Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction

I.D. Elliott, BSc, AMIEE
I.L. Sayers, PhD, AMIEE

Indexing terms: Microprocessors, Errors and error analysis

Abstract: The need for reliable integrated circuits is becoming of paramount importance as they are increasingly used in a range of safety critical applications or domestic products. In the past reliability has been achieved at the IC level by comprehensive testing of the device after manufacture. The use of scan design and BILBO techniques have assisted designers in achieving the necessary high test coverages with little effort. However these methods only address the problem of testing for permanent faults after fabrication or periodically during the lifetime of a system. These 'classical' techniques do not tackle the more serious problem of intermittent faults, which will come to dominate VLSI circuits as device geometries decrease. To deal with intermittent faults and maintain reliable operation concurrent test methods need to be used. The paper will present one possible method of detecting and correcting single intermittent faults that occur during normal operation and also assist the designer in post fabrication testing. The chosen technique uses information redundancy in the form of a SEC/DED Hamming code and will be illustrated by the design of a 32-bit CMOS RISC processor. The processor is capable of detecting and correcting errors arising from faults as they occur without the need to halt normal operations or recourse to any specialised software. A detailed appraisal of the costs involved in using this technique will be given in terms of the extra silicon area needed and the reduction in throughput of the processor.

1 Introduction

The pervasive use of integrated circuits in an ever increasing number of critical applications has brought with it a need for increased reliability and durability. Several possible solutions exist to improve overall reliability, these range from complex 'burn-in' procedures to eliminate weak devices before inclusion in the circuit, to fault tolerance techniques, where faults within a system can be masked by the use of redundant hardware or software. The use of 'burn-in' procedures can cause perfect devices to age prematurely; they may then fail when in service, usually without warning and with possibly catastrophic effect. Hardware fault tolerance techniques on the other hand offer a possible solution by masking faults and allowing a system to degrade gracefully or remain serviceable until repairs can take place. A further benefit of using hardware fault tolerance techniques is the need to eliminate the potentially serious problem of intermittent faults in future very small geometry integrated circuits. The detection of intermittent faults would have the added advantage of simplifying the maintenance of the next generation of potentially very complex systems. It has been estimated that 75% of all errors in future devices will be temporary in nature [1], thus fault tolerance techniques must be considered for the next generation of integrated circuits which may have reduced voltage levels, decreased noise margins and smaller geometries, all of which contribute to the problem of intermittent faults.

The current test strategies employed in the IC industry are aimed mainly at detecting permanent faults directly after fabrication by allowing the test engineer greater access to the internal nodes of the circuit. In some cases the internal test structures of the IC can be configured to allow periodic testing of a system by suspending normal operations to apply carefully preselected test patterns. Obviously this has several drawbacks:

(*a*) When a permanent fault occurs it may be some time before the fault is detected.

(*b*) Using this technique, intermittent faults will not be detected unless they occur during the system test.

Therefore 'classical' test methods are inadequate for reliable operation.

The detection of faults during normal system operation requires the use of concurrent error detection (CED) techniques to make the system fault-tolerant. CED techniques are capable of detecting an error immediately it occurs, therefore both permanent and temporary errors can be dealt with before they cause any serious problems. If the chosen technique then indicates which device is faulty, the mean time to repair (MTTR) can be significantly reduced thus enhancing the overall performance of the system. Unfortunately, to date most of the techniques proposed to achieve the CED goal have simply been 'paper' designs. One of the very first attempts to quantify the extra hardware overheads involved in the design of a reasonably complex VLSI circuit was in 1980 by Sedmak and Lierbergot [2] when they presented a complete processor and system design. In 1984 Marchal

Paper 6981E (C2), first received 30th September 1988 and in revised form 6th April 1989

I.D. Elliot is at the Department of Electrical and Electronic Engineering, Newcastle upon Tyne Polytechnic, Ellison Building, Newcastle NE1 8ST, United Kingdom

I.L. Sayers is at the Department of Electrical and Electronic Engineering, Newcastle University, Newcastle NE1 7RU, United Kingdom

et al. [3] presented the first critical appraisal for the application of CED techniques to a complete microprocessor, in this case the MC68000. Yen *et al.* [4] have demonstrated how a CED technique may be applied to a microprogram control unit which was slightly restricted in its application. The author [5] has also demonstrated how a CED technique, in this case a residue code, can be applied to a whole range of VLSI devices. This culminated with the successful fabrication of a complete 8-bit NMOS datapath [6]; this is believed by the authors to be one of the few times that a practical technique has gone beyond the 'paper' design stage. All the above techniques have one drawback: they are only capable of detecting errors. Therefore to continue with normal system operations some alternative higher level system needs to correct the fault before the computation or operation can continue. A technique that is capable of both detecting and correcting a fault without reference to a higher level would thus be advantageous provided implementation costs, in terms of silicon area and the overall speed of the device could be minimised.

This paper will discuss the design of a 32-bit fault-tolerant RISC (FT-RISC) processor in 2 μm 2-layer CMOS. The processor is capable of not only detecting single faults but also of correcting faults without the need for extra specialised hardware or software; this is achieved by the use of a SEC/DED Hamming code [7]. The Hamming code was chosen since it is easy to implement and the checking elements are very small. A further advantage of using a Hamming code is that memory arrays are usually protected using this form of code, consequently the checkbits of the code already exist and are normally discarded once they enter the processing elements; the processor will thus be able to make use of this information. A RISC processor was chosen to illustrate the technique as it is reasonably complex and performs a wide variety of operations; also RISCs have become a major force in current computer design strategies. For simplicity the processor is considered to consist of two parts, a datapath and a controller. The datapath is further subdivided into an information processing section and a Hamming code processing section. Thus the Hamming code section could be used in other designs if required. The concurrent error detection/correction (CED/C) technique to be discussed was first outlined in Reference 6. Although this technique is primarily aimed at providing reliable systems it is also capable of simplifying the initial test of a device.

2 Description of checking method

The processor datapath, shown schematically in Fig. 1, consists of an information processing section and a parity processing section; a word size of 32 bits was chosen for this implementation of the information section, therefore the number of parity bits required for a SEC/DED Hamming code is 7. A 32-bit architecture was chosen to give a more realistic analysis of the implementation costs involved and to produce a useful chip after fabrication. Table 1 shows how each of the seven parity bits can be generated from the information bits, a • indicates that the corresponding information bit is to be included in the modulo 2 sum used to generate that parity bit. To detect a double error an overall parity bit P_0 is required. The P_0 parity bit is the overall modulo 2 sum of the information bits and the parity bits, however since the parity bits contain some of the information bits and modulo 2 addition is equivalent to modulo 2 subtraction cancellation

Table 1: Hamming encoder bit table

Bit position	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	
Data bit				0		1	2	3		4	5	6	7	8	9	10		11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26		27	28	29	30	31		
Parity bit	0	1	2		4				8								16																32								
P_1				•		•		•		•		•		•		•		•		•		•		•		•		•		•		•		•		•		•		Even
P_2				•			•	•			•	•			•	•		•			•	•		•			•	•			•	•			•		•		•	Even
P_4						•	•	•			•		•	•	•	•			•	•	•		•	•			•		•	•	•				•	•				Even
P_8										•	•	•	•	•	•	•									•	•	•		•	•	•									Odd
P_{16}																		•	•	•	•	•	•	•	•	•	•												Odd
P_{32}																																		•	•	•	•	•	•	Even
P_0				•		•	•	•		•	•	•	•	•	•	•		•	•	•	•	•	•	•	•	•	•	•	•	•	•	•		•	•	•	•	•		Even

• Indicates data bit is included in module 2 sum

occurs within the equation for P_0, such that only 18 of the information bits are required in the final sum. Inspection of Table 1 shows that if there are an even number of

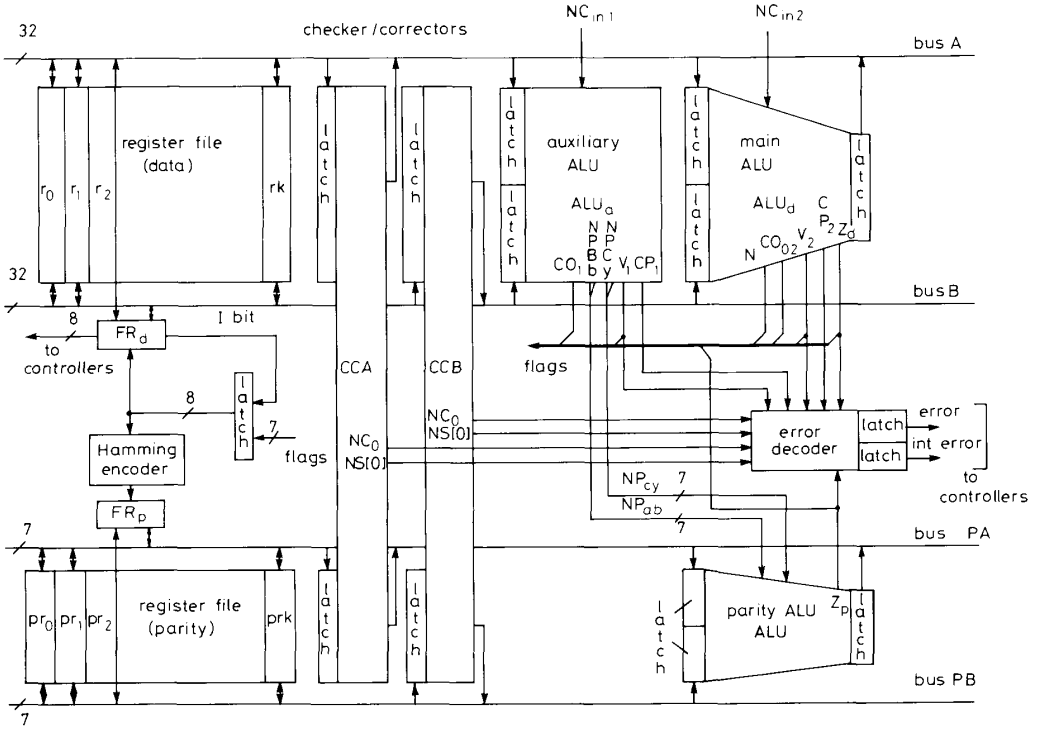

Fig. 1 *Self-checking and correcting 32-bit datapath*

• in any column (excluding the P_0 row) then the corresponding information bit is included in the P_0 sum. This combination of information and parity bits forms a valid Hamming code with an overall even parity. The datapath makes use of two checker/corrector units (CCA and CCB) to check and if possible correct errors in the data being transferred by buses A and B. Contained within each checker/corrector unit is a Hamming decoder which is used to decode the 39-bit code into a seven-bit syndrome.

$$\text{syndrome} = C_{32}\,C_{16}\,C_8\,C_4\,C_2\,C_1C_0$$

The syndrome bits are generated in a similar manner to that used to produce the parity bits, except that the parity bits themselves are included in the modulo 2 sum along with the relevant information bits. To form C_0, the overall parity of the Hamming code, all of the information and parity bits are summed modulo 2; the cancellation which is used to produce a simpler encoder cannot be exploited in the decoder since the exclusion of any of the information bits would result in those bits being uncovered. When decoded the six most significant bits of the syndrome indicate the position of the erroneous information or parity bit, in the event of a single bit error. If a double bit error had occurred the syndrome would be nonzero, however, the overall parity bit would be zero, indicating that two bits had changed.

Table 2 lists the four possible cases for the syndrome and the necessary steps to take for each case; the column containing $NS[0]$ represents the active low zero output of the syndrome decoder. In the event of a single bit error the syndrome decoder output is used to toggle the erroneous bit to the correct state. A double error is indicated by a nonzero syndrome and unchanged overall parity; in this case an error flag must be raised and corrective action initiated. If the syndrome decoder $NS[0]$ output is active and the overall parity is odd, the bit in error is P_0. The basic function of the parity bit processing section of the datapath is to maintain the relationship between the information and parity bits of the result such that they constitute a valid Hamming code following an arithmetic or logical operation, i.e. the parity processing section is used to predict the parity of the result produced by the information processing section; this is achieved as shown in the following sections [10]. For the purposes of demonstrating the method, an eight-bit information word will be used which requires five parity bits and therefore clarifies the description.

$$I_7\,I_6\,I_5\,I_4\,I_3\,I_2\,I_1I_0 \quad P_8\,P_4\,P_2\,P_1P_0$$

Table 2: List of possible checker/corrector syndromes

Syndrome decoder output $NS[0]$ (active low)	Syndrome overall parity bit NC_0 (1 = even parity) (0 = odd parity)	Necessary action
1	1	Even no. of errors. Raise error flag
1	0	Correct single error
0	1	No errors
0	0	Toggle overall parity bit P_0

Consider two operands A and B, which are 13 bits each, split into information (I_A and I_B) and parity (P_A and P_B) sections. Now, consider the following operations using these operands.

2.1 Logical operations

2.1.1 AND

The result of a bitwise *AND* operation is

$$I_{R_i} = I_{A_i} \cap I_{B_i} \quad i = 0, 1, \ldots, 7$$

where I_R is the result of the *AND* operation. The parity of this result I_R is given by

$$P_R = P_{AB}$$

where P_{AB} is the parity of the result produced by *AND*ing I_A with I_B.

Example:

	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
I_A	1	0	1	1	0	1	1	1
I_B	1	0	0	1	1	0	1	1
I_R	1	0	0	1	0	0	1	1

P_8	P_4	P_2	P_1	P_0	
1	1	1	1	0	P_A
0	1	0	0	0	P_B
0	0	1	1	0	P_R

Note that $P_R \neq P_A \cap P_B$ and must be generated when required.

2.1.2 OR

Using the above result it is possible to derive a relationship for the parity bits that does not depend directly on generating the OR, but only on the *AND* parity bits as follows:

$$P_R = P_A \oplus P_B \oplus P_{AB}$$

Example:

	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
I_A	1	0	1	1	0	1	1	1
I_B	1	0	0	1	1	0	1	1
$I_A \cap I_B$	1	0	0	1	0	0	1	1
I_R	1	0	1	1	1	1	1	1

P_8	P_4	P_2	P_1	P_0	
1	1	1	1	0	P_A
0	1	0	0	0	P_B
0	0	1	1	0	P_{AB}
1	0	0	0	0	P_R

2.1.3 EXCLUSIVE OR

The parity bits for the results of an *EOR* operation can be generated without recourse to P_{AB} as follows:

$$P_R = P_A \oplus P_B$$

Example:

	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
I_A	1	0	1	1	0	1	1	1
I_B	1	0	0	1	1	0	1	1
I_R	0	0	1	0	1	1	0	0

P_8	P_4	P_2	P_1	P_0	
1	1	1	1	0	P_A
0	1	0	0	0	P_B
1	0	1	1	0	P_R

2.2 Arithmetic operations

2.2.1 Addition

The parity of the result for the addition of two numbers is given by

$$P_R = P_A \oplus P_B \oplus P_{CY}$$

where P_{CY} are the parity bits generated from the carries of the addition operation.

Example:

	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
I_A	1	0	1	1	0	1	1	1
I_B	1	0	0	1	1	0	1	1
Carries	0	1	1	1	1	1	1	0†
I_R	1	0	0	1	0	0	1	0

P_8	P_4	P_2	P_1	P_0	
1	1	1	1	0	P_A
0	1	0	0	0	P_B
1	1	0	0	0	P_{CY}
0	1	1	1	0	P_R

† represents carry in. The carry out from column 8 is not used in the generation of P_{CY}.

2.2.2 Bitwise complementation

The complement of a number A is formed by inverting each bit. The parity of the complemented number is obtained by inverting those parity bits of P_A which are the result of summing an ODD number of information bits, as inverting an even number of parity bits in modulo 2 arithmetic will produce an identical result. In the present example P_1 and P_2 are the ODD parity bits.

Example:

	I_7	I_6	I_5	I_4	I_3	I_2	I_1	I_0
I_A	1	0	1	1	0	1	1	1
I_R	0	1	0	0	1	0	0	0

P_8	P_4	P_2	P_1	P_0	
1	1	1	1	0	P_A
1	1	0	0	0	P_R

For a 32-bit information word with 7 parity bits, P_8 and P_{16} are inverted to produce P_R.

Using the basic functions described above, all of the required datapath operations can be implemented. Table 3 shows a complete listing of the operations that are supported by the information and parity processing elements of the self-checking datapath. The operations increment (INCA), decrement (DCRA) and negate (NEGA) are synthesised by raising the carry-in to logic one while performing an add (ADD) or subtract (SUB); for the negate (NEG) operation the operand is complemented before addition takes place. Forcing the carry-in to logic one eliminates the need for a constant register containing unity.

The self-checking datapath does not directly support a shift left operation. However, a single bit left shift can be performed using an ADD (or ADDC) operation with A equal to B; A and B are the operands latched into the ALU from the two buses during an instruction, therefore by driving both buses from the output of the same register B can be made equal to A. All operations which involve addition or subtraction use the parity of carries signal P_{CY}; similarly the logical *AND* and *OR* operations

Table 3: Operations supported by ALU

Mnemonic	Operation (information bits)	Operation (parity bits)
ADD	$A+B \rightarrow R$	$P_A \oplus P_B \oplus P_{CY} \rightarrow P_R$
ADDC	$A+B+C_{IN} \rightarrow R$	$P_A \oplus P_B \oplus P_{CY} \rightarrow P_R$
SUB	$A-B \rightarrow R$	$(P'_A \oplus P_B \oplus P_{CY})' \rightarrow P_R$
SUBC	$A-B-C_{IN} \rightarrow R$	$(P'_A \oplus P_B \oplus P_{CY})' \rightarrow P_R$
SUBR	$B-A \rightarrow R$	$(P_A \oplus P'_B \oplus P_{CY})' \rightarrow P_R$
SUBRC	$B-A-C_{IN} \rightarrow R$	$(P_A \oplus P'_B \oplus P_{CY})' \rightarrow P_R$
COMPA	$A' \rightarrow R$	$P'_A \rightarrow P_R$
COMPB	$B' \rightarrow R$	$P'_B \rightarrow P_R$
NEGA	$A'+1 \rightarrow R$	$P'_A \oplus P_{CY} \rightarrow P_R$
NEGB	$B'+1 \rightarrow R$	$P'_B \oplus P_{CY} \rightarrow P_R$
INCA	$A+1 \rightarrow R$	$P_A \oplus P_{CY} \rightarrow P_R$
INCB	$B+1 \rightarrow R$	$P_B \oplus P_{CY} \rightarrow P_R$
DCRA	$A-1 \rightarrow R$	$(P'_A \oplus P_{CY})' \rightarrow P_R$
DCRB	$B-1-R$	$(P'_B \oplus P_{CY})' \rightarrow P_R$
CLR	$0 \rightarrow R$	$0 \rightarrow P_R$
LITA	$A \rightarrow R$	$P_A \rightarrow P_R$
LITB	$B \rightarrow R$	$P_B \rightarrow P_R$
AND	$A \cap B \rightarrow R$	$P_{AB} \rightarrow P_R$
OR	$A \cup B \rightarrow R$	$P_A \oplus P_B \oplus P_{AB} \rightarrow P_R$
EOR	$A \oplus B \rightarrow R$	$P_A \oplus P_B \rightarrow P_R$

A and B are two 32-bit data values, R is 32-bit ALU result
P_A and P_B are 7-bit parity codes for A and B, respectively, P_R is parity of ALU result
P_{CY} is parity of carries $C_{in}, C_1, C_2, \ldots, C_{31}$
P_{AB} is parity of data value resulting from logically *AND*ing A with B
For complementation only parity bits P_8 and P_{16} are inverted

require the signal P_{AB} to produce the parity of the result. Both P_{AB} and P_{CY} depend on the information bits being supplied to the information ALU and so two Hamming encoders are required within the information processing section of the datapath to generate the P_{AB} and P_{CY} signals; these are located within the Auxiliary ALU (ALU_a). The encoders along with the checking and correcting hardware constitute the two points of interaction between the information and parity sections of the datapath, otherwise both sections operate independently.

3 Description of checked datapath and checking hardware

The self-checking and correcting datapath is shown schematically in Fig. 1. As previously discussed, the system is divided into two distinct sections, the information datapath using buses A and B to communicate and the parity datapath using buses P_A and P_B. The two sections interact with each other within the two checker/corrector units CCA and CCB and also when P_{CY} or P_{AB} are needed to predict the parity of the result. The use of two buses within both the information and parity sections of the datapath allows operations involving two operands to be executed within one bus cycle with the result written back during the next bus cycle; this architecture is similar to that used in Mead and Conway's OM2 machine [8]. During a typical instruction one or two operands are transferred from the register files into the ALUs, via the two checker/correctors. The information ALU (ALU_d) produces the 32-bit result, whereas the Auxiliary ALU (ALU_a) produces the parity P_{CY} of the carries and the parity P_{AB} of the *AND* operation; the latter may or may not be used in the calculation of the result parity depending on the type of instruction being executed.

The parity ALU (ALU_p) produces the 7-bit parity code for the result using a specific combination of the input parity signals. The result, along with its associated parity, is then driven back into the register file via checker/corrector A (CCA) where they are decoded to produce a syndrome, which can then be used to take corrective action in the event of a single error or flag a double error. The two checker/corrector units (CCA and CCB) latch the data to be checked during the initial part of a clock cycle and output the checked/corrected data at the start of the next cycle when the data is either operated on by the ALU or stored in a register. This strategy allows the checker/correctors to be located at any position within the datapath where they have access to both buses, also peripheral registers may be located with the same degree of freedom since they only need have access to either bus (A or B) to be able to communicate with a checker/corrector unit. Both the information and parity sections of the datapath include a register file for storage of data and parity bits, respectively; each register file contains 16 registers made up of register cells which have dual port read and write capabilities. A circuit diagram of one such register cell is shown in Fig. 2. The registers can all read either bus (A or B) and drive data onto either bus; as previously mentioned one register drives both buses for a shift operation.

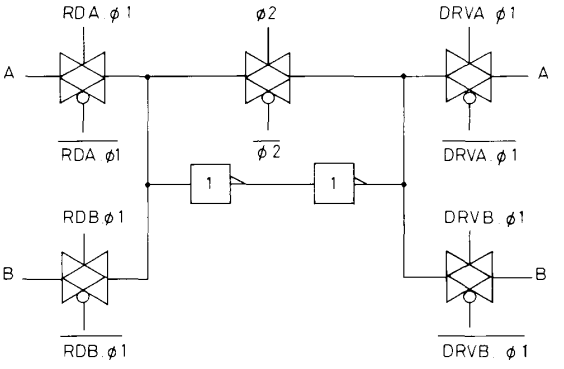

Fig. 2 *Dual-port, read/write register cell*

When external data is brought into the datapath the corresponding parity bits are also input, allowing the data to be checked before being stored in the register file; similarly both data and parity bits are made available at the output pins to allow external devices to also perform checking.

3.1 Main and auxiliary ALUs

The 32-bit result is produced by ALU_d during a machine cycle; to produce the result it is necessary to generate the carry information from the input data. The outputs of the main carry chain located within ALU_d cannot be used to generate P_{CY} owing to the fact that an error at the output of the main carry chain will produce an erroneous result in both the data bits and the parity of carries signal P_{CY}. The parity P_R of the result is dependent upon P_{CY} and so the combined output is a valid but incorrect Hamming code word. The incorrect but valid result and parity would then pass through the checker/corrector unaltered into the register file. Simulation has shown that the above problem will occur for all carry circuit errors in addition to errors in those signals which determine the carries, such as the carry generate signals.

A possible solution to this problem involves using two disjoint carry chains, the extra carry chain being supplied by a separate latch and residing within the auxiliary ALU (ALU_a). The duplicate carries are used to generate P_{CY} which means that an error in the main carry circuit output can only affect result bits. However, a potential

problem occurs when the auxiliary carry chain output contains a single error; this is due to the way in which P_{CY} is generated. From Table 1 it can be seen that each column contains an odd number of entries, i.e. 3 or 5; this means that a single input error will corrupt either 3 or 5 of the P_{CY} encoder output bits, resulting in an equal number of erroneous bits within the parity of the result. Such an error will force the checker/corrector to toggle the incorrect bit, resulting in possibly four errors.

The above problem can be overcome by checking the main and auxiliary carry chain outputs against one another; this is done by comparing the overall parity of the two carry signals CP_1 and CP_2 and flagging a 'carry error' if they differ. Using this method an odd number of carry errors can be detected. Logic diagrams of one bit slice of the main and auxiliary ALUs are shown in Figs. 3 and 4. Both the main and auxiliary carry chains are implemented as an array structure based on Brent and Kung's [9] regular adder layout; the so-called binary look ahead carry scheme offers a good compromise between speed and area, as well as having a regular structure which can be adapted to any word size. The worst-case carry delay corresponds to a delay of 16 gates in series, making the carry chain four times faster than a straightforward ripple carry chain. In the main ALU a logic circuit is used to produce the required carry generate and propagate signals which are fed into the carry array for addition; for logical operations the carry circuit outputs are suppressed by control line 'NCKL', which forces all carries to logic 1, and control lines CR_1 and CR_2 determine the logical function.

The use of two carry chains means that both the carry-in and carry-out signals are duplicated, providing a further degree of protection. The carry-in signals NC_{in1} and NC_{in2} are generated using a completely independent circuit and the two carry-out signals CO_1 and CO_2 are maintained within the internal flag register FR_d for use by two independent controllers.

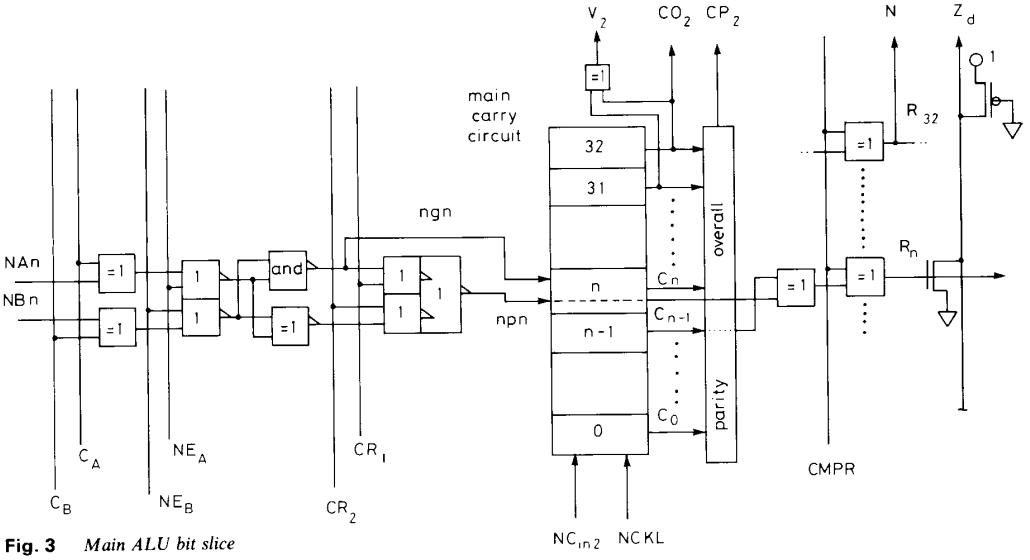

Fig. 3 *Main ALU bit slice*

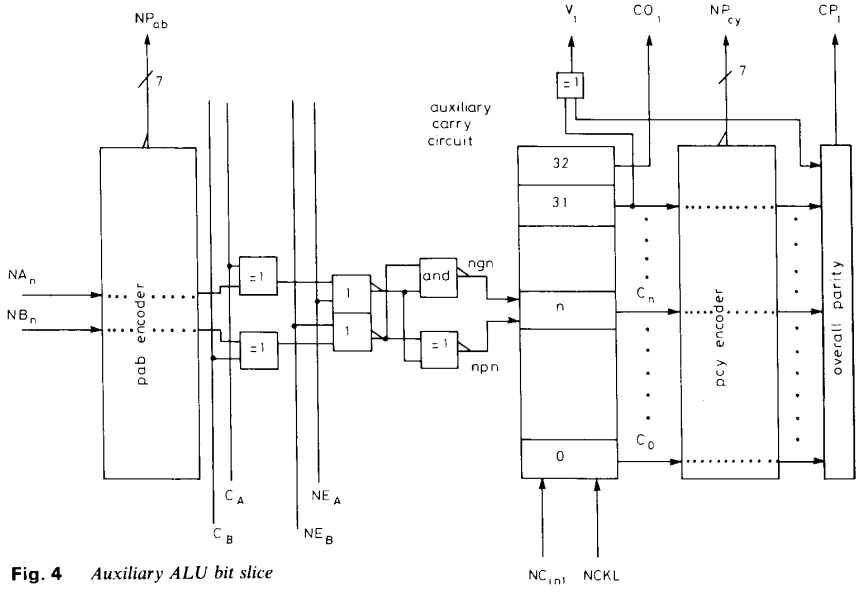

Fig. 4 *Auxiliary ALU bit slice*

3.2 Flags

The bit allocation of the flag register FR_d is shown in Fig. 5. This register along with an associated parity register FR_p are mapped into the general purpose register address space of the datapath; this allows the contents of FR_d to be checked and/or modified by the use of any instruction which can modify the bit pattern stored in a register (e.g. '*ANDI*', the *AND* immediate instruction).

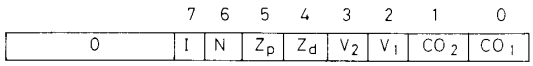

Fig. 5 *Flag register FRd bit map*

I: interrupt enable bit
N: sign bit
Z_d: zero bit (DATA)
Z_p: zero bit (PARITY)
V_2: main ALU two's complement overflow
V_1: auxiliary ALU two's complement overflow
CO_2: main ALU carry out
CO_1: auxiliary ALU carry out

In the proposed RISC design all arithmetic and logical functions affect all flags, and as these instructions modify the contents of registers within the register file a hardware trap is built in to the controller to prevent the contents of the flag register being updated if it is itself the destination of the ALU result. As shown in Fig. 5 the carry out, two's complement overflow and zero flags are all duplicated within the flag register owing to the fact that each is generated by separate hardware within the datapath. For example, the duplicate zero flag Z_p is generated by the parity bits directly since if $R = 0$ then $P_R = 0$.

To maintain the contents of the parity of flags register FR_p, a simplified Hamming encoder is used to produce a 7-bit parity code from the 8-flag bits. Every time the ALU performs an operation all the flags are latched into a temporary latch along with the interrupt enable bit (I) from the flag register itself; the reduced Hamming encoder then produces the updated parity bits which can be loaded along with the new flags into the FR_p and FR_d registers, respectively, under control of an update flags signals from the controllers.

3.3 Parity ALU

From Table 3 it can be seen that the parity ALU performs a selective modulo 2 addition of input partiy bits to produce the result parity P_R. Fig. 6 shows a logic diagram of one bit slice of ALU_p. Seven control lines are used to determine the components of the final sum output P_R. In certain operations the data bits are complemented, resulting in the need to produce the corresponding parity of the complemented data; this is achieved by simply inverting bits P_8 and P_{16} of the parity code using control lines CPA, CPB and CPR. All operations performed by ALU_p are bitwise modulo 2 additions, therefore single or double faults within the parity ALU will at worst result in single or double errors in P_R. The control information required by the main and auxiliary ALUs is identical to that required by the parity ALU; this results in the ability to check the outputs of each controller against one another using very simple hardware.

3.4 Hamming encoders

The function of the Hamming encoders within the ALU is to produce a 7-bit parity code from a 32-bit data input such that the 39-bit result is a valid Hamming code. The Hamming encoder is organised in such a way that a single fault at any point within the encoder circuit can at worst produce a single bit output error. This is achieved by the use of completely independent circuits to generate each parity bit output, as shown schematically in Fig. 7. The data bits enter the encoder on metal layer 1 of the 2-layer process, and the parity bits emerge from the top in complement form on metal layer 2. Each column of the encoder is designed to have a worst-case delay corresponding to the delay of 7 *EOR* gates in series.

3.5 Checker/corrector unit

A schematic diagram of a checker/corrector unit is shown in Fig. 8. The unit consists of three distinct sections; the syndrome generator, syndrome decoder and the corrector logic. The syndrome generator is identical to the Hamming encoder except that all are bits used in the generation of the overall parity bit C_0, and the parity bits P_1 to P_{32} are used in the generation of the Hamming syndrome bits C_1 to C_{32}. The syndrome decoder consists of 39 six-input *NAND* gates, the active low outputs of which are used to toggle the erroneous input bits using *EXNOR* error correcting gates. A two-input *OR* gate is used to detect the case where the overall parity bit P_0 is corrupted, i.e. $NS[0] = NC_0 = 0$ (see Table 2). Single bit faults at any position within the checker/corrector unit will result in a single bit output error in either the parity

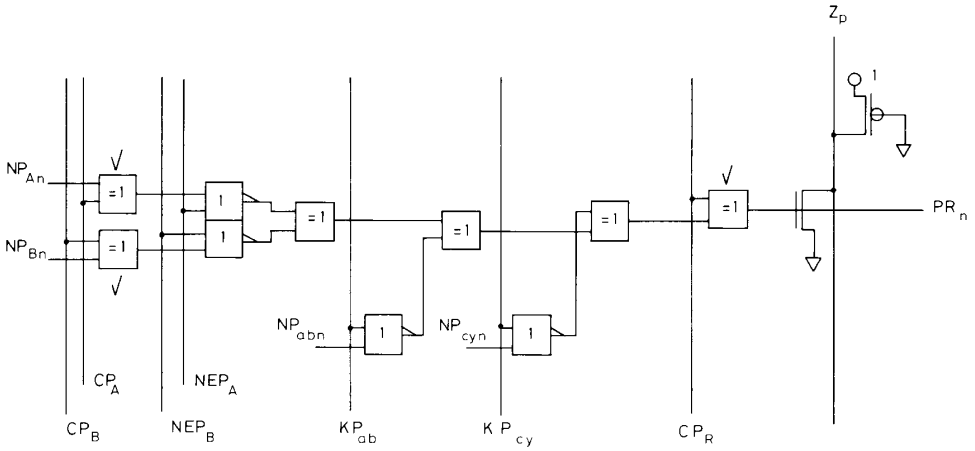

Fig. 6 *Parity ALU bit slice*

√ for NPA/B8 and 16 only

or information bits. If the data entering the unit has a single bit error in addition to the fault within the unit itself then a double error could occur at the output.

3.6 Error decoder

To maintain an ability to detect single faults at any position, an error decoder is used to generate a dual-rail coded error signal which will be used as both a diagnostic output from the device and as inputs to the datapath controllers so that the proper corrective action may be taken in the event of a double error. A logic diagram of the error decoder is given in Fig. 9. There are four sources of error flags from within the datapath:

(*a*) CCA
(*b*) CCB
(*c*) carry chains
(*d*) zero and overflow flags

It is important that the zero, overflow and carry flags are protected in this way since an undetected error in any of these flags will be compounded by the fact that the flag Hamming encoder outputs will also be incorrect. The condition of the syndrome output of each checker/corrector which signifies a double error is

$$NS[0] = NC_0 = 1$$

This indicates that errors have occurred, as the syndrome is not zero and the unchanged overall parity indicates a double error (or an even number of errors). The above

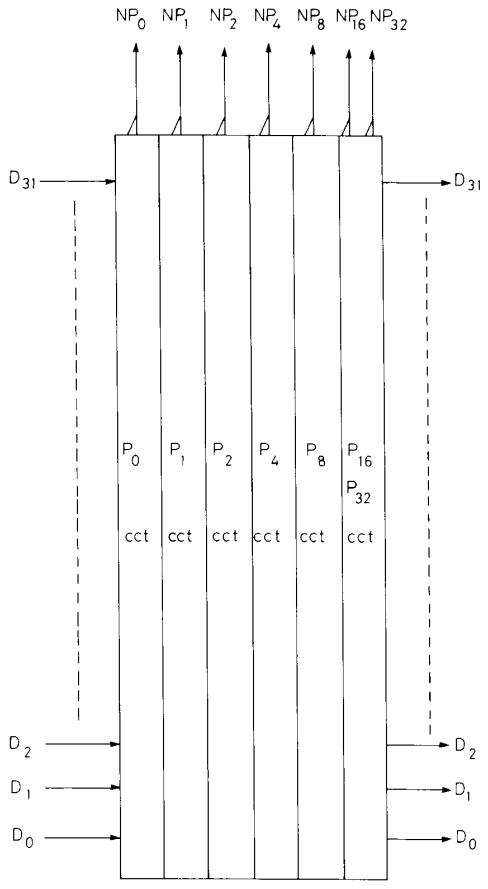

Fig. 7 *Schematic diagram of Hamming encoder*

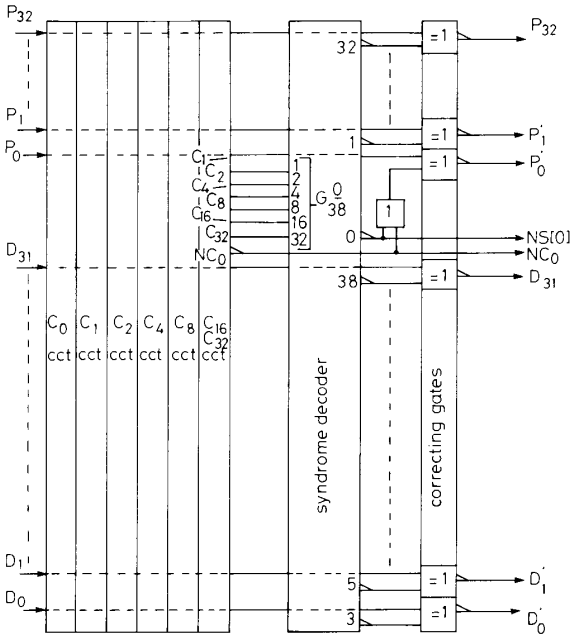

Fig. 8 *Schematic diagram of checker/corrector*

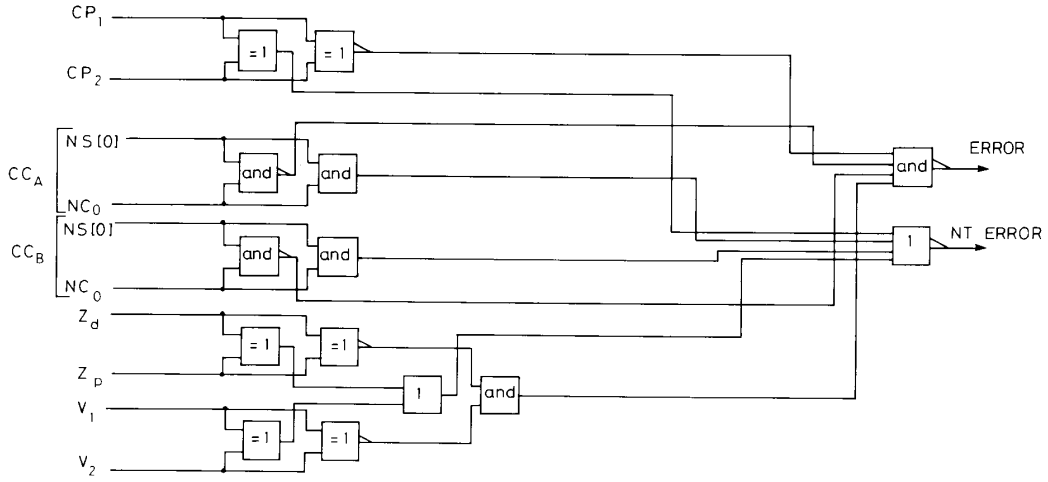

Fig. 9 *Logic diagram of error decoder*

condition is detected by an *AND* gate and a *NAND* gate to produce active high and active low error signals, respectively.

All error flag sources are grouped together to form an active high and active low group of error signals; for each signal at logic 1 within the active high group there must be a corresponding signal at logic 0 within the active low group. To produce a single dual-rail output code, all active low lines are logically *OR*ed to produce an active high *ERROR* output using a *NAND* gate, whereas all active high lines are logically *OR*ed to produce an active low *NTERROR* output using a *NOR* gate. A single error at any node within this circuit will be indicated by both output lines being at the state state, the exception being the syndrome outputs of checker/correctors A and B since the distance between valid outputs is unity.

3.7 Implementation of checked datapath

The fault-tolerant RISC (FT-RISC) is to be fabricated using a two-layer metal CMOS process. The self-checking and correcting datapath forms the central data processing element of the FT-RISC; it is designed so that bus lines run horizontally in metal 1 across its entire width, whereas control signals generally run in a vertical direction on metal 2. This will allow the datapath controllers to be situated directly above and below the regions of the datapath which they control, an arrangement which will allow both controllers to check each other. At the boundary between datapath and controllers, signals such as flags are duplicated wherever possible so that errors occurring at such points can be detected via the controller outputs.

A two-phase nonoverlapping clocking strategy is used throughout the datapath to synchronise all bus transfers and ALU operations. During the high period of clock signal $\phi 1$ two operands are driven onto buses A and B and latched into the two checker/corrector units CCA and CCB; any errors are flagged when clock signal $\phi 2$ goes high.

During the next $\phi 1$ the checked operands are latched into the ALU input latches and the required operation performed. The result is then clocked out of the ALU output latches during the following clock cycle and into the checker/corrector unit CCA where it is checked and if necessary corrected. Finally the checked results are latched into the register file.

Fig. 10 shows a scaled floorplan of the datapath. The shaded regions indicate those areas of the layout which constitute checking/correcting hardware; the clear areas are the main ALU and register file sections of the datapath, i.e. the checked hardware. The percentage area increase due to incorporating the error checking/correcting hardware can be estimated from the areas of the shaded and unshaded regions of the layout; the following values are based on the use of a 2 μm CMOS process. Total area of checking/correcting hardware

$$A_c = 12.93 \times 10^6\ \mu m^2$$

total area of unchecked datapath

$$A_d = 11.51 \times 10^6\ \mu m^2$$

The percentage cost in terms of increased layout area is given by

$$\text{percentage area increase} = \frac{12.93}{11.51} \times 100\% = 112\%$$

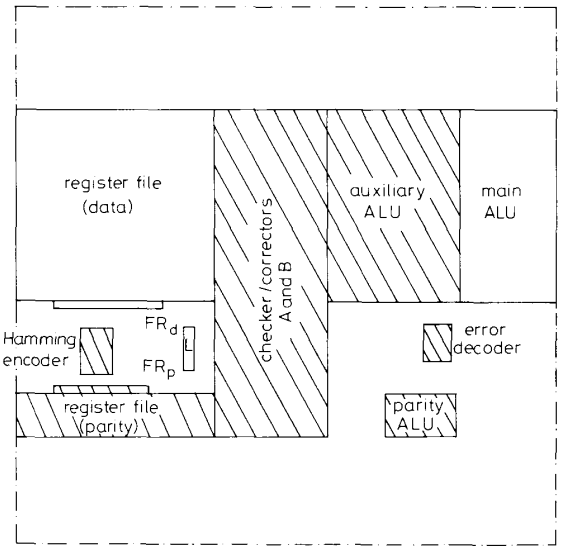

Fig. 10 *Floorplan of self-checking datapath*

4 Fault coverage

In this Section of the paper consideration is given to the fault coverage provided by the additional checking and correcting hardware incorporated into the self-checking datapath. In the discussion that follows a fault is assumed to be either one or two gate outputs which are permanently stuck at logic 0 or logic 1, resulting in errors in the information and parity sections of the datapath.

For the purpose of establishing the fault coverage it is necessary to divide the instruction types into three categories, i.e. those which are purely bitwise not requiring the extra parity bits P_{AB} or P_{CY}, logical instructions which involve the use of P_{AB} and arithmetic instructions which involve the use of P_{CY}. The following is obtained from Table 3:

Group 1: LITERAL (LIT), CLEAR (CLR), COMPLEMENT (COMP), EXCLUSIVE OR (*EOR*). The parity of the result is totally independent of the information bits and only depends on P_A and P_B.

Group 2: Logical AND, logical OR. Parity of result depends on P_A, P_B and P_{AB}.

Group 3: ADD, SUBTRACT (SUB), DECREMENT (DCR), INCREMENT (INC), NEGATE (NEG).

All involve addition and use the parity of carries signal P_{CY}, along with P_A and P_B, to predict the parity of the result.

Group 1 operations: For the operations within Group 1, P_R is solely dependent on the parity bits of the operands P_A and P_B, this means that all sections of the information and parity datapaths will be covered for faults to the extent that single errors will be corrected and double errors detected. The exceptions to this are the flags and the error decoder, both of which are covered for single error detection only.

The above is confirmed in Tables 4 and 5 which list the fault coverage expressed as the percentage of faults which can be detected and/or corrected for each hardware component of the information and checking sections of the datapath. The information presented in

Table 4: Total fault coverage for Group 1, 2 and 3 operations

Location of fault	Coverage	% area
Data register file	100% SEC/DED	66
Flag register FR_d	100% SEC/DED	1.2
ALU_d result circuit	100% SEC/DED	3.4
ALU_d output latch	100% SEC/DED	2.2
Flags V_2, CO_2, Z_d	100% SED	0.3
N and I Flags	NONE	<0.1
Overall carry parity CP_2	100% SED	1.7

a

Location of fault	Coverage	% area
Parity register file	100% SEC/DED	14
Flag register FR_p	100% SEC/DED	1
CCA/CCB parity bits section	100% SEC/DED	6.5
ALU_p input latches	100% SEC/DED	1.1
ALU_p output latch	100% SEC/DED	0.4
ALU_p circuit	100% SEC/DED	3.3
Z_p flag	100% SED	0.1
Overall carry parity CP_1	100% SED	1.5
Flags CO_1 and V_1	100% SED	0.2
Error decoder	100% SED	1.4

b

a Information hardware
b Checking hardware
SEC: single error corrected
DED: double error detected SED: single error detected

Tables 4 and 5 was obtained by the use of gate level simulation. Two HILO [11] models of the datapath, a faulty and a fault-free version, were exercised by an exhaustive set of input test vectors with various stuck-at faults applied to the faulty version. For each test vector the outputs of the faulty and fault-free datapaths were compared with one another to establish if the fault had been corrected (SEC), detected (SED) or not detected (END), the latter case being indicated by different faulty and fault-free datapath responses and an inactive error flag. Tables 4 and 5 also list, for each hardware component of the datapath, the area of the component expressed as a percentage of the total area of: (*a*) the checking hardware if the component is a sub-element of the checking hardware, or (*b*) the checked hardware if the component is a sub-element of the checked hardware. These figures give a measure of the relative contribution of each datapath element to the total area of the checking and checked sections of the system.

Of the two checker/corrector units CCA and CCB, CCB is checked by CCA when an ALU result is written back to the register file. An error in the output of the checker/corrector CCA will produce a faulty result which when checked by the faulty checker will produce a double error in the data stored in the register file. Alternatively if the fault in CCA is temporary then an error introduced by it during one cycle may be corrected during the next cycle.

Table 5: Partial fault coverage for Group 1, 2 and 3 operations

Location of fault	Group 1	Group 2		Group 3	% area
		AND	*OR*		
ALU_d input latches	100% SEC/DED	100% SEC/DED	100% SEC/DED	75% SEC 21% SED	5.9
ALU_d *NPN* circuit	100% SEC/DED	100% SEC/DED	100% SEC/DED	83.6% SEC 13.7% SED	3.9
ALU_d *NGN* circuit	—	—	—	57.8% SEC 39% SED	3.9
Main ALU carry in NC_{in2}	—	—	—	50% SEC 50% SED	<0.1
ALU_d carry generator	—	—	—	100% SED 100% DED	11.5

a

Location of fault	Group 1	Group 2		Group 3	% area
		AND	*OR*		
CCA/CCB data bits section	100% SEC/DED	75% SEC 25% END	75% SEC 25% END	75% SEC 25% END	26.5
ALU_a input latches	—	75% SEC 25% END	75% SEC 25% END	75% SEC 23.4% SED	5.3
ALU_a *NPN* circuit	—	—	—	83.6% SEC 15.6% SED	2.6
ALU_a *NGN* circuit	—	—	—	57.8 SEC 39% SED	2.6
ALU_a carry generator	—	—	—	100% SED	10.1
Auxiliary ALU carry in NC_{in1}	—	—	—	50% SEC 50% SED	<0.1
P_{AB} encoder	—	100% SEC/DED	100% SEC/DED	—	13.4
P_{CY} encoder	—	—	—	100% SEC/DED	10

b

a Information hardware
b Checking hardware
SEC: single error corrected DED: double error detected
SED: single error detected END: error not detected

It is worth noting the fault coverage for both the data and parity register files which account for 66 and 14% of the checked and checking hardware areas, respectively. For all types of ALU operations, faults within these elements are covered to the extent of 100% SEC/DED by checker/correctors CCA and CCB. This is of particular importance since any data residing within the datapath for any length of time will spend the majority of that time stored in the register file and consequently is more likely to be affected by soft errors.

Group 2 operations: Both the logical *AND* and *OR* operations use the output of the P_{AB} encoder to predict the parity of the result. If the fault lies in the encoder itself then the resulting errors will be corrected and/or detected by one of the checker/corrector units; thus the P_{AB} encoder is fully covered for SEC/DED and Group 2 operations. If a fault occurs at the output of the main ALU (ALU_d) input latch the resulting errors will be detected and/or corrected, as P_{AB} is generated by the output of a different latch. The same argument applies to the carry propagate logic (NPN) within ALU_d and so this is also fully covered for SEC/DED.

If the auxiliary ALU input latches are faulty then depending on the data values there will be either no errors or an undetectable error in P_{AB}; in terms of percentages, 50% of faults which cause an error will produce an error that can be corrected, and the remaining 50% will cause an error that cannot be detected. Hence 75% of input test patterns resulted in identical faulty and fault-free datapath responses, and the remaining 25% produced different responses without flagging the error.

If one of the checker/corrector units, CCA or CCB, has a faulty output within the information bits, then both the auxiliary and main ALUs will source faulty data. This situation is a combination of the cases discussed above and the resulting fault coverage is identical to the previous case as shown in Table 5. Faults within CCA will be covered to the same extent as faults within CCB provided they are temporary in nature.

Group 3 operations: All of the operations within Group 3 involve addition and as such use the parity of carries signal P_{CY} generated by the P_{CY} Hamming encoder within the auxiliary ALU. If errors occur within the parity bits P_A, P_B or P_{CY} then these can be detected and corrected; hence the P_{CY} encoder, and all sections of the hardware which manipulate the parity bits are covered to the extent of 100% SEC/DED.

The remaining sources of errors to be considered are the main and auxiliary ALUs, the checker/correctors CCA and CCB (data side only) and the two carry inputs NC_{in1} and NC_{in2}. The percentages given in Table 5 show that the main cause of undetectable errors are faults within the checker/corrector units themselves due to the fact that both the main and auxiliary ALUs receive erroneous data under such circumstances; however only 25% of input test vectors result in an undetectable error, and since it is reasonable to assume that checker/corrector faults will be a rare event the above figure can be considered acceptable. Of the remaining elements, the next most significant in terms of undetectable faults are the main ALU input latches. Faults within these latches will cause undetectable errors for 4% of input test vectors.

5 Description of complete fault-tolerant RISC processor

The self-checking and correcting datapath discussed in the preceding Sections forms the functional element of the fault-tolerant reduced instruction set computer (FT-RISC). Fig. 11 shows a block diagram of the complete

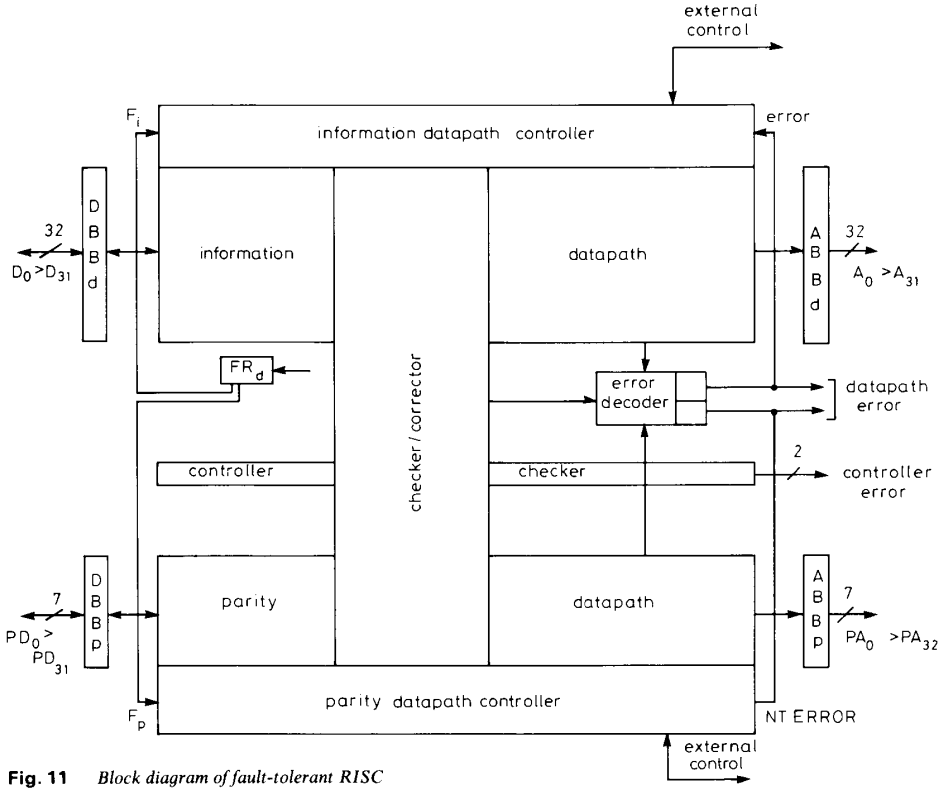

Fig. 11 *Block diagram of fault-tolerant RISC*

FT-RISC; from this diagram it can be seen that the information and parity processing sections of the datapath are controlled by two independent but synchronised controllers, the information datapath controller and the parity datapath controller respectively. The two controllers are identical logic circuits and they produce matching sets of signals during each instruction cycle; these signals are continually compared with each other by means of an overall parity circuit located between the information and parity processing sections of the datapath, and the resulting 'controller error' output provides a single bit error detection capability at the controller level. Datapath signals which are used as qualifying inputs to each controller are duplicated by means of separate hardware within the datapath itself; for example the data flag register FR_d contains two flags for each status bit, one being used to supply the information datapath controller and the other supplies the parity datapath controller. This provides an additional degree of protection against faults due to the fact that a single error in the output of the data flag register will force the controllers to produce differing outputs during the current machine cycle; if the discrepancy between controller outputs adds up to an odd number of inequalities then the result will be an active 'controller error' output.

Communication with the outside world is achieved by means of a unidirectional address but and a bidirectional data bus, both 32 bits wide. To allow the flow of check bits between system elements and maintain a consistent check of the whole computer system, both the data and address buses have an associated 7-bit parity bus which is used to carry the relevant check bits during data transfers to and from the device. The remaining external input/output signals are used for memory read/write control, error flags and interrupts.

5.1 Instruction set

The instruction set supported by FT-RISC is given in Table 6, each instruction has an identical format to that shown in Fig. 12. The majority of instructions perform an

Table 6: Fault-tolerant RISC instruction set

Operation	Operands			Description
ADD	rs_1	rs_2	r_d	register/register addition
ADDC	rs_1	rs_2	r_d	register/register addition with carry
SUB	rs_1	rs_2	r_d	register/register subtraction
SUBC	rs_1	rs_2	r_d	register/register subtraction with borrow
ADDI	rs_2	dd	r_d	register/immediate data addition
ADDCI	rs_2	dd	r_d	register/immediate data addition with carry
SUBI	rs_2	dd	r_d	register/immediate data subtraction
SUBCI	rs_2	dd	r_d	register/immediate data subtraction with borrow
COMP	rs_1		r_d	complement register
NEG	rs_1		r_d	negate register
INC	rs_1		r_d	increment register
DCR	rs_1		r_d	decrement register
CLR			r_d	clear register
MVCHK	rs_1		r_d	move/check register
AND	rs_1	rs_2	r_d	register/register bitwise AND
OR	rs_1	rs_2	r_d	register/register bitwise OR
EOR	rs_1	rs_2	r_d	register/register bitwise exclusive-OR
LITI		dd	r_d	immediate data literal
ANDI	rs_2	dd	r_d	register/immediate data bitwise AND
ORI	rs_2	dd	r_d	register/immediate data bitwise OR
EORI	rs_2	dd	r_d	register/immediate data bitwise Exclusive-OR

a

Operation	Operands			Description
LOAD	rs_2	dd	r_d	load register from memory $((rs_2) + dd) \rightarrow r_d$
STORE	rs_2	dd	r_d	store contents of register $(r_d) \rightarrow \text{addr}((rs_2) + dd)$

b

Operation	Operands			Description
JMP	rs_2	dd		jump always $(rs_2) + dd \rightarrow r_0$ {program counter} For branch operation $rs_2 = r_0$
JC	rs_2	dd		jump if carry set
JNC	rs_2	dd		jump if carry clear
JP	rs_2	dd		jump if sign flag clear
JM	rs_2	dd		jump if sign flag set
JV	rs_2	dd		jump if overflow flag set
JNV	rs_2	dd		jump if overflow flag clear
JZ	rs_2	dd		jump if zero
JNZ	rs_2	dd		jump if not zero
JSR	rs_2	dd	r_d	jump to subroutine
RET	rs_2	dd	r_d	return from subroutine/interrupt $(r_d) \rightarrow r_0$ {r_0 = program counter}

c

a Arithmetic and logical instructions
b Memory reference *c* Transfer of control
Registers rs_1, rs_2 and r_d can be any of the general purpose registers
Immediate operand dd is a 16-bit number
Shift left can be achieved by addition with $rs_1 = rs_2$

operation on the contents of two source registers rs_1 and rs_2, placing the result in a destination register r_d. The

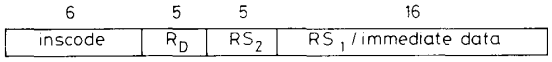

Fig. 12 *FT-RISC instruction word format*

least significant 16 bits of the instruction word are interpreted as an immediate data operand for certain instructions such as *AND* Immediate (*ANDI*); alternatively the immediate data can be used in the calculation of the effective address for those instructions which refer to main memory, such as jump and branch. The instruction code field (INSCODE) of the instruction word is used to encode one of the 34 possible instructions listed in Table 6. There are no explicit instructions for the manipulation of flag register bits; the flags are modified by use of the arithmetic and logical instructions which involve immediate data such as 'Exclusive-*OR* Immediate' (*EORI*). In this case the flag register is used as both the source register and the result destination register. All of the instructions within the arithmetic and logical group can effect the flags directly by changing the value stored in the ALU output latch; the new flags are latched into the flag register under control of an *UPDATE FLAGS* signal. To prevent the ALU flags overwriting the result of an instruction which is manipulating the flags themselves, the *UPDATE FLAGS* signal is inhibited during these instructions.

Included within the arithmetic and logical group of instructions is the so called 'move and check' (MVCHK) instruction. This instruction takes the contents of the source register specified by rs_1, checks for and corrects any errors and stores the result in a destination register r_d. If rs_1 and r_d are equal, the original contents of the register are overwritten by the checked/corrected version. This operation can be carried out on any of the general purpose datapath registers thus providing a useful self testing facility. Two memory reference instructions are provided to allow communication with external memory. In both instructions the effective address is calculated by adding the 16-bit immediate offset to the contents of the register specified by rs_2.

Transfer of control instructions can be either branches or jumps depending on whether or not the program counter r_0 is used as the base register when calculating the effective address. To facilitate the use of subroutines and interrupts, the jump to subroutine (JSR) and return from subroutine/interrupt (RET) instructions are provided; in both cases the register specified by r_d is used as a temporary storage location for the program counter. In the case of subroutine calls and returns this may be any of the general purpose registers, however, due to the asynchronous nature of interrupts one particular register within the register file must be set aside by the programmer/compiler for use solely as a temporary storage location for the program counter in the event of an interrupt; this will prevent registers being overwritten during the interrupt service routine.

Fig. 13 shows the register allocation used within FT-RISC. A total of five registers are set aside for particular uses while the remaining eleven registers are completely general purpose. Register number 2 is a read-only register containing the number 0000FFFFH; this register is used in the process of extracting the 16-bit immediate operand from the instruction word. Once an instruction word is fetched into the device the instruction code is used to determine whether or not part of the word is to be used as immediate data; if this is the case the entire instruction word is logically *AND*ed with the constant 0000FFFFH, thereby forcing the most significant 16 bits to zero. The parity ALU simultaneously produces the parity of the masked result, allowing the immediate data to be checked/corrected before it is used in further calculations.

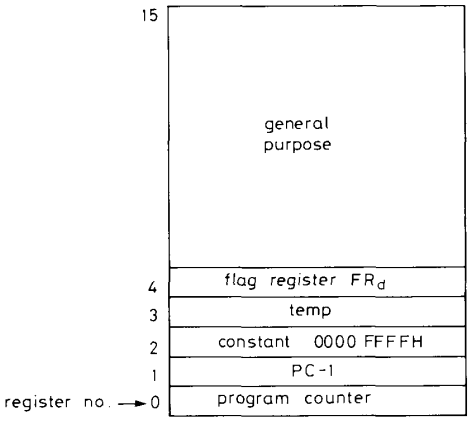

Fig. 13 *FT-RISC register allocation*

FT-RISC employs instruction prefetch to increase data throughput; this involves fetching the next instruction from memory while executing the current instruction, therefore the program counter r_0 always contains the address of the next instruction to be executed. In the event of a datapath error, the controller forces the datapath to repeat the entire instruction within which the error occurred. The error flags become active during the initial part of the instruction which immediately follows the instruction within which the error occurred, and so the remedial action involves repeating the previous instruction in an attempt to correct the situation. FT-RISC achieves this by decrementing the contents of the shadow program counter PC-1, which contains the address of the instruction currently being executed, and fetching the instruction contained at the resulting address. The shadow program counter makes the re-execution of jump instructions straightforward, as the address of the previous instruction can be easily obtained. Without a shadow program counter it is not possible to determine the address of the previous instruction once the main program counter has been loaded with the jump address.

5.2 Instruction timing

Every single item of data that is operated on by the datapath must pass through the checker/corrector units at some point during an instruction. For example the program counter is checked both before and after being incremented. Also the instruction code, after being read into the data bus buffer must be checked before being loaded into the instruction register. The increased activity taking place within the datapath during an instruction results in an unavoidable degradation in performance in terms of instruction execution speed. Fig. 14 shows the timing of events during an arithmetic and logical register/register type instruction. The instruction lasts for a total of 6 clock cycles; 3 of the instruction cycles (T_2, T_4 and T_6) involve the checking/correcting of data and these cycles would not be necessary in a comparable processor which did not possess fault-tolerant capabilities. The clock period is determined to a large extent by the time it

takes the ALU to produce a 32-bit result; this time period is greater than the delay caused by the checking and correcting process although a full clock cycle elapses while the latter takes place. The above leads to the conclusion that the cost, in terms of processing speed, of having concurrent error detection is a doubling of the overall instruction execution time when compared to a similar processor not performing CED/C.

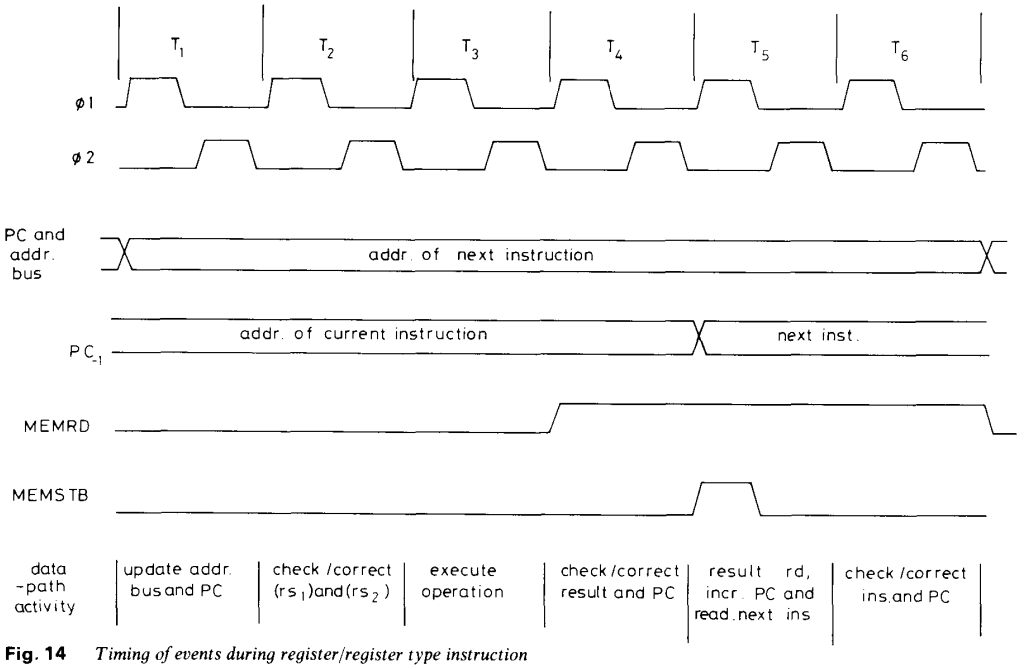

Fig. 14 *Timing of events during register/register type instruction*

6 Discussion

Fault tolerance in the computer industry has in general been obtained at the higher levels of the design. Many systems exist that use modular redundancy methods, in the form of TMR or NMR, at the board level. Although it is acceptable to replicate a board several times this replication is not desirable at the integrated circuit level. The extra silicon area needed to implement a TMR system on a chip would be unacceptable in most cases at >200%. This paper has presented a possible alternative to TMR for error detection and correction in an IC. The area overheads have been reduced to more acceptable levels and the techniques employed are very straightforward to implement.

The need for CED in future VLSI devices has been well documented, however, most proposals concentrate on simply detecting the error. The correction phase of the error condition is normally 'glossed over' as relatively trivial. The problem of how to deal with the error is nevertheless very serious and cannot be dismissed lightly. Most systems rely entirely on the controller to resolve the problem via the microprogram. This can lead to complex and intricate programming to deal with all eventualities. The solution proposed in this paper does not require any specialised controller. However the controller does need to take action when an error occurs and this takes the form of repeating the erroneous instruction. The majority of single errors can be dealt with concurrently by the code checkers/correctors during the additional check cycles occurring within the instruction. Although this results in a decrease in data processing speed, the inherent advantages of concurrent error correction enhance overall system reliability.

The gain in performance resulting from the use of the chosen code is well worth the extra area needed for its implementation. The hidden benefits gained are just as significant:

(*a*) As the entire code word is both taken on and off-chip all intercommunications between blocks are checked. This is very difficult to achieve with TMR, because of the excessive pin count, hence the board level implementations.

(*b*) Initial testing of the device is greatly simplified since the designer need only apply preselected patterns and observe the error signal outputs. Obviously a few noncode inputs could also be applied to exercise the checking hardware.

(*c*) The repair of systems containing the CED/C devices is eased. The faulty device is indicated immediately allowing the replacement to be made. The system would also degrade gracefully in the event of a permanent failure that could be dealt with successfully by the correction hardware.

Although this hardware fault tolerance method was demonstrated on a RISC processor the authors feel that many other systems which use similar operations could employ this technique at possibly reduced area costs. The RISC processor design will be evaluated as a real device once it has been fabricated.

7 Acknowledgments

The authors would like to acknowledge the efforts of Mrs. K. Kidd for typing this manuscript, also the Department of Electrical and Electronic Engineering in both Newcastle University and Sunderland Polytechnic for their computing and design resources. The authors would

also like to thank staff at ES2 Bracknell for their time in dealing with the fabrication aspects of the design.

8 References

1 TENDOLKAR, N., and SWANN, R.: 'Automated diagnostic methodology for the IBM 3081 processor complex', *IBM J. Res. Dev.*, 1982, (2), pp. 78–88

2 SEDMAK, R.M., and LIEBERGOT, H.L.: 'Fault tolerance of a general purpose computer implemented by very large scale integration', *IEEE Trans.*, 1980, **C-29,** (6), pp. 492–500

3 MARCHAL, P., NICOLAIDIS, H., and COURTOIS, B.: 'Microarchitecture of the MC68000 and the evaluation of a self checking version'. NATO Advanced Study Institute on Microarchitecture of VLSI computers, 1984

4 YEN, M.M., FUCHS, W.K., and ABRAHAM, J.A.: 'Designing for concurrent error detection in VLSI: Application to a microprogram control unit', *IEEE Trans.*, 1987, **SC-22,** (4), pp. 595–605

5 SAYERS, I.L., KINNIMENT, D.J., and CHESTER, E.G.: 'Design of a reliable and self testing VLSI datapath using residue coding techniques', *IEE Proc. E*, 1986, **133,** (3), pp. 169–179

6 SAYERS, I.L.: 'An investigation of "Design for Testability" techniques in very large scale integrated circuits'. PhD thesis, Newcastle University, May 1986

7 HAMMING, R.W.: 'Error detection and error detecting codes', *Bell Tech. J.*, 1950, (2), pp. 147–160

8 MEAD, C.A., and CONWAY, L.A.: 'Introduction to VLSI systems' (Addison-Wesley, 1980)

9 BRENT, R.P., and KUNG, H.T.: 'A regular layout for parallel adders', *IEEE Trans.*, 1982, **C-31,** (3), pp. 260–264

10 WAKERLEY, J.F.: 'Error detecting codes, self checking circuits and applications' (Elsevier North-Holland, New York, 1978)

11 HILO-3 User Manual, Genrad Ltd.