FASTCHART – A Fast Time Deterministic CPU and Hardware Based Real-Time-Kernel

Lennart Lindh[1]
University of Erlangen-Nürnberg
Institute for Computer Aided Circuit Design
Prof. Müller-Glaser
Wetterkreuz 13
D-8520 Erlangen

Frank Stanischewski
University of Erlangen-Nürnberg
Institute for Computer Science (IMMD)
Lehrstuhl für Rechnerstrukturen (3)
Martensstraße 3
D-8520 Erlangen
email : stani@immd3.informatik.uni-erlangen.de

Abstract

The designer of hard realtime systems requires deterministic behaviour of the system. Today there are problems because of the hardware and the real-time kernel. So one gets only statistic statements regarding timing. This article describes a new hardware structure that is deterministic, fast and includes a real-time kernel in hardware. But this structure is limited to small real-time systems

1 Introduction

With today's hardware it is not possible to get absolute timing for a real-time system, one can get only statistic timing. The first problem are the non deterministic instruction cycles of the CPU because there is pipeline, cache or DMA ([1]). The next problem in calculating absolute timings are the interrupts to the CPU and the unknown number of task switches, that gives one statistic time delays. And the last problem is that the real-time operating system has different execution times for various numbers of tasks.

Time deterministic implies that we need a constant execution time for instructions, the real-time functions and operating system. Therefore, we need a simple system without pipeline, cache and interrupts. But if we omit these things we get slower in overall execution time. For that reason we implement the Real-Time-Kernel (RTK) in hardware. Because of that we get a second benefit, we get deterministic execution time of real-time functions and task switch without any CPU time delay.

2 Overview of FASTCHART

Before we give an overview over FASTCHART we have to describe the simplifications we have made. First we consider only small systems with less than or equal to 64 tasks and 8 priorities. This number is a free choice to get closer to a realistic implementation. Secondly we have reduced the number of possible state transitions for the tasks. This leads to figure 1.

From figure 1 one can infer that we have only 3 real-time function calls for each task. These are:

- **Activate Task :** activates another task.
- **Terminate Task :** terminates itself, afterwards it can only be activated by another task.
- **Delay Task :** deactivates task for a constant time.

Now we can introduce the functions of FASTCHART. In FASTCHART there is a small RTK implemented as described in figure 1, which runs automatically and concurrent to the task. Therefore we had to divide our hardware into two parallel running parts. One is the normal Central Processing Unit CPU, the other is the Real-Time Unit RTU (see figure 2).

Instead of the known ways to implement real-time-systems (e.g. [6], [7], [8]), FASTCHART contains the whole real-time-operating-system in hardware. That means there are no microcoded or ROM-based operating-system-instructions like TRON ([5]) or Transputer or in [7]. Also FASTCHART is designed for general real-time-applications instead of the system described in [8].

The RTU contains the *Task Control Blocks* TCB's, the *scheduler*, the *Ready Queue* and the *Wait-* and *Terminate-Queue*. It also includes the *system timing*. In our CPU we have implemented a RISC-CPU-core with a load-store-architecture and the ability to do a task switch after the execution of every instruction. Task switch means that the CPU can switch to another task during one cycle. Therefore we need two register-files, which are exchanged only during this task-switch. Also the CPU has the above described real-time function calls as instructions to synchronize with the RTU and send parameters to the RTU.

[1] Currently guest reseacher at University of Erlangen-Nürnberg. Home address : University of Eskilstuna/Västerås, Institute of Data and Electronics, P.O. Box 11, 72103 Västerås, Sweden

0-8186-2212-1/91/0000/0036$01.00 © 1991 IEEE

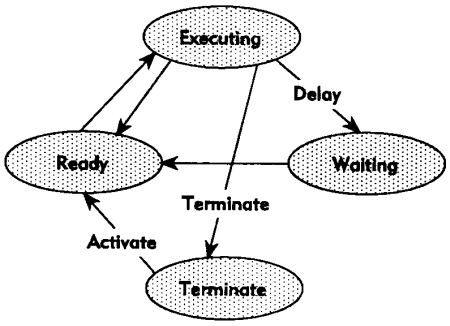

Figure 1: State diagram

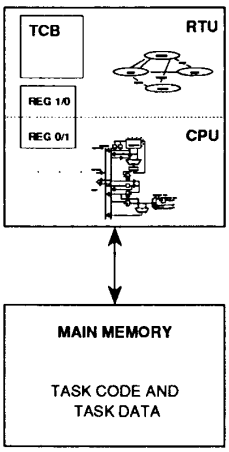

Figure 2: Overview over FASTCHART

3 The Central Processing Unit CPU

Our purpose in designing the central processing unit was to obtain a CPU with a deterministic time behaviour. One deterministic CPU-concept we considered was the FORTH-machine that we initially considered as a basis for this architecture ([2],[3]). First we changed the data stack to a register-file to meet our need for a fast and easy task-switch. The second modification is the return stack, that we located outside the CPU in the main memory. We thereby obtain the following programming model with a *Program Counter* PC, *Status Register* SR and a register-file with 8 registers, where the first register R1 has the function of the *Return Stack Pointer*.

As can be seen in figure 3, the main memory space for each task is divided into four parts. First we have the task program code space, the memory space for task global data, then space for as many data stacks as are needed and lastly in the task memory space the return address stack.

The instruction set of the CPU is similar to a LOAD-STORE-architecture without indirect addressing modes. The CPU recognizes instructions for ALU and Shifter operations, Load or Store from or to main memory, conditional and unconditional branches and call and return subroutines. Additionally the above described function calls are in the instruction set. Also we have the possibility to combine instructions as for example :

LOAD R3, (R4)+

This operation loads a data addressed by register R4 from main memory in register R3 and also increments the value in R4. So we can obtain a high concurrency in our CPU that accelerates execution time compared with the lack of pipeline and cache.

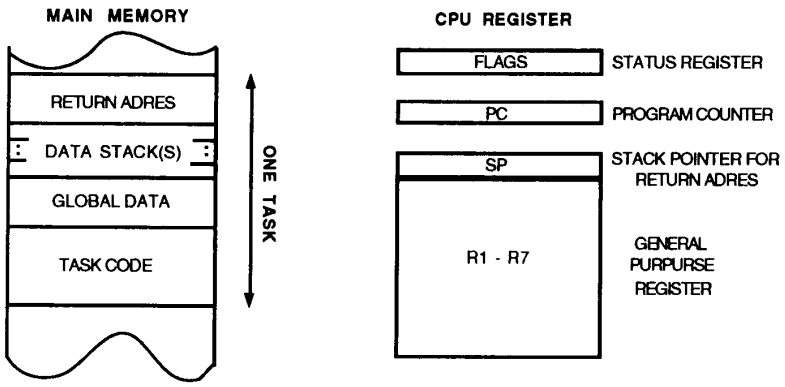

Figure 3: The Programming Modell of FASTCHART

The normal instructions need only one CPU cycle. Instructions which have an additional main memory access need two cycles. An example of such an instruction is Call Subroutine:

CSR $ 1000 Call Subroutine at $1000

In the first CPU cycle the instruction is fetched and decoded, in the second cycle the value of the *Program Counter* is pushed onto the return address stack and the *Program Counter* is loaded with the subroutine address. The stack pointer is modified automatically.

Figure 4 shows the logical schematic of the CPU. The shaded parts in this schematic are the double existing registers which are exchanged during a task-switch. Like every RISC there is no microcode and therefore the instruction decoder is very simple. It controls the program execution and data manipulation units. Also it takes the control of the synchronization between the CPU and RTU. It decodes the real-time functions and delivers them to the RTU. A further function of the decoder is set and reset of the *Not-Switch-Flag*. The job of the *Not-Switch-Flag* is first to suppress task-switching in the execution of two- or three-cycle instructions and second to avoid task-switching in critical program parts. During these program parts the flag is set and reset from the program.

4 The Real-Time Unit RTU

In our Real-Time Unit RTU we implement a complete real-time-kernel in hardware. It can manage 64 concurrent tasks independently of the CPU. Therefore we copy our task state diagram directly in hardware. For every state in the diagram there is a block in the RTU. Additionally there is a *Control Unit* that controls the overall execution of te RTU and receives the synchronize instructions from CPU.

The shaded part in the RTU schematic stands for the execution state and the task-switch part of RTU. The register named OLD contains the task ID of the current task. The other register NEW contains the ID of the next task. After a task switch, when the register-files are exchanged, the values of all registers of the *old* task are written back to TCB memory in the location given by OLD-task-ID multiplied by TCB size. Then the contents of task register NEW are transferred to register OLD. After that a new task ID is fetched from *Ready Queue*. Using this new ID the registers are addressed from this task in TCB memory and transfered to the register-file. The values in TCB memory contain the register R0 to R7, the *status register* SR, *the program counter* PC and the *instruction latch* IL for every task. As one can infer from figure 5, the registers NEW and OLD also store the priorities of the tasks. Therefore it is not necessary to look in TCB memory when a task changes its state and we need its priority in the new state.

There two ways the current (OLD) task can change its state; first by itself, second when a higher privileged task is in *Ready Queue*. In the second case the ID from register OLD is written into the *Ready Queue* before the contents of this register is overwritten with the value of NEW. If the current task changes its state by itself, the ID is not written into *Ready Queue*, but in *Wait-* or *Terminate-Queue*.

The scheduler algorithm in the *Ready Queue* is a static priority-driven algorithm and it is implemented with 8 FIFO's, one for each priority level with depth of 8. On demand – that means after a task switch when the contents of the NEW register are transferred to OLD – the *Ready Queue* gives at this time the highest priority ID to the NEW register. The highest priority ID is detected by looking for the highest privileged non empty FIFO and the first ID in that FIFO is used.

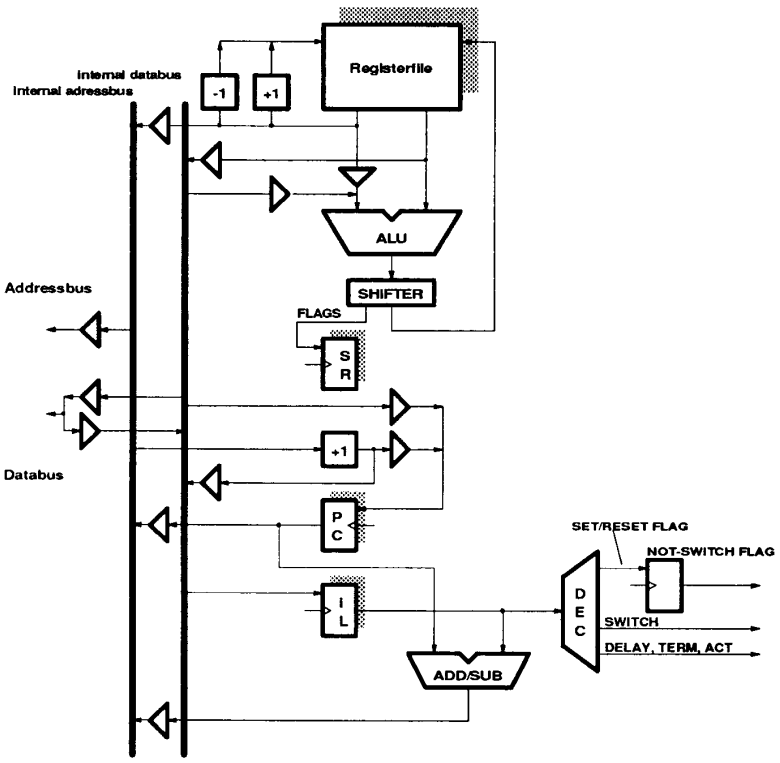

Figure 4: The CPU schematic

If a task changes to ready state, the priority of this task is used to select the correct priority-FIFO and the ID is put into this FIFO. So one only needs the ID of a task in the *Ready Queue* because the priority of that task is encoded in the number or priority of the FIFO.

If a task terminates, its inactive status is written to the *Terminate Queue*. The *Terminate Queue* is implemented as a RAM where each ID has its own location. Each location contains a flag INAC which indicates whether the task is inactive or not. If a task is terminated via a function call, the flag INAC is set and a task switch is initiated. If a task activates another inactive task, the function call needs two parameters, the ID of the inactive task and the new priority. Then the flag INAC for the activated task is reset and the ID of this task and the priority from function call is writted to *Ready Queue*. If the activated task is already active, an error code is given back to CPU.

When the current task will delay for a number of time ticks, the CPU gives the delay time to the RTU and then a task switch is initiated. A down-counter in the *Wait Queue* is loaded with the *Delay Time*. The down-counter in the *Wait Queue* is selected by the ID of the delayed task. Therefore 64 counters are needed in the *Wait Queue*. The down-counters count with the system time tick and every counter gives a signal when it is empty. With this signal the *Control Unit* is activated to transfer the task ID given by the number of the counter to the *Ready Queue* as described above. After the transfer, the *Control Unit* gives an acknowledge and the down-counter is set to an inactive status. If at the same time more than one counter gives a signal, the counters are served one after the other in order of priority.

5 Conclusions

Our approach with this article is to show that a real-

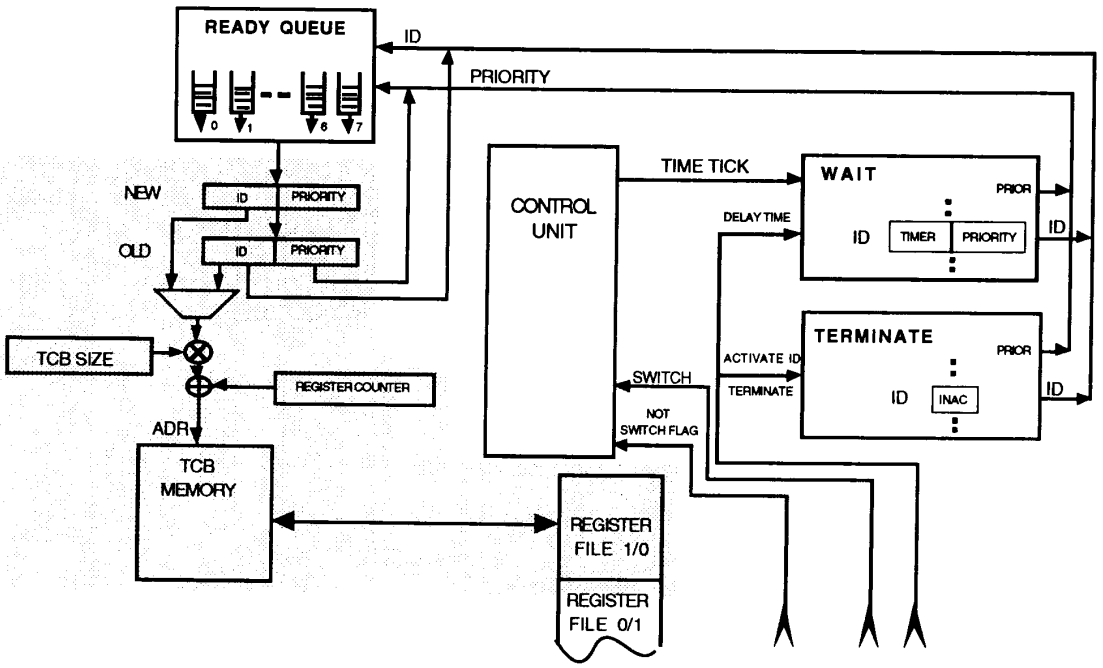

Figure 5: The RTU schematic

time-kernel can be implemented in hardware and also that a time deterministic CPU can be built. Our first model is a simulation model in a logic synthesis language that we try to implement direct in hardware. We estimate that we need 100,000 gates if we implement 16-bit FASTCHART in one gate array. But for our first implementation we want to use smaller gate arrays because so we can change much more easier parts or functions of FASTCHART. Also we are implementing a programming environment for FASTCHART which seems to be much simpler than other environments.

In our simulation model the exchange of the task register contents from register file to TCB memory and vice versa is possible in one CPU cycle. This is very difficult to implement we think, because one has only two possibilities to reach this. One possibility is that the RTU runs much more faster than the CPU, so the data can exchange successively through a small data path. The other possibility is to widen the data path and use less cycles to exchange data. Both possibilites are not optimal so one has to make compromizes.

In our future work we want to expand the real-time functions of FASTCHART for functions like rendezvous. If it is possible we are looking for a cooperation to implement FASTCHART in one chip. Also we want to design systems with more than one CPU and with a communication coprozessor.

References

[1] John A. Stankovic and Keirti Ramanmritham, Hard Real-Time Systems, pages 361-370, Computer Society Press of the IEEE, 1988

[2] C.H. Ting, Footsteps in an empty valley, Offete Enterprises, 1986

[3] RTX2000, Data Sheet, Harris Corporation, 1988

[4] The Transputer Databook, INMOS, 1988

[5] Ken Sakamura (ed.), TRON Project 1989, Springer, 1989, ISBN 0-387-70050-1

[6] Joachim Roos, The Design of a Real-Time Coprocessor for ADA Tasking, pages 17.0-17.12, NORSILC/NORCHIP Seminar 1989, Stockholm, Sweden

[7] John Tinnon, Real-Time Operating System Puts Its Execution on Silicon, pages 137-140, Electronics, April 21, 1982

[8] T. Juntunen, J. Kivelä, A. Reinikka, M. Sipola, J.-P. Soininen, K. Tiensyrja, T. Tikkanen, Real-Time Structured Analysis in System Level Design of Embedded ASICs, pages 449-454, Microprocessing and Microprogramming (24), 1988