Castle: An Interactive Environment for HW-SW Co-Design

Markus Theißinger, Paul Stravers and Holger Veit

GMD, Schloß Birlinghoven, St. Augustin, Germany

Abstract

We introduce CASTLE, a design environment for embedded systems. Starting from an algorithmic specification in C++/VHDL, CASTLE helps a designer to quickly find a suitable, cost-effective implementation of his system. The designer manually partitions the algorithmic specification into hardware and software components and refines the hardware architecture step by step. CASTLE provides immediate feed-back by displaying the feasibility and consequences of each partitioning decision. After partitioning, CASTLE automatically outputs the hardware and software components as VHDL and C++ programs. These can then be simulated to validate the design partitioning. Highlights of the CASTLE design environment include support for product maintenance, arbitrary hardware architectures and full design control by the designer.

1 Introduction

Embedded digital computers find their way more and more into a wide variety of industrial products. Their application ranges from automobiles to television sets, from navigation equipment in pleasure yachts to simple controllers in a washing machine. The production volume of embedded processors is huge. Currently, about 2000 million microprocessors are sold yearly, of which only 2 % find their way into a PC of some kind. The rest is for embedded systems.

An important characteristic of embedded systems is that they operate in a real-time environment. This environment often imposes a limited response time on the embedded system. On the other hand, the environment often dictates low power consumption, a wide operating temperature range, small sizes, etc.

Another important observation is that the commercial success of an embedded product is much more likely when the product idea is quickly implemented in a cost-effective way. After its initial implementation, it must be possible to maintain the design and adapt it to the evolving operating environment and market demands.

Given this context, there clearly is a need for a design environment to assist a designer with the transformation of an algorithmic specification into a suitable implementation. Depending on the operating environment of the embedded system and the algorithmic specification, implementations can range from a simple single chip microprocessor to a high-end multiprocessor configuration. In both cases it may be necessary to add application specific hardware to meet performance constraints.

In the remaining part of the paper we present CASTLE, our codesign environment. In section 2 general requirements on a design environment are discussed. Section 3 describes the design flow to transform an algorithmic specification into a cost effective implementation. Section 4 then applies this theory to a real-world example, the *Gzip* program for data compression. Finally, section 5 summarizes conclusions and indicates future directions.

2 Design environment requirements

A design environment for embedded systems should have at least the following properties:

1 *it starts with an algorithmic description in a high-level language.* This enables quick execution and debugging of the algorithm before any attempt at implementation is made. A high-level specification is also crucial for successful maintenance of the embedded system and helps documenting it.

2 *it handles a wide variety of hardware architectures.* The designer must be given room for experiments with various hardware configurations. This not only includes exchanging one microprocessor type with another type, but it also includes changing the overall hardware architecture. For instance, the designer should be able to compare an implementation with a single powerful processor to an implementation with three weaker processors communicating through shared memory.

3 *it is interactive and leaves control in the hand of the designer.* The task of implementing an algorithmic specification is too complex and it involves too many (non-technical) considerations than that it can be left to a computer program, however sophisticated. Of course, computer programs are extremely useful to perform many subtasks in the design flow,

This work has been sponsored by the Bundesministerium für Forschung und Technologie BMFT, project 01M2897A SYDIS.

0-8186-6315-4/94 $04.00

such as analysis of the decisions made by a human designer. A computer program may even recommend certain components and a hardware—software partitioning, as long as the designer has the last word on it.

4 *it presents the designer relevant statistical data about his algorithm.* Most of this data can be obtained from profiling sessions, where the algorithm is run with typical input data to identify functions and operations that are most heavily used.

5 *it presents the designer relevant data about the (partial completed) implementation.* Estimations of production cost, power consumption, weight and size and operating temperature range are important data when the designer is deciding how to proceed. Also data regarding the communication overhead between hardware and software modules must be given to the designer. Other relevant data may include an address trace of one or more processors that the designer can feed into a cache simulator to see if the implementation would benefit from a cache or not.

6 *it is capable of reusing hardware and software components from a library.* In industrial environments there is a strong tendency to reuse proven concepts and components. This is actually another reason why the design environment should accept the hardware architecture that the designer imposes, not the other way around.

7 *it supports maintenance of the implementation.* This means that the design environment must be able to relate changes in the algorithmic specification to possible changes in the implementation. It should not be necessary for the designer to re-implement his algorithm from scratch while the algorithm only changes gradually.

The CASTLE design environment presented in the next section was designed to comply with the properties listed above. Comparing these properties to related work by other authors [SB91,EH93,BR92,GC92] the properties 2, 5 and 7 have not been reported before and are therefore considered the highlights of our system.

3 Design flow and methodology

Figure 1 shows the general design flow with CASTLE (*C*odesign *A*nd *S*ynthesis *T*oo*L* *E*nvironment). A designer models his system by an algorithmic description. Our codesign environment accepts C++ and VHDL or a mix of both languages. The algorithmic description can be a manually written text file, or an output of a code generator or CASE tool. The latter could provide a limited validation and rule checking in advance.

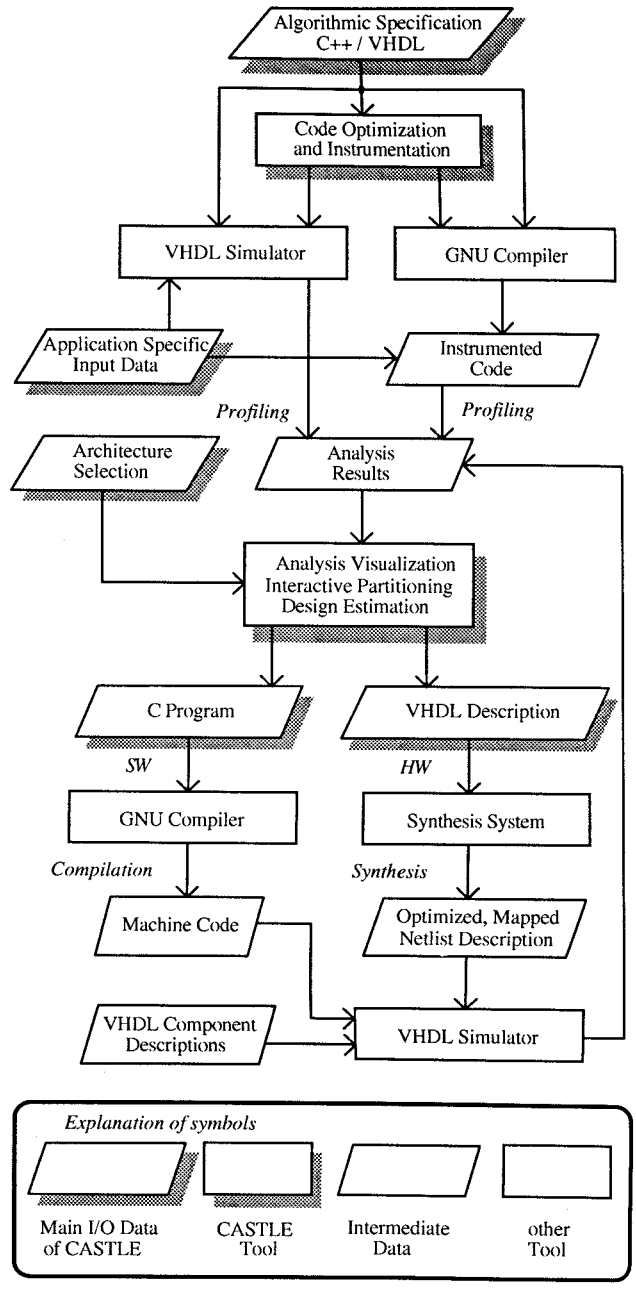

Fig. 1: Design flow.

3.1 Design analysis

The first design step with CASTLE is to analyze the given algorithmic description. This can be done statically or, more accurately, dynamically, i.e. profiling the application of the algorithm to selected input data.

Supplied with profiling results a designer can concentrate on the most promising design pieces where optimization gains the highest speed up. Profiling is especially useful if the designer did not write the original

specification but reuses one, developed by another designer.

Important profiling points are functions and basic blocks. A function is a mean of the designer to group operations. A basic block is a straight-line single-entry sequence of instructions with no branch except at the end.

Profiling basic blocks is important because all operations of a basic block are executed as many times as the block itself, thus with the execution frequency of all basic blocks known, the execution frequency of all operations of a program is known. Sometimes it is advantageous not only to determine how often a basic block is executed but also from where the block is activated, i.e. to obtain traces of branch directions taken [LB94]. We call this *block trace* profiling.

CASTLE includes a modified version of GNU gcc 2.4.5 to collect block trace profile data. Where GNU gcc is not sufficient, e.g. in a mixed language description, a special tool of CASTLE can instrument the algorithmic description by adding profiling code into the C++ or VHDL source text.

The instrumented code is simulated/executed with data the designer believes to be relevant for his specific application. All parts of the design should be activated several times to get meaningful statistics. The designer should already have such a suitable set of test data in order to validate his design.

Since profiling execution is already some kind of simulation and thus potentially time consuming, we recommend starting with an algorithmic description in C++ because this code may be executed on any fast computer. This is adequate if no absolute timing information but only execution frequencies are needed. Gathering profiling information quickly allows a designer to test larger sets of input data, as compared to simulation, which reduces the risk to miss some important program behavior.

In contrast to dynamic design behavior CASTLE also analyses a design statically and obtains characteristics like:

- Dependency between operations.
- Distribution of operations in functions and basic blocks.
- Communication statistics for calling functions e.g. amount of data transferred between functions.
- Amount of data transferred between operations.
- Existence of complex data addressing (pointers and arrays).

Static and dynamic information is continuously available during the partitioning process.

3.2 Interactive partitioning

A designer implicitly writes his algorithm with a particular target structure in mind. Automatic partitioning algorithms would either need to detect and recover such implicit structures, or the designer would have to provide additional information to guide the automatic algorithm. Therefore, it is adequate that the designer does the partitioning himself, with a design system being his assistant.

From a list of architectures, the designer selects his preferred target architecture. Known architectures for instance are:

- Processor and additional data path logic, e.g. multiply/divide circuit.
- Processor and intelligent coprocessor, e.g. 80386 CPU with 80387 arithmetic coprocessor.
- Processors and application specific coprocessor.

The architectural library also contains several common communication and synchronization mechanisms, such as dual-port shared memory, FIFOs or single-staged I/O ports with hand shaking.

For the partitioning step CASTLE displays the list of functions of the algorithmic specification and allows the designer to interactively assign each function to hardware or software execution, this is called *component assignment.* Functions can be sorted by estimated execution time. Additional information displayed, includes a leaf-function tag and communication requirements, e.g. what types of variables need to be transferred from one calling function to the called function and back again.

A non-leaf function calls another function. If the former is to be implemented in hardware but the latter is to be implemented in software additional circuitry is necessary to allow the hardware to activate the main processor to execute the software. The leaf-function tag informs the designer about such cases.

An alternative to component assignment is *component flattening* which merges a called function into the calling function. This modifies the topology of the algorithmic description and allows the designer to easily choose a more feasible function structure for his particular subprogram.

During partitioning the designer refines the architecture step by step, e.g. by selecting cost of operations and restricting hardware components. After each partitioning decision, the system estimates the consequences. A master-slave architecture like in [EH93], for instance, is more expensive in terms of communication than a dual-ported shared memory, so the first architecture is efficient if the amount of data to be passed is small. If the number of arithmetic operations is high for a particular function, it could be worthwhile to assign a hardware unit to it. CASTLE then would show an estimation of the gain. There can be also a loss in overall performance if the designer erroneously chooses to assign complex addressing modes like pointer accesses into hardware, because it might

require implementing the whole addressing mechanism of a common CPU in a hardware module.

Eventually the designer has found a satisfying structure. Then CASTLE will perform the actual partitioning of his design into hardware and software parts. CASTLE collects the various parts of the design that were tagged as 'hardware' and generates a set of VHDL modules from that. The interfacing is already determined by the architecture template that was chosen in the beginning (e.g. processor and additional data path). The remaining parts become the software of the design. This is actually a skeleton of the original design, reduced by the parts that have been assigned to hardware now, but modified by the insertion of code to manage the interfacing with the hardware modules (like I/O port handling and synchronization mechanisms). The interface code is taken from a special software library that is tailored to the requirements of the architectural template.

For maintenance support CASTLE allows the designer to modify a previously created implementation. It compares the modified algorithmic specification to the original specification and identifies the subprograms that in any case must be re-implemented, either because they are new or because they are modified. CASTLE then presents the modified list of functions to the designer who then interactively proceeds with the implementation as described above.

An embedded system often has restricted memory for storing the software part. Therefore subprograms may be mapped to absolute memory addresses during compilation. For maintenance of such a system a special compiler/linker is needed which can preserve previous subroutine entry points when rewriting portions of code.

4 Co-design example

This section presents an example of the assistance provided by CASTLE. Here a designer reuses an existing specification formerly unknown to him and wants to improve its performance. With the aid of CASTLEs analysis tools he can immediately focus on the most important design parts and optimize them.

We choose the file compression program Gzip version 1.2.4 from Jean Gailly to demonstrate a typical design flow using the CASTLE system. Gzip is written in C and is freely distributed under the terms of the GNU General Public License. It uses the Lempel-Ziv algorithm [ZL77,We84] to compress files of arbitrary data content.

Gzip is intended to be used in a backup server connected to a network. Data from the network to the backup server should be transferred and compressed with an average rate of about 100 KB / second. Data from the server to the network is automatically decompressed which can be done faster than compression, hence we concentrate on compression only.

4.1 Gzip analysis

Gzip allows to trade compression quality for compression calculation time. We, as a designer, decide to optimize the design for highest compression quality (Gzip runtime option -9). First we select relevant input data to profile Gzip. Our selection consists of the C-source files of the Gzip program itself, which are ASCII text files, and the Gzip executable binary file generated by gcc -pg. Tab. 1 characterizes this data.

File type	File size [KB]
ASCII C program files	226
Binary executable files	192
Total	418

Tab. 1: Input data selected for profiling.

Basic block profiling is done on a SPARCStation 10 with the modified GNU C compiler. Some function call frequencies are displayed in Tab. 2. Fig. 2 shows the ten most frequently executed basic blocks and Fig. 3 presents the innermost loop of Gzip.

One profiling run to obtain all function, basic block and block trace execution frequencies requires 82 seconds, so we could easily try some other sets of input data to profile Gzip and see how this influences profiling results.

The global statistic output of CASTLE is useful for

Function name	Call frequency
longest_match	128087
updcrc	34
deflate	15
ct_tally	137009

Tab. 2: Call frequency of some selected functions.

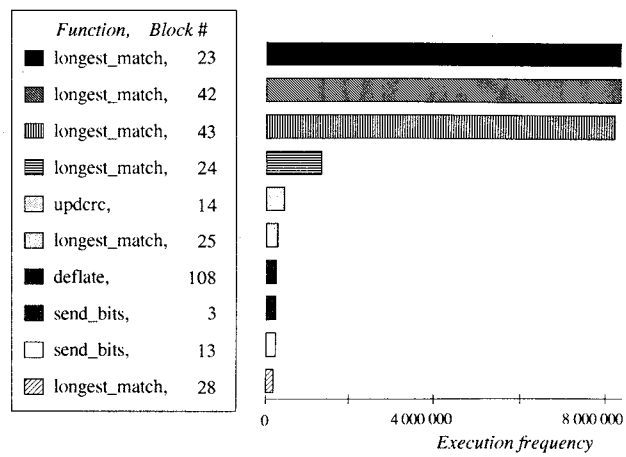

Fig. 2: Top 10 execution frequencies of basic blocks.

```
Exe. freq.    Gzip program text                                   Block #
          do {
            ...
8357608   • match = window + cur_match;                            23
            ...
8357608   • if (match[best_len]     != scan_end    ||              23
1326392     match[best_len-1]   != scan_end1   ||                  24
 288050     *match              != *scan       ||                  25
 137870     *++match            != scan[1] ) continue;             26
              .
              .
              .
8357541   • } while((cur_match =
                        prev[cur_match & WMASK]) > limit           42
8229758   •       && --chain_length != 0);                         43
```

Fig. 3: Innermost loop of Gzip.

Operation	Cost
load mem	2
store mem	3
move reg	1
add, sub, cmp	1
and	1
shift	1
branch conditional	1
call	1
return	1
coprocessor load/store mem	2

Tab. 3: Basic costs selected for architecture.

judging the quality of our input data selection: Gzip consists of 1291 basic blocks. 550 of these are activated at least once with the given input data but only 96 blocks are executed more than 10000 times. 727 different transitions between basic blocks are taken but only 125 transitions are taken more than 10000 times.

From the profiling results we see

- The function longest_match of Gzip contains the most frequently executed basic blocks and thus is likely to consume the largest amount of computation time.
- Longest_match and ct_tally are functions that are called very frequently while the functions updcrc and deflate are called very rarely. This gives us a hint on communication needs between functions.

We can not conclude the true cost of a function in terms of execution time solely from the execution frequencies. In order to estimate the true function cost we have to select a cost for every operation appearing in the function. Assigning costs to operations is already part of the architecture template selection of CASTLE. We choose

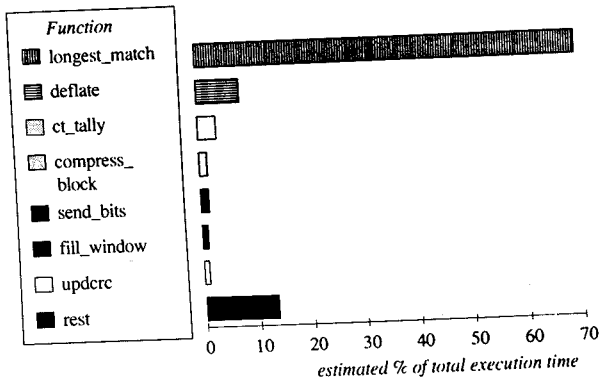

Fig. 4: Cost distribution of functions of Gzip.

operation execution costs from [Cy90] which specify the number of cycles needed on a SPARC processor (Tab. 3).

With these selected costs CASTLE presents us function timing estimations as shown in Fig. 4 that allow to compare the relative importance of each function. The execution cost c_f of a function f is calculated by summing the costs of all basic blocks of f.

$$c_f = \sum_i \left(n_{f,i} \cdot c_{f,i} \right)$$

where $n_{f,i}$ is the number of executions of basic block i of f and $c_{f,i}$ is its execution cost.

As the operation costs here are given in execution cycles on a SPARC, we can estimate the absolute execution time by multiplying the cost with the length of one processor cycle. In our example we get an estimated execution time of 6.6 seconds. This estimation compares well to the measured runtime on a SPARCStation 10 which is 6.5 seconds and therefore differs by only 1.5 %. However the measured runtime is not considered to be accurate because effects of task switching and network access are included. Without these factors the estimation could differ by up to 20 %.

4.2 Architecture refinement steps

Compressing 418 KB of input data in 6.5 seconds gives a data rate of 64 KB / second, which does not fulfill our goal of 100 KB / second, hence we will try to improve this rate by using special purpose hardware.

The functions longest_match and deflate seem to be the most promising candidates for a hardware implementation because longest_match consumes 69 % of the total execution time while deflate consumes 8 %. For brevity we concentrate on longest_match only.

Now we have to refine our architecture template. We choose to implement our system as a single processor, in our case a SPARC, with an additional application specific coprocessor. Processor and coprocessor will communicate

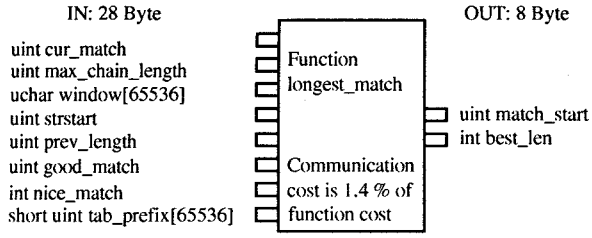

Fig. 5: Communication requirements of function longest_match.

through shared memory. The refinement enables us to define costs for communication operations of processor and coprocessor (load/store operations in Tab. 3) which CASTLE uses to estimate communication overhead.

We try to move the function longest_match into the coprocessor. Longest_match requires to transfer 36 bytes between processor and coprocessor upon each call (Fig. 5). The relative communication cost is calculated as:

$$c_t / c_f$$

where c_t is the execution time cost of all data transfer needed when applying Gzip to the sample input data. c_t is calculated as the product of the number of data units to transfer times the cost to transfer one unit. Here one data unit is 32 bit and the cost for one transfer is taken from Tab. 3.

The next step is to design the coprocessor. The coprocessor will consists of several functional units which are specially suited to execute function longest_match. First we take a look at the distribution of operation costs for function longest_match (Fig. 6). The cost of ALU operations (add, sub, cmp, and, shift) is 1.8 times the cost of load-store operations. This suggests that our coprocessor should have twice as much ALUs as load-store units.

Additionally we consider the operations of the most demanding basic blocks (Fig. 3 and Fig. 7). Now we

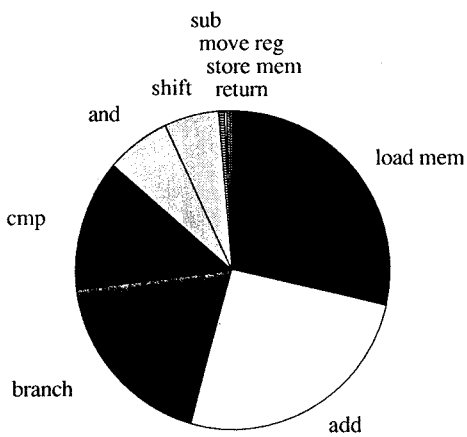

Fig. 6: Cost distribution of operations in function longest_match.

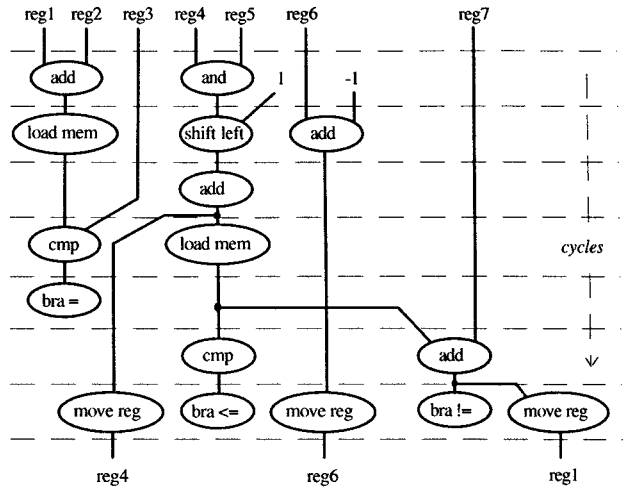

Fig. 7: Schedule of the three most demanding basic blocks.

specify resource restrictions for functional units in the coprocessor. This specification further refines our architecture template. We use two ALUs and a register file with 20 registers of 32 bit each. CASTLE schedules the operations of longest_match using a list scheduling algorithm for the basic blocks. Operations of a sequence of the most demanding blocks are merged into one block and are also scheduled using list scheduling. This block is shown in Fig. 7. The original dependencies between operations are preserved using register renaming. In the last cycle shown in Fig. 7 the renamed registers are copied to their true destinations (move reg instructions).

With the new scheduled operations and the profiled block transition frequencies CASTLE estimates the execution time costs for longest_match. As the list scheduling does not move operations across block boundaries the set of operations in a block remains the same. Execution time saving for block i of function f is calculated by

$$(c_{f,i} - c_{l,i}) \cdot n_{f,i}$$

where $n_{f,i}$ is the number of executions of basic block i of f and $c_{f,i}$ is its execution cost on the processor while $c_{l,i}$ is its execution cost on the coprocessor depending on the list scheduling. For the merged blocks, estimation is more difficult as the execution frequency of the sequence of blocks is not directly known from profiling. Block trace profiling determines only single transitions but not longer sequences. Estimation simply uses the minimum of each single transition as the execution frequency of the sequence. This overestimates cost savings but will be sensible for the most important inner loops of a program.

In our example the estimated speed up of longest_match is 2 (150,788,439 clock cycles for a software solution compared to 75,810,539 cycles for a hardware solution). This will improve the complete Gzip

algorithm by a factor of 1.4 but will not quite reach our desired data rate of 100 KB / second. We examine the schedule of Fig. 7 and define that one of our ALUs can execute the 'shift' and the following 'add' operation in one cycle. This shortens the critical path by one cycle. Now CASTLE estimates the speed up of longest_match to be 2.23 and the speed up of Gzip to be 1.6, which yields the desired data rate of 100 KB / second. We then tell CASTLE to output this configuration in hardware and software and simulate it to verify the estimations.

4.3 Gzip coprocessor

The generated description of the datapath of the coprocessor consist of 370 lines of VHDL. We used Synopsys Design Compiler version 3.1a to synthesize the necessary components. Statistics are shown in Tab. 4. The coprocessor datapath is shown in Fig. 8. Synthesis of the datapath including all components on a SPARCStation 10 took 23 hours and used 120 MB swap space and 80 MB RAM. The datapath is controlled by a sequencer which has 200 words of microcode, with 45 Bits each word. To allow reconfiguration of the coprocessor, the microcode can be stored in a fast RAM inside the coprocessor. This RAM is taken from a component library and not synthesised. If reconfiguration is unnecessary the microcode is stored in ROM instead.

Components	Area/NAND gate equiv.	Delay/ns
ALU1	1037	13.2
ALU2	766	7.4
Mux	160	0.8
RegFile	14122	2.4
LdStore	527	2.4
tot. datapath	18147	18.2

Tab. 4: Datapath statistics.

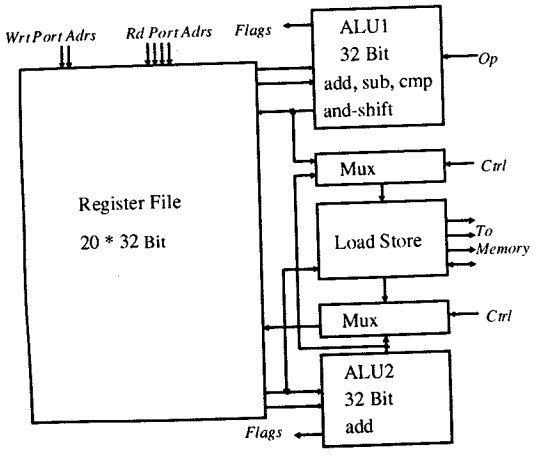

Fig. 8: Datapath of coprocessor.

One important factor of the selected shared memory architecture is that the coprocessor accesses memory at the same speed as the main processor, that is, both either use the same cache or the processor does bus snooping to be aware of memory changed by the coprocessor. Without this, coprocessor memory access cost would increase too much to gain a considerable speed up.

5 Conclusion and Future Work

The CASTLE design environment helps the user to quickly find a suitable, cost-effective implementation for an algorithm. Compared to other design environments, CASTLE aims at several new goals. It handles different hardware architectures, it provides detailed information about the implementation at any stage during the design, and once the design is ready it can maintain the implementation.

Currently the CASTLE environment is still under construction. Future research aims include tools to suggest hardware—software partitioning schemes to the designer. Several algorithms to automatically find such partitioning have already been presented, for example [EH93], [BR92], [GC92]. Further research must show how these algorithms compare to each other and how they can provide helpful information to the user.

References

[BR92] E. Barros and W. Rosenstiel, "A Method for Hardware/Software Partitioning", Proceedings of the Third European Conference on Design Automation, 1992.

[Cy90] Cypress Semiconductor, Ross Technology Subsidiary, "Sparc Manual", *Sparc Risc User's Guide*, p. 2-1—2-88, February 1990.

[EH93] Rolf Ernst, Jörg Henkel, Thomas Benner, "Hardware-Software Cosynthesis for Microcontrollers", *IEEE Design & Test of Computers*, pp. 64-75, December 1993.

[GC92] R.K. Gupta and C.N. Coelho Jr. and G. De Micheli, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components", Proceedings of the 29th Design Automation Conference, 1992.

[LB94] James R. Larus, Thomas Ball, "Rewriting Executable Files to Measure Program Behavior", *Software — Practice And Experience, Vol. 24(2)*, pp. 197-218, February 1994.

[SB91] M.B. Srivastava and R.W. Brodersen, "Rapid Prototyping of Hardware and Software in a Unified Framework ", *Proceedings of the International Conference on Computer Aided Design*, pp. 152-155, 1991.

[We84] Terry. A. Welch, "A Technique for High-Performance Data Compression", *IEEE Computer*, pp. 8-19, June 1984.

[ZL77] J. Ziv, A. Lempel, "A Universal Algorithm for Sequential Data Compression", *IEEE Transactions on Information Theory, Vol 23, No. 3*, pp. 337-343, May 1977.