R. Ernst, J. Henkel: Hardware-Software Codesign of Embedded Controllers Based on Hardware-Extraction, International Workshop on Hardware-Software Codesign, Estes Park, Colorado, 30.9.-2.10.1992.

Hardware-Software Codesign of Embedded Controllers Based on Hardware Extraction

R. Ernst, J.Henkel
Technische Universität Braunschweig
Germany

Abstract

We present a system for hardware-software codesign of embedded controllers. We show that the *full set of a standard programming language* (C) with real-time programming extensions can be used as input, including dynamic structures. This is feasible due to a deviation from the standard data representation in high-level synthesis. The design process starts with an *all-software solution* and shall *extract* code segments for implementation in hardware, but only where timing constraints are violated. Three classes of hardware shall be extracted, interface primitives, coprocessors and second cores. We explain why we think the latter two classes are the most important ones for hardware-software codesign. We discuss the extraction side effects and conclude that a dual-loop iterative extraction process is a good choice. Clustering known from high-level synthesis can be applied but counterexamples show that current clustering criteria are not sufficient. We propose additional extraction functions. An example shows the viability of the approach.

Introduction

Embedded control systems are a constantly growing field with a large share of the semiconductor market. Applications range from office automation, telecommunication and consumer products to industrial and automative control. Embedded control requires reactive systems, i.e. systems which react in real time to external asynchronous events, rather than process an input and produce an output after some time, such as classical data processing.

The system architecture is a combination of programmable microprocessor cores with memory and hardwired or field programmable peripheral devices. Hardware and software together form the control system.

Embedded control applications are of increasing complexity including e.g. 3D-signal processing, computer vision and fuzzy logic. Consequently, the architectures become more complex and have caught up with workstation technology using 32bit RISC processors.

At the same time there is strong incentive to speed up embedded control system design to meet tight time-to-market requirements. Hardware design must often start when the specification is still subject to changes. All this makes controller design increasingly difficult.

As a result, embedded controller design is changing. In particular at the high end, assembler coding is gradually replaced by higher-level-languages, often C.

To minimize customized hardware, libraries of standardized peripheral components are currently developed, such as in the European OMI-project. Though this approach allows fast design turn-around and quick modifications, design space is severely limited. Right at the controller interface, the library approach might be acceptably efficient, because most interface functions are either relatively simple (counters, timers, serial-parallel conversion) or they are standardized (CAN-Bus, ISDN, Ethernet,...) or they are analog (ADC,...). It is, however, much more difficult to decide which processor core(s) to use, and whether one or more - possibly different - processor cores should be used, and if so, how to distribute the workload. Application specific coprocessors could be very cost effective, if they can be targeted to those often small parts of the software where most of the computation time is spent.

All these decisions need intricate knowledge of the system which a hardware designer usually does not have, and they must be re-evaluated in case of modifications. All this is not covered by the library approach.

So, in general, the designer will stay on the safe side and tend to overdesign processor performance, even in cost sensitive volume markets.

An overdesign can have a high impact on chip area, because the e.g. the difference between a 32bit RISC and a 16bit processor may be several 100k transistors including additional memory for increased instruction and program size. As an example, we have investigated an HDTV chromakey algorithm [Ri92], which is discussed in this paper.

So, hardware-software codesign can be more effective than logic synthesis. It would be sufficient if it could only be used for estimation.

A software oriented approach to hardware-software codesign

Our hardware-software codesign approach is based on the standard architecture consisting of a processor core, memory and customized hardware.

As many operations as possible are implemented in software running on the processor core. There are several reasons for this choice: high memory density, availability of optimally adapted compilers, careful verification and field test of standard cores, simpler software debugging and problems of hardware synthesis efficiency for larger applications, and last not least high flexibility in case of modifications.

External hardware is only generated when timing constraints are violated. Exceptions are basic and inexpensive I/O-functions, such as the standard processor interface (address bus, data bus and control signals) and serial and parallel I/O and except for user provided peripheral functions, such as an optimized field bus interface, e.g. selected from a library.

While timing constraints for interface control signals, such as Request and Acknowledge signals, concern a smaller part of the whole control task and, such, leave little architectural choice, other timing contraints are more global, such as control process cycle times, dead times, data sampling rates, and process intercommunication. These more global constraints concern extended code sections and give much choice which part of a function to implement in hardware. If multiple tasks are executed concurrently, one might even decide to move part of another task to a hardware function to save processor time for the critical task.

The problem is to analyze the software and to extract an appropriate part of the software for implementation in hardware, in order to meet timing contraints.

An advantage of this approach is the flexibility in function selection. At this point, we would like to be able to use libraries, synthesis tools and user experience. So, we would like to extract the following circuit types:

- Primitive structures at the interfaces.
  Examples: counters, timers, ...
  Currently, we assume synchronous circuit structures which can be implemented with a synthesis system. We further assume that interface functions which are asynchronous to the program flow are described as seperate processes. Moreover, the user should be able to provide the hardware structure.

- Coprocessors.
  Coprocessors should be small, such that they can be synthesized by high-level synthesis.

- Second core processor.
  If a coprocessor is not appropriate because there is no distinct critical software function or because it is too large, another (possibly different) standard core could be implemented.

In all three cases, the user must be able to support the extraction.

This extraction is a partitioning problem. Because analysis and extraction are done on the software functions, we call our approach "software oriented" hardware-software codesign.

Previous work in the area of hardware-software codesign has recently been published in [GM92]. It uses a similar system architecture. The input language is HardwareC, a very limited subset of C designed for hardware description, with integer as the only data type. The codesign starts with a configuration where all functions, except for program constructs with unbounded delay, are implemented as hardware modules. Only the rest is implemented as software on a core processor. Then, the design system tries to gradually move hardware functions to software regarding timing constraints and synchronism. The hardware is generated with the high-level synthesis system OLYMPUS. This is a *hardware oriented* approach. In contrast to our software oriented approach, only constructs which could initially be implemented in customized hardware can move to software, that means there are strong limitations to dynamic structures and system complexity.

Control system modeling

As mentioned in the introduction, C seems to emerge as an important language for programming embedded control systems. So, we chose C as input language to our design system. Because we understand that software development, software efficiency and verification are the dominant problems of control system design, we avoided to impose any limitations on the C language. By choosing suitable internal data representation and translation, we were able to support the full C language including dynamic data structures.

Because C has no notion of timing and no task concept, we added few language constructs for timing and task intercommunication. Timing can be defined as t_{min}, t_{max} and $t_{duration}$ (see e.g. [Da85]), where each timing contraint refers to two C-labels. Again, these extensions are not a result of hardware-software codesign but a necessity for the description of control tasks. Also, there is an *assertion* statement, e.g. to specify loop bounds for a precise run time analysis [PS91]. The syntax of these constructs is not significant and could be adapted to another real time-C dialect.

As required, the designer can provide hardware structures. The behavior must be described in a C function. This is well known from synthesis (e.g. [MKM90]). The same is possible the other way round, i.e. the designer may choose that a C-function must not be implemented in hardware to allow modifications until after hardware design.

The resulting superset of C is called C^*.

Design system structure

Fig. 1 shows the coarse design system structure without partitioning details.

The C^*-system description is parsed into a syntax graph CGL including all constraints. A data and control flow graph, as it is typically used for design representation in synthesis systems [FPC90], would not be sufficient to represent dynamic structures. For verification and debugging, there is a simulator for CGL format. Partitioning - which is described later - is performed on this graph on a C-statement basis. Those statements which shall be implemented in software are then translated to regular C. Unlike a data and control flow graph as in synthesis tools, the original program structure is kept throughout the partitioning process. So, a (hopefully) good and efficient program structure is preserved for the following compilation with a standard optimizing C-compiler. Currently, we are using the GNU compiler, which can generate code for a multitude of processor cores. Moreover, because the program structure is kept in the CGL, the user can understand the steps of the CGL-simulation and might even be able to evaluate the partitioning effects with a line-by-line comparison.

When a second core shall be implemented, code for this processor can be generated the same way.

For customized hardware implementation, we currently use the Stanford OLYMPUS system [MKM90]. It uses HardwareC as input language. So, the hardware which can be extracted is, at the moment, limited to the scope of operations which can be synthesized by the OLYMPUS system.

<u>Partitioning</u>

Partitioning requires to identify if and where timing constraints are violated. Run time analysis on the programming language level [PS91] is not precise enough because it does not reflect timing overhead as a result of hardware extraction. This will be described in a moment. Even assembly language level analysis (e.g.[PK90]) lacks sufficient precision. In pipeline architectures, e.g., performance can be seriously degraded by pipeline interlocks as shown with RT-level processor simulation of a control system benchmark [HEZ92]. While in a related project we are working on run time analysis regarding the processor architecture, the only method which provides sufficient precision today is extensive simulation (also used in [GM92]). Extensive simulation is very time consuming. At the moment, our only simulator is for SPARC-architectures. We have coupled this simulator to the Verilog simulation system.

A difficult problem of partitioning is overhead as a side effect of hardware extraction. The important effects are:

- Communication between processor core and peripheral hardware:

  - Communication time overhead:
    Additional I/O-operations of the processor core require additional time. Load-store architectures which are typical for RISC-controllers even require an extra instruction for each read and write operation. In an extreme case, the overall timing might be worse than before extraction.

  - Communication overhead:
    Besides obvious wiring overhead, communication can require buffers. Buffer size estimation is not always a simple problem.

- Interlocks:
  If variables are moved to a peripheral register, they might not (yet) be available by the time the software would be able to process them. This will lead to waiting times in the software.

- Compiler effects:
  When a program is fragmented by the extraction of statements, the efficiency of compiler optimization will change. Also, pipeline efficiency will be different. These effects are hard to predict. On the other hand, extraction on the assembler level is not usefull because the assembler code is based on processor details.

We concluded that the partioning which we want to use for hardware extraction needs iteration. Iteration through the whole design loop, however, would be too time consuming considering the time for compilation and simulation. So, a second, inner loop is added which is based on cost estimation.

The design system including partitioning loops is shown in fig. 2.

For cost estimation and determination of the next partitioning step in the inner loop, we might use closeness critieria, as defined in [LTh89]. Originally, these closeness criteria have been employed for clustering in synthesis. Out of these closeness criteria, three seem applicable:

- The *control closeness* of two operations or clusters a and b is the probability that the control flows from a to b or b to a.

- The *data closeness* of two operations or clusters measures the number of common variables relative to all variables of the operations or clusters.

- The *operator closeness* quantifies the similarity of operations of two clusters.

Control closeness can be derived from the syntax graph CGL; data closeness can be calculated using the C^*-compiler symbol table and a similar table can be generated for operator closeness determination.

The closeness criteria alone are not sufficient in our case. We give two examples. Suppose two program segments count some events. Because both increment and compare integer variables several times, the will have very high operator closeness. A designer, however, would hardly share hardware but would use two counters to avoid control and interconnection overhead. So, we might decide to put higher weight on data closeness and control closeness. High emphasis on control closeness and data closeness, however, would prevent the use of coprocessors, such as a floating point unit. Data closeness also overestimates communication costs, whenever a dual port memory can be used.

We might use closeness criteria for a first selection of candidates for partitioning and then look for patterns which give hints to clustering for coprocessors, second cores or interfaces. [GM92] gives a first citerion (external non-determinism) for interfaces. The definition of such patterns is one goal of our current work.

However, we are in a somewhat better position than synthesis, because synthesis has to implement every operation in hardware while in our case the design algorithm may pick a suitable part, which may be small compared to the possibly several ten thousand lines of control system code. So, rather than relying on closeness criteria, we are also looking for cost functions, which are taylored to a specific class of hardware, which we would like to extract. We will give a first example in the next section.

For the cost function, two more components are necessary, area overhead and timing improvement caused by additional hardware. Both must be estimated because synthesis is currently not in the partitioning loop. In particular, we have to regard scheduling in the synthesis system to estimate interlocks. Control and interconnect overhead estimation is difficult, but we hope for improved estimations from high-level synthesis research.

For the partitioning process (inner loop), we concentrate on stochastic algorithms. Stochastic algorithms allow arbitrary cost functions and iteration steps and allow a trade-off between run time and result quality. So, they are well suited to partitioning experiments, even if the relation of run time to quality might not be optimal. At the moment, we are implementing Simulated Annealing.

An experiment

The system is not completely running yet and we have not yet implemented an automatic partitioning, but we decided to try an example with some manual support to show the viability of our approach.

For this first experiment, we have chosen coprocessor extraction. For simplicity, processor and coprocessor are memory coupled with a CSP (coroutine) protocol. The simple hardware solution is shown in fig. 3. When the processor accesses a predefined address, a start signal is issued to the coprocessor and a HOLD signal to the processor core (SPARC: BHOLD: Hold signal, AOE: address bus enable, DOE: data bus enable). When finished, the coprocessor sends a DONE signal which ends the hold state. Core and coprocessor are clocked at the same speed.

We use a very simple cost function:

- for each operation, except for floating point, a constant speedup is assumed. This speedup is estimated to be equal to the number of function units in the coprocessor times the number of C-statement executions transferred to the coprocessor. This is a very rough estimation which must later on be corrected by a better scheduling estimation. It assumes that the execution of a statement takes the same time on coprocessor or processor if the coprocessor has a single functional unit, and that all coprocessor units will permanently be used. This cost function, however, will prefer loops, which are good candidates for coprocessors. Because it is so simple, it is well suited to a stochastic algorithm.

- communication costs are fixed due to the simple communication mechanism. For each variable to be transferred from memory to a coprocessor register, one memory cycle is required, and each switch between processor and coprocessor takes another cycle.

The simple cost function is then:

$$t = t_{proc} - t_{avg} * s * (1 + 1/n_F) + v,$$

where t_{proc} is the execution time when the system is executed exclusively on the processor, t_{avg} is the average execution time of a C statement, n_F is the number of functional units, s is the number of all C statements executed on the coprocessor regarding iterations, and v is the number of additional variable transfers necessary between coprocessor and memory and processor and memory.

In this experiment, area is not part of the cost function, but the user controls the number of functional units to be used by OLYMPUS.

As an example, we used a chromakey algorithm for HDTV studio equipment developed at another institute of our university [Ri92]. The desired response time is 1s. Using parameters, the precision can be reduced to meet the required response time. For the given precision, the algorithm needs 3.0s on a SPARC 1+. The program has 1400 lines of C code.

As we mentioned, the partitioning is not yet automatic. There is a loop of 34 lines, however, which is iterated 10070 times (fig. 4). This loop takes 90% of the total run time. Two other loops (C,D) are nested within this loop with 120 iterations each. There is a steep decrease in cost when extracting these two inner loops. So, we selected them and after translation to HardwareC, there where synthesized with OLYMPUS, choosing 2 ALUs. The RAM is accessed through an external port. Fig. 5 shows the resulting schedule. The 4 read operations load variables which are used more than once into registers in order to minimize memory access.

Without further interaction, OLYMPUS generated a circuit with 17.300 gate equivalents and 120ns cycle time using the LSI 1.5 ~~0.7~~µm library, corresponding to a coprocessor run time of 260.9us and a total loop run time of 2.63s. We used a gate level timing estimator in OLYMPUS because our current simulation environment would not have able for timing simulation of the overall configuration. By inserting a few drivers into high fanout lines, the minimum cycle time dropped to 20ns with 18.000 gate equivalents. This time was extended to the processor cycle time of 30ns, now reaching 65.2 ~~43.5~~us per coprocessor run and 0.66s for the whole loop. The total algorithm run time would now be approximately 1s (i.e. 33% of the original run time on a SPARC 1+) and thus meet the requirement at the much less cost than a second SPARC processor. A SPARC 10 class system might reach the same level of performance at much higher cost.

We tried other numbers of functional units and the extraction of a larger part of the system, but we had some problems with synthesis, which could not be fixed yet.

Conclusion and further steps

We have presented a system for hardware-software codesign of embedded control circuits consisting of a standard core processor and application specific on-chip hardware. Using a syntax graph for internal representation and hardware extraction on the statement level, no compromises in input language, semantics and programming style are necessary. The system starts with a pure software approach and tries to extract hardware until the timing requirements are met. The largest design space is available to coprocessor and second core extraction in conjunction with global time contraints. Side effects of hardware extraction require an iterative extraction process. A dual-loop approach seems to be necessary to remove the expensive run time evaluation from the primary extraction loop. Clustering criteria known from synthesis alone are not sufficient and cost functions which are taylored to a specific type of extracted hardware may be a promising approach. This will be one focus of our future research.

Except for the inner partitioning loop and the interface to the synthesis system OLYMPUS, the system is for the most part implemented and currently under test.

Acknowledgement

We would like to thank U. Holtmann and Th. Benner for some of the experimental work.

Literature

[Da85] B.Dasarathy. *Timing constraints of real-time systems: Constructs for expressing them, methods of validating them.* IEEE Trans. on Softw. Eng. , Jan. 1985, pp. 80-86.

[FPC90] M.C.McFarland, A.C.Parker, R. Camposano. *The high-level synthesis of digital systems.* IEEE Proc., Feb. 90, pp. 311-318.

[GM92] R.K.Gupta, G.D.Micheli. *System-level synthesis using re-programmable components.* Proc. EDAC 92, pp. 2-7.

[HEZ92] Cl.Hardewig, R.Ernst, H.Ch.Zeidler. *RISC-processors in real-time systems - a case study.* Submitted for publication.

[LTh89] E.D.Lagnese, D.E.Thomas. *Architecural partitioning for system level design.* Proc. DAC 89, pp.62-67.

[MKM90] G.D.DeMicheli, D.C.Ku, F.Mailhot et al.. *The OLYMPUS synthesis system for digital design.* IEEE Design&Test, Oct. 90, pp. 37-53.

[Ri92] Ch.Ricken. *Optimierung der automatischen Einpegelung eines HDTV-Chromakey-Mischers (in German).* Master Thesis, Technische Universität Braunschweig, 1992.

[PK90] R.Puschner, Ch.Koza. *Calculating the maximum execution time of real-time programs.* Journal of Real-time Systems, Kluwer Academics, 89, pp.159-176.

[PS91] C.Y.Park, A.C.Shaw. *Experiments with a program timing tool based on source- level timing schema.* IEEE Computer, May 91, pp. 48-57.

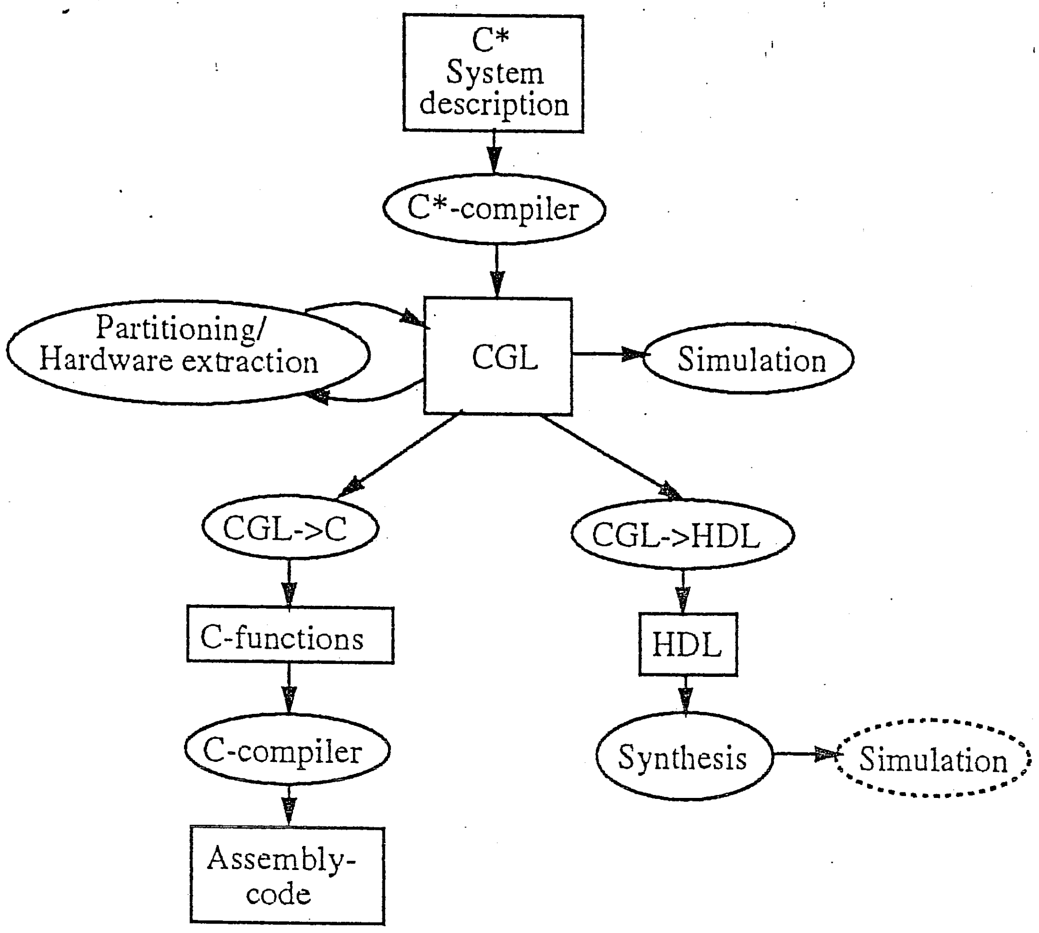

Fig.1: Coarse design system structure

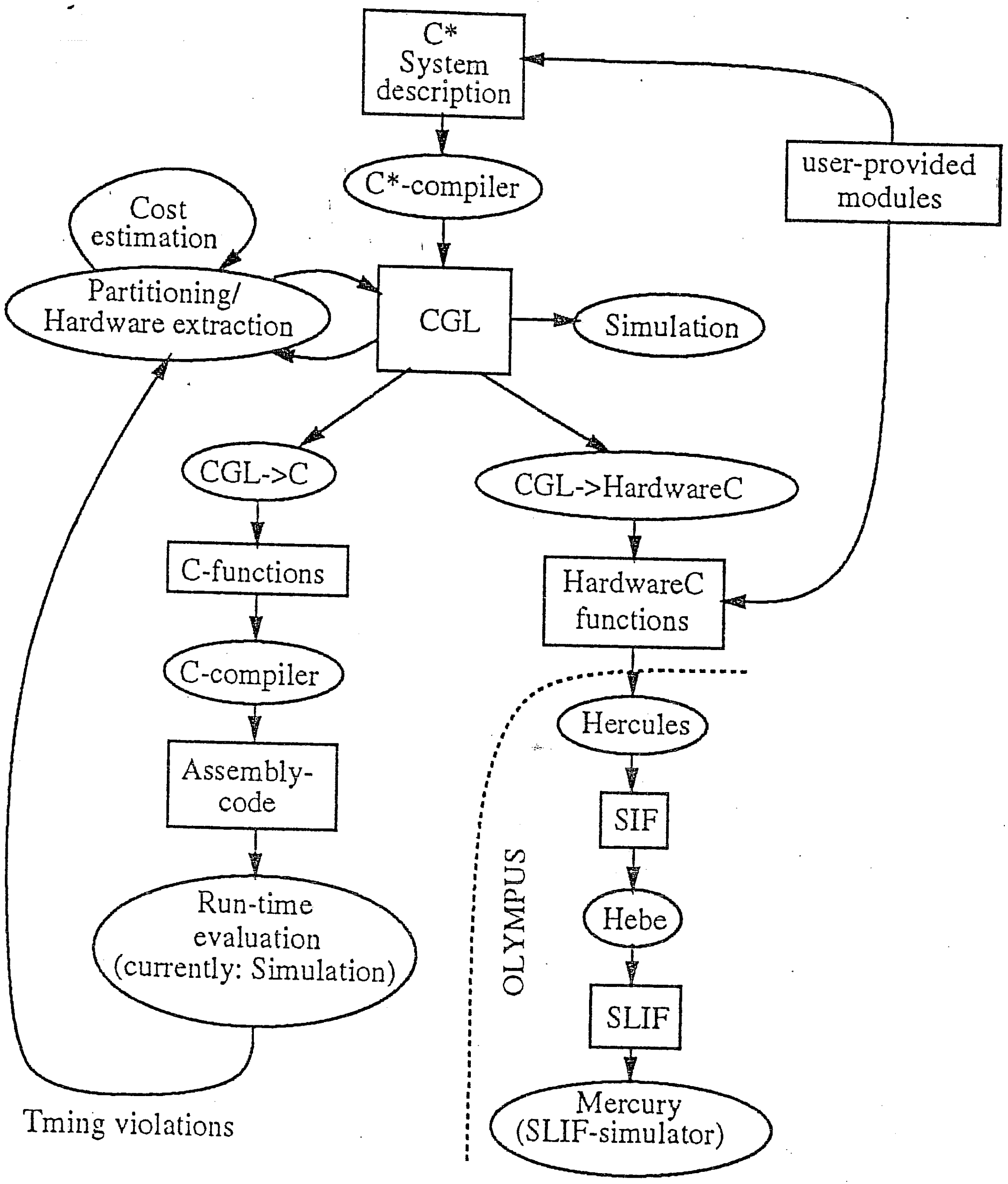

Fig.2: System for hardware software co-design

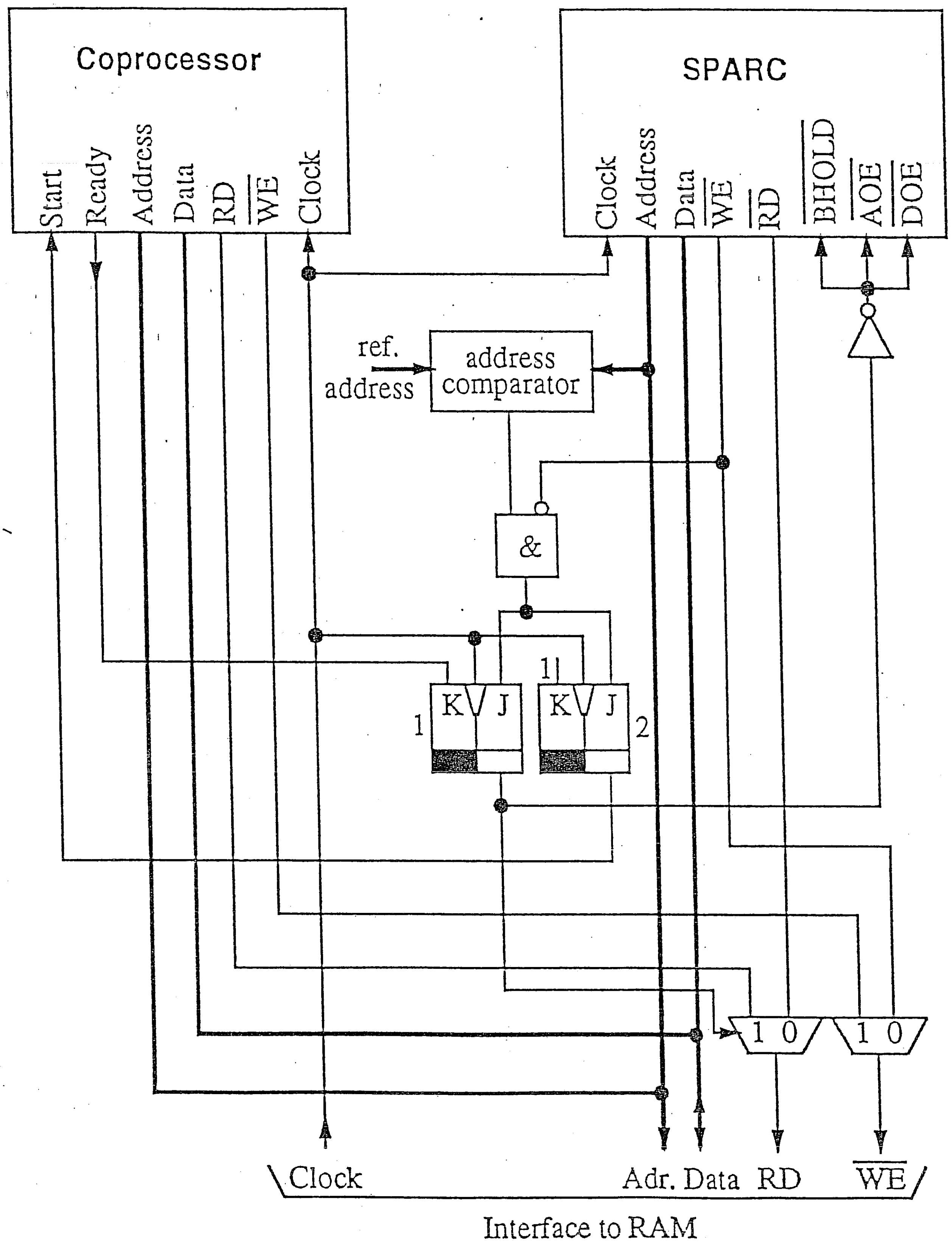

Fig. 3: Circuit for memory coupling

```
/* C description of parts 30c and 30d of blue-screen program "key"
*/
    while (cr <= cr2 + keyl) {
      cb = cbl + keyl;

      while (cb <= cb2 + keyl) {
        /*xx*/ MARK (130b); /*yy*/
        if (cb > vtab[cr]) {
          if (cb >= htab[cr])
            kt[cr][cb] = 255;
          else {
            iabsv = 512;    /* = 256+256 */

/* ============================ Bereich 30c ============================== */
   |
   |          for (i = crl; i <= cr2; i++) {
   |            ihilf = labs(cr - i) + labs(cb - vtab[i]);
   |            if (ihilf < iabsv) {
C  |              iabsv = ihilf;
   |            }
   |          }  /*of for i*/
   |
   |          iabsh = 512;    /* = 256+256 */
/* ============================ Bereich 30d ============================== */
   |
   |          for (i = crl; i <= cr2; i++) {
D  |            ihilf = labs(cr - i) + labs(cb - htab[i]);
   |            if (ihilf < iabsh) {
   |              iabsh = ihilf;
   |            }
   |          }  /*of for i*/
/* ======================================================================= */

            kt[cr][cb] = iabsv * 255 / (iabsv + iabsh);
          }  /*of else*/
        }
        FORLIM = min(cr + keyr, cr2);

        for (v = max(crl, cr - keyl); v <= FORLIM; v++) {
          FORLIM1 = min(cb + keyr, cb2);

          for (u = max(cbl, cb - keyl); u <= FORLIM1; u++) {
            kt[v][u] = kt[cr][cb];
          }

        }

        cb += keyf;
      }  /*of while cb <= cb2*/
```

Fig. 4: Loop in chromakey algorithm

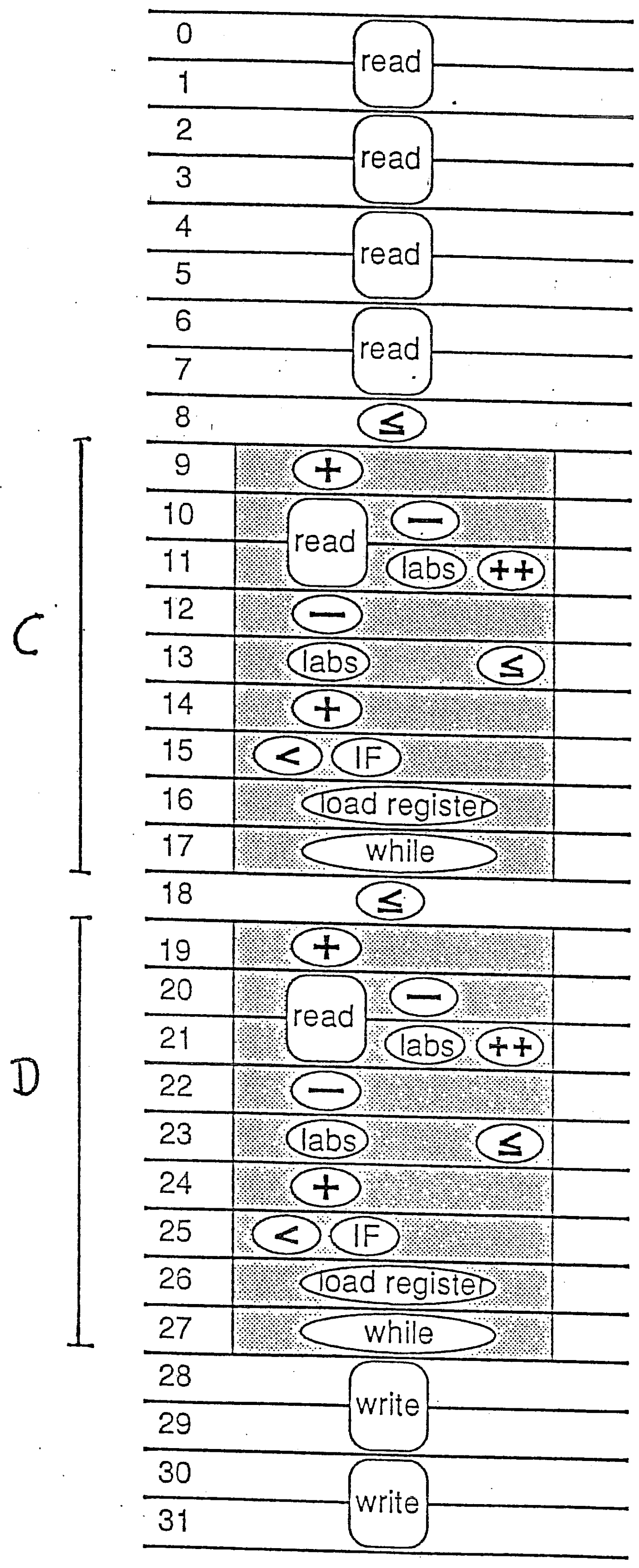

Fig. 5: Scheduled loops