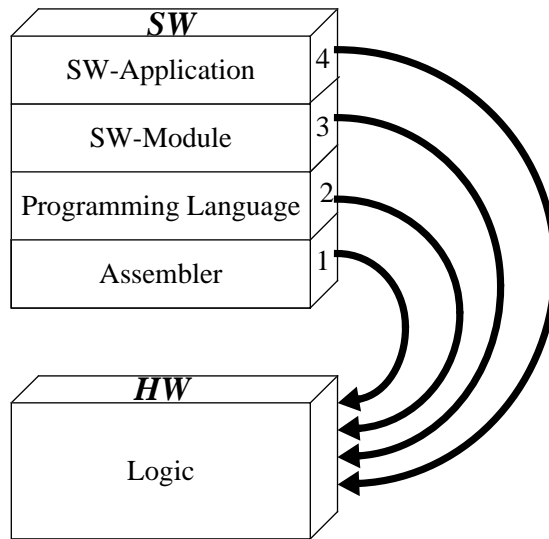


FIGURE 1. Levels of abstraction

## 2.0 HW/SW Codesign Environment

Among the many architectural choices for HW/SW implementations we chose a given microprocessor with additional hardware attached to it, i.e. a “coprocessor”. The well known sparc [Cyp90] architecture is based on a powerful general purpose processor and allows a wide variety of configurations and extensions, e.g. additional cache units, one floating point processor and one additional coprocessor. A standard configuration is shown in Figure 2.

Due to our chosen architecture we used a sparc based workstation (ELC) and the GNU [Stal92] compiler for primary research regarding the software part of a design.

## 3.0 Comparison of Hardware- and Software Designs

Hardware designs implemented in the same technology can in principle be compared. Criteria used for comparison of synchronous hardware designs are size (area) and time (delay). Furthermore, testability, design time and probably power consumption must be taken into account.

Software applications are evaluated by the amount of source code (lines of code), the size of the executable code (specific for a given compiler, machine and operating system) and the necessary data size. The runtime is mostly data and system dependent. Other advantages such as ease of changes (programmability), support, etc. are not considered here.

The criteria for hardware and software described above are very different and do not allow direct comparisons of mixed systems. For our HW/SW codesign activities we define a basic comparison restricted to

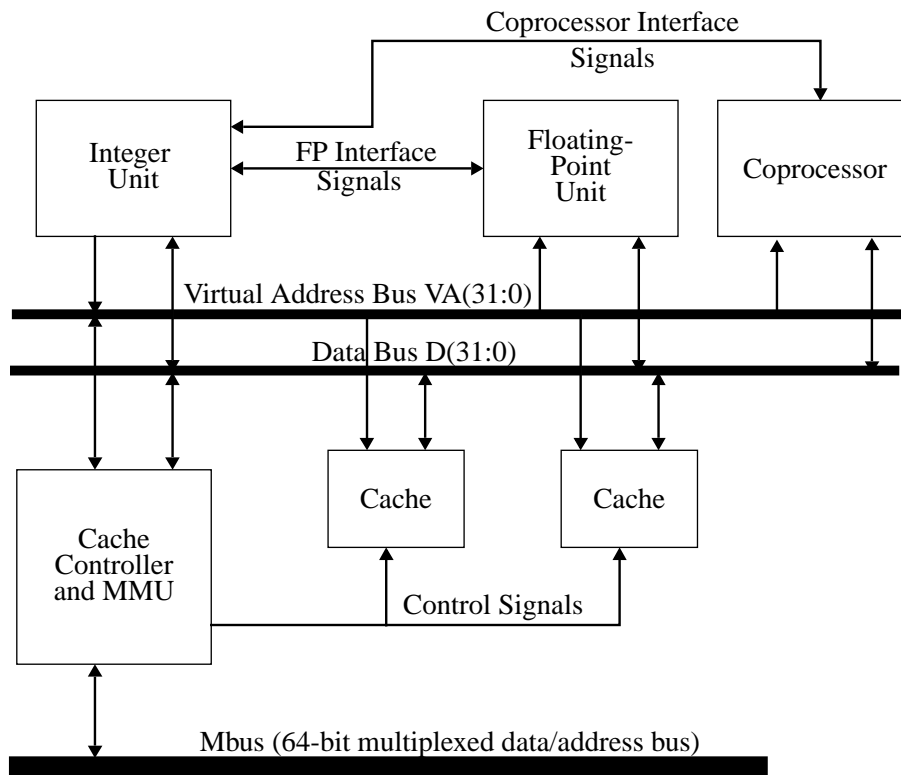


FIGURE 2. Standard configuration of a sparc architecture [Cyp90]

the two factors area and time. In the next two subsections we explain how to obtain this data for hardware and software.

### 3.1 Area

We decided to use an absolute measure unit ( $\mu\text{m}^2$ ) to compare the area of designs. This allows to implement parts of the design in different hardware technologies and also in software. There are some definitions necessary which are described next.

#### 3.1.1 Hardware

Hardware today is often described by a HDL<sup>1</sup>. The description is synthesized and implemented in a given technology, e.g. CMOS. Tools derive synthesis results which are presented in many different forms and on different abstraction levels (i.e. modules, gates, cells, transistors). Usually the wiring area is only included in the final layout. We decided to take the active area into account including all modules and registers. We map areas to a CMOS  $2\mu\text{m}$  standard library [Miet04] which allows to compute an absolute area. For the purpose of comparing this in an “standard” area.

1. Hardware Description Language

### 3.1.2 Software

The more difficult part is to define the area of a software implementation. At this point our particular architecture becomes important. We assume that there is a general purpose machine which is extended by some special hardware. The difference found by moving parts of the software into hardware can be measured by the different amount of used memory.

Table 1 shows the characteristics of a 4MB memory circuit in 0.5  $\mu\text{m}$  technology [Naka92]. The chip

Feature	Characteristic
Organization	1M x 4
I/O Interface	TTL
Power	single 3.3V
Cell size	1.43 $\mu\text{m}$ x 3.04 $\mu\text{m}$
Chip size	6.2 mm x 10.6 mm
Design	0.5 $\mu\text{m}$

TABLE 1. Characteristics of a 4 MB memory chip

size (65.72  $\text{mm}^2$ ) includes the area of control and wiring. Following our concept, using general purpose hardware, we suppose there is memory available. Hence use of memory is measured by the memory cells area only. The control hardware will not be taken into account, since it is needed anyway in a system with memory. (This argument does not hold 100%, but it is a fair approximation for “small “ software.) The area is given by cell sizes, 139  $\mu\text{m}^2$  per 32 bit word. Notice that the technology used for our memory is significantly smaller than the technology used for the “custom” hardware (chapter 3.1.1). This is usual practice.

## 3.2 Time

Synchronous systems are controlled by clock signals. The clock cycle time depends on the chip technology and the system configuration. This relative time base can be identified easily for different architectures. Hence we measure time by an approximation of the required execution cycles, both for software and hardware. This assumes also that the processor and the additional hardware use a common clock, and that the additional hardware does not slow down this clock (i.e. it is as fast or faster than the processor). Note that processors are usually fairly fast, so this assumption does not “waste” speed in the additional hardware.

### 3.2.1 Hardware

Our chosen architecture is controlled by a 33 MHz clock rate. The special purpose hardware is controlled by the same clock. This must be imposed as a constraint to the scheduler to limit the amount of chaining. Trade-offs which may be found by comparing systems with special purpose hardware controlled by clock rates with multiple frequencies are not evaluated at this point. The number of cycles is computed during scheduling in high level synthesis. We used the values published by HIS [Camp91] to compute the number of cycles used by the hardware.

### 3.2.2 Software

The time to execute a software part can be approximated by the number of machine instructions needed times the average number of cycles taken per instruction. RISC<sup>1</sup>- processors reach nearly an execution rate of one instruction per cycle [Cyp90] computing sequences of instructions. Control instructions (i.e. jump, interrupts, traps, etc.) lower this rate because of pipeline hazards. Also floating point operations usually need more than one machine cycle.

We examine the assembler code generated by the GNU “C”-compiler [Stal92]. Our static analysis computes the number of instructions used and the number of cpu cycles needed for execution on a sparc architecture. Cycle computation includes some pipeline effects and the parallelism of instruction and floating point unit. In addition the amount of allocated data is summarized and a small statistic of used instruction types is computed. This leads to a more detailed approximation of execution time than is given by the average instruction execution time. Table 2 partly lists the number of cycles we take into account per instruction. Note that for small software pieces the average number of cycles per instruction is not a good measure anyway. (Since the instruction mix is not representative.) Moreover, cache effects can often be neglected completely since the code may fit into the cache.

For large applications cache faults must be considered [Smith82], [Smith86], [Good83]. We are extending our software to consider also cache and memory access more exactly.

Instruction	Cycles	Floating Point Instruction	Cycles
conditional jump	2	FADDs	5
unconditional jmp	3	FCMPs	4
load	2	FDIVs	23
load double floating point	3	FDIVd	37
store	3	FSQRTs	34
arithmetic/logical	1	FSQRTd	63

TABLE 2. Instruction cycle count [Cyp90]

## 4.0 Trade-Offs

### 4.1 Assembler Level

The assembler level is defined by the instruction set of the processor. Many different instruction sets are known and can be grouped for example into the two categories of RISC- and CISC<sup>2</sup> designs minimizing either execution time per instruction or number of instructions per intended action. The used instructions

1. Reduced Instruction Set Computer
2. Complex Instruction Set Computers

are counted and the instruction execution time is set into relation to the frequency of use in order to define an optimal instruction set [Patt90]. This has generated for example the trend to RISC-architectures. Considering HW/SW codesign at this level moving short sequences of instructions to hardware can be compared with designing instruction sets. Thus the granularity of elements is rather fine and the improvements may be consumed by the slowing down of all instructions. Instruction set design is an old discipline [Lund77], [Keem67] and we do not expect dramatic improvements for HW/SW codesign at this level.

## 4.2 Programming Language Level

Raising the level of abstraction next programming languages have to be examined. Their constructs are implemented by sequences of machine instructions generated by the compiler. These sequences are larger than those which are necessary to implement complex machine instructions. We chose the well known language "C" for an illustrative analysis. Its more complex constructs can be grouped into four classes: assignment, comparison, call and control. Starting from the software point of view, the implementation of these constructs is examined.

- Assignment

Assign constructs can be distinguished by the assigned value (constant, variable, implicit given value). The software implementation needs at least three cycles, whereof two are used for data transfers.

- Comparison

The number of needed cycles for this basic construct depends on the types of compared values. The standard constant zero is implemented by the compiler in the same way than a user defined constant. The data transfer is here also the dominant cycle consumer. Typical cycle amounts range from 3 to 5.

- Call

Function calls are frequent in software implementations. They build the interface to software blocks used multiple times. The basic call without return value and without parameters requires five cycles which are necessary to store (restore) the stackpointer and to execute the jump-instruction. Additional cycles are introduced to handle the parameters and the return value. This is already supported by hardware. The sparc-processor uses register windowing, which allows efficient exchange of parameters. Notice that this is the programmers (compilers) point of view which may be called the best case. The limited number of available register windows implies some operation system activities if a window overflow occurs.

A drawback of Sparc-architectures is slow handling of certain data types. For example a call that returns an eight bit value requires 2 additional cycles.

- Control

The simple control constructs (if-then-else) are based on compare constructs and jump-instructions. Compiler optimization will eliminate the code sequences entered in case *if(0)*. The more complex control constructs define loops (for, while). A substitution of these constructs by basic constructs discussed before leads to the same amount of cycles.

Figure 3 displays this result. The dark area marks the average of the basic variant of all five categories and gives an impression of elements granularity at this level.

The conclusion at this point is that even constructs of an abstract programming language are so simple that the execution is basically done in three to five cpu-cycles. Even this amount is mainly caused by data transfers. An implementation by hardware would result in a computer architecture which slightly expands the cpu's instruction set (at the expense of increased hardware complexity).

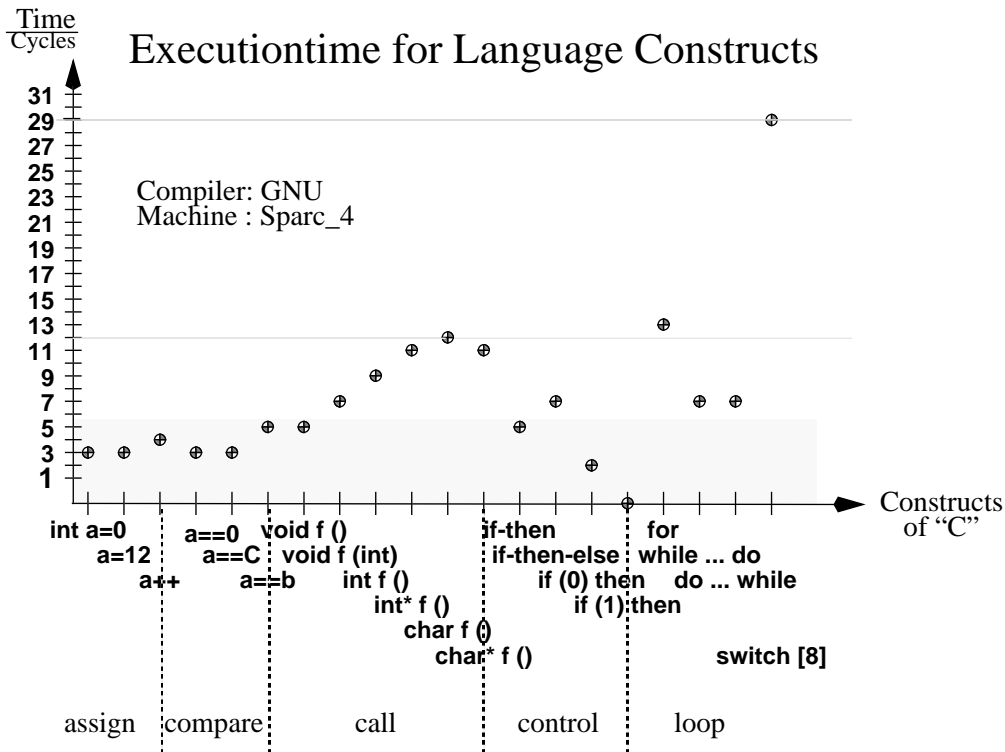


FIGURE 3. Implementation of C-constructs

There is only one more construct which may allow some gain in hardware.

- Switch

The switch construct is more complex. Figure 3 shows the use of 29 instructions for eight different cases. This result depends on the chosen algorithm to implement this construct. Most compilers apply three different strategies with respect to the interval in which the case-values are spread and the number of possibilities which may be chosen. The compiler uses nested *if-then-else* constructs to implement the switch construct if there are only a few cases to distinguish. Decision trees are preferred if the case values are widely spread. The third algorithm deals with many cases. The target addresses of each case's instruction sequence is stored in an address table and the algorithm computes the address's place in this table. This explains the amount of necessary instructions.

A typical switch construct is given in Figure 4. The break is necessary in each case to avoid side effects, i.e. execution of several cases during one loop.

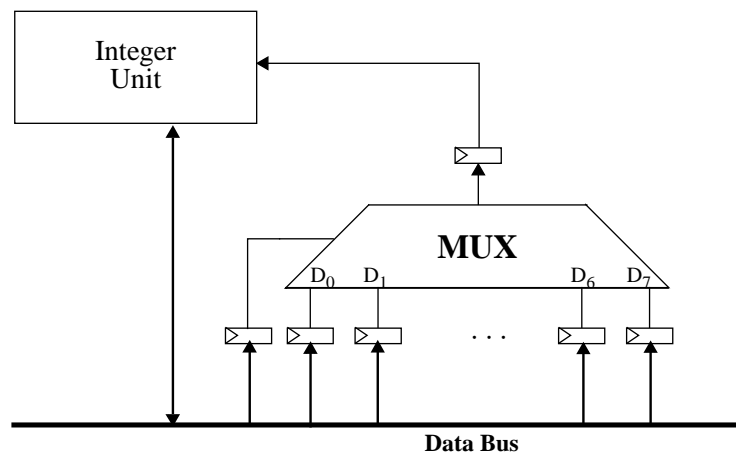
```

switch ( int A )
{
    case 0 :    { sequence of C-commands}    break;
    case 10:   { sequence of C-commands}    break;
    case 20:   { sequence of C-commands}    break;
    .
    .
    .
    case n:    { sequence of C-commands};    break;
}

```

**FIGURE 5. Standard switch construct**

The outline of a hardware implementation restricted to 8 cases (including default) is shown in Figure 5. The registers linked to the data inputs can be load from memory with the relevant addresses during run time. This basic description was synthesized and compared with the software implementation (the “case” entry in Table 3, Table 5, Table 6). Considerable time improvements are found, if no communication overhead is taken into account, neglecting the register load cycles. It is relevant for fixed jump addresses, i.e. either hardwired or loaded once for ever, which is probably the case in small applications. In the worst case all registers must be load in sequence while the cpu is waiting. This values are given in parentheses and the time improvement is rather small. Hence there is little use of adding case-coprocessors to the given hardware..



**FIGURE 5. Hardware implementation of standard switch construct**

### 4.3 Software-Module Level

Most programming languages allow the structuring of code sequences in procedures and functions (i.e. “modules”). At this level we examined modules defined by source code functions. We converted some

small benchmarks [Ben92] given in VHDL to C. This allows the comparison of hardware and software implementation. For hardware implementation the results reported by the HIS system [Camp91] of IBM were used here and normalized as explained in chapter 3.0. The greatest common divisor (gcd) was implemented for four bit, the diffeq and the elliptic filter for 16 bit. Also the synthesis results of the switch construct are given. Table 3 lists the area of the hardware implementation of this examples.

Benchmark	Area mm <sup>2</sup>
gcd	0.77
diffeq	12.96
ellipticF	12.0
case	6.5

**TABLE 3. Area of hardware designed benchmarks in 2  $\mu$ m CMOS technology, standard cells**

In comparison to this we give the area of some well known processors from [Patt90] in Table 4.

Processor	Area mm <sup>2</sup>
Sparc	56
Intel 80486	160
MIPS R3000	72
Motorola 88100	81

**TABLE 4. Area of common processors in ceramic PGA technology**

Time and area for the software implementation of these examples are computed as defined in chapter 3.0 and summarized in Table 5 together with the results of hardware implementations. The time complexity of the hardware implementation is based on path based scheduling. Restrictions were given for diffeq (function units limited to 1 adder, 1 subtractor and 2 multipliers) and for elliptic filter (function units limited to 1 adder and 1 subtractor).

Here first trade-offs of moving software into hardware can be identified. Considerable time improvements of 4 to 30 times are achieved. It has to be mentioned that this holds for one iteration of all loops in the module. Each further iteration may raise this but the number of iterations cannot be determined by source code examination because it is mostly data dependent and not known at compile time.

Area comparison faces another problem caused by the bit width of the hardware modules. The corresponding software implementation allows a common width of 32 bits, while hardware implementations are mostly smaller, tailored to the real problem (4 bits for the gcd, 16 bits for diffeq and ellipticF). The area of n-bit data paths implemented by bit-slice-like structures can be easily computed by multiplying the area of one bit slice by the width. Table 6 lists the design trade-offs for the given examples for time and area. The approximated area of the modules extended to 32 bit data path is listed explicit.

Benchmark	Area of HW design in $\mu\text{m}^2 * 10^3$	Area of SW application in $\mu\text{m}^2 * 10^3$	Average Time of HW design in cycles	Time of SW application in cycles
gcd	770	7.3	2.5	84
diffeq	12,960	10.9	5	144
ellipticF	12,000	22.1	13	312
case	6,541	5.9	1 (10)	44

TABLE 5. Benchmarks implemented in HW and SW

Benchmark	Area HW/SW	Area HW/SW <sub>32</sub>	Time SW/HW	Control-Instruction (%)
gcd	105	840	33.6	18.8
diffeq	1188	2376	28.8	2.53
ellipticF	542	1084	24	1.2
case	1108	1108	44 (4.4)	29

TABLE 6. HW-SW trade-off

These examples can be classified considering the number of control instructions used. The percentage of jump instructions is given in Table 6. Correlating these values with the time improvements our preliminary results seem to point out that control dominated modules are better HW/SW codesign candidates. Notice that this tendency is clearly seen for the case example's best case.

Considering area the hardware implementation requires nearly 1000 times as much as the software implementation. On the other hand, the size of the additional hardware compared to the processors (Table 4) is relatively small. If the communication between hardware and software parts which requires additional clock cycles is taken into account the improvement is lowered to about 10% (example case). Hence improvements in system design moving parts from software to hardware can only be expected at higher levels. To introduce HW/SW codesign in high level synthesis at least the aspects mentioned above must be examined.

#### 4.4 Software-Application Level

Software applications may consist of several software modules. Usually libraries are used. Especially system dependent functions, input and output routines, etc. are only available as compiled library. We have to extend our workbench to examine also software available only as object code. On this level only few activities can be found. A software oriented approach to synthesize embedded controllers from C [Ernst90] applies runtime analysis. Using runtime restrictions to determine what to implement in hardware leads to a different kind of HW/SW codesign motivation. Another approach starting from an initial

hardware design [Gupt92] implements data dependent loops in software. The chosen example, a ethernet coprocessor, was successfully implemented with a hardware reduction of about 20 percent. We will emphasize applications which are often used in a standard system. E.g. the blitter [Ben92] is a coprocessor to perform special data access. This example will allow to examine HW/SW codesign specific module definitions. An even more complex application will be a restricted compiler.

For preliminary research we examined a compiler which accepts a subset of ALGOL [Lin77] and generates assembler code. This application is implemented by nearly 650 lines of C-code. We fed this into our tool for source code examination (chapter 3.2.2) and extracted a control dominated module. Therefore a hardware implementation was generated and compared with the approximated runtime of the software implementation. The results are briefly summarized below.

This restricted compiler consists out of 47 modules with an average percentage of control instructions of 10.7% and a maximal percentage of 27.8%. The most control dominated module (27.8%) is called during a compilation of a program to compute prime numbers 1196 times. It consists out of several *if*-statements. We converted this into a standard *switch* construct and mapped it onto the hardware implementation described in Figure 5 adding some logic of comparable small size (55 cells, 189.520  $\mu\text{m}^2$ ). Comparing the execution time of the hardware and software implementation of this module a speedup of 56 times is reached by introducing hardware. This is consistent with the results found on software module level.

We are implementing a non recursive restricted compiler in software and generating a hardware implementation to compare them. Approaches designing high performance architectures to support efficiently the C programming language such e.g. CRISC [Ditz87] provide optimized hardware designs which gives good guidance in this field.

## 5.0 Conclusion and Future Work

In this paper we examined the trade-offs in HW/SW codesign at different levels. A way of comparing hardware- and software designs is defined. Trade-offs are found only at the software module level and above. Further research concerning the application level is under way. Design descriptions are transformed into the ISF synthesis format [Camp93] which allows a more detailed examination such as synthesis in hardware and software of larger modules and applications. Additional aspects must also be considered, e.g. separation of control and arithmetic part and the definition of modules characteristics that indicate an advantageous implementation in either hardware or software.

## 6.0 References

- [Ben92] "Benchmarks for the 6th International Workshop on High-Level Synthesis." Available through electronic mail at ics.uci.edu, 1992.
- [Camp91] R.Camposano, R.A.Bergamaschi, C.E. Haynes, M.Payer, S.M.Wu. *High-Level VLSI Synthesis*, chapter 4, pages 79–104. Kluwer Academic Publishers, 1991.
- [Camp93] Andreas Hoffmann, Heinz Josef Eikerling, Wolfram Hardt, Reiner Geneveriere, Raul Camposano. "Isf- eine Entwurfsdarstellung für High-Level-Synthese." *Workshop des SFB 358 der TU Dresden und der ITG/GI*, 1993.
- [Cyp90] Cypress Semiconductor. "Sparc/RISC User's Guide." Ross Technology Subsidiary, 3901

North First Street, San Jose, CA 95134. 1990.

- [Ditz87] Daniel R. Ditzel, Hubert R. McLellan. "The Hardware Architecture of the CRISP Microprocessor" *ACM Computer Architecture News*, Vol 15 No.2, pages 309-319, 1987.
- [Ernst92] Rolf Ernst, Ulrich Holtmann. "Some experiments with low-level speculative computation based on multiple branch prediction for the synthesis on high-performance, pipelined coprocessors." *Sixth International Workshop on High Level Synthesis*, pages 146–158, 1992.
- [Gupt92] Rajesh K. Gupta and G. De Michelli. "System synthesis via hardware- software co-design." *CSL TR-92-548*, 1992.
- [Henk92] R.Ernst, J.Henkel "Hardware-Software Co-design of Embedded Controllers based on Hardware Extraction" *Proceedings of the ACM Workshop on Hardware-Software Co-design*, 1992.
- [Good83] J.R.Goodman J.E.Smith. "A study of instruction cache organizations and replacement policies." *Proc. Tenth Annual Symposium on Computer Architecture*, pages 132–137, January 1983.
- [Patt90] D.A.Patterson, J.L.Hennessy. *Computer Architecture A Quantitative Approach*. MORGAN KAUFMANN PUBLISHERS, INC., 1990.
- [Keem67] W.M. Mc Keeman. "Language directed computer design." *Proc. 1967 Fall Joint Computer Conf., Washington, D.C.*, pages 413–417, 1967.
- [Lin77] C.H. Lindsey, S.G. Van der Meulen. "Informal introduction to ALGOL 68" NORTH-HOLLAND PUBLISHING COMPANY, 1977.
- [Lund77] A. Lunde. "Emperial evaluation of some features of instruction set processor architecture." *Comm. ACM*, pages 143–152, March 1977.
- [Miet04] Mietec document S2SM 1.0.0. *Mietec Standard Cell User Manual 2.4 um CMOS*. EUROCHIP Document MIE/F/04.
- [Naka92] K. Nakagawa, N.Akiyama, T. Ohta, T. Someya, A.Tamba, Yuji Yokoyama a.o. "Circuit Technologies for a 12ns 4Mb TTL BiCMOS DRAM at 3.3V Operation." *Symposium on VLSI Circuits Digest of Technical Papers*, pages 62–63, 1992.
- [Smith82] A.J.Smith. "Cache memories." *Computing Surveys*, pages 473–530, September 1982.
- [Smith86] A.J.Smith. "Bibliography and readings on cpu cache memories and related topics." *Computer Architecture News*, pages 22–42, January 1986.
- [Stal92] Richard M. Stallmann. *Using and Porting GNU CC*. Free Software Foundation Inc., December 1992.