Microcode with Embedded Timing Constraints

Bernhard Fechner

Department of Computer Science
FernUniversität in Hagen
58084 Hagen
Bernhard.Fechner@fernuni-hagen.de

Abstract: Watchdogs are a well-known and widespread means to increase the safety of microprocessors. The programmer or the compiler must insert instructions to reset the watchdog. If the programmer or compiler chose the wrong timing values or forgot to insert instructions to reset the timer, the processor will never be able to fulfill its task, because it will be set back to an initial (known) state each time it encounters a timing violation. We eliminate the need to insert special instructions and dedicated external watchdog hardware. Our strategy is able to detect transient control-flow faults in state automata and faulty BUSY-signals of execution units in microcode-based microprocessors. The innovation is to introduce fixed timings for each microcode so explicit instruction sequences to reset the watchdog timer are not necessary any more. Each execution unit receives a timing value from the microcode ROM. A unit-specific cycle counter is set to the timing from the microcode (μcode) when the execution starts. Due to possible different execution runtimes (e.g. floating point division), we include the possibility to select the timing accuracy. If the timing is *not accurate*, the timing value is set to the maximal timing of the concerned operation. Then, a fault will only be signaled if the cycle-counter value is greater than the maximal timing. The scheme can be implemented very fast at small additional hardware cost. An FPGA-based implementation of microcode timing as an extension of a multi-cycle 32 bit microprocessor with support for forwarding showed a hardware increase of less than 1.3% using normal place and route effort with a maximal execution time of 16 cycles for each microcode.

1 Introduction

Watchdogs can detect crashes in microprocessors. They are cost-efficient and well-suited to increase safety [5]. To detect a crash an autonomous timer unit (the watchdog) has to be set to a specified value. While executing, the timer counts down. If it reaches zero, an interrupt will be generated.

The two main causes for such an interrupt are:

1. The programmer/ compiler chose the wrong timing values or forgot to insert instructions to reset the timer.
2. The processor crashed.

The watchdog value is set well above the actual timing of the program e.g. because of the dynamic execution in superscalar processors. This coarse-grained timing relates to a late detection of the crash, leading to data loss or the loss of system-relevant functionality during mission-critical phases. We eliminate the need to insert instructions to reset the watchdog. This leads to less energy consumption and to a more reliable system. Less energy is consumed because the processor does not have to fetch the watchdog-related instructions over the bus. Reliability is increased, because the error-prone insertion of watchdog instructions is not necessary. Each microcode-related execution will be monitored by a counter. Faults can be detected very early, because microcode timing works cycle-based.

The fault model assumes one fault at a time for a component and transient faults in the form of Single Event Upsets (SEUs). Single Event Upsets (SEUs) are transient errors which are caused by high-energetic particles hitting the die. SEUs are modeled by bit-flips of the corresponding latches or memory cells [4]. This modeling closely matches the real faulty behavior [3]. The rest if this paper is organized as follows: Section 2 presents related work. Section 3 explains microcode timing in detail. In Section 4 an evaluation of the circuit costs is done. Section 5 concludes the paper.

2 Related Work

Watchdogs can be found in early fault-tolerant computing systems such as the SEL-88 or the HP Systemsate/1000 [2]. These systems used low-resolution timers for the detection of faults. Macroinstruction control-flow monitoring divides the application program into blocks. Blocks are checked instruction by instruction for control-flow faults [5][6]. In [7] signature instruction streams (SIS) were introduced. A checksum is computed over the instruction stream and inserted into the binary code after a branch. The monitor reads and compares this checksum with the computed checksum. An error is detected if the checksums do not match. With a probability of a branch occurring every forth to tenth instruction, the overhead to store the signatures is between 10% and 25% of the original program code. Because the monitor is much simpler than the processor it monitors, performance degrades (because of extra memory cycles). The effectiveness of SIS was verified by hardware fault-injection for a Motorola 68000 system [7][8]. SIS raised the error detection rate to 25% in comparison to the original system.

The Dynamic Implementation Verification Architecture (DIVA) [9][10] adds fault-tolerance to fight permanent (design) errors and single-event upsets (SEUs) to any processor design by adding the DIVA core, consisting of two pipelines and a checker unit to the commit phase of the processor. Additionally, a watchdog is run with a maximum value gained from the latency of each instruction. Unfortunately a slowdown of execution (maximal ~14%, without any extra cache ports or register files, average 3%) and a space increase occurs. With contemporary microprocessors (like the Intel Pentium 4), it is possible to use *performance counters* [11] to trigger an interrupt upon overflow for sampling. Performance counters are normally used to monitor hardware events. It is possible to save external watchdog hardware by using performance counters as watchdogs. The resolution of a performance counter is very high and errors can be detected very early at program level. But one problem still persists: to detect the error very early, we have to know the exact execution time. Additionally, we have to choose the right hardware events to monitor.

3 Microcode Timing

To implement timing constraints in the microcode, we have to enhance every microcode by a timing entry. The additional entry represents the timing requirements for the microcode originating from the specification. The exact timing values can be won by using cycle-accurate simulators. Let *mlen* be the number of microcode entries. We define the length (operator) of a binary string as

$$|\ \ |: \{0,1\}^b \to \mathbb{N}$$
$$\left|\{0,1\}^b\right| := b.$$

A microcode entry e_i, $i \in \{1,...,mlen\}$ is defined as (where $\{0,1\}^{|control_i|}$ is the microcode entry without timing extensions):

$$e_i := \{0,1\}^{|control_i|}\{0,1\}^{|timing_i|}\{0,1\}^{|accurate_i|}.$$

There can be two types of microcode timings, signaled by the flag *accurate*. If the flag is set, the microcode i exactly executes in $timing_i$ cycles. If not, we set $timing_i=cmax_i$, because the execution time for some microcodes might differ. This can happen if e.g. the design of a multiplier is hybrid, applying different algorithms at different input combinations so that different runtimes result. $Cmax_i$ is the maximal number of cycles the execution of µcode i needs. Figure 1 shows the microcode-timing scheme. Before the execution starts a cycle counter C is initialized with zero. The counter is incremented every cycle. If it reaches the beforehand loaded timing constraint *T* from the microcode, but the execution has not finished (signaled through the BUSY-signal of the unit), a fault will be signaled. If both values are equal and the unit is not busy any more, no fault occurred. If a particle changed the value of C this will be detected, because the counter will certainly reach value T earlier (later) than expected while the BUSY flag of the concerned unit is enabled (disabled).

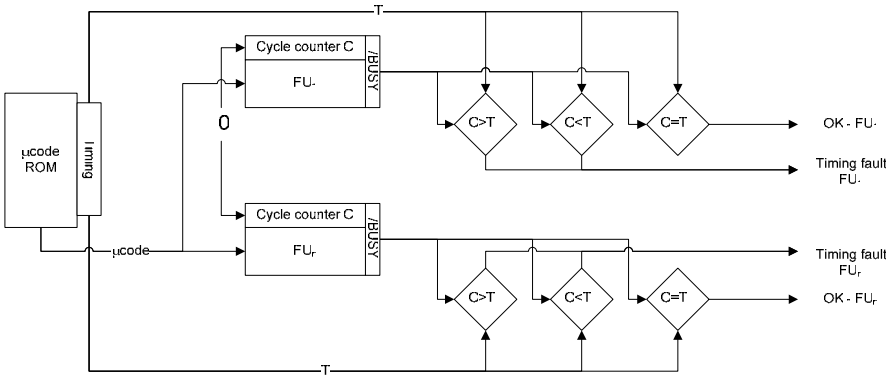

Figure 1: Microcode-based timing

We can identify three states for the checked execution of a microcode:

- S1 (LOAD): At the start of execution the cycle counter C is set to zero (`ccountval=0`). The BUSY-flag is set and the compare units are loaded with the timing value T from the microcode (`mcodeval`). Then we go into state S2.
- S2 (RUN): The cycle counter is incremented. Here we have two options:
  - SAFE-MODE: We assume that the BUSY-signal from unit FU_i can be faulty. In this mode we compare the counter each cycle with the loaded values to detect timing violations (see algorithmic description below).
  - N-MODE: On (N)ormal mode operation, we wait for BUSY to get low again. If this happens, we go to state S3.
- S3 (STOP): As in SAFE-MODE, we compare `ccountval` and `mcodeval`. If we have accurate timing enabled, we signal a timing violation if the values are not equal. If we have inaccurate timing, we signal a fault if `ccountval>mcodeval`.

In the states S2 (SAFE-MODE) and S3, we need to compare `ccountval` and `mcodeval` and signal a fault if applicable. Since we can have multiple causes for a fault, we assign a higher priority to a fault, if it has a higher probability. State automata have more flip-flops than a single flag. Thus, there is a higher probability for a SEU in a state machine than for a single flag. Since the clock tree usually spans the whole chip, there is a higher probability for a fault than for a single flag. For clarity, we show the algorithmic description for the signaling of faults in pseudo-code in Figure 2.

```
#########################################################################
#                                                                       #
#           Microcode with Embedded Timing Constraints                  #
#                                                                       #
#                       - Signaling of faults -                         #
#                                                                       #
#########################################################################


ccountval:=0;

while (BUSY) {

  if (SAFE-MODE) {
    if (ccountval=mcodeval): signal ”Flag fault.”
    if (ccountval>mcodeval):
       signal ”Timing violation. Unit run slower than expected
                    State automata fault.
                    Clock line fault.
                    Flag fault.”

   } # end if
   ccountval++;

}   # end while

#########################################################################
# Now we are not BUSY any more. Do this for N-MODE and SAFE-MODE #
#########################################################################

if (ccountval<mcodeval && (accurate)):
       signal ”Timing violation.
                    Unit ran faster than expected.
                    State automata fault.
                    Flag fault.”

if (ccountval=mcodeval)
       signal ”Unit has terminated execution correctly.”

if (ccountval>mcodeval)
       signal ”Timing violation. Unit ran slower than expected.
                    Counter/State automata fault.
                    Clock line fault.”
```

Figure 2: Algorithmic description for the signaling of faults

Remember that a cycle counter has to be implemented for each functional unit of a superscalar processor.

4 Cost Analysis

To measure the space requirements, we discuss the synthesis results from the proposed microcode scheme for FPGAs (Field Programmable Gate Array) with normal place and route effort. The circuit was implemented in VHDL as an integrated part of a multi-cycle processor with support for forwarding. As a target we used the Xilinx Virtex-E XCV1000, bg560, speed grade -8 FPGA [1]. Table 1 shows the resource usage. The column '*before'* holds the number of slices before the implementation of microcode timing. We included the number of external IOBs (IO blocks) because logic was mainly routed to external IOBs.

Table 1: Resource usage for microcode timing

	Before	**After**	**Increase**	**Total**
Number of External IOBs	294	299	~1.23%	404
Number of Slices	151	152	~8.14‰	12288

In relation to the total number of FPGA slices (12288), the overall space increase was lower than 1.23814%. For the synthesis we assumed a maximal execution time of 16 cycles for a microcode. Larger values can be implemented at a linear cost increase, since only one column of the microcode must be changed. Contemporary ALUs only take a few cycles to execute. Thus, only small values must be stored in the timing entries of the microcode ROM. For all estimations, we took into account that timing values coming from the microcode must be stored in every structure dealing with microcodes until the designated execution unit is reached. For superscalar processors, e.g. the microcode and the timing values must be stored in the dispatch queue.

5 Conclusion

In this paper we proposed a novel and innovative scheme to enhance the reliability of microcode-based microprocessors by adding timing constraints to the microcode ROM. The scheme is very simple and can be implemented easily with minor changes to the existing micro-architecture if cycle-accurate information about microcode execution times is available. To prove this, we synthesized the proposed circuit in VHDL as an integral part of a multi-cycle processor with support for forwarding. As a target we selected a Xilinx Virtex-E XCV1000bg560 speed grade -8 FPGA [1].

The hardware cost increased by less than 1.3% in comparison with the original design (normal place and route effort). Microcode timing has no effect on performance, because cycle counters are incremented each clock cycle in parallel to the execution. The scheme can be used to detect transient control-flow faults in state automata (e.g. floating point, integer, processor control), because a transient fault will have an impact on the timing of the concerned unit by changing its state. Furthermore, it is able to detect faulty BUSY-signals by comparing the state of the BUSY-signal, the cycle counter and the value from the microcode. It is also possible to detect a transient fault in one of the clock lines because a transient fault will lead to a longer unit execution time if the units and the designated counter are clocked synchronously from different sources. Faults covered by watchdogs are transient control-flow faults on program-level and permanent processor faults. Faults covered by microcode timing are mainly transient control-flow faults in programs and transient faults on micro-architectural level such as faults in state automata. As shown in Figure 1, faults can be assigned to a functional unit and thus be located. Furthermore, the detection speed is different. Microcode timing enables to detect faults within a very short period – one cycle after the actual occurrence of the fault. This fast detection enables a precise localization of the fault in time, whereas other watchdog schemes will take several hundred or even thousand cycles. This enables fast and simple chip-based recovery schemes.

References

[1] Xilinx: *Virtex-E 1.8 V Field Programmable Gate Arrays*, 2002. http://direct.xilinx.com/bvdocs/publications/ds022.pdf

[2] O. Serlin: Fault-Tolerant Systems in Commercial Applications, In *IEEE-Computer*, pp. 19-30, August 1984.

[3] R.W. Wieler, Z. Zhang, R.D. McLeod: *Simulating static and dynamic faults in BIST structures with a FPGA based emulator*. In Proc. of IEEE International Workshop of Field-Programmable Logic and Application, pp. 240-250, 1994.

[4] J.P. Hayes: *Fault Modeling*, IEEE Design& Test, pp. 88-95, April 1985.

[5] J.C. Laprie (ed.): Dependability: Basic Concepts and Terminology, Springer-Verlag 1992.

[6] M. Namjoo. "Techniques for Concurrent Testing of VLSI Processor Operation". In Proc. of the 12th Int'l. Symp. On Fault-Tolerant-Computing, IEEE Computer Society, Santa Monica, CA, June 1982, pp. 461-468.

[7] T. Sridhar, S.M. Thatte. "Concurrent Checking of Program Flow in VLSI Processors." In Digest of the 1982 Int'l. Test Conference, IEEE 1982, paper 9.2, pp. 191-199.

[8] J.P. Shen, M.A. Schuette. "On-Line Self-Monitoring Using Signatured Instruction Streams", IEEE Proc. 13th Int'l. Test Conference, Oct. 1983, pp. 275-282.

[9] C. Weaver, T. Austin, "A Fault Tolerant Approach to Microprocessor Design", IEEE Intl. Conference on Dependable Systems and Networks (DSN-2001), July 2001

[10] T.M. Austin, DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design, In Proc. of the 32nd Intl. Symp. On Microarchitecture, Nov. 1999.

[11] Intel. IA-32 Intel Architecture Optimization. Reference Manual. Document Number: 248966-009, 2003.