ELSEVIER

Microprocessors and Microsystems 25 (2002) 409–419

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.com/locate/micpro

The impact of instruction pointer corruption on program flow: A computational modelling study

Royan H.L. Ong, Michael J. Pont*

Department of Engineering, Control and Instrumentation Research Group, University of Leicester, University Road, Leicester, LE1 7RH, UK

Received 1 June 2001; revised 16 August 2001; accepted 29 August 2001

Abstract

Corruption of the instruction pointer, the main cause of program flow errors, is a common failure mode in the presence of high levels of electromagnetic interference. In this paper, results from a sub-cycle level simulator intended to provide an insight into the impact of EMI on such systems are described, and a detailed model of the impact on EMI on embedded processors is presented.

Keywords: Instruction pointer corruption; Electromagnetic interference; EMI; Program flow errors

1. Introduction

The increasing use of programmable electronics devices (microprocessor, microcontroller and DSP chips) in safety-related applications, such as passenger cars and aircraft, offers many opportunities to improve safety: however, it also introduces new risks. One key problem is the detrimental impact of electromagnetic interference (EMI) on the normal operation of programmable devices [1,10].

In a previous paper [8], we examined one of the key effects of EMI-induced errors: corruption of the instruction pointer (IP). A model for predicting the probability of a system detecting and correcting IP errors solely based upon the program code was also presented. While the model of EMI-induced errors was the most comprehensive model then available, it was still not able to explain all possible EMI-induced errors relating to corruption of the instruction pointer. A more complete model is presented in this paper.

Previously, the impact of EMI was assessed using an 8051 simulator manufactured by Keil Electronik GmbH. This simulator is primarily intended for use in debugging C programs: as a consequence, it operates at the instruction-cycle (or higher) level. This level of performance makes it well-suited to its intended purpose: however, in order to provide more detailed analysis; we require simulation at a finer level of resolution. A suitable simulator was therefore developed in our laboratory, and was used in the investigations discussed in this paper.[1]

The paper begins by providing an overview of the previous model. Section 3 discusses the new model while a description of our sub-cycle level simulator is provided in Section 4 together with simulation procedures and results. Section 5 discusses the results and conclusions are drawn in Section 6.

2. Overview of the previous model

IP is only one of many registers in a processor and there is no evidence to suggest that it is any more or less susceptible to EMI than the others. However, the impact of corruption of the IP is arguably the most serious result of EMI, as it normally results in serious disruption to the program flow.

Since the IP of an 8051-based microcontroller is a 16-bit wide register, we make the assumption that — in response to IP corruption — the IP is equally likely to take on any value in the range 0–65535. In these circumstances, the 8051 processor will fetch and execute the next instruction from the code memory location pointed to by the corrupted IP register. This code memory location may contain program code, data constants, or it may be empty. To

* Corresponding author.
E-mail address: m.pont@leicester.ac.uk (M.J. Pont).

[1] As with Refs. [7,8], the results in this paper were obtained using the 8051-family of microcontrollers. However, the results and discussion are generally applicable to devices with multibyte instructions.

0141-9331/02/$ - see front matter
PII: S0141-9331(01)00132-6

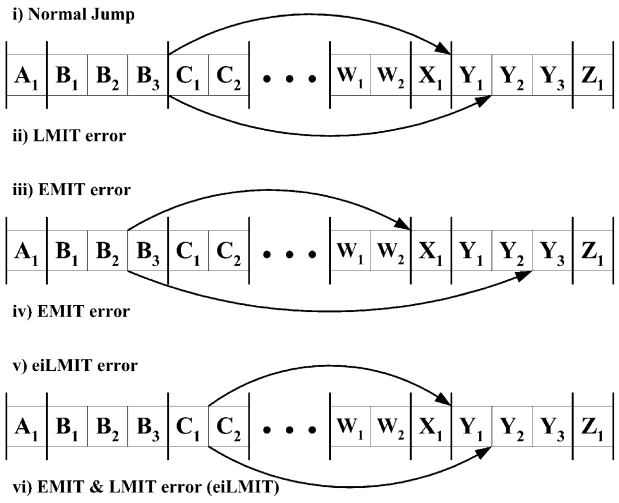


Fig. 1. Scenarios of program-flow branching due to IP errors.

complicate matters further, the amount of physical code memory is usually less than that addressable by the IP: that is, less than 65536 bytes. This results in an effect known as memory aliasing.[2]

From this starting point, a statistical model that may be used to assess the relative safety of a program based on its executable code was developed [8]. In this model, all physical code memory locations (in ROM) were identified as belonging to one of three categories: 'Safe Locations' (SL), 'Unsafe Locations' (UL) and 'Dangerous Locations' (DL). When comparing two programs with otherwise identical features, the implementation with the larger number of safe locations was shown to be more reliable in the presence of IP corruption.

This model also took into account the effects of two software-based error detection techniques, generally known as 'NOP Fills' (NF) and 'Function Tokens' (FT) [3,4,6,12].

2.1. Limitations of the previous model

The model described in [8] has some limitations. In the absence of a sub-cycle simulator, it was necessary to make the assumption that corruption of the IP took place between instructions: that is, that the IP was not corrupted during the execution of a multibyte instruction. Thus, the study reported in our previous paper assumed that only what is referred to here as 'Late Multibyte Instruction Trap' (LMIT) errors could occur. Such errors happen at the location where the IP 'lands'. In this paper, we consider not only LMIT errors but also what we refer to as 'Early MIT' (EMIT) errors.

EMIT errors are caused when the program branches due to IP errors before all the bytes of a multibyte instruction have been read and decoded by the processor. Listing 1 illustrates the nature of such errors.

```
0100 759850 MOV   98H,#50H
0103 438920 ORL   89H,#20H
0106 758DFD MOV   8DH,#FDH
```

Listing 1. Original instruction sequence.

For example, if we assume that, in Listing 1, an IP error occurs after the byte at location 0×0101 is read, but before the byte at location 0×0102 is read, the program will then branch erroneously to 0×0104, and the processor will interpret the instruction sequence as shown in Listing 2.

```
0100 7598
0104 89     MOV   98H,#89H
0105 20758D JB    75H,8DH
```

Listing 2. Modified instruction sequence due to EMIT error.

The original instruction starting at 0×0100 will still be correctly interpreted since, in the 8051-instruction set, the first byte alone uniquely determines the instruction type. However, because part of the operand has changed, the instruction will be executed with the wrong operand(s). The severity of this form of error can be significant, ranging from incorrect arithmetic results to 'random' call and branch jumps.

After an EMIT error occurs, it is possible that an LMIT error will also occur, as shown in Listing 2, starting from address 0×0105. This is due to the fact that the instruction bytes are no longer aligned with the instruction boundary of the original sequence. It is also possible, under some situations, for LMIT errors not to occur even when the IP 'lands between' multibyte instructions.

Fig. 1 shows some of the possible IP error scenarios, where 'A', 'X' and 'Z' are single-byte instructions, 'C' and 'W' are double-byte instructions and 'B' and 'Y' are triple-byte instructions. The bold vertical lines (e.g. between 'B' and 'C') denote the original instruction boundary.

The first two scenarios, (i) and (ii), were discussed in Refs. [8,9] and can be simulated with instruction-level simulators. However, the other scenarios involve program branches between instructions (EMIT errors); these can only be reproduced with a sub-clock-cycle (or oscillation) level simulator [7].

Like LMIT errors, EMIT errors can only occur with multibyte instructions. In Fig. 1, Scenario (iii) is typical: one or more operands would be made up of byte(s) at the IP's 'landing' location while the subsequent instructions are correctly interpreted. However, as mentioned earlier, EMIT errors can nullify the effects of LMIT errors as in (iv), or, on the other hand, cause EMIT-induced LMIT (eiLMIT) errors as in (v).

2.2. IP error scenarios

EMIT errors occur under three situations: when the program branch happens between the first and second byte of a double-byte instruction (type D_{EMIT}), between the first and second byte of a triple-byte instruction (type T_{EMIT1}) and

[2] All of these possibilities were discussed in detail in Ref. [8]. These discussions will not be repeated here.

Table 1
eiLMIT conditions for double-byte instructions (D_{EMIT})

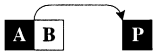
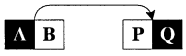
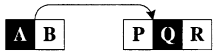
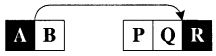
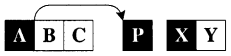
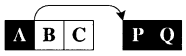
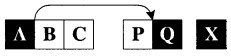
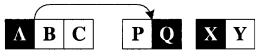
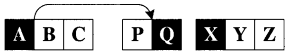
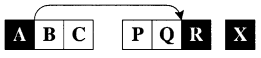
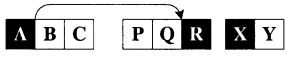
	Schematic	L_{POS}	L_{1S}	L_{2S}	Outcome
1		0	1	X	Normal
2		0	2	X	eiLMIT (excl. D2)
3		1	2	X	Normal
4		0	3	X	eiLMIT (excl. T23)
5		1	3	X	eiLMIT (excl. T23 and T3)
6		2	3	X	Normal

Table 2
eiLMIT conditions for triple-byte instructions — position 1 (T_{EMIT1})

	Schematic	L_{POS}	L_{1S}	L_{2S}	Outcome
1		0	1	1	Normal
2		0	1	2	eiLMIT (excl. D2)
3		0	1	3	eiLMIT (excl. T23)
4		0	2	X	Normal
5		1	2	1	Normal
6		1	2	2	eiLMIT (excl. D2)
7		1	2	3	eiLMIT (excl. T23)
8		0	3	X	eiLMIT (excl. T23 and T3)
9		1	3	X	Normal
10		2	3	1	Normal
11		2	3	2	eiLMIT (excl. D2)
12		2	3	3	eiLMIT (excl. T23)

between the second and third byte of a triple-byte instruction (type T_{EMIT2}). Each situation has its own set of conditions for eiLMIT errors to occur; three tables (Tables 1–3) are used to show all possible combinations.

The schematic representations shown in the tables are detailed in Fig. 2. The term 'take off' and 'landing' refer to the code memory location where the IP value changes and the location that it points to, respectively.

Only the last instruction before program branches and the first or first two instruction(s) where the branch 'lands' are shown in the scheme. Instruction branch and 'land' positions are denoted numerically (0–2), starting from the left boundary of the first byte of an instruction. The colour-inverted bytes are those that will be interpreted as one instruction.

In the tables, L_{POS} denotes the 'landing' position of the branch while L_{1S} and L_{2S} are the instruction sizes (in bytes) of the 'landing' and the next instruction respectively. An 'X' for L_{2S} means that instruction does not play a part in determining the four possible outcomes ('Normal', EMIT, LMIT or eiLMIT).

Two important points can be made about the scenarios presented in the tables. In the case of D_{EMIT} and T_{EMIT2}, only the 'landing' byte is necessary to determine if an eiLMIT error has occurred. However, the possibility of eiLMIT errors occurring for some T_{EMIT1} scenarios can only be ascertained by knowing the instruction-type of the instruction following the 'landing' instruction. This situation arises when the 'landing' instruction will only provide one of the two operand bytes needed.

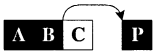
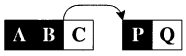
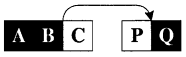
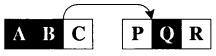
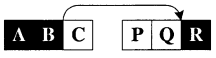
Another point to note is the fact that not all eiLMIT conditions will produce errors. If the remaining bytes of the 'landing' instruction that do not form part of the erroneously decoded instruction (e.g. byte 'Q', condition 2, Table 3) have the value of 0×00, an eiLMIT error does not occur: this is noted in bold in the 'outcome' column. This is because the value of 0×00 corresponds to the 'No Operation' (NOP) instruction in the 8051 instruction set; a detailed explanation is given in [8]. The symbols 'D2', 'T23' and 'T3', also taken from [8], are described in Section 3.

2.3. Other program flow error mechanisms

Apart from IP errors, stack pointer (SP) errors will also cause program flow problems. As the corrupted SP will point to the wrong RAM location, this value will, in turn, be copied to the IP. This scenario has the same effect as LMIT or 'Normal' IP errors. However, it does not give rise to EMIT and eiLMIT errors.

A similar program-flow error scenario occurs when the operand(s) of branching instructions are corrupted (for

Table 3
eiLMIT conditions for triple-byte instructions — position 2 (T_{EMIT2})

	Schematic	L_{POS}	L_{1S}	L_{2S}	Outcome
1		0	1	X	Normal
2		0	2	X	eiLMIT (excl. D2)
3		1	2	X	Normal
4		0	3	X	eiLMIT (excl. T23)
5		1	3	X	eiLMIT (excl. T23 and T3)
6		2	3	X	Normal

example, in code memory, on the bus or in registers), before being copied to the IP.[3] The effect will be similar to IP errors. The corruption of operands (for non-branching instructions) may not directly cause program-flow errors, but may, of course, still have serious repercussions.

3. Extending the model to handle EMIT errors

To incorporate EMIT-based errors, the original model described in [8] has been extended, taking into account all of the conditions considered in Section 2.2. This model is discussed in the remainder of this paper: it is shown schematically in Fig. 3.

To simplify the statistical modelling process, the model is divided into three sections, each of which is described separately here. Referring to Fig. 3, the first section of the model determines the probability of an EMIT error occurring, denoted by α. As the only other possible situation is that an EMIT error does not occur, its probability is simply $1 - \alpha$. The other two sections split the 'EMIT/no EMIT' probability into four categories in a binary-tree manner. β ('Normal' or LMIT) and λ (EMIT or eiLMIT) denote the probability of each split. As with $1 - \alpha$, $1 - \beta$ and $1 - \lambda$ are the conjugates, respectively.

[3] The error is most likely to occur in systems where code is stored in RAM, or while code is transferred in internal buses or registers: code stored in ROM is much more resilient.

3.1. Definition of terms and symbols

The different instruction-types, some of which were used in [8,9], are described in Table 4. This table also lists their associated symbols.[4]

The probability of each instruction-type is denoted in the form of $P(X)$.

Static instruction-type probabilities (*S*, DN, *D2*, *D*, *T*, TN, *T23*, *T3* and UP) — which are measured from the program code itself — are given by the ratio:

$$\frac{\text{Number of bytes which are part of instruction-type X}}{\text{Total code memory availabe (in bytes)}}$$

Dynamic instruction-type probabilities (I11, I12, I14, I21, I22 and I32) — which are determined via simulation — are given by the ratio:

$$\frac{\text{Number of clock cycles spent executing instruction-type X}}{\text{Total number of clock cycles in program run}}$$

3.2. Dissecting the model

Calculating the probability of eiLMIT and LMIT errors only requires static instruction-type probabilities as the probability of an IP error 'landing' at any code memory location is exactly the same (1/physical code memory size).

To model the probabilities of EMIT errors (α), dynamic instruction-type probabilities are needed.

The equations in Section 3.4, Section 3.6 and Section 3.7 are used in conjunction with those from Section 3.3 to form the complete model (Section 3.8).

3.3. Probability of EMIT occurring (α)

To calculate EMIT errors, it is necessary to know how the 8051 processor fetches and executes instructions. The fetch–execute cycle consists of 12 oscillation cycles (equivalent to a clock cycle), which are divided into 6 states (S1 to S6) with two phases each (P1 and P2). Bytes are only read during State 1 Phase 2 (S1P2) and State 4 Phase 2 (S4P2). The first byte of all instructions is read during S1P2 of its first clock cycle. Double-byte one-cycle (I21) instructions are read during S1P2 and S4P2 while triple-byte (*I*32) instructions during S1P2, S4P2 and S1P2 of the next clock cycle [13,14]. We assume that for double-byte two-cycle (I22) instructions (e.g. ORL), both bytes are read during S1P2 and S4P2 of the first cycle.

As a result, EMIT errors will happen between states S1P2 and S4P1 for a double-byte instruction during its first clock cycle, and between S1P2 of the first and following clock cycle for triple-byte instructions. In terms of the proportion of time taken to execute each instruction, I21, I22 and I32 instructions will cause EMIT errors in 6 out of 12, 6 out of 24 and 12 out of 24 oscillation cycles, respectively. Hence

[4] Note that: T = TN + T23 + T3 and D = DN + D2.

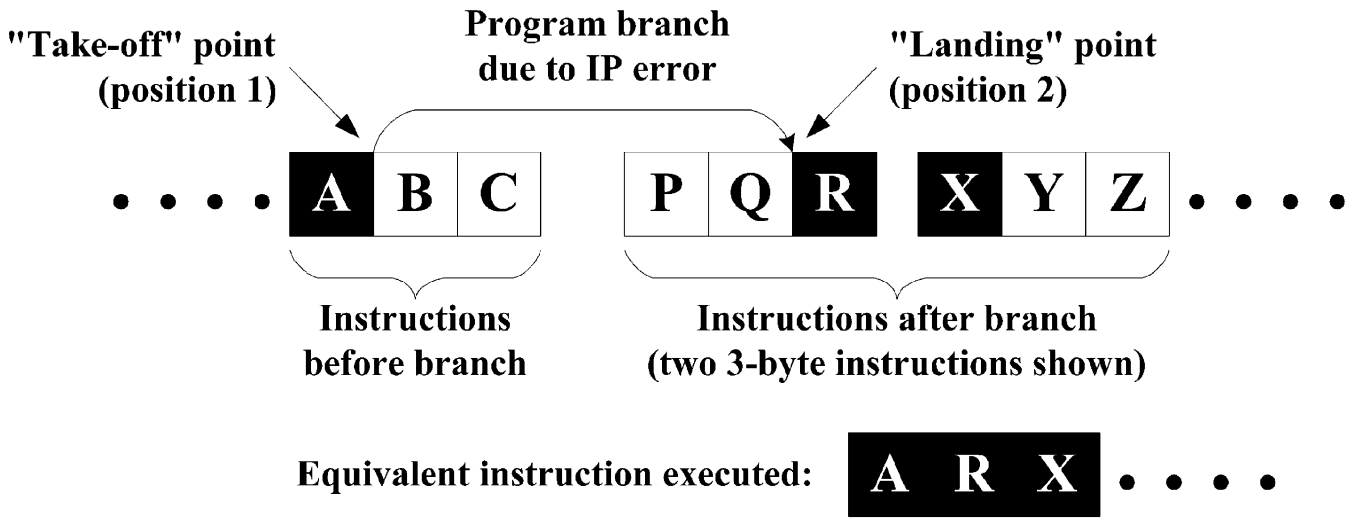


Fig. 2. Description of schematic diagram used in tables.

this probability is determined as follows:

$$\alpha = P(\text{I21})/2 + P(\text{I22})/4 + P(\text{I32})/2$$

The possibility of EMIT error not occurring will be:

$$P(\text{No EMIT}) = P(\text{I11}) + P(\text{I12}) + P(\text{I14}) + P(\text{I21})/2 + 3P(\text{I22})/4 + P(\text{I32})/2$$

Since only two possible conditions exist, this equation can be simplified as follows:

$$P(\text{No EMIT}) = 1 - \alpha$$

which is also true in the case of β, and λ

3.4. Probability of LMIT occurring (β)

This part of the model has already been explained and proven in [8]. It is presented here in a slightly different form, to ensure consistency with the other material presented in this paper.

$$\beta = P(\text{DN})/2 + 2P(\text{TN})/3 + P(\text{T3})/3$$

$$P(\text{Normal}) = P(\text{SUP}) + P(\text{DN})/2 + P(\text{D2}) + P(\text{TN})/3 + 2P(\text{T3})/3 + P(\text{T23}) = 1 - \beta$$

3.5. Dividing eiLMIT error calculations

Calculating the probability of eiLMIT errors occurring is rather complicated, due to the various possible combinations. To simplify discussion and modelling, the three possible eiLMIT scenarios, D_{EMIT}, T_{EMIT1} and T_{EMIT2}, are considered separately. Only two unique equations, λ_1 and λ_2, are derived since D_{EMIT} and T_{EMIT2} are the same (compare Tables 1 and 3). In any case, the 'take off' instruction is not considered here, as it has already been included while calculating α.

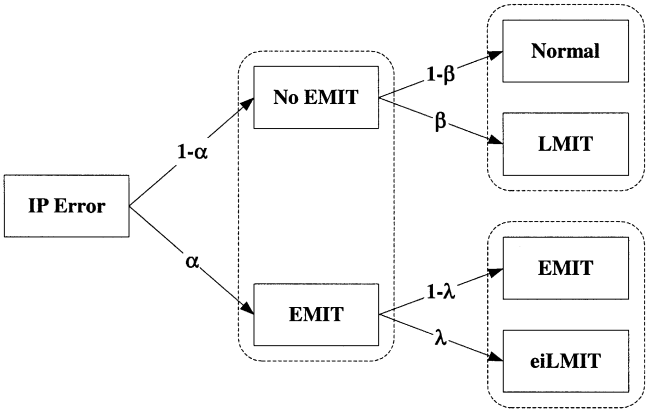


Fig. 3. IP error model.

Table 4
Instruction-type classification and description

Instruction type		Description
Statically determined	S	Single-byte instruction
	D	Double-byte instruction
	DN	Double-byte instruction with all non zero-valued bytes
	D2	Double-byte instruction with zero-valued second byte
	T	Triple-byte instructions
	TN	Triple-byte instructions with all non zero-value bytes
	T23	Triple-byte instructions with zero-valued second and third byte
	T3	Triple-byte instructions with zero-valued third byte
	UP	Unprogrammed code memory locations
	SUP	Sum of S and UP
Dynamically determined	I11	Single-byte one-cycle instruction
	I12	Single-byte two-cycle instruction
	I14	Single-byte four-cycle instruction
	I21	Double-byte one-cycle instruction
	I22	Double-byte two-cycle instruction
	I32	Triple-byte two-cycle instruction

The second point mentioned in Section 2.2 regarding zero-valued, unexecuted, trailing byte(s) (e.g. condition 2, Table 1), has to be taken into account as it has an impact on the results.

3.6. Calculating eiLMIT for D_{EMIT} and T_{EMIT2} conditions (λ_1)

For double-byte instructions, there will be a 50% chance of the IP 'landing' at position 0, which would cause an eiLMIT error. However, D2-type instructions will not cause eiLMIT errors: only DN-type instructions will do so.[5]

Triple-byte instructions will cause eiLMIT errors when the IP 'lands' at position 0 or 1 (66% likelihood). Their eiLMIT conditions are calculated in the same manner as that for double byte instructions. By summing the terms of each condition, the overall eiLMIT probability for D_{EMIT} and T_{EMIT2} conditions is as follows:

$$\lambda_1 = P(\mathrm{DN})/2 + [P(\mathrm{TN}) + P(\mathrm{T3})]/3 + P(\mathrm{TN})]/3$$
$$= P(\mathrm{DN})/2 + 2P(\mathrm{TN})/3 + P(\mathrm{T3})/3$$

$$P(\text{No eiLMIT}) = P(\mathrm{SUP}) + P(\mathrm{D2})/2 + P(D)/2 + P(\mathrm{T23})/3 + [P(\mathrm{T23}) + P(\mathrm{T3})]/3 + P(T)/3$$
$$= P(SUP) + P(\mathrm{D2})/2 + P(D)/2 + 2P(\mathrm{T23})/3 + P(\mathrm{T3})/3 + P(T)/3$$
$$= 1 - \lambda_1$$

[5] The sum of D2 and DN-type instructions constitutes all double-byte instructions.

3.7. Calculating eiLMIT for T_{EMIT1} conditions (λ_2)

T_{EMIT1} eiLMIT error calculations are similar to λ_1 but more complicated. Under some conditions, an additional instruction following that of the 'landing' instruction is needed to determine if eiLMIT errors have occurred. This is due to the fact that two bytes at the 'landing' location will become part of the operand (Table 2). This condition has an impact on 'landings' for all single-byte instructions, 'landings' at position 1 of double-byte instructions and 'landings' at position 2 of triple-byte instructions.

When a second instruction is needed to fulfil the eiLMIT criteria, the probability of the next instruction type occurring is multiplied by the probability of the first 'landing' instruction. Here, eiLMIT exclusion conditions, due to zero-valued trailing bytes, are taken into account in exactly the manner described for D_{EMIT} and T_{EMIT2}. Each eiLMIT condition is summed to produce λ_2 as follows:

$$\lambda_2 = [P(\mathrm{SUP}) \times P(\mathrm{DN})] + [P(\mathrm{SUP}) \times \{P(\mathrm{TN}) + P(\mathrm{T3})\}] + [P(\mathrm{D})/2 \times P(\mathrm{DN})] + [P(\mathrm{D})/2 \times \{P(\mathrm{TN}) + P(\mathrm{T3})\}] + P(\mathrm{TN})/3 + [P(\mathrm{T})/3 \times P(\mathrm{DN})] + [P(\mathrm{T})/3 \times \{P(\mathrm{TN}) + P(\mathrm{T3})\}]$$
$$= P(\mathrm{SUP}) \times [P(\mathrm{DN}) + P(\mathrm{TN}) + P(\mathrm{T3})] + P(\mathrm{D})/2 \times [P(\mathrm{DN}) + P(\mathrm{TN}) + P(\mathrm{T3})] + P(\mathrm{T})/3 \times [P(\mathrm{DN}) + P(\mathrm{TN}) + P(\mathrm{T3})] + P(\mathrm{TN})/3$$
$$= [P(\mathrm{SUP}) + P(\mathrm{D})/2 + P(\mathrm{T})/3] \times [P(\mathrm{DN}) + P(\mathrm{TN}) + P(\mathrm{T3})] + P(\mathrm{TN})/3$$

$$P(\text{No eiLMIT}) = [P(\mathrm{SUP}) \times P(\mathrm{SUP})] + [P(\mathrm{SUP}) \times P(\mathrm{D2})] + [P(\mathrm{SUP}) \times P(\mathrm{T23})] + P(\mathrm{D})/2 + [P(\mathrm{D})/2 \times P(\mathrm{SUP})]\ [P(\mathrm{D})/2 \times P(\mathrm{D2})] + [P(\mathrm{D})/2 \times P(\mathrm{T23})] + P(\mathrm{T23})/3 + P(\mathrm{T3})/3 + P(\mathrm{T})/3 + [P(\mathrm{T})/3 \times P(\mathrm{SUP})] + [P(\mathrm{T})/3 \times P(\mathrm{D2})] + [P(\mathrm{T})/3 \times P(\mathrm{T23})]$$
$$= P(\mathrm{SUP}) \times [P(\mathrm{SUP}) + P(\mathrm{D2}) + P(\mathrm{T23})] + P(\mathrm{D})/2 + P(\mathrm{T})/3 + P(\mathrm{D})/2 \times [P(\mathrm{SUP}) + P(\mathrm{D2}) + P(\mathrm{T23})] + P(\mathrm{T23})/3 + P(\mathrm{T})/3 \times [P(\mathrm{SUP}) + P(\mathrm{D2}) + P(\mathrm{T23})] + P(\mathrm{T3})/3$$
$$= [P(\mathrm{SUP}) + P(\mathrm{D})/2 + P(\mathrm{T})/3] \times [P(\mathrm{SUP}) + P(\mathrm{D2}) + P(\mathrm{T23})] + P(\mathrm{D})/2 + [P(\mathrm{T23}) + P(\mathrm{T3}) + P(\mathrm{T})]/3$$
$$= 1 - \lambda_2$$

Table 5
IP error classification probability

Error category		Error condition	Equation
EMIT	LMIT		
No	No	No error	$(1-\alpha)\times(1-\beta)$
No	Yes	LMIT	$(1-\alpha)\times\beta$
Yes	No	EMIT	$[P(D_d)/2\times(1-\lambda_1)]+[P(I32)/3\times(2-\lambda_1-\lambda_2)]$
Yes	Yes	eiLMIT	$[P(D_d)/2\times\lambda_1]+[P(I32)/3\times(\lambda_1+\lambda_2)]$

3.8. Overall model for IP error

By combining all of the equations from previous sections in the manner shown in Fig. 3, it is possible to calculate the probability of each of the four IP error classifications from the static and dynamic instruction-type profile of a program. The error classifications will be a product of these probabilities, as shown in Table 5.

Though α has been calculated as a single value, it is necessary to work with each term separately when it comes to calculating EMIT and eiLMIT errors. This is due to the fact that both λ coefficients representing D_{EMIT}, T_{EMI1} and T_{EMIT2}, have to be multiplied by each of the three terms in α's equation before being summed. Hence the equation for eiLMIT is as follows:

$$P(\text{eiLMIT}) = [P(D_d)\times\lambda_1] + [P(I32)/3\times\lambda_1] + [P(I32)/3\times\lambda_2]$$

where $P(D_d) = P(I21) + P(I22)$. The probability of EMIT is then:

$$P(\text{EMIT}) = [P(D_d)\times(1-\lambda_1)] + [P(I32)/3\times(1-\lambda_1)] + [P(I32)/3\times(1-\lambda_2)]$$

Table 6
Static instruction-type breakdown

Instruction type		Programs		
		Alarm	Prog	Krider
S	#	642	810	111
	P(S)	0.0784	0.0989	0.0135
DN	#	722	386	222
	P(DN)	0.1763	0.0942	0.0542
D2	#	115	31	2
	P(D2)	0.0281	0.0076	0.0005
TN	#	288	72	25
	P(TN)	0.1055	0.0264	0.0092
T23	#	0	0	0
	P(T23)	0.0000	0.0000	0.0000
T3	#	33	7	14
	P(T3)	0.0121	0.0026	0.0051
UP	#	4913	6311	7516
	P(UP)	0.5997	0.7704	0.9175

Table 7
Dynamic instruction-type breakdown (10000000 cycles)

Instruction Type		Programs		
		Alarm	Prog	Krider
I11	#	810168	4600789	309511
	P(I11)	0.0812	0.4601	0.0319
I12	#	69436	67891	151444
	P(I12)	0.0139	0.0136	0.0312
I14	#	97124	9775	0
	P(I14)	0.0389	0.0039	0.0000
I21	#	1225045	2541691	2131189
	P(I21)	0.1228	0.2542	0.2198
I22	#	834183	1239630	3323564
	P(I22)	0.1672	0.2479	0.6855
I32	#	2873750	101510	153259
	P(I32)	0.5760	0.0203	0.0316

Or, simply:

$$P(\text{EMIT}) = 1 - P(\text{Normal}) - P(\text{LMIT}) - P(\text{eiLMIT})$$

Table 5 summarises the error classification equations.

4. Experimental procedure and results

In order to validate the statistical model, a 'Monte Carlo' style simulation was carried out on three real-world programs, 'Prog', 'Alarm', and 'Krider'. The experimental procedure and the results are described in this section.

4.1. Sub-cycle level simulator

For reasons considered at the start of this paper, it was necessary to construct a microcontroller simulator capable of sub-cycle resolution in order to correctly model both EMIT and LMIT errors.

The resulting program simulates the generic 8051 microcontroller core with 8 kB of internal ROM and all on-chip peripherals, with the exception (in the version used here) of the serial port module. External RAM (XRAM) is also available to the user, but the address, data bus and control signals are not emulated at the ports (P0, P2 and P3).

This simulator operates at the oscillation cycle level. All major events such as instruction reads, timing register updates and port updates are carried out at the precise system state and phase according to the details in the relevant data sheets [13,14].

The simulation engine itself was written in Visual C++ as a dynamically-linked library (DLL) while the graphical interface was developed in Visual Basic. This configuration gives good speed performance and, more importantly, allows the use of Microsoft's Scripting Control (the VBScript parser).

4.2. Program description

All the embedded programs were written in C and

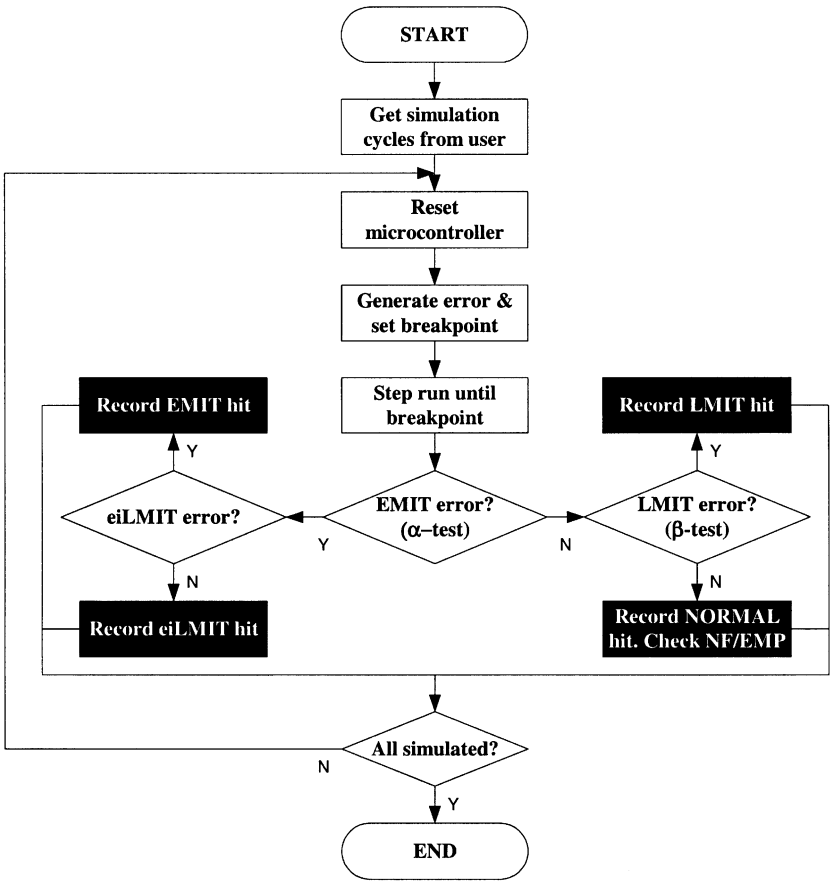


Fig. 4. Simplified error checking routine flowchart.

compiled with the Keil C51 v5.20 cross-compiler. They utilised standard I/O ports and timers that are present in all variants of the 8051 family of microcontrollers.

'Prog' is identical to 'Prog_A' used in [7–9], and is used here to compare the behaviour of the current and previous models. 'Alarm' [7,9] is an alarm clock that has a user interface made up of four buttons and a liquid-crystal display. 'Krider' is another simple application that turns eight LEDs on and off, creating the 'Knight Rider' effect.

Please note that the programs employ use two different system architectures. 'Prog' and 'Krider' are written in a procedural manner. By contrast, 'Alarm' is task-based and runs using a co-operative scheduler [11]. In addition, 'Prog' also uses code memory locations to store the coefficients for a finite impulse response (FIR) filter.

Table 6 shows the static instruction-type profile for the test programs. The dynamic profiles, obtained by simulating each program for 10000000 clock cycles, are shown in Table 7. The total clock cycles are less than 10000000 as the handling of each interrupt generates a hidden 'Long CALL' (LCALL) instruction requiring two clock cycles. The probability of each instruction-type occurring is based on the number of clock cycles in which instructions were executed (not the total number simulated).

'Alarm', 'Prog' and 'Krider' required 3281, 1889 and 684 code memory locations, respectively. In these programs, 86.6, 52.2 and 93.7%, respectively, of the execution time was spent executing multibyte instructions.

4.3. Simulation procedure

As with the previous simulation [8], a large number of simulation cycles were required in order to provide a high level of confidence in the results obtained. These simulations were executed without user intervention under the control of test scripts.

The simulation procedure is graphically represented as a flowchart in Fig. 4. Each program was subjected to 1000 simulation cycles with an error randomly injected between the first and 10000000th clock cycle. Before each cycle commences, the microcontroller was reset and the erroneous clock cycle (EIC) and IP value (EIP) were pseudo-randomly determined. A breakpoint was then set at EIC and the simulator began the program execution.

When the breakpoint was reached, it was cleared and an EMIT test (α-test) was carried out. This involved checking the multibyte flag (part of the simulator): this is set when a multibyte instruction is being executed and some bytes

Table 8
Simulation results for 1000 error cycles

Program	eiLMIT		EMIT		LMIT		Normal	
	#	%	#	%	#	%	#	%
Alarm	58	5.8	314	31.4	101	10.1	527	52.7
Prog	9	0.9	190	19.0	48	4.8	753	75.3
Krider	5	0.5	242	24.2	27	2.7	726	72.6

remain unread. If the multibyte flag was set, the simulation script tested for EMIT or eiLMIT errors (λ-test). If the multibyte flag was clear, a test for 'Normal' or LMIT errors (β-test) was conducted.

λ-testing involves checking for valid eiLMIT conditions (as listed in Table 2): this requires fewer test conditions than are required when testing for non-eiLMIT errors. The λ-test itself involves testing for (1) D_{EMIT} and T_{EMIT2} conditions and (2) for T_{EMIT1} conditions. Detected eiLMIT errors were recorded as eiLMIT hits while non-eiLMIT errors were recorded as EMIT hits.

Testing for 'Normal' or LMIT errors (β-test) is simpler as it involves interrogating a table containing all LMIT locations: this table was generated with an in-house program. LMIT errors were logged as LMIT hits while non-LMIT errors were recorded as NOR hits.

Upon completion, the entire cycle was repeated until the required number of cycles was simulated.

Please note that in the tests here, unlike those in our previous study [8], no additional cycles were simulated to determine the actual 'damage' that was caused by the IP error once it had been injected. This step was not carried out as it is very heavily implementation dependent: for example, in a system controlling an industrial robot, the impact could be a fatality, while in an alarm clock, the impact will be less severe. All we can say with certainty is that, in the event of an IP error, the program will not operate as intended.

Programming techniques that can help ameliorate the impact of IP corruption are discussed in Section 5.3.

4.4. Test results

The experiments were carried out on two Pentium II class machines with 128 MB RAM. Each 1000-cycle simulation took, on average, 12 hours to complete. The results and summary, shown in Table 8, were stored as text files during the simulation.

To gauge the accuracy of the model, both the simulated and calculated results are shown together in Table 9. By comparing the simulated and calculated α, β and λ coefficients (Table 10), the accuracy of each part of the model (Fig. 3) can be determined.

5. Discussion

5.1. Modelling of IP errors based on program code

Through simulation, we have shown that the model described in this paper is able to predict the probability of IP errors occurring: this may be useful when estimating a program's reliability. Specifically, the correlation between the simulated and calculated results falls within 5%. The remaining differences are due to the small number of simulation cycles and the fact that some unprogrammed code memory locations within the program code have not been taken into consideration in the simulations. In addition, constants stored in code memory (e.g. in 'Prog') are not considered in the simulations.

Table 10 compares the coefficients to give an indication of the accuracy of each part of the model. As expected, λ generally showed the biggest difference between the simulated and calculated results, primarily due to its complex conditions. However, the α coefficient for 'Krider' also shows a significant difference: further studies are required to explain this.

5.2. Simulation results

From Table 8, it is evident that, in addition to LMIT errors, eiLMIT and EMIT errors do indeed occur. In fact, EMIT and eiLMIT errors occur more frequently than LMIT errors in these test programs. This is to be expected: the number of EMIT and eiLMIT errors is related to the number of multibyte instructions. In all three programs, more than 50% of the program duration is spent executing such instructions.

The impact of differences in programming methodologies can be seen in Table 7 by comparing 'Prog' and 'Krider' with 'Alarm'. In terms of instruction-types executed, 'Alarm' spends 58% of the time executing I32 instructions compared to 2–3% for 'Prog' and 'Krider' respectively.

Table 9
Comparison between calculated and simulated results (%)

Program	eiLMIT			EMIT			LMIT			Normal		
	Sim.	Cal.	Dif.	Sim.	Cal.	Dif.	Sim.	Cal.	Dif.	Sim.	Cal.	Dif.
Alarm	5.80	7.99	**2.19**	31.40	31.13	**0.27**	10.10	9.89	**0.21**	52.70	50.99	**1.71**
Prog	0.90	1.34	**0.44**	19.00	18.59	**0.41**	4.80	5.25	**0.45**	75.30	74.83	**0.47**
Krider	0.50	1.06	**0.56**	24.20	28.64	**4.44**	2.70	2.45	**0.25**	72.60	67.84	**4.76**

Table 10
Comparison for α, β and λ coefficients

Program	Coefficient α			Coefficient β			Coefficient λ		
	Sim.	Cal.	Dif.	Sim.	Cal.	Dif.	Sim.	Cal.	Dif.
Alarm	37.20	39.12	**1.92**	16.08	16.25	**0.17**	15.59	20.42	**4.83**
Prog	20.20	19.92	**0.28**	6.11	6.56	**0.45**	4.52	6.70	**2.18**
Krider	24.70	29.71	**5.01**	3.59	3.49	**0.10**	2.02	3.58	**1.56**

This large difference is attributed to the triple-byte instruction used for the 'Idle Loop': this loop is entered by the scheduler when no tasks are due to execute.

It might be concluded from these results that time-triggered (and event-driven) programs are likely to prove more prone to EMIT and eiLMIT errors because of the time the processor spends in the Idle Loop (or 'Super Loop'). However, such a conclusion would be premature. In many time-triggered (and event-triggered) applications, the processor will be placed in 'idle' mode when it enters the Idle Loop [11]. This action is carried out primarily in order to reduce power consumption. However, it may also mean that the system has greater immunity to IP errors during this period. The simulator used in this study was not able to reproduce the impact of idle mode, and these issues were therefore not addressed in the results presented here.

The probability of EMIT and eiLMIT errors happening is governed by the percentage of time spent executing I22 and I21 instructions. However, if we compare the percentage of time executing multibyte instructions (from Section 4.2: 'Alarm'-86.6%, 'Prog'-52.2%, and 'Krider'-93.7%) with the percentage of EMIT and eiLMIT hits recorded ('Alarm'-37.2%, 'Prog'-19.9%, and 'Krider'-24.7%), we do see an inconsistency. 'Krider' recorded fewer EMIT and eiLMIT hits than 'Alarm' even though the percentage of time it spends executing multibyte instructions is higher. This is due to the fact that 'Krider' had a disproportionately high percentage of I22-type instructions, which only causes EMIT or eiLMIT errors 1/4 of its entire instruction cycle. The same trend is also shown in the calculated results (Table 9).

5.3. Reducing the impact of IP corruption

If IP corruption occurs, there are ways of reducing the impact it will have on the system itself. For example, two software-based techniques for detecting and correcting IP errors, 'NOP Fills' (NFs) and 'Function Tokens' (FTs) were discussed in [8].

If we consider only the 'Normal' IP error category (ignoring the effects of MIT errors for the moment), examination of the generated log files showed that 60.3, 78.6 and 92.3%[6] of IP errors for 'Alarm', 'Prog' and 'Krider', respectively, would have vectored program execution to unused program code locations. As a result, had NF been implemented, these errors would most probably be detected. IP errors vectoring to populated code memory locations can also be detected using FTs: however, as discussed in [8], the use of FTs has a number of disadvantages.

When taking MIT errors into account, NF and FTs will fail to detect program-flow errors under certain conditions. For example, if EMIT errors occurred on a branch instruction, or LMIT/eiLMIT errors 'create' branch instructions, there is the risk that the processor may become trapped in an Idle Loop. Such a possibility exists for scheduler-based and interrupt-driven systems, and with many real-time operating systems. This can be particularly problematic if the interrupt system is inactive or disabled.

An example is shown in Listing 3a

```
0100 94C2    SUBB  A,#C2H
0102 AF80    MOV   R7,80H
0104 FC      MOV   R4,A
0105 6E      XRL   A,R6
```

Listing 3a. Processor caught in permanent 'Idle Loop' due to MIT error.

where an IP error vectoring processing to location 0×101 would produce the equivalent instruction shown in Listing 3b.

```
0101 C2AF    CLR   AFH
0103 80FC    SJMP  0101H
0105 6E      XRL   A,R6
```

Listing 3b. Processor caught in permanent 'Idle Loop' due to MIT error.

The first 'created' instruction disables all interrupts while the following instruction causes the processor to loop to the first instruction, 'hanging' the processor.

Another scenario where NF and FT may not detect MIT errors is when the processor gets caught in an inescapable loop due to unmeetable loop-exit conditions. As a result the processor would end up executing the same code segment repeatedly.

Though these situations can happen, they will only occur when a particular combination of code sequence and program-flow error occurs.

While software techniques have been shown to be effective and feasible [2,5], the best solution will often be to use them as an adjunct to hardware-based protection.

[6] Note that these programs occupy 40.1, 23.1 and 8.3% of the 8 kB program code memory, which makes these figures within 2% of its actual value.

6. Conclusion

In this paper, results from a sub-cycle level simulator intended to provide an insight into the impact of instruction-pointer corruption on computer-based systems has been described, and an extension to the model described in [8] has been shown to be accurate enough to allow the prediction of the likely impact of IP errors. We have also shown that different programming and design methodologies can have an impact on a system's reliability.

References

[1] K. Armstrong, What EMC is, and some examples of EMC problems caused by software, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, 1998 (Conference code 98/471).
[2] N.Q. Burnham, C.F. Cowling, Fault-tolerant software in real-time single-chip microcontroller systems, Electronics Components and Applications 6 (1) (1984) 7–14.
[3] D. Campbell, Defensive software programming with embedded microcontrollers, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, Nov 1998 (Conference code 98/471).
[4] D.R. Coulson, EMC techniques for microprocessor software, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, 1998 (Conference code 98/471).
[5] M.D. Ker, Y.Y. Sung, Hardware/firmware co-design in an 8-bits microcontroller to solve the system-level ESD issue on keyboard, Microelectronics and Reliability 41 (3) (2001) 417–429.
[6] A. Niaussat, Software techniques for improving ST6 EMC performance, ST Application Note 1998 AN1015/0398, 1998.
[7] H.L.R. Ong, M.J. Pont, Empirical comparison of software-based error detection and correction techniques for embedded systems, 9th International Symposium on Hardware/Software Codesign (CODES 2001), Copenhagen, Denmark, 2001.
[8] H.L.R. Ong, M.J. Pont, W. Peasgood, A comparison of software-based techniques intended to increase the reliability of embedded applications in the presence of EMI, Microprocessors and Microsystems 24 (10) (2001) 481–491.
[9] H.L.R. Ong, M.J. Pont, W. Peasgood, Hardware–software tradeoffs when developing microcontroller-based applications for high-EMI environments, IEE Colloquium on Hardware-Software Co-Design, London, UK, 2000 (Conference code 00/111).
[10] C.R. Paul, Introduction to Electromagnetic Compatibility, Wiley, New York, 1992.
[11] M.J. Pont, Patterns for Time-Triggered Embedded Systems: Building Reliable Applications with the 8051 Family of Microcontrollers, ACM Press, New York, 2001.
[12] M.J. Pont, R. Kureemun, H.L.R. Ong, W. Peasgood, Increasing the reliability of embedded automotive applications in the presence of EMI: A pilot study, IEE Colloquium on Electromagnetic Compatibility of Software, Birmingham, UK, 1998 (Conference code 98/471).
[13] Siemens C515C 8-bit CMOS Mirocontroller User's Manual 08.96, Siemens AG, 1996.
[14] Atmel Microcontroller Data Book, 1997.