FASTCHART – Idea and Implementation

Lennart Lindh[1]
University of Eskilstuna/Västerås

Frank Stanischewski[2]
University of Erlangen-Nürnberg

Abstract

In this article there is a description of a new hardware structure for small real-time systems which is time deterministic in the execution of CPU instructions and the real-time operating system. It also includes the studies of practicability and the difficulties in implementing the structure.

1 Introduction

The designers of hard real-time systems want to know whether the response time for all functions will be shorter than a specified maximum time. Achieving this requirement the real-time system will be predictable. This article shows how one can get predictable real-time systems on a microscopic level and an implementation of this principle. In the article we define a new unit named RTU, Real Time Unit, which works concurrently with the CPU.

2 Idea

In this section we give only a short overview of FASTCHART. For a full description see [9]. Before we describe the functionality of FASTCHART we explain the design goals we want to reach with FASTCHART.

2.1 Design goals

In today's real-time systems it is normal that one can not get an absolute execution timing because of the use of pipeline, cache or DMA. Also the operating system is not predictable because the execution time of the real-time operating system depends on the number of tasks which are currently active or in delay status etc.

With FASTCHART we want to get a predictable real-time system, so FASTCHART must be deterministic in :

- Execution of CPU instructions
- Execution of the real-time operating system

[1] University of Eskilstuna/Västerås, Institute of Data and Electronics, P.O.Box 11, 72103 Västerås, Sweden, FAX : +46 21 114922

[2] University of Erlangen-Nürnberg, Institute of Computer Science 3 (IMMD 3), Martensstraße 3, 8520 Erlangen, FAX : +49 9131 39388,
e-mail : stani@immd3.informatik.uni-erlangen.de

2.2 Overview

Because of the defined design goals FASTCHART can not have pipeline, cache, DMA or interrupts. For this reason the performance of the CPU slows down. So we transfer important parts of the real-time kernel RTK into hardware to give the CPU more time to spend in the working tasks.

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].

Therefore FASTCHART has two concurrent running parts. One is a CPU designed to our purposes and the other part is the Real Time Unit RTU.

2.3 The Central Processing Unit CPU

The concept of the CPU is similar to the FORTH-machine described in [2] and [3]. But we have changed the data-stack into a register file and moved the return stack into memory. So we have a very simple CPU where normal instructions need one CPU-cycle and instructions with an extra memory fetch need two cycles.

The programming model consists of a Program Counter PC, a Status Register SR, eight or sixteen general purpose registers R0 - R7/R15 and an Instruction Latch IL. The register R0 has the function of the return stack pointer. One can also use every register as a data stack pointer.

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 three real-time functions (see section 2.4) are in the instruction set. We also have the possibility to combine instructions as for example :

LOAD R3, (R4)+

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

CH3040-3/91/0000/0401$01.00 © 1991 IEEE

To meet our design goals we have to implement two registerfiles which means all above named registers and latches exist twice. One registerfile is currently used by the CPU, the other can be accessed by the RTU. In figure 1 one can see this by the shadowed registers.

If a task switch occurs the RTU changes the registerfiles and the CPU can immediately run the next task by accessing the other registerfile. A detailed description of a task switch is in the following section.

2.4 The Real-Time Unit RTU

The Real-Time Unit RTU can manage 64 tasks with eight priorities. This is an arbitray choice to get closer to a realistic implementation. Every task can be in one of four states (see figure 2).

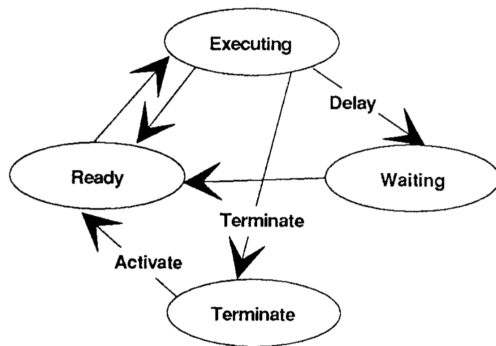

Figure 2: State diagram

From figure 2 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.

Because we have made the real-time kernel RTK so simple we can copy the state diagram directly into hardware. For each state one can find a block in the schematic of the RTU. Additionally there is a *Control Unit* that controls the overall execution of the RTU and receives the synchronize instructions from the CPU.

The shaded part in the RTU schematic stands for the execution state and the task-switch part of the 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 transferred to the register-file. The values in TCB memory contain the register R0 to R7/R15, the *status register* SR, *the program counter* PC and the *instruction latch* IL for every task.

There are two ways in which 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 a depth of 8. 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. To get the task with the highest priority one has only to look for the highest privileged non empty FIFO.

When the current task wants to be delayed 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 counter counts with the system time tick. If the value in the counter reaches zero, the ID connected to the counter is loaded in the *Ready Queue*.

If a task terminates, its ID is written in the *Terminate block* and the task can only be activated by another task.

3 Implementation

This section describes how we implemented our first prototype and what kind of problems we had with it.

3.1 First prototype

We built up a first prototype to show that the idea of FASTCHART will really work. The prototype is realised in a manner called "Rapid Prototyping". Rapid prototyping allowed us to develop the first FASTCHART in eight weeks. But we had to make some compromises for the implementation.

The prototype "FASTCHART ONE" is described in VHDL, a logic synthesis language, compiled and simulated with ViewLogic and then it is built up from small gate arrays (FPGAs from the company ACTEL). Because of the limited complexity we had to shrink the features of FASTCHART ONE. So we have only two priorities and eight tasks. And the CPU can only perform instructions like No-Op, Jump, and the real-time functions.

The building consists of six FPGAs and has a complexity of 10,000 gates. The RTU needs about thirty state-machines and has a clock ten times faster than the CPU. The CPU has an 8 bit data bus and an 8 bit address bus.

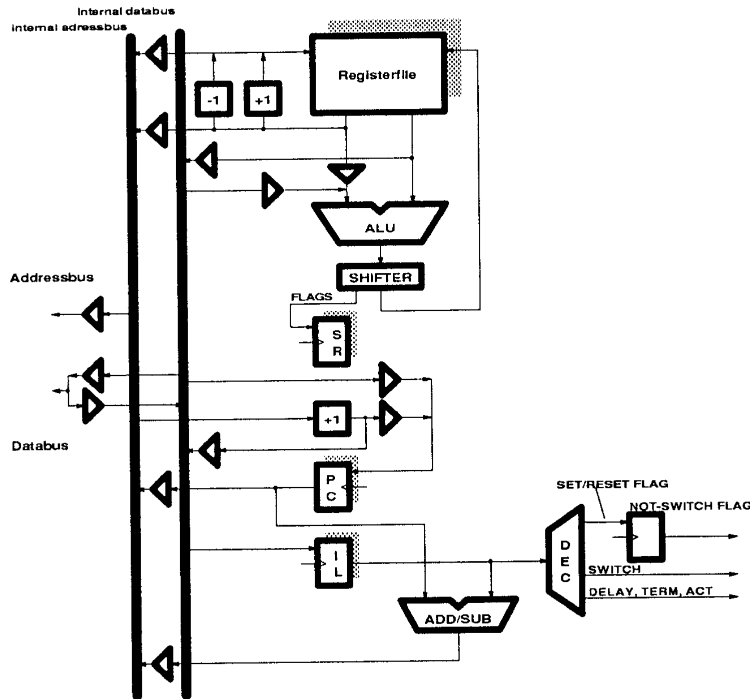

Figure 1: The CPU schematic

```
INIT_TASK :

ACTIVATE (TASK_1,PR_0);
STARTADDRESS (20H);
ACTIVATE (TASK_2,PR_1);
STARTADDRESS (40H);
ACTIVATE (TASK_3,PR_0);
STARTADDRESS (60H);
TERMINATE;

TASK_1 :
begin   DELAY (16 CLOCKS);
        JMP begin;

TASK_2 :
begin   DELAY (16 CLOCKS);
        JMP begin;

TASK_3 :
begin   DELAY (16 CLOCKS);
        JMP begin;
```

Figure 4: Prototype program

To show that FASTCHART operates we test FASTCHART ONE with a small program (see figure 4). In this program there are four tasks. The first task (INIT_TASK) initialises and starts the other tasks with different priorities and then it terminates. The other tasks (TASK_1, TASK_2, TASK_3) are periodic, run in a loop and have a delay of 16 CPU clock cycles in the loop.

3.2 Difficulties

In the structure of FASTCHART we assume that it is possible to transfer the contents of one registerfile in TCB-memory and back in one CPU cycle. For a real implementation this transfer is difficult to achieve. First one has the possibility to run the RTU faster than the CPU. That means that one transfers one register after the other into the TCB memory and vice versa. But then one needs very many RTU cycles. The other way is to widen the data path to the TCB memory so that one does not need so many cycles for the transfer. Which way one chooses is a question of the implementation and what speed one wants to achieve.

A second problem is the dependence of the TCB memory size on the number of tasks one would currently run. For each task one needs about 50 bytes in the memory. So if the number of tasks were to increase, the TCB memory would have to be enlarged, making it impossible to implement on the same chip.

3.3 Design discussion

When we implemented FASTCHART we saw that there was the possibility of problems when running a real application on it, because FASTCHART doesn't know a minimum execution time for a task. So it might be that some tasks with low priorities never

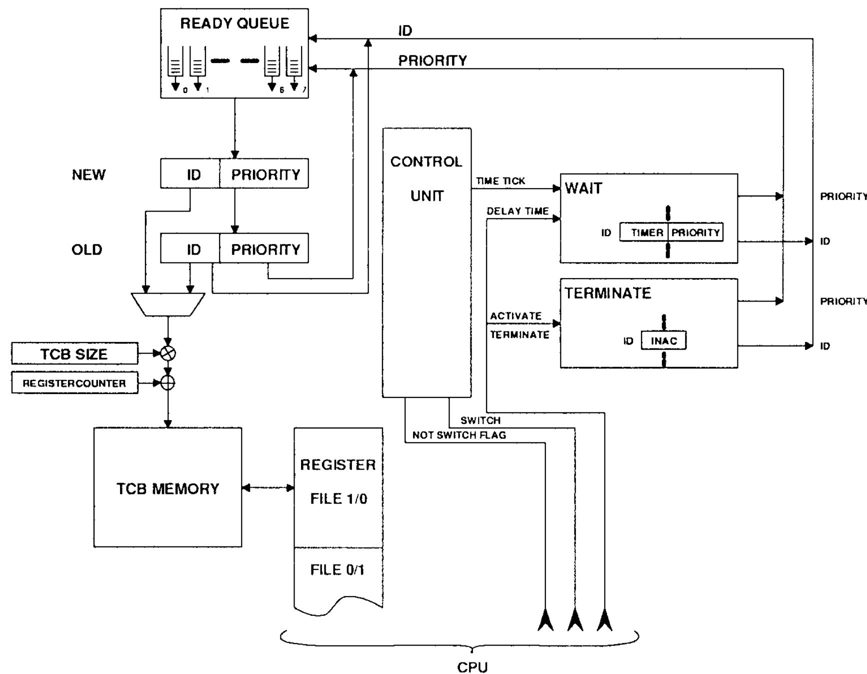

Figure 3: The RTU schematic

execute their job. We also have no maximum time so tasks with high priorities must terminate on their own otherwise no other task can change into execute state.

4 Conclusions

Today it is possible to design a predictable CPU for real-time systems. We have also shown that it is possible to implement a real-time kernel in hardware working concurrent to the CPU. But to get a 100 percent predictable system one has to make a number of concessions e.g. decrease of performance of the CPU or complex implementation.

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

[9] L. Lindh, F. Stanischewski, FASTCHART - A Fast Deterministic CPU and Hardware Based Real-Time-Kernel, accepted for EUROMICRO '91 Workshop on Real-Time Systems, 1991