Concurrent error-detection and modular fault-tolerance in an 32-bit processing core for embedded space flight applications

Jiri Gaisler
On-board Data Division
European Space Research and Technology Centre
2200AG Noordwijk, The Netherlands
email: jgais@wd.estec.esa.nl

Abstract

This paper describes the concurrent error-detection methods employed in the ERC32, a 32-bit processing core for embedded space flight applications. The processor core consists of three devices; an integer unit, a floating point unit and a memory controller. All three devices are provided with internal concurrent error-detection, mainly to detect transient errors. Over 98% of all latched errors are detected. Depending on the error location, errors can be removed by instruction retry or by software intervention without loss of context. A program flow control mechanism is provided to detect execution anomalies due to undetected errors. To further increase the error-detection coverage, each device can be operated in master/checker mode.

1 Introduction

On-board computers in spacecrafts are used in both life-critical and mission-critical applications, where continuous operation over the mission lifetime is crucial. To achieve the required reliability and availability, various fault-tolerance techniques are used. A common approach in un-manned spacecraft is to use two self-checking computers in a cold stand-by configuration. For this configuration to be successful, the self-checking computers must have high error-detection coverage, combined with low detection latency and strong error isolation. However, the error-detection mechanisms implemented so far have been limited to periodic self-tests, checkbits on memory and a watchdog.

In 1990, the European Space Research and Technology Centre (ESTEC) began a program to develop the ERC32 radiation-tolerant 32-bit processing core. The ERC32, which is still in development, is to be a high-performance, general-purpose scalar computer for space-based applications. Rather than only develop a microprocessor, the ERC32 includes all the required functionality to form an embedded computer and to host a real-time operating system. The core is configured for a particular application by adding external memory and application specific I/O devices.

Various different processor architectures were considered, both CISC and RISC. After two independent studies [1,2], the SPARC architecture was chosen, mainly because it is a widely supported, open architecture available without license requirements. Although a proprietary architecture could offer a higher level of customisation, the risk of choosing a unique, single-source architecture was considered too high.

Considering the increased complexity of a 32-bit computer, an error-detection scheme with higher coverage than used in previous on-board computers was desirable. This paper describes the error-detection scheme implemented in the ERC32, and an ERC32-based self-checking computer.

2 General concept

In fault-tolerant systems, faults are either detected or masked. In life-critical space applications, faults are typically masked, using a pool of computers in NMR configuration. In unmanned missions, error-detection and recovery/reconfiguration is preferred, since the redundancy overheads are smaller. If errors are detected, a system can either recover from them or reconfigure around them.The fault-tolerance in ERC32 is based on error-detection and recovery/reconfiguration with the following characteristics:

- **Concurrent error-detection.** Errors are detected concurrently with ERC32 operation, without any special test instructions.
- **Error isolation.** Detected errors do not propagate to the outside system, thereby corrupting its state and function.
- **Error handling.** In case of transient errors, a correct state is restored and operation is resumed with minimum delay. In the case of permanent errors, the ERC32 halts and reports this to the outside system.

Several investigations have shown that transient errors largely prevail over permanent [3]. In the space environment, transient errors are mainly caused by heavy ions, and in low-earth orbit also by protons. The errors are manifested as bit flips in registers or memory elements. The sensitivity for these errors, or single event upsets (SEUs), depends both on the manufacturing process and design style, and can usually only be derived through testing of the actual device. Tests of devices manufactured in the same process as the ERC32 chip-set, indicates that the 3-chip processing core could experience up to one SEU induced error per month in geostationary orbit under worst-case conditions[4]. In comparison, the failure rate for permanent errors, calculated from MIL-HDBK-217F, is less than one error in 100 years.

The ERC32 error-detection scheme is based on the assumption that transient errors, mainly caused by SEUs, occur at least one order of magnitude more frequent than permanent errors. They are also more likely to occur in dense sequential blocks such as register files than in combinatorial logic. The following error-detection functions are provided in the ERC32:

- Parity checking on register files
- Parity checking on other critical registers
- Parity checking on external address, data and control bus
- Program flow control
- Master/checker mode for 100% error-detection coverage

An other important consideration is software compatibility. The error-detection and handling scheme is fully compatible with the SPARC V7 ISA, no registers or instruction have been added or modified.

In space applications, special attention has to be given to error recovery. Spacecrafts can operate autonomously without ground coverage for several days, and automatic error isolation and removal is thus crucial for mission success. Rather than reconfiguring at system level after each transient error, the ERC32 supports local error isolation and recovery. The detected errors are handled according to the following rules:

- If the error can be isolated without corrupting the system state then execution shall continue and necessary clean-up operations be performed.
- If the error cannot be isolated then the ERC32 shall inform the above layer and then restart itself.
- If a permanent error occurs such that execution cannot continue, then the ERC32 will halt and signal its condition to the above layer.

3 ERC32 architecture

The ERC32 consists of three devices; a SPARC integer unit (IU), a floating point unit (FPU) and a memory controller (MEC). The ERC32 interfaces directly to external memory and IO devices. The MEC includes all system functions required to form an embedded computer and to host a real-time operating system. The most important features are:

- Address decoding and memory interface
- Interrupt controller
- Block protection unit
- 32-bit SEC/DED EDAC
- Two 32-bit timers
- Two UARTs
- Boot prom interface
- DMA interface
- Error manager
- Watchdog

The ERC32 does not use a cache memory, it runs directly from a fast, SRAM-based main memory. There is no memory management unit (MMU), address translation and paging is usually not used in space-based embedded systems.

3.1 Integer unit

The integer unit (90C601E) is based on the 7C601 from Cypress Semiconductors. It has a four stage pipeline consisting of a fetch, decode, execute and write stage [5]. A total of 140 32-bit registers are accessible to the programmer, divided into 136 general and four special purpose register. The SPARC architecture uses register windowing, the general purpose registers are divided into windows of 24 registers, with an overlap of eight. Only one window at a time is accessible, selected through the Current Window Pointer (CWP) in the Processor Status Register (PSR).

Two types of exceptions (traps) are supported, synchronous

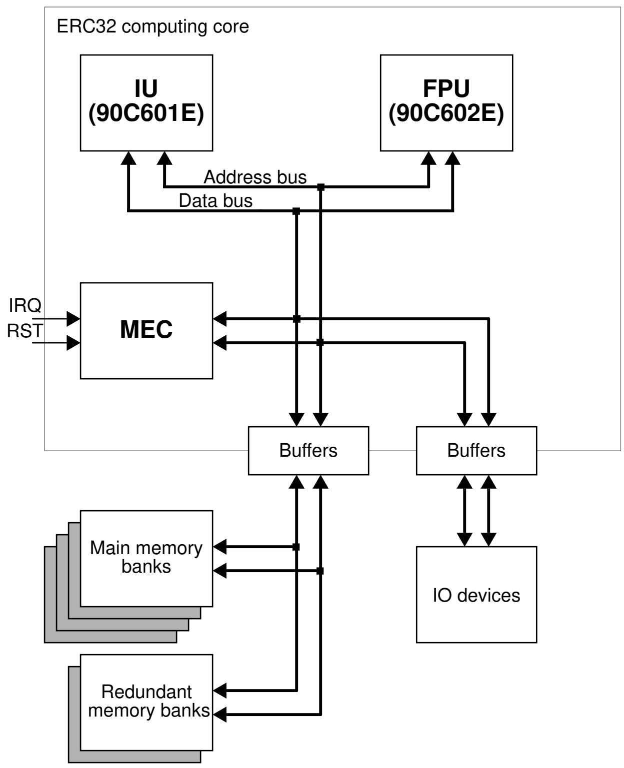

Figure 1: ERC32 architecture

and asynchronous. The asynchronous traps are generated by external interrupts and can be masked, while the synchronous traps originate from internal events and cannot be disabled. Once a trap is taken, further traps are disabled. If a new synchronous trap occurs while traps are disabled then the processor enters error mode and halts. During the trap operation a new window is allocated, the program counters (PC and nPC) are copied into two local registers and the *tt* field in the trap base register (TBR) is updated to reflect the trap type. The processor then branches to the appropriate exception handling routine as indicated by the TBR.

For most synchronous traps, the trap criterion is examined during the execute stage of each instruction and the trap is taken before the instruction reaches the write stage. A trapped instruction therefore does not change the processor status. This feature is used for handling errors; when an error is detected, the failing instruction is trapped before the processor state is further corrupted.

General purpose registers. The register file, containing the general purpose registers, is provided with one parity bit per register. The parity bit is generated and written together with the data in the write stage of the pipeline. The bits of adjacent registers are physically interleaved to reduce the probability of multiple errors in one register caused by a single SEU. The three-port register file is accessed during each cycle, but the parity is only checked if the fetched values are used by the current instruction.

Special purpose registers. The four special purpose registers (WIM, TBR, Y and PSR) are divided into a number of bit fields. The bits in each bit field are updated together and each bit field is provided with one parity bit. The parity bit is generated and written when the field is updated and checked when the field is used in an operation. Some fields are used in every instruction, and are consequently checked continuously.

Temporary registers. There are several internal registers used for instruction decoding and data pipelining. A majority of those are provided with parity bits. The parity of these registers is checked during each instruction.

Table 1 shows the number of latches provided with parity bits versus the total number of latches. As can be seen, more than 98% of all latches in the IU are covered.

Module	# latches	# protected	ratio
Register file	4352	4352	100%
Main datapath	852	812	95.3%
PSR,Y,WIM,TBR	300	300	100%
Temporary regs.	585	545	93.2%
Total	6,089	6,009	98.7%

Table 1: IU parity protection summary

External bus parity. To check the integrity of the external address bus an address parity bit is generated. The address bus parity is not checked by the IU since it is an output only. A parity bit on the external data bus is generated during stores and checked during loads and instruction fetches.

Three parity bits are used to protect the control buses. One bit contains the parity of the control signals going from the IU to the FPU (checked by the FPU), one bit contains the parity of the control signal going from the FPU to the IU (checked by the IU), and one bit that contains the parity of the remaining output control signals (checked by the MEC). The remaining input control signals are not protected; they can be generated by different external units, and a unified parity bit would be difficult to generate

Program flow control. To complement the register-targeted error-detection methods, a program flow control functions is included in the IU using the embedded signature-monitor technique (ESM). The concept is to calculate a signature from

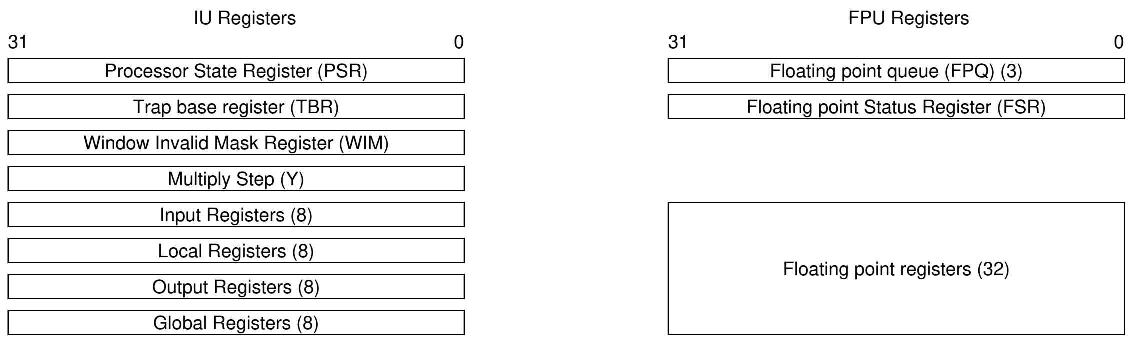

Figure 2: SPARC V7 programming model

each executed instruction and to compare this signature at appropriate points with a predefined checksum, to insure that the correct instructions have been executed.

Flow control is implemented by XOR-ing all instruction codes into a signature until a check-point instruction is reached. During the check-point instruction, the calculated signature is compared with the reference checksum (calculated by the compiler), contained in the check-point instruction. If a mismatch is detected, an error trap is immediately taken. The check-point instruction also resets the signature generator.

To preserve software compatibility with the SPARC ISA, the check-point instruction is implemented as a modified NOP. The modification is minor; the original NOP is a SETHI %g0, 0, the modified is SETHI %g0, CHK_SUM. A program using flow control is divided into branch-free blocks ended by a branch and a check-point instruction in the branch delay slot. To insure compatibility with software compiled without checksum insertion, the original NOP will disable the subsequent checking. Checking is also disabled when taking a trap or returning from a trap (RETI).

Error handling. Adhering to the general exception mechanism, the internal error-detection logic is evaluated during the execute stage of each instruction. If an error is found, an error trap is taken. The error traps are divided into six types, grouped after error location and possible recovery action (table 1).

Error group	Error description	Trap type
1	Restartable, precise error	0x60
2	Non-restartable precise error	0x61
3	Restartable, late error	0x62
4	Non-restartable, imprecise error	0x63
5	Register file error	0x64
6	Program flow control error	0x65

Table 2: Error traps

A restartable, precise error is defined as an error which can be removed by retrying the failing instruction and for which the saved PC and nPC in the trap window indicates the correct address of the failing instruction. Recovery is performed by simply returning from the trap routine, which will resume execution at the location of the failing instruction. Errors of this type originate from parity errors in the temporary registers. When the failing instruction is retried, these registers are reloaded, and the error is effectively removed.

Non-restartable, precise errors are errors which will not be removed if the failing instruction is re-tried, but where the failing instruction is known (correctly saved PC and nPC). Removing the error will require software intervention, typically by restarting the current task. Since the failing instruction is identified, the error is isolated and will not propagate. This type of errors originate from parity errors in the user-visible special purpose registers (PSR,Y,WIM,TBR).

The most serious type of errors are non-restartable, imprecise errors. These errors are not removable by instruction retry, and cannot be tied to a particular instruction. Error isolation is still guaranteed, these errors affect all instructions and the first instruction in the trap handler will also encounter the error and will cause the IU to go to error mode and halt. A reset is the only way to recover from these errors.

Initialisation. At power-up, the register check bits are not set and have to be initialized by software. Since registers are only checked when used, no special initialisation mode needs to be entered and registers (and check bits) can be initialized in the same way as in a normal SPARC processor. Care has to be taken not to read a register before it has been written and its check bits initialised. However, using registers before they are initialized is normally not recommended even without error checking.

3.2 Floating-point unit

The floating-point co-processor (90C602E) is based on the MEIKO floating-point core. It is tightly coupled to the integer unit which fetches and decodes all floating-point instructions. The floating-point instructions are started by the IU using the INS1/INS2 signals and then execute independently inside the FPU. If an exception (e.g. overflow) occurs during the execution of an floating-point instruction, the FPU will assert $\overline{\text{FEXC}}$ and enter pending_exception state. The IU will recognize the floating-point exception at the start of the following floating-point instruction and take a floating point trap. The exception type will be indicated in the *ftt* field in the floating-point status register.

The FPU error-detection scheme is similar to the IU scheme. All registers are provided with parity bits, and FPU generates and checks parity bits for all buses (address, data and control). The error-handling is slightly different; the FPU cannot handle the detected errors on its own, but flags them to the IU which have to take corrective measures. When and error is detected, the FPU enters pending_exception state and indicates the error type in the *ftt* in the floating point status register. Three types of errors are defined; restartable error, non-restartable error and data bus error.

A restartable FPU error is defined as an error which was detected before the FPU state was changed, and where the failing instruction can be re-executed. These error originate from instruction decode and data pipeline registers, or from a control bus parity error. A non-restartable error is an error which was detected after the FPU state was changed, and can therefore not be removed be re-executing the instruction. A data bus error indicates a bus parity error during a floating point load instruction. This error does not affect the FPU state and the load instruction can be re-executed.

Module	# latches	# protected	ratio
Register file	1024	1024	100%
Datapath	756	756	100%
Temporary regs.	406	406	100%
Total	2,186	2,186	100%

Table 3: FPU parity protection summary

At power-up, the register check bits are not set, and have to be initialized by software. As for the IU, no special initialisation mode needs to be entered and the registers (and check bits) can be initialized in the same way as in a normal SPARC FPU. Again, care has to be taken not to read a register before it has been written and its check bits initialised.

3.3 Memory controller (MEC)

The MEC implements important system support functions such as chip select decoding, waitstate generation, EDAC, timers and USARTs. Error-detection is implemented by providing each MEC register with a parity bit which is continuously checked. The parity of the external address, data and part of the control bus is checked by the MEC.

The MEC also contains an error manager, where the error signals from the IU, FPU and the MEC itself are sensed. For each error type, the error manager can be programmed to either ignore the error, reset the ERC32 or halt.

3.4 Error-detection overhead

The introduced error-detection scheme has a relatively low overhead; less than 15% in terms of silicon area. The timing impact is basically the maximum delay through a 32-bit parity generator, approximately 8 ns on a 1 µm CMOS technology. The flow control scheme gives a negligible hardware overhead, but results in a run-time performance degradation. If a program consists of 10% branches, and the schedulability of the delay slot is 50%, the total performance degradation is 5% (0.1 * 0.5). Five additional pins are added to the IU and FPU for bus parity and error flagging.

4 System configurations

4.1 Duplex configuration

All three devices contain comparators and logic to be able to work in a duplex (master/checker) configuration. The master and checker have all inputs and outputs connected together, but only the master drives the outputs. During each cycle, the slave compares the values of the outputs (driven by the master) with its own internal values. A mismatch is indicated on the CMPERR output.

4.2 Modular fault-tolerance

The ERC32 can be used in three configurations, simplex, duplex and duplex with masking. In the simplex configuration, errors are only detected with the built-in error detection functions, and recovery is based on instruction retry or software intervention. In the duplex configuration, errors are also detected by the checker devices, giving almost 100% error-detection coverage. Recovery is performed as in the simplex configuration.

In a duplex configuration with masking, the ERC32 is provided with an additional reconfiguration unit (RU). The RU monitors the two error signals from each core device, HWERR and CMPERR. HWERR indicates that an error was detected by the internal error-detection, CPMERR indicates that the checker found a output mismatch. When an internal error is detected, it takes in the most cases two clock cycles to propagate to the outputs. If the reconfiguration unit detects HWERR before CMPERR is asserted from any device, it can halt the system, switch the master and checker, and resume the operation, thereby masking the error. The ERC32 can continue operation without interrupts, but with reduced error-checking coverage. The master and checker can be re-synchronised by performing a soft-reset at a non-critical point in the mission.

Config	Error-detection coverage	Fault-tolerance	Overhead
Simplex	> 98% of all latched errors	Instruction retry Software action	15%
Duplex	100%	Instruction retry Software action	100%
Duplex + Masking	100%	Device switching	100% + RU

Table 4: Configuration characteristics

5 Conclusion

A SPARC-based computing core with internal error-detection has been described. The error-detection scheme has several benefits:

- The processor stays fully compliant with the SPARC architecture, even with error-detection enabled.
- Detected errors are isolated, to prevent further error propagation.
- The up-time is increased by ignoring errors from unused registers or bit fields.
- Error-handling is transparent to the application software and mapped on the exception mechanism.
- Fault-tolerance can be achieved with low overhead.

- The error detection scheme is expandable without requiring additional registers or instructions if further error detection mechanisms are introduced.
- The implementation overhead is within the limits of space-qualified device technology.

6 Acknowledgments

This work was performed under ESTEC contract 9848/92/NL by a consortium consisting of Matra MHS (F), Matra Marconi Space (F) and Saab Ericsson Space (S).

References

[1] "RISC Evaluation study - final report", SAAB Space 1990, ESTEC report reference CR(P)3188.

[2] "RISC architecture and technology - final report", SAGEM 1990, ESTEC report reference CR(P)3190.

[3] "Architecture of fault-tolerant computers", D.P. Siewiorek, Proceedings of IEEE, 79(12), 1991.

[4] "Memory design considerations in space grade VLSI", T.Bion and A.Dantec, MATRA MHS, 1993.

[5] "SPARC RISC User's guide", Matra MHS SA, 1992

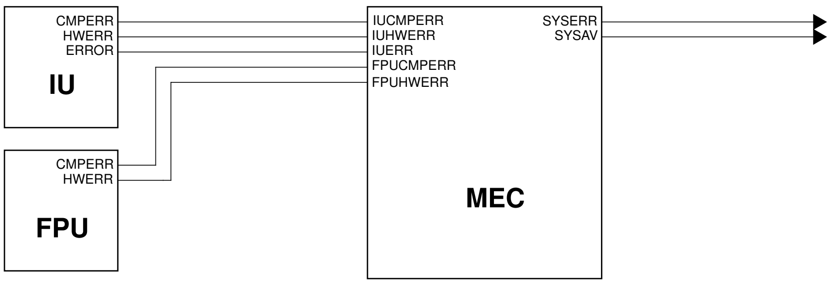

Figure 3: ERC32 simplex configuration

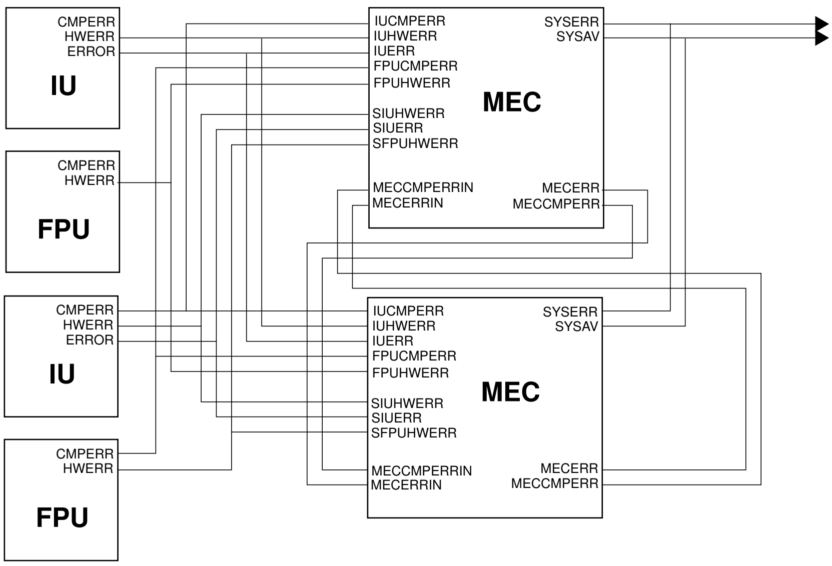

Figure 4: ERC32 duplex configuration