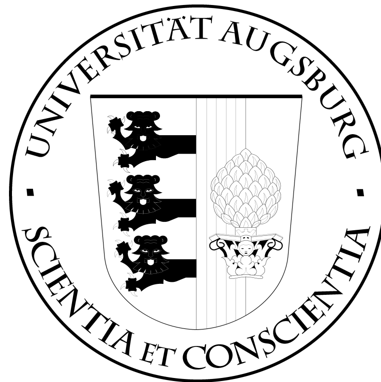
Universität Augsburg



Reconfigurable Extension for CarCore

Stefan Maier

Report 2005-17 September 2005

Institut für Informatik
D-86135 Augsburg

Copyright © Stefan Maier
Institut für Informatik
Universität Augsburg
D–86135 Augsburg, Germany
http://www.Informatik.Uni-Augsburg.DE
— all rights reserved —

Reconfigurable Extension for CarCore

Stefan Maier

University of Augsburg
Institute of Computer Science
Eichleitnerstr. 30, 86159 Augsburg, Germany

Abstract

Reconfigurable processing units promise to boost the performance of computational intensive applications. Thereby the kernel of an application is executed in the reconfigurable hardware instead of being executed in software. This report presents a reconfigurable extension for the CarCore embedded processor. The extension combines reconfigurability with simultaneous multithreading and real-time capabilities of the CarCore. The prototype implementation in SystemC shows an overall application speedup of 4 compared to a single threaded pure software execution.

1 Introduction

This report describes a reconfigurable extension for the simultaneous multithreaded CarCore microcontroller and its implementation in SystemC [5]. The CarCore is derived from the Infineon TriCore 1 and obtains binary compatibility to it. The CarCore features hardware based scheduling of four threads. The threads are isolated from each other and one thread cannot influence the runtime behavior of the other threads. The processor is designed to deal with real-time threads and the timing behavior of the threads should be predictable. A performance gain is reached by using the latencies of highly prioritized threads for execution of instructions from other threads.

To further increase performance integrating reconfigurable hardware is considered to be a feasible solution. A reconfigurable unit speeds up computational intensive parts of an application by executing them in hardware. The reconfigurable extension for the CarCore described in this report is similar to the MOLEN Reconfigurable Processor proposed in [6].

Section 2 gives a short introduction to the preconditions of the work. In particular the architectures of the CarCore embedded processor and the MOLEN polymorphic processor are described. Section 3 describes the CarCore reconfigurable extension in detail while the last section presents the evaluation results.

2 Preconditions

Two existing processor architectures are the preconditions of the reconfigurable extension described in this work. The CarCore simultaneous multithreaded processor itself serves as basis for the reconfigurable extension and was developed during a diploma thesis at the University of Augsburg. The MOLEN Polymorphic Processor was designed at the Delft University of Technology as a general purpose processor with a reconfigurable processor attached.

2.1 CarCore

The CarCore [5] consists of a five stage pipeline containing the stages instruction fetch, schedule, decode, execute and write back. Similar to the TriCore 1 the pipeline is divided into an address and a data pipeline where the fetch and schedule stage are common for both pipelines. Therefore two

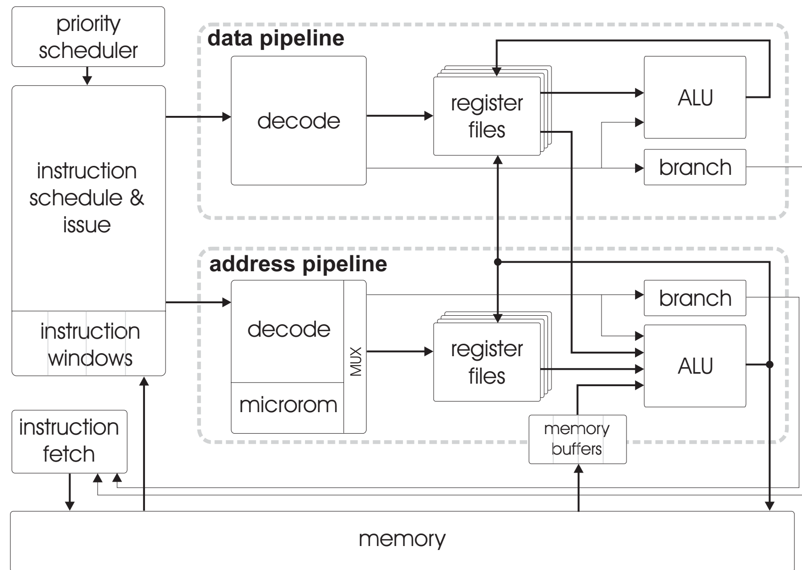


Figure 1: Block Diagram of CarCore.

instructions can be executed in parallel. As the CarCore features simultaneous multithreading the two instructions may belong to the same thread or to different threads. Figure 1 shows the block diagram of the CarCore. Each pipeline contains one register set per thread (overall 8 register files) and its own ALU and branch unit. Besides the register sets other parts of the processor context are duplicated for each thread, e.g. program counter, status and core registers.

The fetch unit maintains the four program counters and four instruction windows decouple the fetch stage from the rest of the pipeline. The scheduler keeps track of the thread priorities and issues instructions to the data and to the address pipeline. Some complex instructions of the TriCore instruction set, i.e. CALL and RET, need several cycles to be executed. If such a complex instruction would block the pipeline, the predictability of the timing behavior of the other threads would be destroyed. Therefore these instructions are executed as interruptible sequences of microinstructions. The microcode unit maintains a microinstruction pointer for each thread and hence allows to interrupt microprograms and to interleave microinstructions with assembler instructions from different threads.

Every clock cycle the scheduler chooses two instructions from the instruction windows and issues them to the pipelines. Thereby it follows the priorities of the threads wich are given by the priority unit. Two priority schemes are supported. The fixed priority scheme associates fixed priorities to the hardware threads, i.e. thread 0 has always highest, thread 3 lowest priority. This scheme prefers one thread which can be considered as a real-time thread. The round-robin scheme circularly changes the thread priorities and therefore equally disposes processing time.

The scheduling works as follows: First an instruction taken from an unblocked thread with a priority as high as possible is issued to the integer pipeline. Second an operation in the address pipeline is scheduled. This operation may be a microinstruction to start or continue a microprogram or to issue an instruction out of the instruction windows to the decoder of the address pipeline. Thereby the scheduler prefers operations of highly prioritized threads. According to the fill level of the instruction windows and the priorities of the threads the scheduler selects a thread to fetch a code block from memory in the next cycle.

Though memory accesses may last several cycles the address pipeline must not be blocked by one thread and potentially disturb the runtime behavior of other threads. To address this problem the CarCore executes load accesses as split-phase loads. The scheduler separates address generation and data write-back in two operations. Valid data from memory is stored in a memory buffer until the scheduler explicitly inserts the data into the register file.

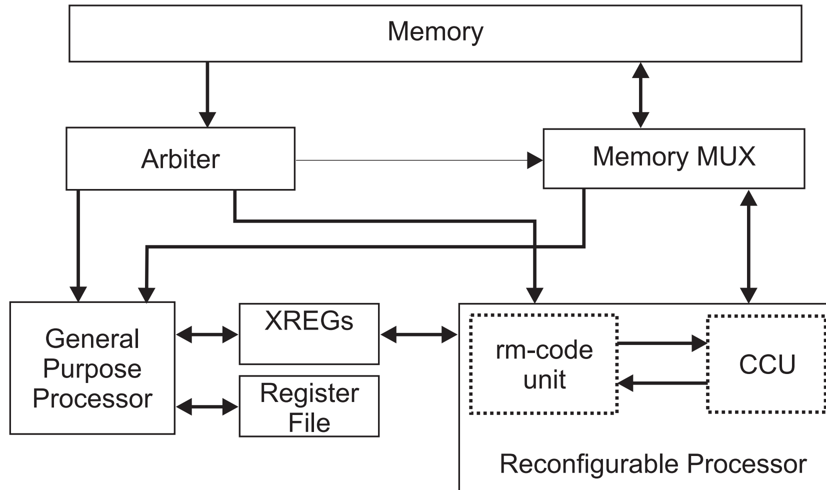


Figure 2: The Molen Organization.

2.2 MOLEN Polymorphic Processor

The MOLEN Polymorphic Processor [6] combines a general purpose processor with a reconfigurable custom computing unit. The reconfigurable hardware allows the designer of an application to extend the processor's functionality at design time and to speed up the application by executing parts of the application in hardware. Figure 2 shows the Molen architecture with the general purpose processor (GPP) and the reconfigurable processor (RP). The Arbiter predecodes and issues the instructions to either of the processors. The exchange registers (XREGs) allow the exchange of data between the GPP and RP. The reconfigurable processor itself mainly contains the reconfigurable microcode unit ($\rho\mu$-code unit) and a custom configurable unit (CCU). To use the functionality implemented in the CCU a general one-time extension of the instruction set is necessary. The basic operations of the RP are set and execute. The set instructions configure the CCU and the execute instruction performs a computation on the CCU. All together eight instructions are proposed to extend the instruction set. Six of them are controlling the reconfigurable hardware:

- The **partial set** (pset) instruction configures the hardware for the common part of multiple functions whereas **complete set** (cset) configures the remaining blocks of the CCU, not set by the pset instruction.
- The **execute** instruction starts a microcode on the CCU configured by pset or cset.
- The **set prefetch** and **execute prefetch** instructions prefetch a microcode needed by succeeding set or execute instructions.
- For synchronization of the core processor with the reconfigurable processor the **break** instruction is introduced.

Two further instructions are used for accessing the XREGs from the pipeline of the core processor:

- **movtx** Xa, Rb copies the value of the general purpose register (GPR) Rb to the exchange register Xa.
- **movfx** Ra, Xb moves the value from exchange register Xb to GPR Ra.

As not all instructions are required for different implementations of reconfigurable processors, the minimal instruction set extension covers only cset, execute, movfx and movtx. Set and execute instructions (including the prefetch versions) are using pointers to *reconfigurable microcode*

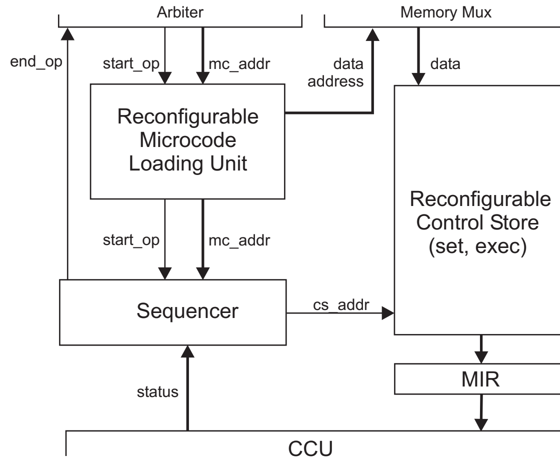


Figure 3: Molen Reconfigurable Microcode Unit.

as operands which are emulated by the $\rho\mu$-code unit. Therefore microcode controlling the configuration of the CCU and microcode executing on the CCU is distinguished. Figure 3 shows the internal organization of the $\rho\mu$-code unit. To execute a set or exec instruction the $\rho\mu$-code unit checks its residence table if the microcode specified by the address is already loaded into the corresponding control store. The control stores are separated into a set and an execute control store holding the microcodes for set and execute respectively. If the microcode is not available the $\rho\mu$-code loading unit loads the microcode into the control store which is acting like a cache. For performance reasons some of the frequently used entries in the control stores can be marked as fixed and therefore cannot be overwritten by other microcodes. To distinguish between fixed and pageable microcodes one bit of the address field is used. After the microcode is available in the control store the sequencer starts to generate the control store address in order to determine the next microinstruction and to write it to the microinstruction register (MIR). There it controls either the reconfiguration of the CCU or in case of an execute instruction is executed on the CCU. During the microcode execution the sequencer follows the last CCU status to generate the next ρ-CS-address. After finishing the CCU execution the $\rho\mu$-code unit signals the arbiter the end of the reconfigurable operation.

The prototype implementation described in [4, 3] implements the minimal ISA extension as described above and prevents the GPP and the RP to execute in parallel. Before issuing a reconfigurable instruction the arbiter clears the pipeline of the GPP and drives it into a wait state until the end of the execution of the RP is signaled. Furthermore the arbiter occupies the memory for use by the CCU. Then the active CCU can exclusively access the exchange registers and the memory without the need for synchronization. Every implementation of a CCU must only comply to the interface specifications to the XREGs, the memory and the $\rho\mu$-code unit. The interfaces include data buses, write signals and clock signals.

3 CarCore RU Extension

The main target of the CarCore reconfigurable extension is to speed up threads running on a simultaneous multithreaded processor by the utilization of a custom configurable unit. In one configuration the reconfigurable extension may be used exclusively by one thread preserving the real-time capability for this thread. Meanwhile other non real-time threads continue their execution using the latencies of the main thread. The other scenario is to concurrently use the reconfigurable extension within all threads.

Problems arise when the reconfigurable unit and another thread executing a load or store instruction try to access the memory at the same time. The concurrent memory accesses have to

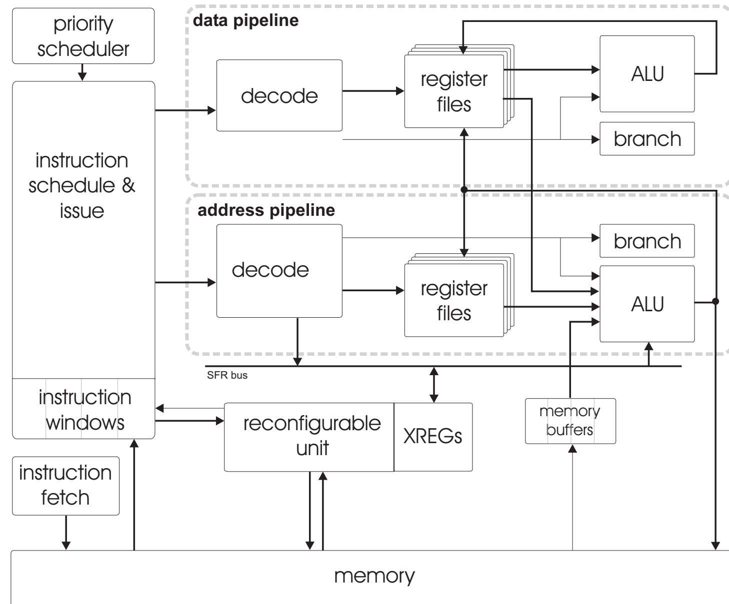


Figure 4: The CarCore with reconfigurable unit.

be arbited and the priorities of the threads considered. The challenge is to preserve the real-time behavior at least for the thread using the reconfigurable part and to enable the scheduler to decide about the memory accesses. In the case of more threads using the reconfigurable extension the access to it also must be arbited. But since the RU may need many cycles to complete its job other threads may have to wait until the RU is free again.

3.1 Interface

The reconfigurable extension is called reconfigurable unit(RU) and is integrated into the existing CarCore like a co-processor following the MOLEN idea. The RU directly interfaces with the scheduler and the memory controller. Furthermore the RU is connected to the special function register (SFR) bus to enable the pipeline to read and write the exchange registers (XREGs) which are part of the RU. Figure 4 gives an overview over the pipeline with the reconfigurable unit interfaced.

The interface of the RU depicted in Figure 5 is kept simple and very similar to the MOLEN interface to the reconfigurable processor. The functionality of the MOLEN arbiter is taken on by the scheduler of the CarCore. When the scheduler recognizes a set or execute instruction it sets `start_op` and writes the instruction word to `instr`. The instruction word is decoded by the reconfigurable unit itself and a set or execute operation is executed accordingly. During the operation of the RU the corresponding thread as well as the RU is locked. Other threads may continue executing instructions in the processor pipeline. After the RU has finished its operation, `end_op` signals the scheduler the completion of the set/execute instruction and the thread as well as the RU are unlocked again.

For memory accesses the RU has its own bus to the memory controller. It consists of read and write signals, an address bus, a data read and write bus and a signal indicating valid data from memory. The memory controller does not arbitrate between memory requests from the pipeline and the RU, so the RU must regard the `mem_rdy` signal. If it's set to one, a memory access from the RU is allowed, otherwise the RU must request a memory access from the scheduler by setting the `mem_req` to one and `req_thread` to the current thread.

To access the XREGs from the processor pipeline the reconfigurable unit is connected to the SFR bus. Hence the XREGs can be read and written like other special function registers, e.g. the

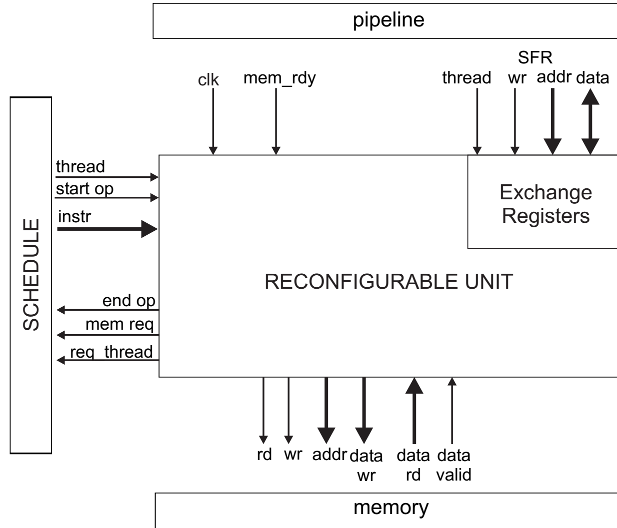


Figure 5: Interface of the reconfigurable unit.

program status words or the thread information words, by using the `mfcr` (move from core register) and `mtcr` (move to core register) assembler instructions. Table 1 shows the sfr bus addresses of the XREGs in the current implementation. Note, that in the current implementation of the simulator the threads share the XREGs unlike the general purpose register sets. Theoretically there is no limitation on the number of XREGs within the 16 bit address space of the SFR bus. Normally not the data itself but only pointers to data regions in memory will be exchanged between the processor pipeline and the RU. Therefore only few XREGs are needed.

XREG	**Address**
XREG0	0x8100
XREG1	0x8104
XREG2	0x8108
XREG3	0x810c
XREG4	0x8110
XREG5	0x8114
XREG6	0x8118
XREG7	0x811c
XREG8	0x8120
XREG9	0x8124
XREG10	0x8128
XREG11	0x812c

Table 1: Addresses of Exchange Registers.

The current implementation of the simulator allows to run a multithreaded workload, one or more threads using the reconfigurable unit in turn. However, if more than one thread is using the RU the programmer or compiler must take care of threads using the same exchange registers. The threads may interfere each other if they are reading and writing different values from the same XREGs. Furthermore the real-time behavior of the thread with the highest priority cannot be guaranteed, if this thread must share the RU with another low priority thread.

NB: According to the access rules of the SFR bus, the sfr_thread, sfr_wr and sfr_addr signals are set one cycle before the sfr_data may be written (in case of a write) or is valid (in case of a read). A detailed description of the SFR bus can be found in [5].

3.2 Instruction Set Extension

In order to use the features of the reconfigurable unit at least four instructions are necessary, namely `mol_set`, `mol_exec`, `mol_movtx`, `mol_movfx`. The `mol_movfx` and `mol_movtx` instructions read and write the exchange registers and are directly mapped to the existing instructions `mfcr` and `mtcr`. Hence they are executed by the address pipeline of the core processor affecting the signals of the internal SFR bus.

`mol_set` and `mol_exec` correspond to the MOLEN complete set and execute instructions respectively and extend the instruction set of the CarCore. The instructions are both 32 bit wide and contain a 16 bit address field specifying the microcode. Figure 6 depicts the instruction format of the new instructions. The position of the two opcode fields is the same as in the TriCore RRR instruction format described in [2].

NB: The opcode for mol_exec overlaps with the `sub.f` instruction of the TriCore floating point unit [1].

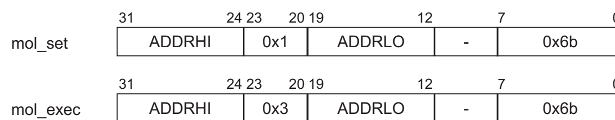


Figure 6: Opcode format of `mol_set` and `mol_exec`.

In order to make the use of the new instructions easier and to avoid changes to the TriCore compilers the new instructions are introduced as preprocessor macros using inline assembler. They are defined in the file *tc_molen.h* (here showing the Hightec GNU compiler style):

```
1   //tc_molen.h
2     #define _mol_exec(addrhi, addrlo) \
3                      __asm( ".word 0x" addrhi "3" addrlo "06b")
4     #define _mol_setc(addrhi, addrlo) \
5                      __asm( ".word 0x" addrhi "1" addrlo "06b" )
6     #define _mol_movtx(xreg, dreg)      \
7                      __asm( "mtcr "xreg", %0" : : "d"(dreg) )
8     #define _mol_movfx(dreg, xreg)      \
9                      __asm( "mfcr %0, "xreg : : "d"(dreg) )
10
11  //Example 1
12    _mol_setc ("00","04");     //setc 0x0004
13    _mol_movtx("0x8100", 15);  //move value of data register d15 to XREG0
14    _mol_exec ("00","64");     //exec 0x0064
15    _mol_movfx(15, "0x8104");  //move value of XREG1 to data register d15
16
17  //Example 2
18  //  XREG0, XREG1, CCU_DCT_ADDR_HI/LO, RMC_DCT_ADDR_HI/LO
19  //  are defined in cc_rusetup.h
20    #include "cc_rusetup.h"
21
22    int param1[2048] = { 200, ... };
23    int param2[2048] = { 200, ... };
24
25    _mol_movtx( XREG0, param1 );  //mov address of array to XREG0
26    _mol_movtx( XREG1, param2 );  //mov address of array to XREG1
27
28    _mol_setc(CCU_DCT_ADDR_HI, CCU_DCT_ADDR_LO);
29    _mol_exec(RMC_DCT_ADDR_HI, RMC_DCT_ADDR_LO);
```

In contrast to the mol_movfx and mol_movtx instructions mol_set and mol_exec are recognized by the scheduler and directed to the reconfigurable unit. They are decoded there and the 16 bit address of the microcode is generated from the `addrhi` (containing the most significant 8 bits) and `addrlo` parts of the instruction.

3.3 Scheduling

The extension of the CarCore with the reconfigurable unit makes changes to the scheduler necessary. The thread information word (TIW) now has two more entries controlling the scheduling behavior for the reconfigurable unit (Figure 7). The other flags of the TIW are described in [5] in its details. The new entries are `RC_ACT` and `RC_MEMREQ` and are maintained by the scheduler itself. `RC_ACT` is set to one, if the reconfigurable unit is active for this thread, `RC_MEMREQ` indicates, that the reconfigurable unit has requested a memory access via the `ru_mem_req` port.

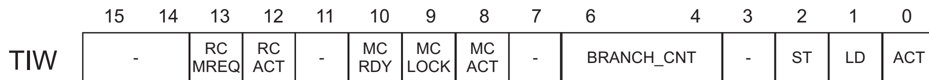


Figure 7: Thread Information Word.

To make a scheduling decision the set/execute instructions and the flags of the reconfigurable unit must be considered. The new scheduling rules are as follows:

1. Dispatch a set or execute from one of the instruction windows to the RU. If more than one thread has a reconfigurable instruction the scheduler takes that one with the highest priority. The RC_ACT flag is set in the corresponding TIW and the thread as well as the RU is locked until the operation has finished. Other threads needing the RU must wait until the the end of operation is signaled and the TIW flags are reseted.

2. Dispatch an instruction to the data pipeline. The selection of the thread follows the same rules as in the original CarCore i.e. a data instruction from a thread with a priority as high as possible is dispatched whereas the thread must not be locked by a branch, load or microcode instruction and a reconfigurable instruction of it must not be executing.

3. Dispatch an operation to the address pipeline. The scheduler chooses a thread with the highest priority and successively considers the following cases until an operation is found.

   A thread executing a reconfigurable instruction normally holds as locked. But if the flag `RC_MEMREQ` of this thread is set the RU requests a memoy access. The scheduler checks if the memory access already takes places in the current cycle by reading the same `mem_rdy` signal as the RU. Otherwise it introduces an empty slot into the address pipeline to give the RU the possibility of a memory access.

   If a thread is locked by a load and the data in the memory buffer is valid, the load insert instruction is inserted into the pipeline. If a thread is locked by a microcode, the next microinstruction of the microcode is executed.

   Finally, if none of the special cases applies, the scheduler tries to dispatch an address instruction from the instruction windows.

These rules allow the scheduler to dispatch the instructions to the two pipelines and to the reconfigurable unit in parallel. Furthermore the pipeline can be used during RU operation by the threads not using the RU. Thereby the restriction applies that only one thread can use the RU at a time, other threads accessing the RU have to wait until the RU finished its operation. The real-time behavior of one thread can only be guaranteed if the highest prioritized thread is the sole accessing the reconfigurable unit.

Concurrent memory accesses of the address pipeline and the RU are also arbited by the scheduler. If the RU cannot immediately access the memory the scheduler decides according to the thread priorities to which thread the access is granted.

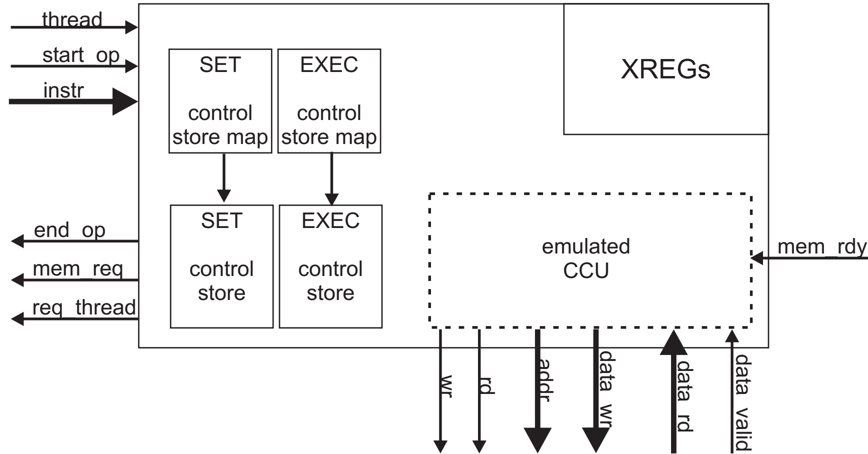


Figure 8: Internal view of the CarCore reconfigurable unit

3.4 Simulation

The reconfigurable unit is written as extension to the existing SystemC simulator of the CarCore. The RU is implemented as module and is emulating the mol_set and mol_exec instructions. Currently the simulation is restricted to microcodes fixed in the control store. Therefore the microcodes are not needed to control the setup of the reconfigurable hardware (there is none in the simulator). Furthermore they are not used to control the execution on the CCU, because the CCUs are emulated by a standard C function call. This section explains how the emulation works and why the experimental results can be considered realistic.

3.4.1 RU Operation

After the `start_op` signal is set to one, the RU decodes the instruction and generates the microcode address. According to the instruction the address is looked up in the set or execute control store map respectively and translated to a position in the control store. The position in the control store determines if the address is valid and which CCU functionality must be emulated by the simulator. Figure 8 shows the internal organization of the RU.

For set instructions no functional emulation takes place, because they only change the internal status of the RU. Set instructions affect the core processor if they access the memory. Therefore a memory profile can be specified performing the memory accesses and generating traffic on the memory bus which is explained later in this section.

In case of a execute instruction, a standard C function is called to emulate the CCU functionality. The XREGs associated with the microcode to be executed are read and the values are passed to the standard C function. If the values of the XREGs are pointers to structures it can be necessary to read out the memory of the simulator and write its contents into a standard C struct passed to the function. After the computation is completed the results of the function are written back to the XREGs. If necessary, the contents of C structs must be copied back to the memory of the simulator. The whole emulation happens in the first cycle the execute instruction is performed on the RU.

3.4.2 Memory Profile

Until now both the processing time of a quite complex CCU operation and the memory accesses made by a CCU are neglected. As concurrent memory accesses of the RU and the core pipeline are crucial for the timing of a multithreaded workload it is important to simulate the actual processing time and the traffic on the memory bus that would be performed by a hardware implementation of the function.

To allow the user of the simulator to easily change the timings of a CCU operation a configuration file specifies the number of cycles the CCU needs and the read and write accesses to the memory. The listing shows an example of a configuration file:

```
1   3025              ## cycles: normalized processing time
2
3   302500            ## divisor
4
5   ## time:         with respect to magnitude
6   ##               the absolute cycle of an operation is calculated
7   ##               by time * cycles / divisor
8   ##
9   ## operation:   LD or ST
10  ## address:     address in memory
11  ## width:       8, 16, 32, 64
12
13  #time   op    address      width
14
15  100     LD    0x00100000   32
16  200     LD    0x00100004   32
17
18  ...
19
20  251300  ST    0x00100000   32
21  251400  ST    0x00100004   32
22  ...
```

The first number of the so called memory profile is the normalized processing time. It represents the number of cycles of the core processor the CCU needs for completion not including latencies of memory accesses. It's noteworthy that all runtimes are counted as core processor cycles, because hardware implementations like the MOLEN prototype may use a slower clock speed for the reconfigurable processor than for the core processor. The second number in the profile is a divisor which all following cycle numbers are divided by. Then a list of memory accesses follows, each in a new line. A specific memory access takes place in the cycle computed by $time * cycles/divisor$ after the begin of operation not counting the cycles of memory latencies. The result of the division is truncated and if more than one access results on the same cycle number only the first one is performed.

NB: The memory accesses are really performed in the simulator, so stores will change the content at the specified address.

3.4.3 DCT Memory Profile

The current simulator only supports one CCU which performs a discrete cosine transformation (DCT) on a block of 128 integer values. The MOLEN project has developed a corresponding hardware implementation of this CCU and validated it against its prototype. The number of memory accesses and the total runtime for the DCT are extracted from the prototype and normalized to get the number of cycles of the core processor. From this numbers the DCT memory profile for the CarCore simulator is derived. They form a generous estimation of the runtime behavior not considering any optimization. The advantage of a changeable memory profile is, that it is easy to check, if a CCU hardware with other timing behavior would boost the performance of the whole multithreaded workload.

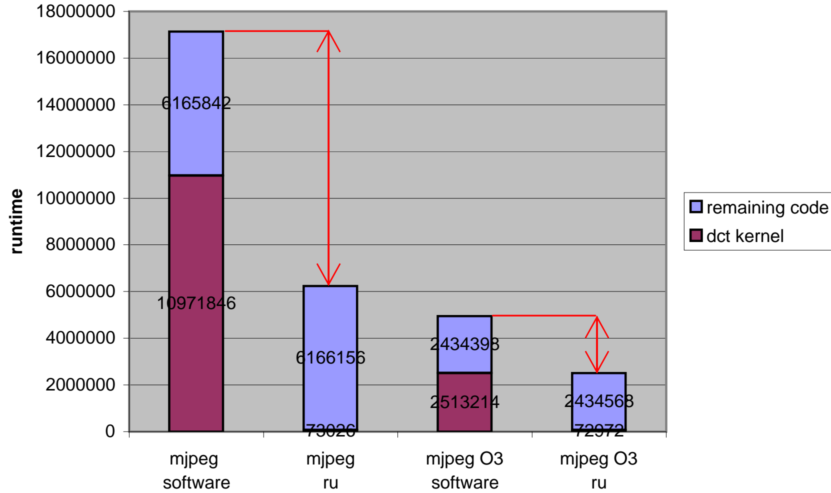


Figure 9: Runtimes of MJPEG benchmark in one single thread.

4 Evaluation

To evaluate the impact of the reconfigurable unit on the CarCore first a workload with a single thread was chosen. A implementation of the MJPEG algorithm serves as load which was compiled in four different configurations: A configuration with pure software execution (not using the RU) and one executing the DCT kernel of the benchmark in hardware (using the RU). Both configurations exist in a version with and without compiler optimizations (-O3 flag). The chart in figure 9 shows the runtimes in core processor cycles of each configuration converting the first frame of a picture sequence to a JPEG image.

The comparison of the unoptimized binaries shows a speedup of the pure DCT function call of about 150 times using the reconfigurable unit. The DCT function is mostly executed by the RU and needs more than half of the processor cycles than the pure software solution. The speedup of the whole application is 2.75 which is similar to the results of the MOLEN project. This is close to the theoretical maximum of 2.78 (assumed a runtime of zero cycles for the DCT). The performance gain of the optimized binaries is 34.44 for the kernel and 1.97 for the whole application (theoretical maximum of 2.03). Although the runtime of the reconfigurable unit is generously estimated, the performance gain is very high and a further optimization of the RU runtime does not seems to be necessary.

For the evaluation of a multithreaded workload all four threads executed the same MJPEG benchmark and independently processed the same picture. Figure 10 shows the runtimes of the optimized binaries using the fixed and the round robin priority scheme. They are compared to the runtime of the sequential execution of the benchmark with and without using the reconfigurable unit. The speedup of the multithreaded workload compared to the sequential execution of the benchmark performing the DCT in the reconfigurable hardware totals to 1.97 and 1.69 for the round robin and the fixed priority scheme respectively. Compared to the sequential execution purely in software the multithreaded speedup nearly doubles to 3.89 (round robin) and 3.34 (FPP) respectively.

5 Conclusion

The CarCore reconfigurable extension speeds up applications with a computational intensive kernel by a factor up to 2.75. In combination with the multithreading features a speedup of 3.89 is

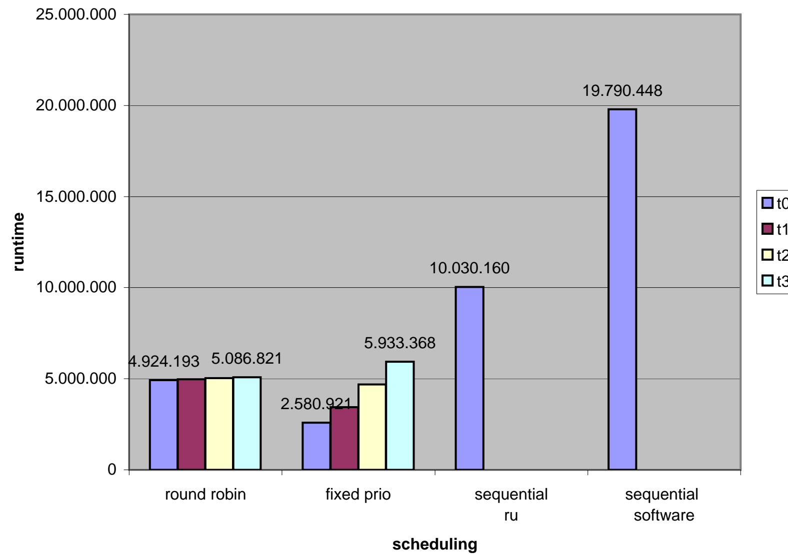


Figure 10: Runtimes of the multithreaded execution of the MJPEG benchmark.

possible. This is achieved by the integration of a reconfigurable unit similar to a co-processor. Changes to the scheduler guarantee the real-time capabilities of the thread with the highest priority in combination with the use of the RU. Therefore the memory accesses of the RU must be controlled by the scheduler.

A future challenge will be to use the reconfigurable unit with more than one real-time thread. A new scheduling scheme has to be integrated and the concurrent use to the reconfigurable unit by more than one thread should be allowed. It must be assured that the threads cannot disturb each other during the usage or even worse the reconfiguration of the RU. Hence memory accesses will be crucial as they may occur concurrently from the instruction fetch stage, the address pipeline and each active CCU in the reconfigurable unit.

References

[1] Infineon Technologies AG, München. *TriCore 1 32-bit Unified Processor Floating Point Unit*, June 2002. Version 1.0.

[2] Infineon Technologies AG, München. *TriCore 1 Architecture Manual*, September 2002. Version 1.3.3.

[3] G.K. Kuzmanov. *The Molen Polymorphic Media Processor*. PhD thesis, Delft University of Technology, December 2004.

[4] G.K. Kuzmanov, G. N. Gaydadjiev, and S. Vassiliadis. The molen processor prototype. In *Proceedings of the IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM 2004)*, pages 296–299, April 2004.

[5] Stefan Maier. Entwurf und Evaluierung eines mehrfädigen TriCore-kompatiblen Prozessorkerns. Master's thesis, Universität Augsburg, June 2005.

[6] S. Vassiliadis, S. Wong, G. N. Gaydadjiev, K. Bertels, G.K. Kuzmanov, and E. Moscu Panainte. The molen polymorphic processor. *IEEE Transactions on Computers*, pages 1363–1375, November 2004.