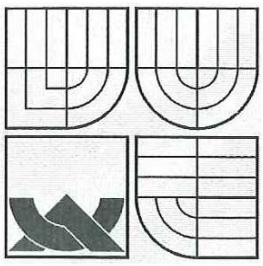
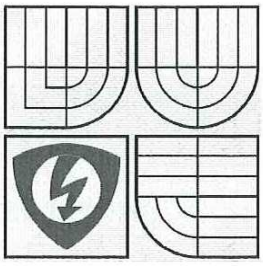


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

BEZPEČNÉ APLIKACE S MIKROKONTROLÉRY

SAFETY MICROCONTROLLER APPLICATIONS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. NIKOLA NACEV
AUTHOR

VEDOUCÍ PRÁCE — Ing. STANISLAV KLUSÁČEK
SUPERVISOR

BRNO 2008

VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ
Fakulta elektrotechniky
a komunikačních technologií
Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student: Nacev Nikola, Bc.
Ročník: 2

ID: 88555
Akademický rok: 2007/08

NÁZEV TÉMATU:

Bezpečné aplikace s mikrokontroléry

POKYNY PRO VYPRACOVÁNÍ:

Seznamte se s analýzou možných poruch vznikajících v mikrokontrolérech při dlouhodobém provozu s ohledem na spolehlivost a bezpečnost embedded systému.
Vypracujte přehled metod hardwarové a softwarové detekce jednotlivých poruch, hodnocení z hlediska spolehlivosti a náročnosti implementace.
Proveďte aplikaci vybraných metod ve spolupráci s vývojovým centrem Honeywell v Brně.

DOPORUČENÁ LITERATURA:

Dle doporučení vedoucího práce a konzultanta.

Termín zadání: 3.12.2007

Termín odevzdání: 26.5.2008

Vedoucí projektu: Ing. Stanislav Klusáček

prof. Ing. Pavel Jura, CSc.
předseda oborové rady

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ Fakulta elektrotechniky a komunikačních technologií
Ústav automatizace a měřicí techniky

UPOZORNĚNÍ:

LICENČNÍ SMLOUVA
POSKYTOVANÁ K VÝKONU PRÁVA UŽÍT ŠKOLNÍ DÍLO

uzavřená mezi smluvními stranami:

1. **Pan/paní**

   Jméno a příjmení: Bc. Nikola Nacev

   Bytem:

   Narozen/a (datum a místo): 22.8.1984, Ústí nad Orlicí

(dále jen "autor")

a

2. **Vysoké učení technické v Brně**

   Fakulta elektrotechniky a komunikačních technologií

   se sídlem Údolní 244/53, 60200 Brno 2

   jejímž jménem jedná na základě písemného pověření děkanem fakulty:

   doc. Ing. Václav Jirsík, CSc.

(dále jen "nabyvatel")

Článek 1
Specifikace školního díla

1. Předmětem této smlouvy je vysokoškolská kvalifikační práce (VŠKP):
   - ☐ disertační práce
   - ☑ diplomová práce
   - ☐ bakalářská práce

   jiná práce, jejíž druh je specifikován jako .....

(dále jen VŠKP nebo dílo)

Název VŠKP: Bezpečné aplikace s mikrokontroléry
Vedoucí/školitel VŠKP: Ing. Stanislav Klusáček
Ústav: Ústav automatizace a měřicí techniky
Datum obhajoby VŠKP: .....

VŠKP odevzdal autor nabyvateli v:

- ☑ tištěné formě - počet exemplářů 1
- ☑ elektronické formě - počet exemplářů 1

2. Autor prohlašuje, že vytvořil samostatnou vlastní tvůrčí činností dílo shora popsané a specifikované. Autor dále prohlašuje, že při zpracovávání díla se sám nedostal do rozporu s autorským zákonem a předpisy souvisejícími a že je dílo dílem původním.
3. Dílo je chráněno jako dílo dle autorského zákona v platném znění.
4. Autor potvrzuje, že listinná a elektronická verze díla je identická.

Článek 2
Udělení licenčního oprávnění

1. Autor touto smlouvou poskytuje nabyvateli oprávnění (licenci) k výkonu práva uvedené dílo nevýdělečně užít, archivovat a zpřístupnit ke studijním, výukovým a výzkumným účelům včetně pořizovaní výpisů, opisů a rozmnoženin.
2. Licence je poskytována celosvětově, pro celou dobu trvání autorských a majetkových práv k dílu.
3. Autor souhlasí se zveřejněním díla v databázi přístupné v mezinárodní síti
   - ☑ ihned po uzavření této smlouvy
   - ☐ 1 rok po uzavření této smlouvy
   - ☐ 3 roky po uzavření této smlouvy
   - ☐ 5 let po uzavření této smlouvy
   - ☐ 10 let po uzavření této smlouvy

   (z důvodu utajení v něm obsažených informací)
4. Nevýdělečné zveřejňování díla nabyvatelem v souladu s ustanovením § 47b zákona č. 111/1998 Sb., v platném znění, nevyžaduje licenci a nabyvatel je k němu povinen a oprávněn ze zákona.

Článek 3
Závěrečná ustanovení

1. Smlouva je sepsána ve třech vyhotoveních s platností originálu, přičemž po jednom vyhotovení obdrží autor a nabyvatel, další vyhotovení je vloženo do VŠKP.
2. Vztahy mezi smluvními stranami vzniklé a neupravené touto smlouvou se řídí autorským zákonem, občanským zákoníkem, vysokoškolským zákonem, zákonem o archivnictví, v platném znění a popř. dalšími právními předpisy.
3. Licenční smlouva byla uzavřena na základě svobodné a pravé vůle smluvních stran, s plným porozuměním jejímu textu i důsledkům, nikoliv v tísni a za nápadně nevýhodných podmínek.
4. Licenční smlouva nabývá platnosti a účinnosti dnem jejího podpisu oběma smluvními stranami.

V Brně dne: 26. 5. 2008

....

Nabyvatel Autor

Vysoké učení technické v Brně
Fakulta elektrotechniky a komunikačních technologií
Ústav automatizace a měřící techniky

Bezpečné aplikace s mikrokontroléry

Diplomová práce

Obor: Kybernetika, automatizace a měření
Student: Bc. Nikola Nacev
Vedoucí práce: Ing. Stanislav Klusáček

Abstrakt:

Záměrem této práce byly popsat metody návrhu bezpečných systémů, provést analýzu vzniku možných poruch v mikrokontrolérech při dlouhodobém provozu, popsat softwarové a hardwarové metody detekce poruch a aplikovat March testy na mikrokontroléry. Pro aplikaci byly zvoleny testy MATS+, PMOVI a March SS. Vybrané testy byly modifikovány na slovně-orientovanou paměť. Dále byla provedena jejich analýza chybového pokrytí, doby trvání testu a velikosti potřebné paměti programu. Pro analýzu chybového pokrytí byla vytvořena virtuální paměť s funkčními poruchovými modely. March testy byly poté porovnány jak mezi sebou, tak i s jiným vzorkovým testem (checkerboard).

Klíčová slova:

FMEA, FTA, analýza poruch, metody detekce poruch, poruchy SRAM paměti, March testy, PMOVI, MATS+, March SS, chybové pokrytí, doba trvání testu, checkerboard test

Brno University of Technology
Faculty of Electrical Engineering and Communication
Department of control, Measurement and Instrumentation

Safety Microcontroller Applications

Thesis

Specialisation of study: Cybernetics, Control and Measurement
Student: Nikola Nacev
Supervisor: Ing. Stanislav Klusáček

Abstract:

The deals of thesis were described methods for designing safety applications, made analysis of possible microcontroller faults of long-run system, described software and hardware methods for fault detection in microcontroller and applied some March test to microcontroller. To application were chosen MATS+, PMOVI and March SS tests. These tests were modified to word-oriented memory. Further it was made analysis of modified tests to determination fault coverage, testing times and program memory requirement. To determination of fault coverage was created virtual memory with fault function models. March tests were compared with each other and with another pattern test (checkboard test).

Key words:
FMEA, FTA, fault analyse, fault detection methods, SRAM faults, March tests, PMOVI, MATS+, March SS, fault coverage, testing time, checkerboard test

Bibliografická citace

NACEV, Nikola. *Bezpečné aplikace s mikrokontroléry*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2008. s. 94, příloh 0. Vedoucí práce Ing. Stanislav Klusáček.

Prohlášení

„Prohlašuji, že svou diplomovou práci na téma Bezpečné aplikace s mikrokontroléry jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne : 26.5.2008 Podpis: Marek Mikola

Poděkování

Děkuji tímto Stanislavu Klusáčkovi a Milanu Kříži za cenné připomínky a rady při vypracování diplomové práce.

V Brně dne : 26.5.2008 Podpis: Marek Mikola

OBSAH

SEZNAM OBRÁZKŮ

SEZNAM TABULEK:

1. ÚVOD

Na bezpečnost systému je v současné době kladen velký důraz, hlavně jedná-li se o systém, který může svým selháním ohrozit život, poškodit majetek nebo životní prostředí. S rostoucí složitostí systémů se zvyšuje riziko jejich selhání. Cílem bezpečných aplikací je zajistit, že v případě poruchy, systém přejde do bezpečného stavu, aby nebyl pro osoby, ale ani své okolí nebezpečný.

Žádný systém není bezpečný. Vždy existuje nějaká posloupnost událostí, které způsobí poruchu systému. Největším zdrojem chyb, které později vedou k poruše systému, je člověk. A je také na něm, aby se chránil sám před sebou.

V dnešní době se pro řízení menších systémů často používají mikrokontroléry, které mají na čipu mnoho periferií. Tyto periferie (paměti, I/O porty, sběrnice, komunikační rozhraní, čítače, …) mohou kdykoliv vlivem okolních jevů, ale také chybou programátora selhat. Proto byly vymyšleny různé způsoby detekce poruch v periferiích, které mohou toto selhání způsobit. Při zjištění poruchy mikrokontrolér udělá příslušné kroky, aby systém přešel do bezpečného stavu.

Práce vysvětluje základní pojmy související s bezpečností a obsahuje popis metod používaných při vývoji softwaru pro mikrokontroléry. Dále je provedena analýza poruch vznikajících v mikrokontrolérech s ohledem na dlouhodobý provoz embedded systému. Pak jsou zde popsány nejčastější hardwarové a softwarové metody detekce poruch vznikajících v mikrokontrolérech. Na závěr tato práce popisuje aplikaci March testů pro detekci poruch v SRAM paměti mikrokontroléru.

2. ZÁKLADNÍ POJMY

2.1 BEZPEČNOST, BEZPEČNÝ SYSTÉM

Bezpečnost je definována mnoha způsoby. Zde jsou uvedeny dvě definice.

1. Bezpečnost je vlastnost systému, že nebude ohrožovat lidský život nebo jeho okolí. [1]
2. Bezpečnost je pravděpodobnost S(t), že systém buď bude pracovat správně nebo bude jeho funkce přerušena, aby nenarušil funkčnost dalších systému, či neohrozil lidské životy. [2]

Jestliže je zjištěno, že systém nepracuje správně, je nutné zajistit, aby přešel do bezpečného stavu. Bezpečným stavem se myslí, aby neohrožoval lidské zdraví, ale ani své okolí. Systém, který toto splňuje, je bezpečný.

Například autopilot letadla. Jestliže tento systém v letadle přestane fungovat, pilot může nadále bezpečně řídit letadlo než doletí na letiště, tam je závada odstraněna. Dalším příkladem může být ventil v chemickém procesu, který se v případě poruchy uzavře.

2.2 CHYBA, PORUCHA A SELHÁNÍ [2]

a) Chyba

Je fyzická závada, která se vyskytne na nějakém zařízení systému nebo subsystému. Tato závada se může vyskytnout na hardwaru i softwaru.

Příkladem takové softwarové chyby je třeba odskok programu na neexistující adresu v programu nebo program uvízne v neplánované, nekonečné smyčce. Běžnou hardwarovou chybou je elektrický zkrat.

b) Porucha

Je následkem chyby, kdy systém už není schopen plnit svou obvyklou činnost.

Například vlivem elektrického zkratu je ventil v procesu neustále otevřen a systém se marně snaží tento ventil zavřít.

c) Selhání

Je následkem poruchy. Systém přestal plnit svou funkci a začal ohrožovat lidské životy nebo okolí.

Pokud budeme pokračovat v uvedeném předchozím příkladu (viz porucha), je to selhání systému. Neustále otevřeným ventilem bude do nádrže přitékat nebezpečná kapalina, která z této nádrže vyteče a začne ohrožovat okolí.

2.3 EMBEDDED SYSTÉM [3]

Embedded systém je systém, který je vytvořen pro jednu konkrétní aplikaci. Je to tedy jednoúčelový systém vykonávající předem definovanou činnost. Někdy se také nazývá vestavěný systém. Ten je řízen počítačem, který je uzpůsobený pro řízení právě jedné konkrétní činnosti, takovým počítačem může být mikrokontrolér. Mikrokontroléry mají na čipu naintegrované periferie zajišťující níže uvedené funkce systému.

Embedded systém je obvykle systém, který pracuje v reálném čase a zajišťuje funkce:

- sběr dat
- zpracování dat
- řízení systému na základě změřených dat
- interakce s člověkem

Běžným embedded systému je třeba autopilot letadla, pračka, telefonní ústředna, mikrovlnná trouba, atd..

3. METODY NÁVRHU SOFTWARU PRO BEZPEČNÉ APLIKACE

Protože je dnes kladen na bezpečnost systémů velký důraz, byly vytvořeny metody pro vývoj softwaru i hardwaru, aby aplikace splňovali podmínky bezpečného systému.

V literatuře se nejvíce píše o metodách FMEA (Failure Mode and Effect Analysis) a FTA (Fault-Tree Analysis). Tyto metody jsou dnes používané výrobci, kteří vyrábějí bezpečné embedded systémy (Honeywell, Siemens, …).

3.1 FMEA [5]

FMEA (Failure Mode and Effect Analysis) je obecná, analytická metoda s vyhodnocováním kvality jednotlivých částí systému. Používá se pro systematické odhalování poruch a odhadování možných rizik spojených s danou poruchou. Hlavním cílem je zabránit vzniku rizik nebo alespoň dostat tyto rizika pod námi stanovenou mez. FMEA je metoda, která se aplikuje jak na hardwarovou část systému, tak i na jeho řídící software.

Metoda je založena na vytvoření tabulky, kde pro jednotlivé možnosti poruch se spočítají RPN (Risk Priority Numbers), které svojí hodnotou od 1 do 1000 určují, jak hodně je daná porucha nebezpečná. Podle RPN hodnoty se dělají v systému úpravy, aby se docílilo nižší hodnoty RPN.

V tabulce jsou sloupce s názvy, jak je zde uvedeno:

Část	Funkce	Druh poruchy	Způsobení poruchy	Následky poruchy	Rating závažnosti	Rating vzniku	Možný způsob detekce	Rating detekce	RPN

tab. 3.1 - Zobrazení názvů sloupců tabulky metody FMEA

RPN hodnoty jednotlivých poruch určují kvantitativní odhad zahrnující 3 kriteria:

- Rating závažnosti – závažnost poruchy je vyjádřená hodnotou od 1 do 10.
- Rating vzniku – pravděpodobnost vzniku poruchy je daná hodnotou 1 až 10.
- Rating detekce – obtížnost detekování poruchy je určena hodnotou 1 až 10.

Hodnota RPN je pak vypočítána podle vztahu (3.1), kde jsou zahrnuty jednotlivé ratingy.

$$RPN = RZ \times RV \times RD \text{, kde} \qquad (3.1)$$

RZ…rating závažnosti poruchy

RV…rating vzniku poruchy

RD…rating detekce

Na této metodě je nejdůležitější přidělování hodnoty ratingu pro jednotlivá kritéria. Ratingy se obvykle přidělují podle tabulek. Rating se přiděluje podle svého charakteristického rysu, a proto je pro každý rating vytvořena tabulka.

Rating	Kriterium: Závažnost poruchy
1	Nerozpoznatelný vliv na systém.
2	Nevratný výskyt AND/OR ojedinělý ohlas zákazníků.
3	Nevratný výskyt AND/OR ohlas zákazníků.
4	Vratný výskyt AND/OR ohlas převážné většiny zákazníků.
5	Omezení uspokojivé/vhodné funkce.
6	Ztráta uspokojivé/vhodné funkce.
7	Omezení primární funkce.
8	Ztráta primární funkce.
9	Problém s bezpečností AND/OR nevhodnost pro ovládání regulace s varováním.
10	Problém s bezpečností AND/OR nevhodnost pro ovládání regulace bez varování.

tab. 3.2 - Přiřazení hodnoty ratingu závažnosti

Rating	Kriterium: Pravděpodobnost výskytu
1	Porucha je nepravděpodobná.
2	větší než 0, ale menší než 0,1 na 1000 zařízení
3	0,5 na 1000 zařízení
4	1 na 1000 zařízení
5	2 na 1000 zařízení
6	5 na 1000 zařízení
7	10 na 1000 zařízení
8	20 na 1000 zařízení
9	50 na 1000 zařízení
10	100 a více na 1000 zařízení

tab. 3.3 - Přiřazení hodnoty ratingu výskytu

Rating	Kriterium: Možnost detekce
1	téměř jistá detekce
2	velmi vysoká šance zachycení
3	vysoká šance zachycení
4	středně vysoká šance zachycení
5	mírná šance zachycení
6	malá šance zachycení
7	velmi malá šance zachycení
8	mizivá šance zachycení
9	velmi mizivá šance zachycení
10	nezachytitelná

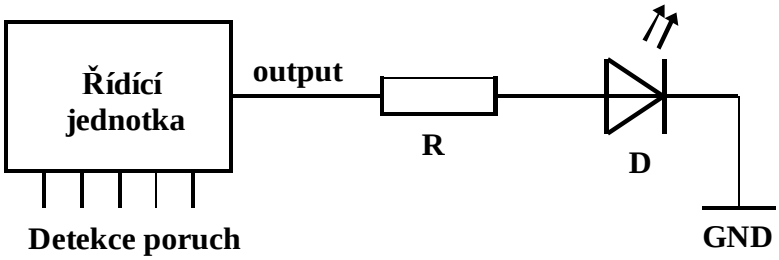
tab. 3.4 - Přiřazení hodnoty ratingu detekce

Po přidělení ratingů se v tabulce pro FMEA metodu dopočítávají RPN a podle výsledků se dělají taková opatření, aby došlo ke snížení. Opatřením může být například vhodná modifikace softwaru pro detekování poruchy a snížení jejího vlivu.

FMEA metoda je více zaměřená na hardware systému. Pro její použití je nutné mít podrobné informace o částech a prvcích systému. Tabulky 3.2 až 3.4 mohou být pro každý systém různé.

3.1.1 Příklad použití FMEA metody

Pro názorný příklad vytvoření tabulky pomocí metody FMEA byla použita signalizace poruchy brzdového systému auta. Na obr. 3.1 je zapojení signalizačního obvodu. Je to jednoduché sériové spojení LED diody D s rezistorem R. Jestliže bude zjištěna porucha na brzdícím systému, řídící jednotka nastaví výstup do logické jedničky (+5V) a LED dioda se rozsvítí. Po rozsvícení LED diody by měl řidič automobilu co nejdříve zastavit.



obr. 3.1– Zapojení obvodu signalizace poruchy brzdového systému auta

Pro vytvoření FMEA tabulky musíme definovat funkční celky nebo prvky, které mohou selhat a způsobit tím, že nebude signalizována porucha brzdového systému. Těmito funkčními celky a prvky jsou řídící jednotka, detekce poruch, rezistor R a LED dioda D. Jakmile známe možné prvky a funkční celky, můžeme určit možné poruchy a jejich následky. Po zjištění následků se určí rating závažnosti (RZ). Potom se určuje rating vzniku (RV), který je dán pravděpodobností vzniku dané poruchy a následně se definuje rating detekce (RD). Ten zjistíme na základě obtížnosti detekce dané poruchy. Hodnoty ratingů se nastavují pomocí tab. 3.2, tab. 3.3 a tab. 3.4. Z hodnot všech ratingů, dále určíme hodnotu RPN, kterou vypočítáme podle vzorce (3.1).

Z výsledných hodnot RPN najdeme největší a pokusíme se udělat v návrhu systému takové úpravy, abychom tuto hodnotu co nejvíce snížili. Můžeme také říci, že nám takové hodnoty RPN vyhovují.

Část	Funkce	Druh poruchy	Způsobení poruchy	Následky poruchy	Možný způsob detekce	RZ	RV	RD	RPN
R (Rezistor)	Omezení proudu tekoucího do diody D	Nevede	Špatně zapájený nebo vadný	Nefunkční signalizace poruchy (nesvítí)	Diagnostika při výrobě	8	5	1	40
		Zkratovaný		Možné zničení diody		8	5	1	40
D (LED)	Signalizace poruchy	Nevede	Špatně zapájená nebo vadná	Nefunkční signalizace poruchy (nesvítí)	Diagnostika při výrobě	8	5	1	40
		Zkratovaná				8	5	1	40
Detekce poruchy	Oznámení vzniku poruchy řídící jednotce	Přerušení kabelu přívodů	Mechanické poškození	Nefunkční detekování poruchy	Kontrola řídící jednotkou	7	3	3	63
		Selhání některé z detekcí	Mechanické poškození	Nefunkční jedna z detekcí poruch		5	4	3	60
Řídící jednotka	Spouštění signalizace a sběr dat od detekcí poruch	Nefunkční	Vadná	Nefunkční signalizace poruchy	Diagnostika při výrobě	8	1	1	8
		Špatný výstup pro spuštění signalizace	Zkratovaný R	Nefunkční signalizace poruchy	Pomocí redundantní řídící jednotky	8	2	4	64
		Nevede (výstup nebo některý z výstupů)	Špatně zapájený	Nesignalizuje některé poruchy nebo nesvití LED	Diagnostika při výrobě	6	5	1	30

tab. 3.5 - FMEA tabulka pro zapojení signalizace poruchy brzd automobilu

3.2 FTA [6]

FTA (Faul-Tree Analysis) je další metoda pro vývoj spolehlivého a bezpečného systému. Jedná se o jednu z analyticko logických technik, která se hojně používá při vývoji systému.

Metoda FTA je založena na vytváření stromů poruchových stavů (diagramu), kde se ke každé možné poruše udělá diagram, za jakých událostí k této poruše může dojít. Jedná se tedy o grafické znázornění posloupností událostí, které vedou k poruše. Poté se vytvářejí opatření, aby se vznik poruchy v systému eliminoval.

Cíle FTA analýzy [7]:

- Identifikovat příčiny nebo jejich kombinace, které mají za následek poruchu.
- Určovat zda ukazatel bezporuchovosti systému, splní určené požadavky.
- Určovat faktory, které nejvíce ovlivňují ukazatele bezporuchovosti a určovat změny, které jsou nutné pro zlepšení ukazatelů.
- Určovat společné události nebo poruchy na základě všeobecných událostí.

3.2.1 Postup vývoje pomocí FTA

Předpokladem úspěšného návrhu systému pomocí metody FTA je důkladná znalost systému. Aby byl systém dobře zidentifikovaný, je potřeba, aby se na jeho popisu podíleli odborníci, kteří danému systému rozumí. Cílem této identifikace je najít vrcholovou událost, která znemožní vykonávat systému požadovanou funkci, jinak může systém přejít do nebezpečného stavu.

Po identifikaci systému a určení vrcholových událostí, je možné začít utvářet stromy poruchových stavů. Poruchové stromy se konstruují dvěma způsoby:

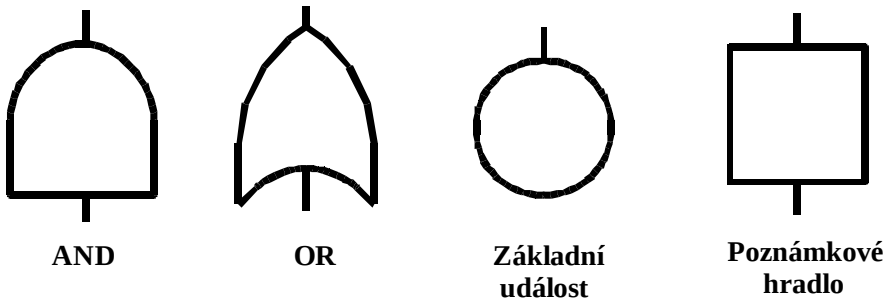
- Svislé – kde vrcholová událost je umístěna na vrcholu stromu (více používaný způsob konstrukce)
- Vodorovně – vrcholová událost je úplně vlevo nebo vpravo

Poruchové stromy obsahují bloky pro přehledné zobrazení událostí, které vedou k vrcholové události.

Obsahují:

- bloky událostí
- bloky pro popis událostí
- logická hradla poruchových stromů
- značky pro přenos dat do/z diagramu

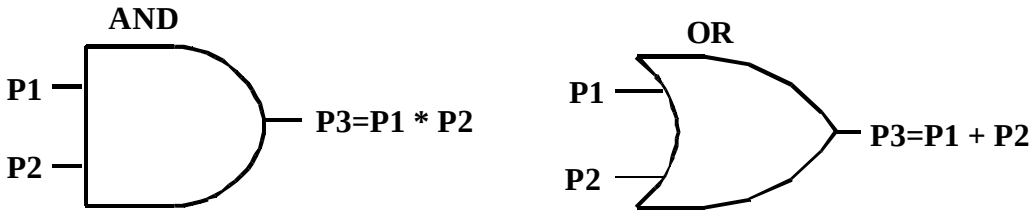
Pro konstrukci poruchového stromu se nejvíce používají logická hradla AND a OR, bloky událostí a bloky popisu, které jsou zobrazeny na obr. 3.2.



obr. 3.2 – Nejpoužívanější bloky při konstrukci FTA [6]

Po zhotovení poruchových stromů se ke každé události připisuje pravděpodobnost jejího vzniku a ta se určuje při identifikaci. Pravděpodobnosti se pak dále počítají až k vrcholové události. Způsob jejího výpočtu je závislý na chybovém toku, tedy na logických blocích mezi událostmi, které zaviňují poruchu systému. U vrcholové události je výsledná hodnota pravděpodobnosti, která určuje možný vznik této události.

Způsob výpočtu pravděpodobností je zobrazen na obr. 3.3, kde se vstupní pravděpodobnosti na hradle OR sčítají a na hradle AND násobí.



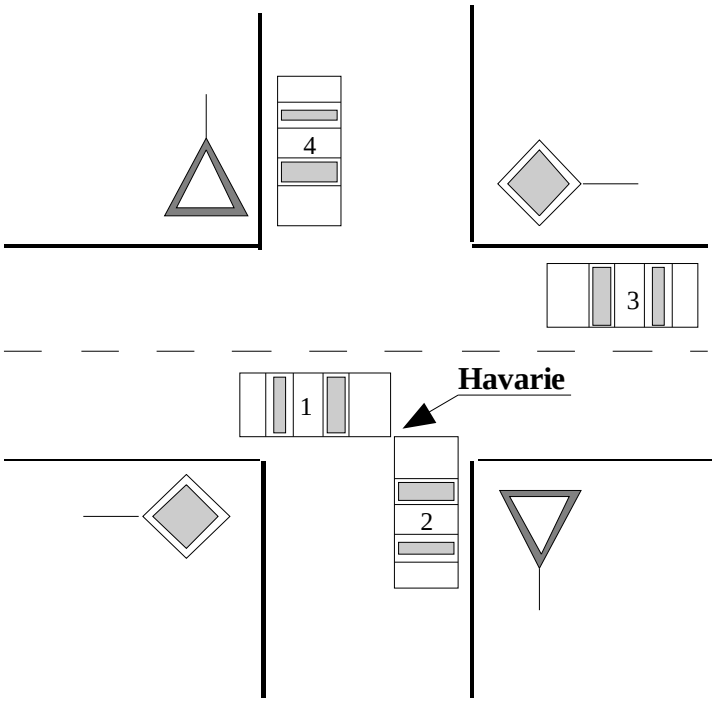
obr. 3.3 – Výpočet pravděpodobností na hradlech OR a AND

Po výpočtu pravděpodobností se vyhodnocují jednotlivé větve poruchového stromu a hledá se ta s největší pravděpodobností (větev, která nejvíce způsobuje možný vznik poruchy). Po nalezení této větve, se dělají takové úpravy v systému, aby se docílilo její minimální pravděpodobnosti. Po úpravách se opět počítají pravděpodobnosti a hledá se větev s největším významem na vzniku vrcholové události, dokud pravděpodobnost vrcholové události není pod určitou námi stanovenou hranicí, kdy považujeme systém za bezpečný a spolehlivý.

3.2.2 Příklad použití FTA metody

Pro názorný příklad zhotovení poruchového stromu pomocí FTA a výpočtu pravděpodobností byla vybrána křižovatka. Na obr. 3.4 je nakreslena situace, pro kterou se poruchový strom vytváří.

Nejprve se musí zidentifikovat možnosti vzniku havárie a rozdělit je možné kombinace. Ty popisují situaci, kdy řidič automobilu na vedlejší silnici vlivem okolností nebo řidičskou chybou nezastaví na hranici křižovatky a tím pádem nedá přednost v jízdě automobilu jedoucímu po hlavní silnici.

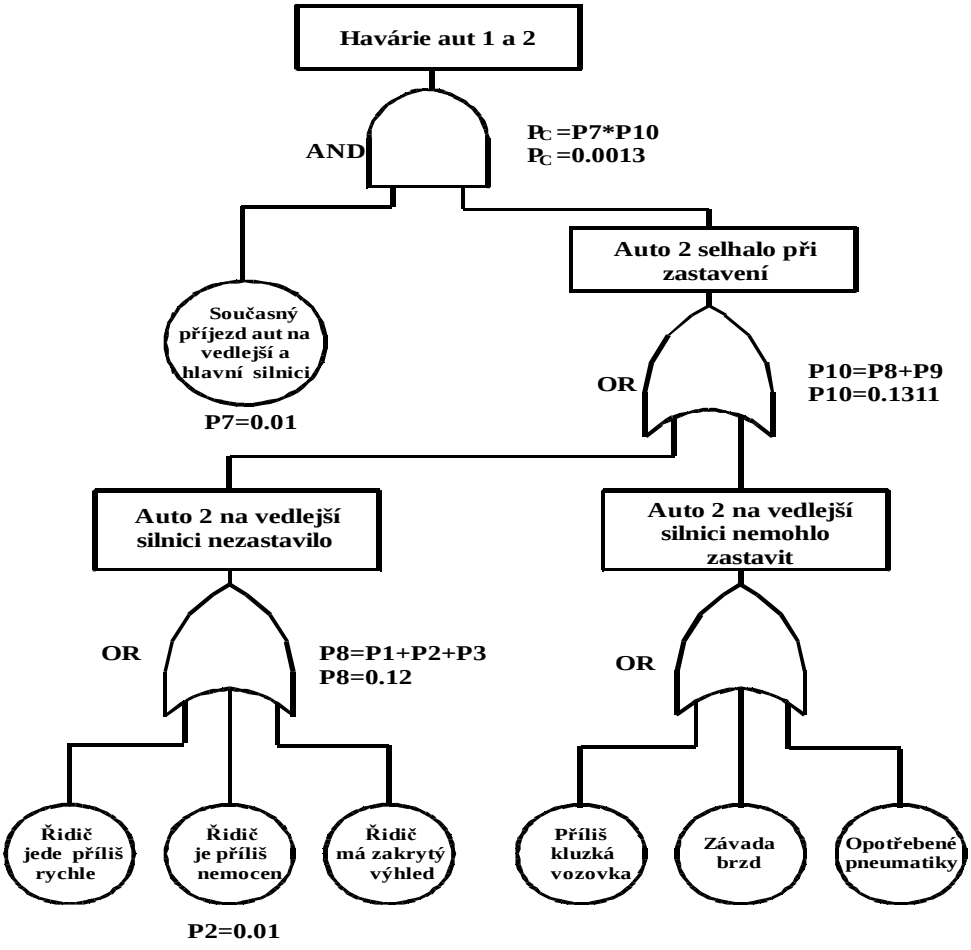


obr. 3.4 – Zobrazení situace na křižovatce

Příklady okolností nebo řidičských chyb:

- Řidič jede příliš rychle (P1 = 0,1).
- Řidič je nemocný (P2 = 0,01).
- Řidič má zakrytý výhled (P3 = 0,01).
- Vozovka je příliš kluzká (P4 = 0,01).
- Závada brzd (P5 = 0,001).
- Opotřebované pneumatiky (P6 = 0,0001).
- Současný příjezd automobilů na křižovatku po hlavní a vedlejší silnici, tak aby si křížili cestu (P7 = 0,01).

Po určení událostí, můžeme vytvořit poruchový strom, který je zobrazený na obr. 3.5.



obr. 3.5 – Poruchový strom pro situaci na obrázku 3.4

Každá okolnost a řidičská chyba je v poruchovém stromu vyznačena jako událost. Ta nastane s určitou pravděpodobností, která je uvedena v poruchovém stromu. Pravděpodobnost s jakou vznikne vrcholová událost říká, že pokud tento případ nastane 6 tisíckrát za rok, kdy auta přijíždí z vedlejší a hlavní silnice současně a kříží si cestu, pak se může očekávat 7-8 havárií ročně.

V poruchovém stromu je nejvýznamnější větev, která začíná událostí *Řidič jede příliš rychle*. Abychom snížili pravděpodobnost této události, musí se v křižovatce udělat taková opatření, aby byl řidič přinucen jet pomaleji. Takovým opatřením mohou být zpomalovací pruhy, které pravděpodobnost této události sníží z 0,1 na 0,03. Pak by výsledná pravděpodobnost byla přibližně $P_C = 0{,}0006$. Z 6000 případů uvedených situací z obrázku 4.3 by se očekávaly 3-4 havárie, tedy poloviční počet nehod než před úpravou křižovatky.

Zobrazeny poruchový strom ukazuje pouze jednu situaci, která může nastat na křižovatce – nejedná se tedy o celkovou analýzu křižovatky.

4. MIKROKONTROLÉRY

V této kapitole jsou převážně popisovány AVR mikrokontroléry od firmy Atmel. Je to jeden z mnoha typů mikrokontrolérů, které se používají jako řídící jednotka pro embedded systémy.

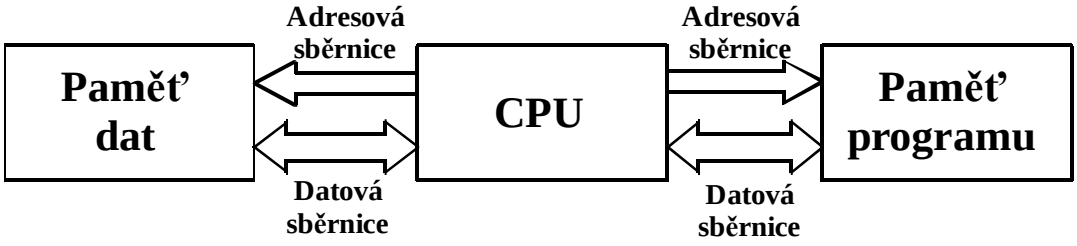
4.1 ZÁKLADNÍ VLASTNOSTI MIKROKONTROLÉRŮ

Architektura:

Mikrokontroléry AVR mají harvardskou architekturu, jejíž principiální schéma je zobrazeno na obr. 4.1. Jejím hlavním znakem je oddělená paměť pro program a pro data. Toto je hlavní rozdíl od Von Neumannovy architektury používané v PC, která má pro program i data společnou paměť.

Výhodou harvardské architektury je možný okamžitý přístup do obou pamětí zároveň, čímž se zrychlí činnost jádra mikrokontroléru.

Nevýhodou tohoto uspořádání je větší počet sběrnicí (2x adresová a 2x datová), což způsobuje větší počet vývodů CPU (složitost) a zároveň menší kapacitu pamětí než u Von Neumannovy architektury (1x datová, 1x adresová).



obr. 4.1 – Blokové schéma harvardské architektury [3]

Kategorie:

Mikrokontroléry AVR od firmy Atmel jsou typu RISC (Reduce Instruction Set Computer). RISC procesory vycházejí z poznatku, že při výpočtech je používáno omezené množství dostupných instrukcí procesoru. Proto není zapotřebí složitých instrukcí, které se provádějí více strojových cyklů. Protože instrukce RISC procesorů jsou běžně vykonávány během jednoho strojového cyklu, je umožněno používat řetězení instrukcí.

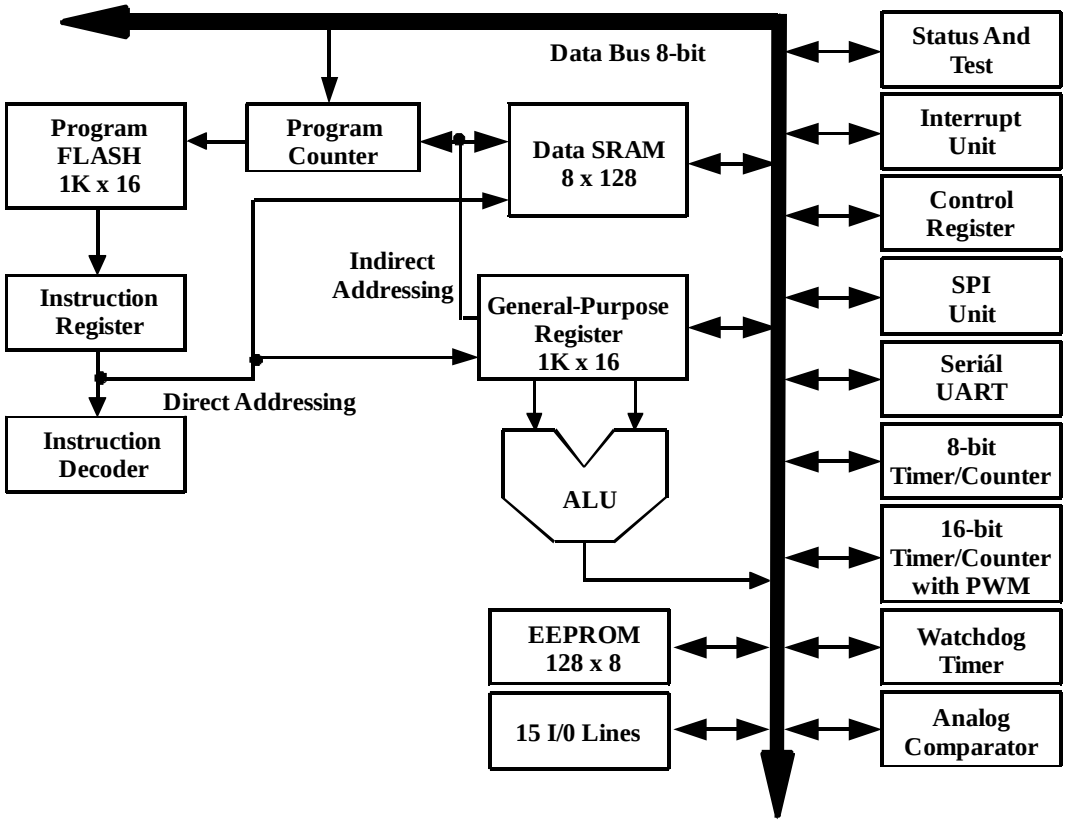
Zřetězení instrukcí (pipelining) vychází z myšlenky rozdělení instrukce mikrokontroléru na několik částí (načtení, dekódováni, provedení, čtení z paměti a zápis výsledku do paměti). Každou část instrukce pak zpracovávají různé části jádra mikrokontroléru. To má za následek, že mikrokontrolér vykonává více instrukcí naráz.

RISC mikrokontroléry se vyznačují základními vlastnostmi:

- omezená instrukční sada – jednoduché instrukce
- instrukce se vykonává jeden strojový cyklus
- využívá se technika řetězení instrukcí

4.2 PERIFERIE MIKROKONTROLÉRŮ

Na obr. 4.2 je znázorněno blokové schéma AVR mikrokontroléru s běžnými periferiemi. Každý typ mikrokontroléru má jinou velikost pamětí a různé periferie.



obr. 4.2 – Blokové schéma AVR mikrokontroléru

4.3 ANALÝZA VZNIKU PORUCH V MIKROKONTROLÉRECH PŘI DLOUHODOBÉM PROVOZU [8]

V mikrokontroléru obvykle vznikají poruchy působením vnějších vlivů nebo špatným návrhem embedded systému.

Nejčastějšími důvody vzniku poruch jsou:

- teplota – nedostatečné chlazení
- kolísavé napájecí napětí – špatně stabilizované napětí
- elektrické pole – silové vedení poblíž mikrokontroléru
- napěťové špičky – v obvodu spínání velkých výkonů

Pro analýzu poruch vznikajících na mikrokontroléru byla použita americká norma (UL Standard for Safety for Software in Programmable Components, UL 1998 [8]). V této normě jsou uvedeny způsoby kontroly a odhalování poruch, které mohou nastat na mikrokontroléru po uvedení do provozu.

Norma rozděluje mikrokontrolér na funkční celky (CPU, Clock, Memory, …) a určuje způsob kontroly jeho jednotlivých částí.

Výrobci mikrokontrolérů neuvádějí možné poruchy na jimi vyrobených mikrokontrolérech s ohledem na dlouhodobou činnost. V současnosti se spíše vymýšlí způsoby odhalování možných chyb, aby se případné poruše předešlo.

4.3.1 Poruchy mikrokontrolérů

Rozlišujeme dva základní druhy poruch v mikrokontroléru:

- zkrat
- přerušení – představuje otevřený obvod (rozpojeno) nebo neměnící se signál.

Dále se pak vyskytují poruchy, které jsou specifické pro určitou periferii mikrokontroléru – přeslechy v paměti, špatná frekvence zdroje hodin, chyba časování komunikačních zařízení atd..

CPU

a) Registry

- Chyba je typu přerušení – neměnící se logická hodnota jednoho či více bitů registru. Této chybě se předchází funkčním testem, kde se pomocí struktury registry otestují zkušebními daty, nebo periodickou samokontrolou (test statické paměti nebo pomocí parity).
- Chyba je typu zkrat – zkrat mezi přívody k registru. Tuto chybu je možné detekovat pomocí redundance CPU.

b) Programový čítač

Chyba programového čítače je poměrně závažná porucha, protože programový čítač ukazuje na adresu paměti programu, ve které je uložen strojový kód. Pokud je porouchaný, vznikají nežádoucí skoky v paměti programu.

V programovém čítači vznikají chyby obou typů, avšak vliv těchto chyb je stejný. Chyba typu přerušení se kontroluje technikou monitorováním logické sekvence programu nebo nezávislým monitorováním programu v časových intervalech. Chyba typu zkrat se kontroluje pomocí techniky periodické samokontroly a monitorování.

c) Adresování

Chyba adresování nastane zkratováním adresových vodičů. Vliv této chyby je stejný jako u programového čítače – nastane riziko, že se program nebude vykonávat sekvenčně a dojde ke skoku do paměti programu, kde není strojový kód nebo naopak nebude přístupná část paměti.

Pro detekci této chyby se používá redundantního CPU, kde se provádí vzájemná porovnání mezi dvěma CPU nezávislým HW komparátorem nebo periodickou samokontrolou používající testování modelu adresových spojů.

d) Datové cesty

K chybě na datových cestách dojde podobně jako u adresování, a to zkratováním datových vodičů. Tím může dojít k poškození přenášených dat.

Detekování této chyby se provádí porovnáváním dat za pomoci redundantního CPU a nezávislého HW komparátoru. Další možností je datová redundance.

Přerušovací jednotka:

Porouchaná přerušovací jednotka způsobí, že v mikrokontroléru se negenerují přerušení nebo naopak se generují příliš často bez žádosti. V případě, že se nevygeneruje přerušení v situaci kdy má, může nastat selhání systému. K selhání může také dojít i v případě, kdy přerušovací jednotka vygeneruje přerušení, ke kterému nebyl důvod (žádný příznak pro přerušení nebyl nastaven).

Detekce poruchy přerušovací jednotky se provádí funkčním testem nebo porovnáváním s redundantním přerušovacím kanálem.

Zdroj hodin

Mikrokontroléry obsahují vnitřní zdroje hodinových pulzů. Obvykle to bývá RC oscilátor. Vnitřní RC oscilátor je hodně závislý na teplotě nebo napájecím napětí mikrokontroléru. To může způsobit, že RC oscilátor špatně generuje hodinové pulsy a dochází k riskantním stavům při časově kritických událostech.

Pro kontrolu správné činnosti se provádí monitorování frekvence.

Paměť

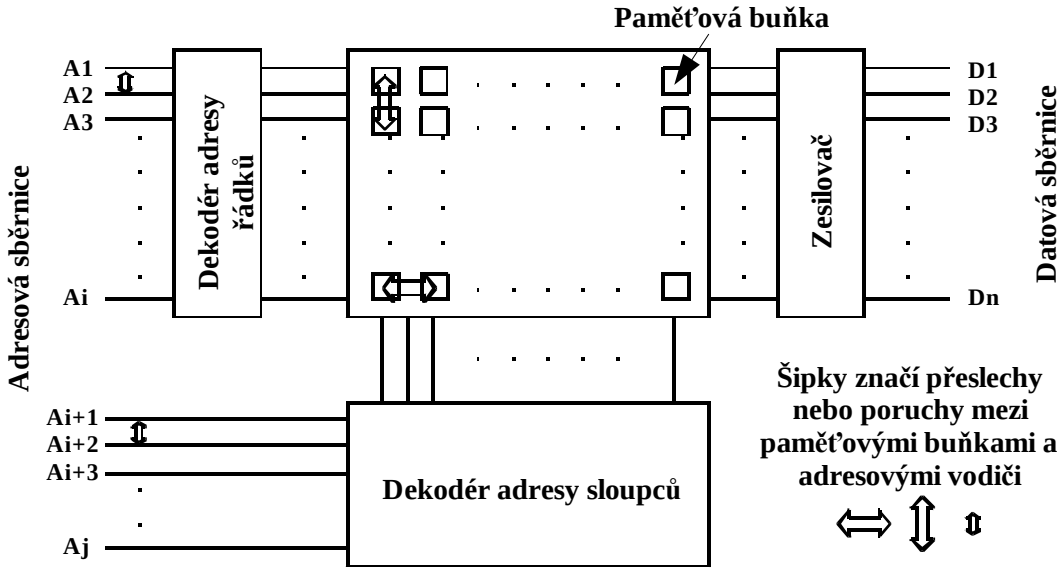
Všechny druhy pamětí v mikrokontroléru mají maticové uspořádáni, které popisuje obr. 4.3. Z něj vyplývá, že může docházet k chybám vlivem přeslechů bitů mezi sousedními byty paměti. Dále může docházet ke zkratům nebo přerušením vodičů adresové i datové sběrnice pamětí.

K detekování chyby vzniklé přerušením vodivé cesty na datové sběrnici se používá ochrana slov paměti pomocí paritního bitu, který se vkládá do každého slova.

Chyba adresování je způsobena zkratem mezi adresovými vodiči. Tato chyba se musí detekovat pomocí redundantního CPU, periodickou cyklickou kontrolou

nebo použitím ochrany slova paměti s využitím vícebitové redundance vkládané do paměťového slova.

Na energeticky závislých pamětích vznikají jiné chyby než na pamětích energeticky nezávislých. Proto jsou i jiné metody odhalování chyb pamětí.



obr. 4.3 – Uspořádání pamětí v mikrokontroléru [13]

a) Energeticky nezávislé paměti:

Na energeticky nezávislých pamětích (paměť programu, EEPROM) vznikají dva druhy chyb:

- jednobitové chyby
- vícebitové chyby

Oba druhy chyb vznikají převážně přeslechy mezi sousedními bity v paměti. Chyba vzniklá přeslechem není chyba stálá (nejedná se o zkrat). Pokud dojde ke zkratu mezi bity, jedná se o trvalé poškození paměti.

Jednobitovým chybám se předchází kontrolními součty (checksum). Používá se periodicky modifikovaný kontrolní součet, kde 1 paměťové slovo reprezentuje obsah všech slov v paměti. Toto slovo se po každém průchodu algoritmem (periodicky) generuje, porovnává s předchozí slovem a ukládá. V případě, že je vygenerované a předchozí slovo rozdílné, dojde k detekování chyby. Dalším druhem

kontrolního součtu je vícenásobný kontrolní součet, který je obdobný jako modifikovaný. Pouze v tomto případě je paměť rozdělena na části a součtové slovo se generuje, porovnává a ukládá pro každou část paměti zvlášť.

Jednobitové i vícebitové chyby se mohou detekovat pomocí redundantního CPU, kde se vzájemně porovnávají paměti jednotlivých CPU pomocí komparátoru. V případě odlišnosti paměti je chyba detekována.

b) Energeticky závislé paměti [4]:

Na energeticky závislé paměti mikrokontroléru mohou vzniknout pouze poruchy typu single-port, protože paměť obsahuje pouze jeden dekodér adresy (sloupce a řádku) a jednu vstupní/výstupní sběrnici. Tento druh poruch se dále dělí na:

- jednobitové (single-cell) chyby
- dvoubitové (two-cell) chyby

Tyto druhy poruch se dále dělí podle charakteristických vlastností a mají svoje označení.

Jednobitové chyby:

1. stavu (SF – State Fault): Trvalá logická hodnota buňky je 0 nebo 1. Nezáleží na prováděných operacích s touto buňkou.
2. přechodu (TF – Transition Fault): Tato chyba se projevuje tak, že po prvním zápisu ji již nejde změnit na opačnou hodnotu. Chyba se projevuje po operaci zápisu z 0 do 1 (0w1) nebo z 1 do 0 (1w0).
3. destruktivního zápisu (WDF – Write Destructive Fault): Zápis stejné hodnoty (0w0, 1w1) do buňky způsobí na výstupu hodnotu opačnou.
4. destruktivního čtení (RDF – Read Destructive Fault): Čtení buňky zapříčiní to, že její obsah se stane inverzní a vyčtená hodnota je nekorektní.
5. klamného destruktivního čtení (DRDF – Deceptive Read Destructive Fault): Čtení buňky způsobí, že obsah se stane inverzní, ale vyčtená hodnota je korektní.
6. chybného čtení (IRF – Incorrect Read Fault): Operace čtení nezpůsobí změnu obsahu buňky, ale výstup je nekorektní (opačná hodnota k obsahu buňky).

Dvoubitové chyby:

1. stavově závislé hodnoty (CFst – State Coupling Fault): Stav jedné buňky je přímo závislí na stavu druhé buňky.
2. destruktivní závislé operace (CFds – Disturb Coupling Fault): Provedení operace (čtení/zápis) nad jednou buňkou způsobí změnu nad buňkou jinou.
3. závislého přechodu (CFtr – Transition Coupling Fault): Jestliže je buňka v určitém logickém stavu, tak selže inverzní zápis (0w1, 1w0) do druhé buňky.
4. závislého destruktivního zápisu (CFwd – Write Destructive Coupling Fault): Bude-li v jedné buňce určitá hodnota (0,1), selže neinvertující zápis do druhé buňky. Hodnota druhé buňky bude tedy opačná než zapisovaná.
5. závislého destruktivního čtení (CFrd – Read Destructive Coupling Fault): Bude-li v jedné buňce určitá hodnota (0,1), pak při čtení druhé buňky se její hodnota zinvertuje a na výstupu bude tato zinvertovaná (nekorektní) hodnota.
6. klamného závislého destruktivního čtení (CFdrd – Deceptive Read Coupling Fault): Bude-li v jedné buňce určitá hodnota (0,1), pak při čtení druhé buňky se její hodnota zinvertuje, ale na výstupu bude původní (korektní) hodnota.
7. chybného závislého čtení (CFir – Incorrect Read Coupling Fault) – Bude-li v jedné buňce určitá hodnota (0,1), pak se při čtení hodnota druhé buňky nezmění, ale na výstupu bude její hodnota inverzní.

Výše uvedené poruchy paměti SRAM se odhalují pomoci různých druhů vzorkových testů. Nejznámějšími detekujícími poruchami jsou March testy.

Externí komunikace:

Při komunikaci s okolím může dojít ke třem druhům chyb:

1. Datová

Jednobitové, dvoubitové a tříbitové chyby se detekují CRC testy (Control Redundacy check) s jedním i více paměťovými slovy. Ty reprezentují všechna posílaná slova nebo se zavádí datová redundance posílaných dat, a také je možné provádět porovnání dat poslaných přes dva funkční kanály.

2. Adresové

Při špatně vygenerované adrese nedojde ke spojení mezi požadovaným zařízením a mikrokontrolérem.

3. Chyba časování

- špatné načasování

- špatná sekvence

I/O periferie:

- Digitální I/O:

Na digitálních I/O vznikají poruchy způsobené obvykle velkým proudem. Proud protékající výstupem nebo vstupem nesmí překročit hodnotu, kterou udává výrobce mikrokontroléru. Jinak hrozí proražení (přerušení) I/O nebo vznikne na I/O zkrat. Takto vzniklé poruchy jsou nevratné.

Poruchy na digitálních I/O se kontrolují pomocí redundantního CPU a komparátoru.

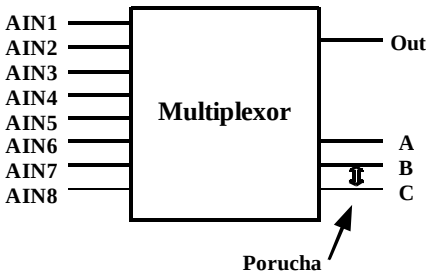
- AD převodník:

Přivedením většího analogového signálu než udává výrobce, může dojít k poruše AD převodníku. Porouchaný AD převodník dává na svém výstupu nesmyslné hodnoty.

Aby se detekovala chyba AD převodníku, používá se redundantní CPU, kde se provádí vzájemná porovnání výsledných digitálních hodnot z AD převodníků obou CPU.

- Analogový multiplexor:

Analogový multiplexor se používá k připojení analogového signálu, například k AD převodníku. Na multiplexoru může dojít k chybě adresování, kde se zkratují nebo přeruší adresové vstupy, jak je zobrazeno na obr. 4.4. Při této poruše nejdou připojit k AD převodníku některé analogové vstupy, jak je vidět v tabulkách na obr. 4.4.



Bez porucha

C	B	A	Out
0	0	0	AIN1
0	0	1	AIN2
0	1	0	AIN3
0	1	1	AIN4
1	0	0	AIN5
1	0	1	AIN6
1	1	0	AIN7
1	1	1	AIN8

S poruchou

C	B	A	Out
0	0	0	AIN1
0	0	1	AIN2
1	1	0	AIN7
1	1	1	AIN8
1	1	0	AIN7
1	1	1	AIN8
1	1	0	AIN7
1	1	1	AIN8

obr. 4.4 – Princip multiplexoru

5. METODY DETEKCE PORUCH V MIKROKONTROLÉRU

Ze semestrálního projektu 1 je patrné, že poruchy se mohou vyskytnout ve všech částech mikrokontroléru. Každá porucha se pak detekuje různými způsoby. Metody pro detekci poruch dělíme do dvou základních skupin:

- Hardwarové (HW) – za pomoci dalších hardwarových prvků je zařízení schopné odhalit poruchu mikrokontroléru.
- Softwarové (SW) – algoritmus, který vykonává sám mikrokontrolér, je schopen nalézt poruchu na určité periferii

Čistě hardwarové metody detekce poruch nejsou v praxi časté. Hardwarová metoda je obvykle závislá na softwarové podpoře. Software obvykle zprostředkovává komunikaci, porovnávání dat nebo signálů. V některých případech ovládá i přidaný redundantní hardwarový prvek. Naopak čistě softwarové metody se běžně používají. Takovým příkladem jsou bezpečnostní kódy pro ochranu dat v paměti (CRC, checsum) nebo SW sledování logického běhu programu.

V bezpečných embedded systémech se obvykle používá současně několik SW a HW metod. To je z toho důvodu, že žádná metoda není schopna vždy samostatně odhalit v reálném čase poruchu, která se může objevit v jakékoliv části mikrokontroléru. Snahou je docílit toho, aby byly odhaleny veškeré poruchy, které mohou zavinit selhání systému. Tím by se systém mohl stát nebezpečný pro své okolí.

Hardwarové metody mají několik výhod, ale také nevýhod. Jejich hlavní nevýhodou jsou náklady na realizaci, protože jsou založeny na redundaci periferií mikrokontroléru nebo obvodové části. Náklady na každou další součástku, která je v systému „navíc“, se projeví v celkové ceně zařízení. To je obvykle v rozporu s požadavky managementu výrobce, jelikož ten se snaží snížit výrobní náklady na minimum. Výhodou HW metod je vyšší spolehlivost a možnost odhalení většiny poruch.

Princip HW metod je založen na komparaci dat nebo signálů z mikrokontroléru a z redundantního HW. Na uvedeném principu některé metody nepracují. Takový příkladem je použití watchdogu.

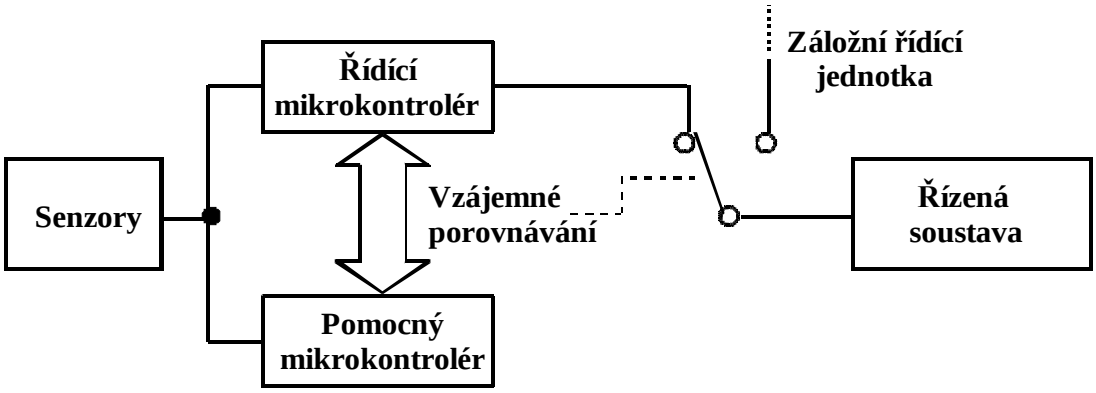
Softwarové metody jsou alternativou k HW metodám. Jejich největší výhodou je cena, a proto jsou častěji používány než HW metody. U SW metod se platí pouze za vývoj nebo pouhou implementaci algoritmu do řídícího programu mikrokontroléru. Tudíž se na celkové ceně systému projeví minimálně. Hlavní nevýhodou SW metod je ta vlastnost, že s nimi nelze odhalit poruchy na všech periferiích mikrokontroléru. Takovým klasickým případem je porucha zdroje hodin mikrokontroléru. SW metody jsou běžně používány pro detekci poruch v pamětích, některých poruch v AD převodníku nebo v programovém čítači.

5.1 REDUNDANCE CPU [8][9]

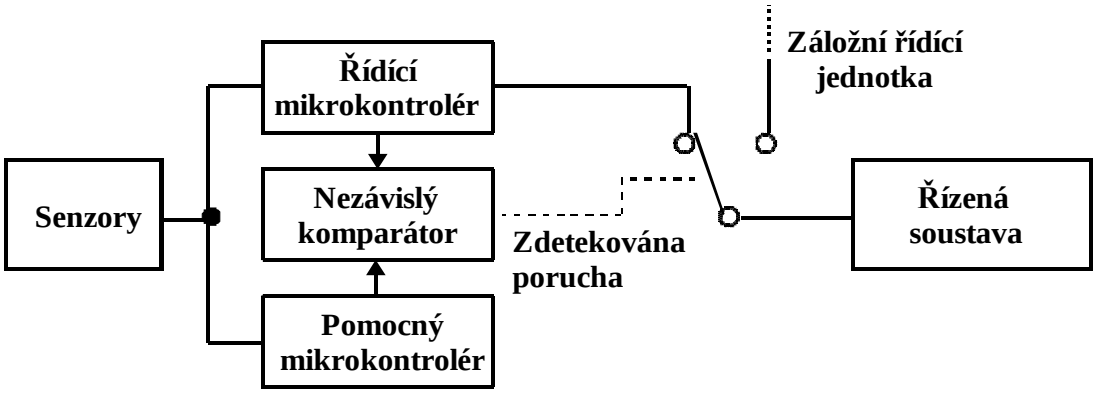
Redundance CPU je nejúčinnější HW metoda pro detekci poruch. Tato metoda vyžaduje minimálně jedno redundantní CPU. To znamená, že systém s touto metodou detekce poruch bude mít více řídících jednotek (více mikrokontrolérů). Jeden mikrokontrolér je aktivní (řídí proces) a ostatní mikrokontroléry jsou pomocné a slouží pouze ke kontrole. Existují dva způsoby založené na tomto principu:

a) Mikrokontroléry se vzájemně kontrolují (obr. 5.1).
b) Mikrokontroléry pro porovnání používají nezávislý komparátor (obr. 5.2).

Základní princip obou způsobů detekce poruch je stejný. Mikrokontroléry porovnávají data nebo signály mezi sebou. Rozdíl je v tom, že v prvním případě dochází k porovnání v mikrokontrolérech, naopak v případě druhém se k porovnání používá nezávislého komparátoru.



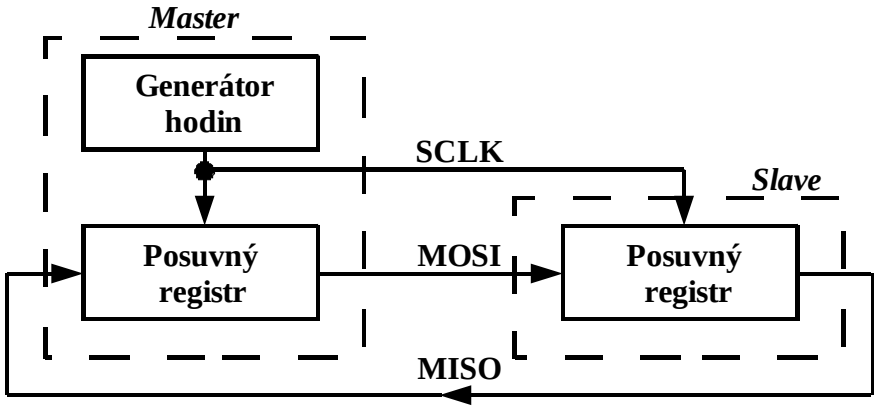
obr. 5.1 – Detekce poruch pomocí redund. CPU se vzájemným porovnáváním



obr. 5.2 – Detekce poruch pomocí redund. CPU s nezávislým komparátorem

V obou případech je velmi důležitá spolehlivá a hlavně rychlá komunikace. K těmto účelům je možné využít sériové komunikační rozhranní SPI. Většina soudobých mikrokontrolérů tuto komunikaci podporuje.

Základní koncepce komunikace SPI je, že v systému může být zapojeno více mikrokontrolérů, kde jeden z nich je *Master* a ostatní jsou *Slave*. Princip komunikace je zobrazen na obr. 5.3. Jedná se o jednoduché posouvání dvou posuvných registrů na základě signálu SCLK. Komunikace probíhá vždy mezi zařízením *Master* a jedním ze zařízení typu *Slave*, které se vybírá pomocí vstupu SS. *Master* generuje hodinový signál SCLK, který je připojen ke všem *Slave* zařízením. Rychlost komunikace závisí na frekvenci hodinového signálu. Ta může být až 2MHz. [10]



obr. 5.3 – Princip rozhraní SPI [10]

Návrh takového systému s redundantními CPU není jednoduchý. Musí se vytvořit protokol pro komunikaci a synchronizaci mikrokontrolérů. Problém může nastat u aplikací, při kterých je kladen důraz na rychlost (například zpracování signálu). Obslužné rutiny pro komunikaci, synchronizaci a kontrolu zatěžují CPU a systém pak nemusí ve stanoveném čase reagovat na podměty (přestává být real-time systém).

Při metodě se vzájemným porovnáváním se zatěžuje mikrokontrolér více než při metodě s nezávislým komparátorem. Musí se zajistit komunikace mezi procesory a vykonává se algoritmus pro detekci poruchy. U metody s nezávislým komparátorem se musí zajistit pouze zaslání dat na vstup komparátoru a jeho výstup pak signalizuje poruchu. Pomocí této metody je možné detekovat poruchy v:

- CPU
- přerušovací jednotce
- pamětích
- I/O jednotce
- AD převodníku
- komunikaci
- čítači/časovači

Jakmile je nalezena porucha, musí aktivní mikrokontrolér buď přejít do bezpečného stavu, nebo přepne na záložní jednotku, která pokračuje v řízení systému.

S touto detekcí poruch je možné se setkat u vysoce spolehlivých systémů, kde je kladen velký důraz na bezpečnost a příliš se nehledí na náklady.

Výhody systémů s redundancí CPU:

- vyšší odolnost proti poruchám
- V případě poruchy aktivního CPU může systém přepnout na záložní CPU a systém pracuje bez přerušení.
- schopnost spuštění diagnostiky k určení důvodu poruchy
- schopnost odhalit téměř všechny poruchy všech periferií

Nevýhody systémů s redundancí CPU:

- složitost systémů
- synchronizace a komunikace
- velké zatížení CPU
- velké pořizovací náklady

5.2 REDUNDANCE PERIFERIE S KOMPARACÍ [8]

Redundance periferie s komparací je obdobná metoda jako metoda s redundanci CPU. Zde se přidává pouze jedna periferie (jeden obvodový prvek), kterou může být například paměť, AD převodník, jednotka pro komunikaci, čítač/časovač a další. Tato varianta je na rozdíl od předchozí metody méně složitá, poněvadž není nutné řešit problém synchronizace dvou řídících jednotek. Nicméně není schopná detekovat chybu v jiné periferii, než kterou má zdvojenou.

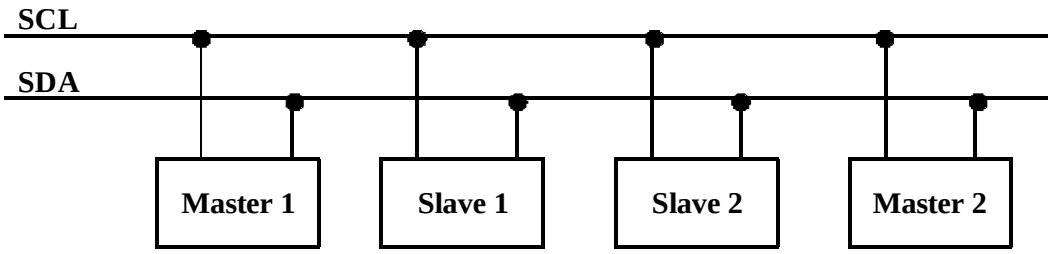
Principielní schéma je zobrazeno na následujícím obr. 5.4. Mikrokontrolér řídí danou periferii a komunikuje s ní přes určité rozhraní. Detekce poruchy provádí mikrokontrolér tím způsobem, že porovná určitá data z interní periferie a z externí

redundantní periferie. Jestliže na základě kontrolního algoritmu mikrokontrolér zjistí poruchu, uvede systém do bezpečného stavu.

obr. 5.4 – Princip metody s redundantní periferií

Stejně jako u předchozí metody je zde důležitá komunikace mezi mikrokontrolérem a periferií. Tato komunikace musí splňovat nároky na rychlost a také spolehlivost. Ve většině případů se návrháři obvodů přiklánějí k SPI nebo I2C rozhraní, která jsou jednak podporována většinou mikrokontrolérů a periferií, ale zároveň splňují uvedené požadavky na komunikaci. Podrobnější popis komunikace SPI je v kapitole 5.1.

I2C je sériové dvouvodičové rozhraní vymyšlené firmou Philips, kde je jeden vodič pro přenos adres i dat (SDA) a druhý je pro hodinový signál (SCL). Koncepce I2C umožňuje připojení více *Master* i *Slave* zařízení. Každé zařízení má svou adresu o délce 7 nebo 10 bitů. Princip I2C je zobrazen na obr. 5.5. Přenosová rychlost I2C může být 100kb/s (Standard mode), 400kb/s (Fast mode) a 3,4Mb/s (High-speed mode) podle módu. [10]



obr. 5.5 – Princip rozhraní I2C [10]

Při této metodě dochází k zatížení CPU, což je způsobeno komunikací a kontrolním algoritmem. Opět však může nastat problém při časově náročných operacích, kdy systém přestane pracovat jako real-time zařízení.

Výhody metody s redundantní periferií:

- méně nákladné a složité oproti metodě s redundantní CPU
- jednoduchá detekce poruch na periferiích
- Pomocí této metody je možné odhalit poruchy, které se prostřednictvím SW obtížně detekuji.

Nevýhody metod redundance periferií:

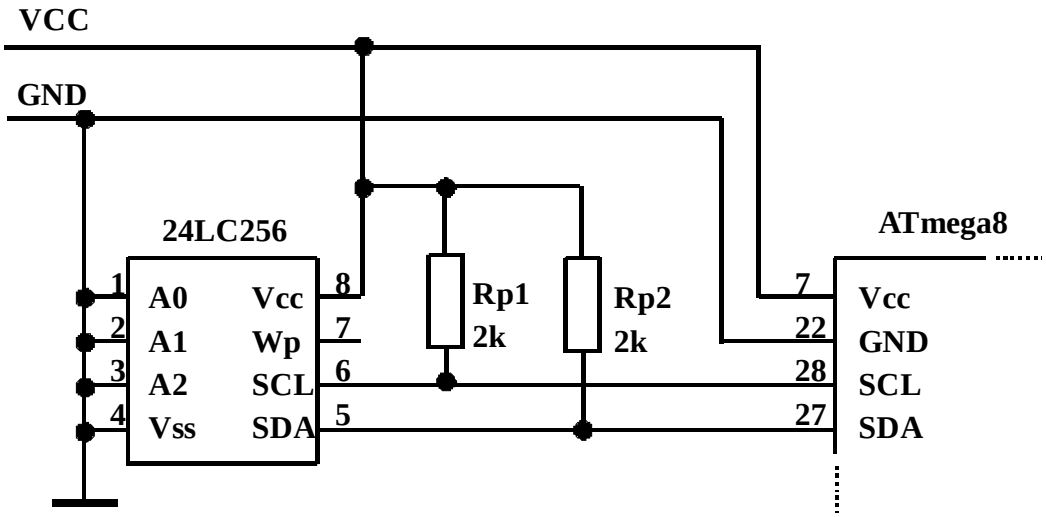
- složitější návrh systému
- vyšší náklady
- zatížení CPU

5.2.1 Příklad redundance EEPROM paměti s komunikací přes I2C

V případě detekce poruch v pamětích, kde jsou uložena data důležitá pro bezpečnost systému, musí systém obsahovat redundantní paměť. Ta je situována mimo mikrokontrolér. Důležitá data jsou pak ukládána do dvou oddělených pamětí obvykle i v různém formátu. Jiným formátem se například myslí to, že uložená data jsou negovaná. Paměť se pak kontroluje pomocí komparace uložených dat.

Na obr. 5.6 je nakresleno zjednodušené zapojení s redundantní pamětí 24LC256 (CMOS EEPROM 256K) přes sběrnici I2C k mikrokontroléru ATmega8 od firmy Atmel. Zapojení je vytvořeno pomocí technické dokumentace k paměti a mikrokontroléru.

Paměť 24LC256 má 3 vstupy A0, A1 a A2, kterými lze nastavit její adresu pro komunikaci přes I2C. Ty se obvykle připojují na určitou logickou úroveň, která pak udává adresu paměti (na obr. 5.6 je to adresa 0). Další dva vývody pro komunikaci přes sběrnici I2C (SDA, SCL) jsou propojeny se stejně označenými vývody mikrokontroléru. K oběma součástkám je připojeno napájení. Na vodiče sběrnice I2C jsou připojeny pull-up rezistory Rp1 a Rp2, jejichž hodnota 2kΩ je odpovídající pro přenosovou rychlost 400kbit/s.



obr. 5.6 – Zapojení ATmega8 s redundantní pamětí 24LC256

Programátor pro komunikaci přes I2C využívá vestavěnou periferii mikrokontroléru ATmega8. Musí nastavit konfigurační registry I2C a komunikovat podle stanoveného protokolu (formát dat i adres). Dále pak naprogramuje kontrolní algoritmus, který porovnává data. Jestliže jsou data uložená v EEPROM paměti mikrokontroléru v normálním formátu a zároveň jsou v redundantní paměti invertována, pak kontrola může probíhat pomocí operace XOR následovně (program v C):

```
EEdata1 = ReadIntEE(adr);    //Čtení dat z paměti mikrokontroléru (interní EEPROM)
EEdata2 = ReadExtEE(adr1);  //Čtení dat z redundantní paměti (externí EEPROM)
test = EEdata1^EEdata2;  // Provedení XOR operace s data1 a data2
if (test != 0xFF)       // Test korektnosti dat-jestliže test je různý od 0xFF, tak v datech je chyba.
{
      num=EEPROM_ERROR; //nastavení druhu nalezené poruchy
      Err(num);  // Funkce Err zajistí, že systém přejde do bezpečného stavu (num=označeni chyby).
}
```

Kontrolní algoritmus pomocí operace XOR zjistí odlišnost dat uložených v proměnných *EEdata1* a *EEdata2*. Jestliže jsou data korektní, pak proměnná *test* bude rovna 0xFFh a program může s daty dále pracovat. Naopak jestli proměnná *test* bude jiné číslo než 0xFFh, pak data jsou nekorektní a nesmí se s nimi nadále pracovat. V tomto případě se podle parametru *num* funkce *Err* provedou takové

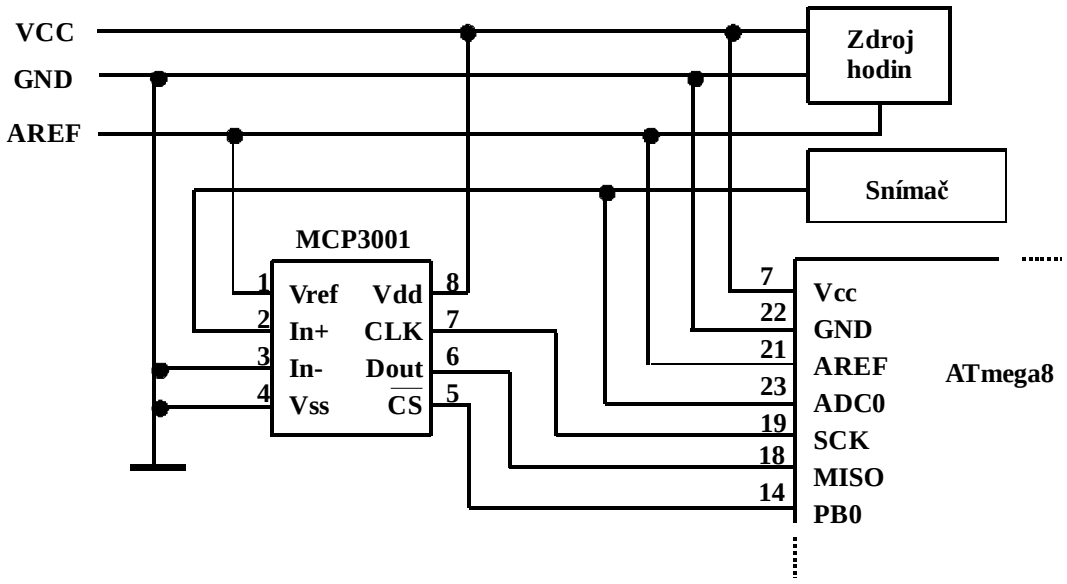
kroky, aby systém přešel do bezpečného stavu. V systému se obvykle detekuje více druhů chyb, proto je nutné pro pozdější diagnostiku tyto chyby odlišit.

5.2.2 Příklad zapojení redundantního AD převodníku s komunikací SPI

V systémech, kde má snímaná veličina velký vliv na bezpečnost systému, se kromě jiných ochranných prvků používá redundantní AD převodník. Pomocí tohoto převodníku je mikrokontrolér schopen kontrolovat správnost převodu integrovaného AD převodníku.

Kontrolu provádí mikrokontrolér pomocí algoritmu, kdy porovnává výsledky z interního (integrovaného) s výsledkem externího (redundantního) AD převodníku. Problém je, že výsledné převedené hodnoty nebývají úplně totožné. Běžně se od sebe liší v nejméně významných bitech (malé rozdíly). Tato odchylka je obvykle způsobena rušením nebo nestejnými referenčními úrovněmi. Pomocí SW a HW filtrů lze do jisté míry odstranit rušení.

Na obr. 5.7 je zjednodušené schéma připojení AD převodníku MCP3001 pomocí SPI sběrnice k mikrokontroléru ATmega8 od firmy Atmel. Zapojení je vytvořeno na základě technické dokumentace mikrokontroléru a AD převodníku.



obr. 5.7 – Zapojení ATmega8 s redund. AD převodníkem MCP3001

Algoritmus pro kontrolu výsledných hodnot z interního a externího 10-bitového AD převodníku může vypadat takto:

```
ADCdata1 = InternalADC();    // vyčtení výsledné hodnoty z AD převodníku mikrokontroléru
ADCdata2 = ExternalADC();   // vyčtení výsledné hodnoty z redundantního AD převodníku
test = ADCdata1^!ADCdata2; //  porovnání hodnot bit po bitu pomocí operace XOR
test = test &= 0x03FC;                    //  maskování dvou LSB a horních 6 (10b)
if (test != 0x0000)
{
     ind++;                    // inkrementace počitadla chybných hodnot, které přišli za sebou
     if (ind >= MAX)           // při překročení hodnoty MAX
     {
          num = ADC_ERROR;          // nastavení příznaku poruchy
          Err(num);                 // Funkce Err zajistí, že systém přejde do bezpečného stavu
     }
}
else
{
      ind = 0;                  // nulování počítadla
      ADCresult=ADCdata1;   // uložení korektních dat – s těmito daty je možné dále pracovat
}
```

Tento algoritmus nalezne nekorektní data i porouchaný AD převodník. Může se stát, že vlivem šumu AD převodníky dodají podstatně odlišná data. Potom nelze rozeznat, jaká data jsou korektní. Obvykle se měření opakuje a data se znovu kontrolují. Jestliže se chyba několikrát po sobě zopakuje, lze usoudit, že jeden z AD převodníku nepracuje správně (porucha, šum nebo rozdílné referenční úrovně).

5.3 WATCHDOG [11][12]

Watchdog (WD) je velmi významný prvek pro bezpečný běh systému. Jedná se o zařízení, které je v dnešní době používáno ve všech embedded systémech řízených mikrokontrolérem.

WD je časovač, který se musí v určitých intervalech nulovat. Jakmile k nulování nedojde v daném čase, pak se automaticky vygeneruje reset. Časový interval WD se nastavuje softwarově nebo externími součástkami.

Pomocí WD je možné detekovat jednak softwarové chyby, kterých se dopustí programátor, tak i HW poruchy. V případě odhalení poruchy WD generuje reset systému. Po resetu nastává problém zjistit periferii, ve které vznikla porucha. Pro její lokalizaci vytvářejí programátoři speciální algoritmy. Ty spočívají v tom, že se pomocí jiné metody nalezne porucha. Metoda spustí funkci, která uloží do paměti číselný kód poruchy, převede systém do bezpečného stavu a pak přestane nulovat WD. Tím je způsoben úmyslný reset systému. Po něm je programátor schopen lokalizovat poruchu. Podle typu poruchy se určí, zda systém setrvá v bezpečném stavu, nebo jestli bude pokračovat v činnosti. Avšak v některých situacích WD vyvolá reset systému neočekávaně (neúmyslný reset). Příklady zdrojů neúmyslných a úmyslných resetu:

- neúmyslné
  - porucha programového čítače
  - porucha přerušovací jednotky
  - porucha zdroje hodin
  - chybné maskování přerušovacího vektoru
  - neošetřena kombinace vstupních dat
  - zacyklení programu
- úmyslné
  - chyba v paměti (neodpovídá kontrolní součet, parita, …)
  - chyba AD převodníku
  - porouchaná externí součástka (tranzistor, relé, …)

Existuje velké množství druhů WD, které se od sebe liší způsobem řízení, funkčními celky, specifickými vlastnostmi, atd. WD se dělí na:

- interní
- externí

5.3.1 Interní watchdog

Interní WD je integrován na čipu mikrokontroléru. V dnešní době je obsažen skoro ve všech dostupných mikrokontrolérech. Buď používá stejný zdroj hodin jako mikrokontrolér, nebo má svůj nezávislý zdroj o určité frekvenci. Časový interval se pak nastavuje pomocí kmitočtové děličky. Vlastní řízení se pak provádí přes konfigurační registry.

Interní WD bývá obvykle chráněn proti neoprávněnému ovládání. Tato ochrana spočívá v použití určité sekvence instrukcí pro jeho spuštění, nulování a vypnutí. Použití posloupnosti instrukcí zaručuje malou pravděpodobnost, že chybně běžící program vynuluje nebo vypne WD. Spuštění WD bývá často povoleno pouze po určenou dobu po resetu. Některé mikrokontroléry pro zvýšení bezpečnosti vůbec neumožňují vypnutí spuštěného WD. Nulování se zpravidla provádí zápisem definovaných hodnot do určeného registru nebo příslušnou instrukcí. Například v mikrokontroléru AT89C55WD se WD nuluje použitím kombinace zápisu hodnot 0x1Eh a 0xE1h do registru WDTRST, ale u mikrokontroléru Atmega8 existuje pro nulování speciální instrukce WDR.

Aby Interní WD pracoval správně, měl by mít následující vlastnosti:

- Reset WD musí trvat dostatečně dlouho, aby se zaručilo, že dojde k opětovnému bezpečnému rozběhu programu (reinicializace programového čítače, reset externích periferií).
- Neměl by být při běhu programu přeprogramovatelný, aby při chybě nedošlo k vypnutí WD.
- Dvojice hodnot, které se zapisují do definovaného registru pro ovládání (nulování, spouštění), by neměly být 0x55h a 0xAAh. Tyto hodnoty se používají při vzorkových testech RAM.
- Měl by mít nezávislí zdroj hodin, aby při poruše zdroje hodin mikrokontroléru došlo k resetu.

U interních WD je po resetu možné zjištění, že reset vyvolal právě WD, což je jejich předností.

5.3.2 Externí watchdog

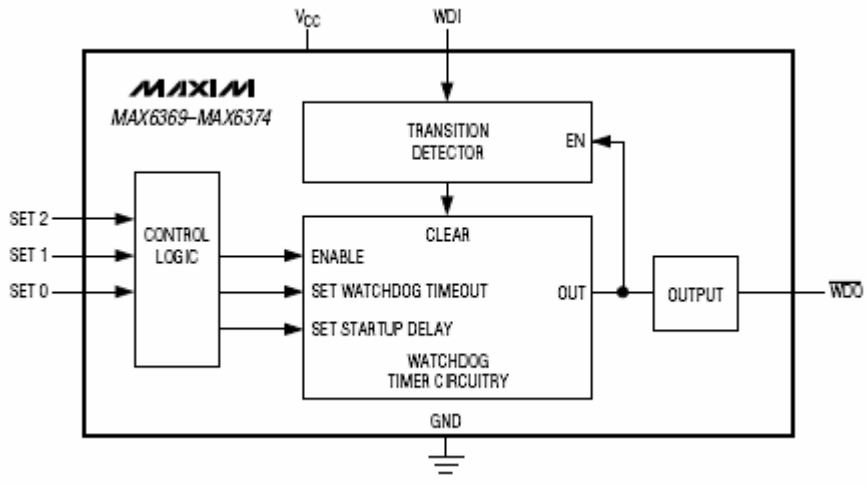
Externí WD je samostatná součástka. Jedná se o obvod, který je zcela nezávislý na mikrokontroléru. Obsahuje autonomní zdroj hodin o určité frekvenci a čítač pulsů hodinového signálu. Jakmile čítač dočítá do jisté hodnoty, je vygenerován na výstupu externího WD pulz o určité délce a úrovni. Ten resetuje mikrokontrolér. Čítač se nuluje pomocí vstupu, na který musí mikrokontrolér v určitých intervalech posílat nulovací signál. Délka čítače určuje časový intervalu, během kterého je možné WD nulovat.

Výrobci nabízejí mnoho variant WD s různými specifickými parametry. Některé typy mají pevně nastavenou délku čítače (časový interval), jiné mají vstupy pro volbu z několika variant proměnné délky. Délka resetovacího pulzu bývá od několika jednotek až po stovky ms.

Výrobci obvykle kombinují WD s dalšími obvody pro bezpečnost. Jsou to například obvody pro sledování napájecího napětí, pamětí, atd.

Nevýhodou externích WD je to, že na desce plošných spojů zabírají místo. Výhodou je velký sortiment těchto časovačů, ve kterém si skoro každý HW návrhář najde nejlepší WD s vyhovujícími vlastnostmi.

Na obr. 5.8 je blokové schéma externího WD od firmy MAXIM s nastavitelnou délkou čítače a zpoždění náběhu. Obrázek je převzat z technické dokumentace součástky.



obr. 5.8 – Blokové schéma WD MAX6369 – MAX6374 od firmy MAXIM

5.4 SOFTWAROVÉ SLEDOVÁNÍ LOGICKÉHO BĚHU PROGRAMU

SW sledování logického běhu programu je velmi často využívaná pomůcka pro detekci poruch, které mohou vzniknout vlivem špatné přerušovací jednotky, AD převodníku nebo externí součástky (např. snímače).

Jedná se o metodu založenou na principu čítače. Programátor nadefinuje proměnnou o určité velikosti (1B, 2B, i více), kterou v jedné části programu inkrementuje a v druhé části dekrementuje nebo nuluje. Jakmile překročí hodnota proměnné definovanou mez, je vyvolána rutina, která uloží identifikační číslo dané poruchy a uvede systém do bezpečného stavu.

5.4.1 Příklad použití metody SW sledování logického běhu programu

Tento příklad je pouze ilustrační pro snazší pochopení této metody. SW sledování logického běhu programu je použito pro kontrolu přerušovací jednotky daného zdroje přerušení. Předpokládá se, že hlavní smyčka programu se opakuje každých 10ms. Zdroj přerušení vyvolá přerušení jednou za 10ms. Z hlediska bezpečnosti je zadán požadavek, že se v případě poruchy přerušovací jednotky pro daný zdroj musí systém dostat do bezpečného stavu po 1s (±0,1s).

Část zdrojového kódu v hlavní smyčce:

```
unsigned char Cnt;  //nadefinování proměnné pro čítač 1B
:
void main (void)
:
If((Cnt++)=>100)     //otestování, zda došlo ke 100 průchodům hlavní smyčky
{
        Err(ERR_INTERRUPT_UNIT);          //funkce, která uvede systém do bezpečného stavu
}
:
```

Část zdrojového kódu k obsluze přerušení:

```
:
Cnt=0;     //nulování čítače
:
```

V hlavním programu je nadefinována globální proměnná *Cnt*, která má funkci čítače. Ta se s každým průchodem hlavní smyčkou inkrementuje. Jakmile je správně zavolána rutina pro obsloužení daného přerušení, proměnná se nuluje. Když nastane

chyba přerušovací jednotky (není volána obslužná rutina), pak je volána funkce, která uvede systém do bezpečného stavu.

5.5 SOFTWAROVÉ METODY DETEKCE PORUCH V PAMĚTÍCH [13]

Nejčastěji používanými metodami detekce poruch v pamětích jsou SW metody. Výhodami oproti HW metodám je menší složitost a nižší náklady. Softwarové metody detekce poruch se dělí podle různých hledisek:

- Podle chování dané metody na:
  - detekční bez lokalizace
  - detekční s lokalizací
- Podle druhů chyb, které odhaluje na:
  - jednobitové
  - vícebitové
- Podle četnosti kontroly paměti na:
  - jednorázové
  - průběžné
- Podle typu metody na
  - existenční testy
  - adresové testy
  - vzorkové (datové testy)

V současné době jsou velmi rozšířeny detekční metody bez lokalizace. Tyto metody zjistí chybu v datech, ale už nejsou schopny přesně určit, kde k ní došlo a v jakém rozsahu. Některé metody neumožňují rozlišit vícebitové chyby – takovou metodou je používání parity. Jednorázové metody jsou takové, které se provedou pouze jednou. Obvykle to bývá po resetu mikrokontroléru. Průběžné metody se provádí v určitých časových intervalech.

Existenční testy jsou obvykle používány při výrobě, když se zjišťuje vlastní funkčnost součástky (paměti mikrokontroléru). Jakmile součástka neprojde

existenčním testem, je další provádění testů zbytečné, protože je součástka vadná a nesmí se použít. V případě, že součástka projde existenčním testem, pak se provádí zbývající testy. Jsou to testy adresové a vzorkové.

Adresové testy se provádí z důvodu ověření, zda v paměti existuje určitý počet adresových míst (velikost paměti), do kterých lze zapisovat data.

Vzorkové testy ověřují, jestli je paměť schopna zachovat data. Myslí se tím, zda je paměť schopna zachovat jakoukoliv binární kombinaci při jakékoliv změně obsahu kterékoliv buňky.

5.5.1 Parita

Parita je jedna z nejjednodušších metod pro odhalování poruch. Rozlišujeme dva druhy parity.

- sudá
- lichá

Sudá a lichá parita určuje hodnotu paritního bitu na základě počtu jedniček v datech. Doplňujeme takovou hodnotu paritního bitu, aby data dohromady s paritním bitem obsahovaly buď lichý počet jedniček (lichá parita), anebo sudý počet jedniček (sudá parita). Například jestliže data obsahují lichý počet jeniček, tak hodnota paritního bitu při použití liché parity bude 0, naopak při použití sudé parity bude 1.

Rozlišujeme dva typy parity.

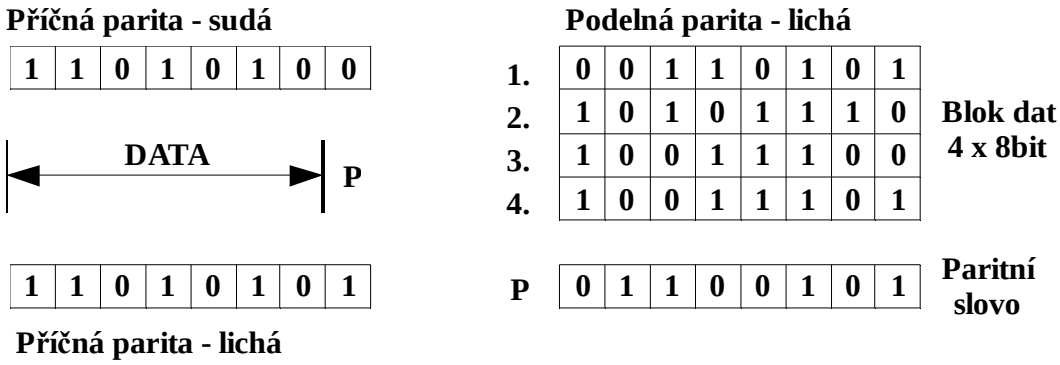
- příčná
- podélná

U příčné parity se jedná o vkládání paritního bitu k jednomu datovému slovu. Jestliže mikrokontrolér má osmibitové slovo, tak 7 bitů z 8 jsou data a zbylý bit je použit jako parita.

Podélná parita se používá pro bloky dat. Určitý počet datových slov se naskládá nad sebe a parita se určuje ze stejnolehlých bitů v datových slovech. Vznikne tak jedno datové slovo s paritními bity jednotlivých sloupců, který se připojí za blok dat.

Pokud při testování správnosti dat neodpovídá počet jedniček s očekávanou paritou, může se předpokládat, že data nejsou korektní. Lze si tedy odvodit, že došlo k chybě jednoho, tří, pěti, … obecně lichého počtu bitů v datovém slovu nebo sloupci bloku dat. Jakmile je získána očekávaná parita, pak není jistota, že byla získána korektní data. Problémová situace nastává v okamžiku chyby dvou, čtyř, šesti, … obecně sudého počtu bitů, kdy je stále dodržen sudý nebo lichý počet jedniček a tudíž touto metodou nemůže být nalezena chyba.

Na obr. 5.9 jsou příklady různých druhů a typů parit (příčná sudá i lichá, podélná lichá).



obr. 5.9 – Příklady příčné a podélné parity

Určitým řešení nespolehlivosti příčné a podélné parity může být využití křížové parity. Křížová parita je aplikování podélné parity na blok dat, kde na jednotlivá datová slova byla použita příčná parita. Tím zvýšíme pravděpodobnost odhalení poruchy, avšak za cenu zvýšení složitosti aplikované metody.

Příčná a podélná parita se v praxi příliš nepoužívá z důvodu nespolehlivosti odhalení dvoubitových chyb. Lze ji použít u systému, kde se očekává malé množství výskytu chyb. Parita je více využívána u přenosu dat. Další nevýhodou příčné parity je snížení velikosti datového slova o jeden bit.

5.5.2 Checksum (Kontrolní součet) [14][15]

Kontrolní součet je další metoda pro ochranu dat v paměti. Touto metodou se zabezpečují celé bloky dat. Z těch se spočítá byt/byty kontrolního součtu a ty se připojí za blok dat, aby se kdykoli mohla zkontrolovat jejich správnost. Na obr. 5.10

je zobrazena reprezentace dat v paměti s kontrolním součtem. Těchto bloků může být v paměti uloženo velké množství.

Blok dat	Kontrolní součet

obr. 5.10 – Blok dat s kontrolním součtem

Data jsou chápana jako binární čísla a kontrolní součet se získá následujícím postupem:

1. Jednotlivá datová slova se sečtou.
2. Na součet se aplikuje operace modulo 2^8 nebo 2^{16} podle toho kolik bytů kontrolního součtu se požaduje.
3. Z výsledného bytu/bytů se vytvoří dvojkový doplněk.
4. Kontrolní součet se připojí za blok dat.

Příklad výpočtu kontrolního součtu:

Blok dat je tvořen 4B s hodnotami: 0x12, 0x2A, 0xC6, 0x8E. Tyto hodnoty sečteme:

$$0x12 + 0x2A + 0xC6 + 0x8E = 0x190$$

Na výsledný součet se aplikuje operace modulo 2^8, tím je získán 1B kontrolního součtu. Modulo je zbytek po celočíselném dělení. V tomto případě to znamená, že výsledek musí být v rozmezí 0x00 – 0xFF.

$$0x190 \bmod 2^8 = 0x90$$

Výsledek po operaci modulo se převede na dvojkový doplněk. To znamená, že se znegují jednotlivé bity a přičte se jedna. Výsledek po této operaci je kontrolní součet pro zadaný 4B blok dat.

$$\text{negace } (0x90) = 0x6F$$

$$\text{kontrolní součet} = 0x6F + 0x01 = 0x70$$

Výpočet kontrolního součtu je velmi jednoduchý. Při každé změně dat v paměti se ovšem musí vždy přepočítat. Porucha se nalezne tak, že se s daty provedou stejné kroky, jako při výpočtu bytu/bytů kontrolního součtu. Takto získaný

výsledek se porovná s uloženým výsledkem za blokem dat. Pokud jsou hodnoty stejné může se předpokládat, že data v bloku jsou správná. V opačném případě jsou data poškozená.

Metoda kontrolního součtu není schopná rozlišit některé typy chyb jako jsou:

- vložení nebo smazaní nulových bytů do bloku dat
- prohození některých bytů
- Vícenásobné chyby, které způsobí vynulování součtu.

Kvůli těmto nedostatkům se tato metoda používá méně a pro softwarovou detekci poruch se více využívá CRC kódu, který tyto nedostatky odstraňuje. Tato metoda je vhodná jednak při ukládání dat, tak i pro jejich přenos.

5.5.3 CRC (Cyclic Redundancy Check) [16][17]

Cyklický redundantní součet se používá pro detekci poruch v uložených datech nebo při jejich přenosu. CRC se využívá k zabezpečení dat. Jedná se o detekční kód, který je schopen odhalit chybu dat v paměti, ale už ji nedokáže opravit.

Principem této metody je připojení k bloku dat jednoho nebo více bytů (obr. 5.10), které jednoznačně charakterizují právě tento blok. Tyto byty jsou získávány aplikováním určitého algoritmu na blok dat a výsledkem algoritmu je zabezpečovací údaj.

Tato metoda je schopna zjistit téměř všechny chyby, které mohou v paměti vzniknout. Podmínkou pro správnou činnost je dostatečně dlouhý zabezpečovací údaj. V praxi je často používaný CRC kontrolní součet o délce 16 bitů. Takto dlouhý údaj zaručuje odhalení chyby v paměti s pravděpodobností na 99,9984%.

Metoda CRC má silné matematické základy. K dokázání vysoké spolehlivosti těchto kódů je zapotřebí pokročilá matematika, a to konkrétně aparát algebry zabývající se okruhy polynomů nad tělesem dimenze 2. To znamená, že se zabývá tělesem polynomů nad celými čísly modulu 2. Jedná se tedy o množinu polynomů, ve kterých jsou koeficienty čísla 0 nebo 1 a algebraické operace jsou odpovídajícím způsobem upraveny. Například binárnímu číslu 110101 odpovídá polynom $x^5+x^4+x^2+1$.

Existuje mnoho druhů CRC, ale všechny typy jsou založeny na stejných matematických úkonech.

- Ke vstupním datům se zleva připojí *n* nulových bitů a takto rozšířená data se stávají dělencem, kde *n* je délka kontrolního součtu v bitech.
- Předurčená *n+1* bitová sekvence nazývaná také generovaný polynom je dělitel.
- *n*-bitový zbytek po dělení je výsledný zabezpečovací údaj nazývaný kontrolní součet.

Z výše uvedeného postupu je zřejmé, že se jedná o velmi zobecněný postup, ale ve výsledku je tvorba CRC pouhé dvojkové dělení, resp. zbytek po dělení. V mikrokontrolérech je implementace dělení poněkud složitější, ale není to neřešitelný problém.

Nejdůležitější částí celého vytváření CRC je zvolit správného dělitele, tedy generovaný polynom. Na něm závisí správná funkčnost, protože při špatném zvolení dělitele může nastat situace, kdy CRC není schopné odhalit některé chyby. Tyto generované polynomy nemusí programátoři vymýšlet nebo odvozovat, protože nejlepší dělitelé jsou již vymyšleni a uvedeni v normách. Programátor musí pouze zvolit správnou délku kontrolního součtu. V tab. 5.1 jsou uvedeny hodnoty dělitele, které určují normy. Hodnota polynomu psána v hexadecimálním tvaru ve třetím sloupku tab. 5.1 je udávána bez nejvyššího bitu – je vždy 1 a zanedbává se.

Typ CRC	**Generovaný polynom**	**Hexadecimální hodnota**
CRC - 1	x+1 (známe jako paritní bit)	0x1
CRC - 4	x^4+x+1	0x3
CRC - 8	$x^8+x^7+x^6+x^4+x^2+1$	0xD5
CRC - 16	$x^{16}+x^{15}+x^2+1$	0x8005
CRC - 32	$x^{32}+x^{26}+x^{23}+x^{22}+x^{16}+x^{12}+x^{11}+x^{10}+x^8+x^7+x^5+x^4+x^2+x+1$	0x04C11DB7

tab. 5.1 – Generované polynomy pro některé druhy CRC vybrané z normy

Pro představu jsou zde uvedeny vlastnosti, podle kterých se vybírá vhodný generovaný polynom (dělitel):

- Nejprve musí být zvolena délka kontrolního součtu s ohledem na schopnost detekce chyb. Generovaný polynom je pak o jeden řád vyšší. Například při velikosti CRC 16 bitů je polynom 17. řádu.
- Nejvýhodnější je použít takový polynom, který je nerozložitelný. Tedy ten polynom, který není dělitelný žádným jiným polynomem kromě sám sebe a 1 beze zbytku.
- Rozložitelný polynom může být použit za cenu snížení schopnosti detekce chyby nebo za určitých podmínek, které snížení schopnosti detekce omezí.

Výše uvedených vlastností generovaného polynomu lze docílit tak, že se využijí následující poznatky:

- CRC s více než jedním nenulovým koeficientem je schopen detekovat jednobitové chyby.
- CRC může detekovat dvoubitové chyby v datech, která jsou menší než 2*k*, kde *k* je délka nejdelší nerozložitelné části generovaného polynomu.
- Generované polynomy jsou schopny naleznout shlukovou chybu, která je kratší (počet chybných bitů) než je řád polynomu.

Kontrolní součet CRC může nabýt 2^n unikátních hodnot, kde *n* je počet bitů kontrolního součtu. Dokud je počet možných uspořádání dat menší než počet unikátních hodnot kontrolního součtu, pak pravděpodobnost, že nastane chyba se určí tímto vztahem $1/2^n$. Naopak pravděpodobnost detekování náhodné chyby se určí vzorcem $1-1/2^n$.

K praktickému využití cyklických kódů není zapotřebí matematický aparát, ale velmi podrobný návod. S takovým návodem se vytváření CRC kódu stává jednoduchou činností.

Příklad výpočtu CRC:

Jsou dána data 11100101, která je zapotřebí zabezpečit pomocí CRC metody. Kontrolní součet je požadován o délce 4bity a generovaný polynom je 11011:

- Nejprve se připojí 4 nulové bity: 111001010000
- Takto získaná data se podělí generovaným polynomem a zbytek je hledaný kontrolní součet. Zde je použita operace modulo 2:

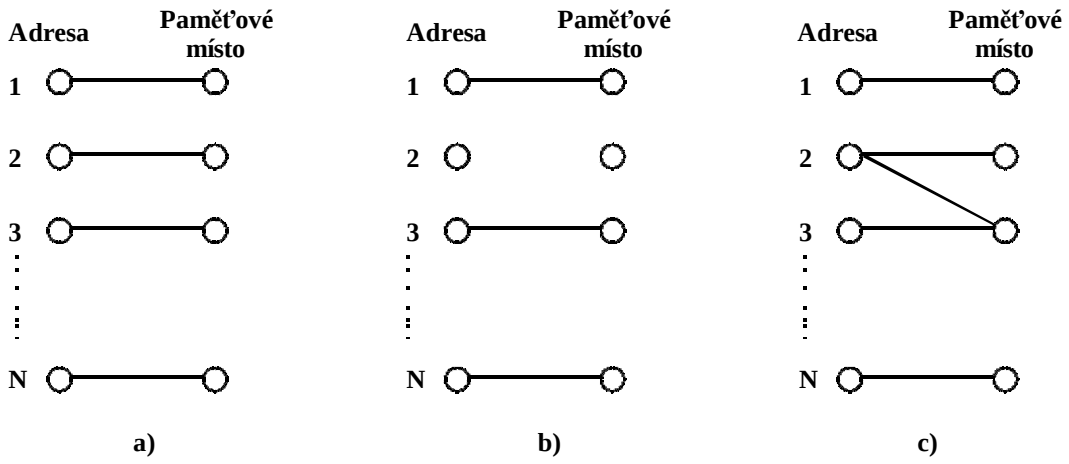
```
1 1 1 0 0 1 0 1 0 0 0 0
1 1 0 1 1
0 0 1 1 1 1
  0 0 0 0 0
    1 1 1 1 0
    1 1 0 1 1
    0 0 1 0 1 1
      0 0 0 0 0
      0 1 0 1 1 0
        1 1 0 1 1
        0 1 1 0 1 0
          1 1 0 1 1
          0 0 0 0 1 0
            0 0 0 0 0
            0 0 0 1 0 0
```

- Zbytek po dělení je 0100, což je výsledný kontrolní součet.

5.5.4 Adresové testy [13]

Adresové testy se provádí z důvodu ověření, zda v paměti existuje určitý počet adresových míst (velikost paměti), do kterých lze zapisovat data. Ověřuje se jimi správnost grafu dekódování adres. Těmito testy se snažíme nalézt dva typy poruch adresování:

- nevybrání adresy
- vybrání dvou adres najednou



obr. 5.11- Grafy dekódování adres [13]

Na obr. 5.11 jsou nakresleny grafy dekódování adres, jednak správného grafu a), ale také grafů, kde jsou nakresleny oba typy poruch b) a c).

Pokud je nalezena porucha typu „nevybrání adresy“ (obr. 5.11b), pak se k určitému paměťovému místu nelze dostat. Obvyklým projevem této poruchy je, že je neustále z paměti vyčítána 0 nebo 1, nezávisle na zapisované hodnotě. Jestliže tedy nastane případ, že se mikrokontrolér pokusí zapsat na toto místo data, dojde k jejich ztrátě. Pokud je nalezena tato porucha opakovaně, je nutné tento mikrokontrolér vyměnit, protože se jedná o trvalou chybu.

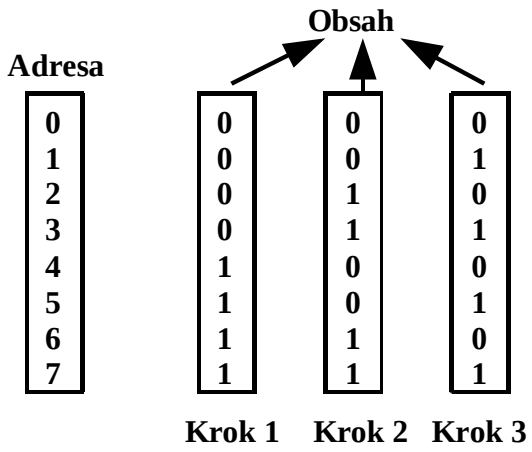
Porucha „nevybrání adresy“ se odhaluje pomocí testovací posloupnosti, která je pro každé paměťové místo následující (je stejný jak test MARCH MATS+, který je popsán níže):

1. zápis nul do paměťového místa
2. Čtení dat z paměťového místa a jejich kontrola, zda odpovídají předchozímu zápisu.
3. zápis jedniček do paměťového místa
4. Čtení dat z paměťového místa a jejich kontrola, zda odpovídají předchozímu zápisu.

V případě odhalení chyby typu „vybrání dvou adres“ (obr. 5.11c), může dojít v případě vyčítání hodnot z paměťových buněk ke kolizi dat mezi 0 a 1. V této

situaci na datové sběrnici paměti dochází k obdobnému chování jako při zkratu. Každá technologie výroby této paměti pak určuje, jakým způsobem se bude daná paměť chovat. Buď bude docházet k logickému součinu kolizních dat, nebo k logickému součtu.

Pro detekci poruch adresování paměti jsou známy také metody, které jsou závislé na datové šířce. Pokud je šířka dostatečná, aby do ní bylo možné uložit celou adresu (2^w=N, *w* je datová šířka paměti a N je počet adresových míst), pak se může využít metoda, pří které je do každého paměťového místa uložena jeho adresu. Poté se pouze kontroluje správná posloupnost adres. V případě, že adresa je delší než datová šířka, je situace o něco složitější. V této situaci se využívá metody, která je na principu binárního vyhledávání. Ta je znázorněna na obr. 5.12, kde je šířka paměti *w* = 1b a počet adres N = 8.



obr. 5.12 - Metoda založena na principu binárního vyhledávání [13]

Vztahy pro výpočet počtu cyklů adresového testu jsou:

a) $2^w = N$: Délka testu = $2 \log_2 N$

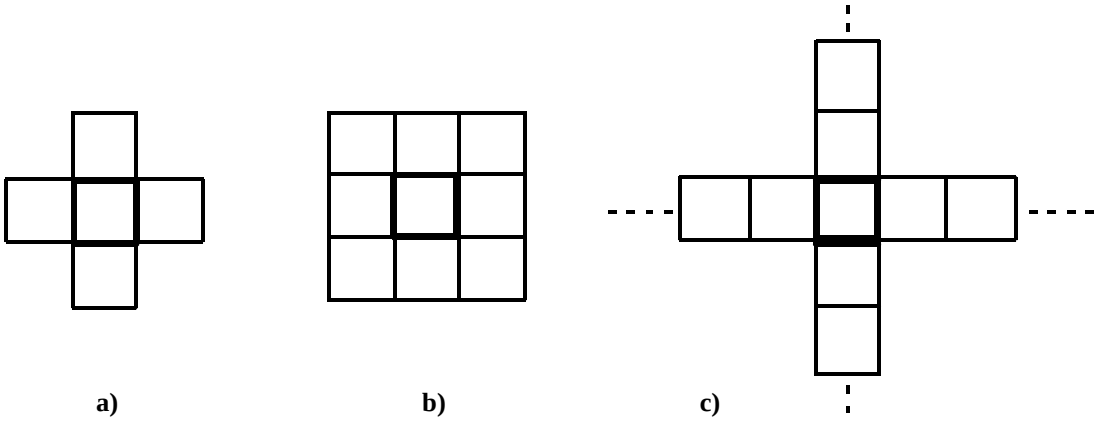
b) $2^w < N$: Délka testu = $2N \log_2 N$

5.5.5 Vzorkové testy (Datové testy) [18], [13]

Vzorkové testy ověřují, jestli je paměť schopna zachovat data. Myslí se tím, zda je paměť schopna zachovat jakoukoliv binární kombinaci při jakékoliv změně obsahu kterékoliv buňky. Z předchozí věty vyplývá, že zde hrají velkou roli sousední

paměťové buňky. Vždy je důležité jaký druh okolí je brán na vědomí. Existuje hned několik druhů sousedství:

- bezprostředně sousední buňky ve směru os X a Y (obr. 5.13a)
- bezprostředně sousední buňky ve směru os X a Y, ale i v obou diagonálních směrech (obr. 5.13b)
- všechny buňky v řádku a sloupci (obr. 5.13c)
- celá paměť



obr. 5.13 - Druhy sousedství paměťových buněk [13]

Délka datového testu je velmi závislá na zvoleném sousedství, které bude spolu s kontrolovanou buňkou testováno. Podle délky testu se pak testy dělí do skupin N, $N^{3/2}$, N^2. N je pro zvolené sousedství bezprostřední, $N^{3/2}$ je pro sousedství se všemi buňkami v řádku a sloupci a konečně N^2 je pro sousedství celé paměti.

Datové testy se skládají ze 3 částí:

- zápis pozadí
- zápis a čtení testované buňky
- čtení sousedních buněk

Zápis pozadí znamená, zapsat do celé paměti jednu logickou hodnotu 0 nebo 1. Vlastní testování pak probíhá podle následujících dvou principů:

- Walk test
- March test

Walk test:

Walk test zjistí pouze přerušené nebo zkratované datové cesty (funkčnost paměťové buňky). Tímto testem nelze zjistit chybu adresace. Jakmile je zapsané definované pozadí paměti, začne se postupně zapisovat do každé paměťové buňky 0 nebo 1. Zpětným vyčtením informace se z datové buňky zkontroluje, zda byla informace zachována. Jestliže došlo ke změně informace z 0 na 1 nebo naopak, pak je paměťová buňka vadná. Po kontrole se do buňky vždy nastaví hodnota předdefinovaného pozadí a pokračuje se na další buňce.

Obdobou walk testu je metoda invertovaných bitů (checkerboard test), která se dnes velmi často používá u mikrokontrolérů. Šířka paměťového místa mikrokontroléru je 8 bitů. Při metodě invertovaných bitu se v prvním kroku nahraje do paměťového místa hexadecimální hodnota 0x55h (binárně 01010101). V druhém kroku se zkontroluje, zda nedošlo ke změně zapsané informace (0x55h). V třetím kroku se zapíše do stejného paměťového místa invertovaná hodnota 0x55h, tedy 0xAAh (binárně 10101010) a v posledním kroku se opět překontroluje, jestli jsou v paměťovém místě uložena zapsaná data (0xAAh).

March testy:

V dnešní době jsou nejznámějšími metodami detekce poruch v pamětích. Je to z důvodů jejich doby trvání testu, ale hlavně z hlediska jejich jednoduchosti. Složitost testů March je lineární. Jedná se o posloupnost March elementů, které jsou složeny z konečné sekvence určitých operací (zápis, čtení) prováděných nad každou buňkou. Existuje několik variant March testů a každý z nich je charakterizovaný určitou posloupností March elementů. Posloupnost a různé druhy těchto elementů pak určují, které poruchy paměti je daná varianta testu schopna nalézt.

March testy lze používat u pamětí, které nemají omezený počet zápisů a jsou dostatečně rychlé. Také je zde důležitá testovaná velikost paměti. Obecně platí čím větší paměť, tím delší test. Předchozím požadavkům odpovídá SRAM paměť mikrokontroléru jednak neomezeným počtem zápisů, ale také menší kapacitou (řadově jednotky kB).

Mezi nejznámější March testy patří MATS+(+), MARCH C- a MARCH B, PMOVI, March SS, March U, March LR, March SR a March G. Existují další, ale v této práci jsou uvedeny pouze výše uvedené.

Pro bližší popis činnosti jednotlivých March testů se zavede následující notace [10]:

r - čtení

w - zápis

r0, r1 - čtení 0, resp. 1 z paměti

w0, w1 - zápis 0, resp. 1 do paměti

↑ - provést zápis 1 do paměti, kde byla 0

↓ - provést zápis 0 do paměti, kde byla 1

↕ - provést inverzi hodnoty v paměti

↑↑ - operace se provádí na buňky se zvyšující se adresou

↓↓ - operace se provádí na buňky s klesající adresou

↕↕ - na směru procházení nezáleží

U každého March testu je uveden počet operací potřebných k otestování 1 paměťového místa. Operací se myslí čtení nebo zápis dat. Celková délka testu se pak určí vynásobením součtu operací počtem paměťových míst N. U každého testu jsou označeny jednotlivé March elementy (Mx). Většina testů má na začátku element ↕↕(w0), což je zápis nulového pozadí do testované paměti.

MATS+:

{↕↕(w0); ↑↑(r0,w1); ↓↓(r1,w0)}

M1 M2 M3

Celková délka testu 5N.

MATS ++:

{↕↕(w0); ↑↑(r0,w1); ↓↓(r1,w0,r0)}

M1 M2 M3

Celková délka testu 5N.

MARCH C-:

{↕↕(w0); ↑↑(r0,w1); ↓↓(r1,w0); ↓↓ (r0,w1); ↓↓(r1,w0), ↕↕(r0)}

M1 M2 M3 M4 M5 M6

Celková délka testu 10N.

MARCH B:

{↕↕(w0); ↑↑(r0,w1,r1,w0,r0,w1); ↑↑(r1,w0,w1); ↓↓(r1,w0,w1,w0); ↓↓(r0,w1,w0)}

M1 M2 M3 M4 M5

Celková délka testu 17N.

PMOVI:

{↓↓ (w0); ↑↑(r0,w1,r1); ↑↑(r1,w0,r0); ↓↓(r0,w1,r1); ↓↓(r1,w0,r0)}

M1 M2 M3 M4 M5

Celková délka testu 13N.

MARCH SS:

{↕↕(w0); ↑↑(r0,r0,w0,r0,w1); ↑↑(r1,r1,w1,r1,w0); ↓↓(r0,r0,w0,r0,w1);

M1 M2 M3 M4

↓↓(r1,r1,w1,r1,w0); ↕↕(r0)}

M5 M6

Celková délka testu 22N.

MARCH U:

{↕↕(w0); ↑↑(r0,w1,r1,w0); ↑↑(r0,w1); ↓↓(r1,w0,r0,w1); ↓↓(r1,w0)}

M1 M2 M3 M4 M5

Celková délka testu 13N.

MARCH LR:

{↕↕(w0); ↓↓(r0,w1); ↑↑(r1,w0,r0,w1); ↑↑(r1,w0); ↑↑(r0,w1,r1,w0); ↑↑(r0)}

M1 M2 M3 M4 M5 M6

Celková délka testu 14N.

MARCH SR:

{↓↓(w0); ↑↑(r0,w1,r1,w0); ↑↑(r0,r0); ↑↑(w1); ↓↓(r1,w0,r0,w1); ↓↓(r1,r1)}

M1 M2 M3 M4 M5 M6

Celková délka testu 14N.

MARCH G:

{↕↕(w0); ↑↑(r0,w1,r1,w0,r0,w1); ↑↑(r1,w0,w1); ↓↓(r1,w0,w1,w0); ↓↓(r0,w1,w0);
M1 M2 M3 M4 M5
↑↑(r0,w1,r1); ↑↑(r1,w0,r0)}
M6 M7

Celková délka testu 23N.

Každý výše uvedený March test je charakterizován svým pokrytím poruch. To určuje počet reálných poruch, které je schopen nalézt. Reálné poruchy jsou popsány v kapitole 4.3.1 v části zabývající se energeticky závislými poruchami.

Aby byla schopnost detekce poruch March testů maximálně využitelná, je nejlepší použít tyto metody na celou paměť najednou. V případě, kdy je požadavek na MARCH test takový, aby běžel na pozadí běžícího programu, je nutné testovat paměť po blokách tak, aby pokaždé otestování bloku posunul v testované paměti o jednu paměťovou buňku vpřed (inkrementování ukazatele).

6. HODNOCENÍ METOD DETEKCÍ PORUCH

Tato kapitola se zabývá hodnocením jednotlivých metod detekce poruch z hlediska jejich spolehlivosti a náročnosti implementace.

Spolehlivost metody se určuje na základě poruch, které je schopna detekovat a podle vlivu této poruchy na bezpečnost systému. Obvykle s rostoucí spolehlivostí roste i náročnost implementace. Návrháři a programátoři musí volit dané metody s ohledem na požadavky, ale také na časovou náročnost dané metody. Některé požadavky udávají normy spojené s daným typem systému a další určuje vedení vývojářské firmy. Normy udávají minimální požadavky, které musí být splněny. V současnosti je cílem vytváření co možná nejspolehlivějších systémů, a to s nejmenšími náklady. Obvykle se musí zvolit určitý kompromis.

Pro hodnocení jednotlivých metod byly použity následující stupnice uvedené v tab. 6.1. Každá stupnice je pro určitou vlastnost.

Spolehlivost	velmi vysoká	Metoda je schopna nalézt poruchy téměř se 100% jistotou.
	vysoká	Metoda je schopna nalézt většinu poruch.
	střední	Metoda je schopna nalézt jen určité typy poruch.
	malá	Metoda je schopna nalézt jen určitou poruchu.
Náročnost implementace	složitá	Pro aplikaci metody je potřeba mít velké množství znalostí.
	střední	Implementace metody je náročnější.
	snadná	Metodu je jednoduché aplikovat.
Časová náročnost	velká	Metoda velmi zatěžuje mikrokontrolér.
	střední	Metoda trochu zatěžuje mikrokontrolér.
	malá	Metoda minimálně zatěžuje mikrokontrolér.

tab. 6.1 - Stupnice pro hodnocení metod

V tab. 6.2 - Hodnocení metod je provedeno hodnocení jednotlivých metod. V ní je uvedena kontrolovaná periferie, spolehlivost, náročnost implementace a časová náročnost. Nejedná se o porovnávání metod mezi sebou, ale o jejich shrnutí. Podrobnější popis (výhody, nevýhody, aplikační náročnost, atd.) metod je možné nalézt v kapitole 5, kde jsou všechny popsány.

Metoda	Periferie	S	N.I.	Č.N.
Redundantní CPU	všechny	velmi vysoká	složitá	velká
Redundantní periferie	všechny paměti	velmi vysoká	střední	střední
	časovač/čítač			
	AD převodník			
	I/O jednotka			
	komunikační jedn.			
Watchdog	programový čítač	vysoká	snadná (interní WD) / střední (externí WD)	malá
	přerušovací jedn.	střední		
	zdroj hodin	střední		
SW sledování logického běhu programu	přerušovací jedn.	střední	snadná	malá
	AD převodník	střední		
Příčná parita	všechny paměti	malá	snadná	střední
Podélná parita	všechny paměti	malá	snadná	střední
Křížová parita	všechny paměti	střední	střední	střední
Checksum	všechny paměti	vysoká	střední	střední
CRC	všechny paměti	velmi vysoká	složitá	velká
Adresové testy	SRAM	malá	snadná	střední
Walk test	SRAM	střední	snadná	střední
MARCH testy	SRAM	podle typu testu střední až vysoká	podle typu testu snadná nebo střední	podle typu testu střední nebo velká
Vysvětlivky: S-spolehlivost, N.I.-náročnost implementace, Č.N.-časová náročnost				

tab. 6.2 - Hodnocení metod

7. APLIKACE VYBRANÝCH MARCH TESTŮ

Pro aplikaci byla zvolena metoda testování SRAM paměti mikrokontrolérů z několika důvodů. Jedním z nich byl velký vliv paměti na bezpečnost systému. SRAM je využívána při výpočtech, a to pro ukládání výsledků a operandů. Potenciální chyba v této paměti může způsobit celkové selhání systému.

March testy jsou dnes nejmodernějšími metodami pro testovaní paměti. Jejich hlavními přednostmi jsou relativně krátká doba testování, vysoké pokrytí poruch vznikajících v paměti a nárůst složitosti testů je lineární.

Hlavní cíle této kapitoly jsou:

- zvolit metody March testů pro další analýzu
- naprogramovat zvolené March testy pro mikrokontrolér
- vytvořit virtuální paměť s modely poruch pro testování funkčnosti testů
- analyzovat testy pomocí virtuální paměti a modelů poruch
- porovnat March testy s jinými často používanými metodami

7.1 VOLBA VHODNÝCH MARCH TESTŮ

Volba vhodného testu je velmi důležitá. March testy se od sebe liší pokrytím různých typů poruch, ale také dobou, za kterou se otestuje jedna buňka. Pokrytí poruch určuje, na jaké závady paměti je daný test citlivý. Jeho citlivost je určena posloupností hodnot a operací, které jsou na danou buňku aplikovány. Dále je také důležitá posloupnost adres (vzrůstající nebo klesající). Délka testu se odvíjí od počtu operací, které test provede.

Z předchozího odstavce vychází zajímavá závislost mezi pokrytím poruch a dobou trvání jednoho testu. Je zřejmé, že se jedná o dvě protichůdné věci. Cílem je samozřejmě dosáhnout co možná nejvyššího pokrytí poruch a zároveň nejkratší doby trvání testu. Musí být tedy zvolen takový test, který efektivně využívá počtu operací tak, aby bylo docíleno co možná nejvyššího pokrytí.

V tab. 7.1 je porovnání vybraných testů podle pokrytí jednobitových (1PF1) a závislých dvoubitových chyb (1PF2). U každého testu je uvedena jeho délka. Jednotlivé chyby jsou popsány v kapitole 4.3.1. U závislých chyb (CFxx) je vždy

uveden dvojnásobný počet možných chyb, než je počet variant vlastní chyby. To je způsobeno tím, že adresa agresivního bitu (a-bitu) může být větší, a nebo menší než adresa chybového bitu (v-bitu). To znamená, že každá varianta má další dvě podvarianty určené adresou agresivního bitu.

Typ poruchy	March testy									
	MATS+	MATS++	PMOVI	C-	G	B	U	LR	SR	SS
SF	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
TF	1/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
WDF	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	0/2	2/2
RDF	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
DRDF	0/2	0/2	2/2	0/2	1/2	0/2	0/2	0/2	2/2	2/2
IRF	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2	2/2
CFst	6/8	6/8	8/8	8/8	8/8	6/8	8/8	8/8	8/8	8/8
CFds xrx	3/8	3/8	8/8	8/8	7/8	7/8	8/8	8/8	8/8	8/8
CFds xwx'	3/8	3/8	7/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8
CFds xwx	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8
CFtr	2/8	4/8	8/8	8/8	7/8	4/8	8/8	8/8	8/8	8/8
CFwd	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	0/8	8/8
CFrd	4/8	4/8	8/8	8/8	8/8	4/8	8/8	8/8	8/8	8/8
CFdrd	0/8	0/8	6/8	0/8	2/8	0/8	0/8	0/8	6/8	8/8
CFir	4/8	4/8	8/8	8/8	8/8	4/8	8/8	8/8	8/8	8/8
Délka	5N	6N	13N	10N	23N	17N	13N	14N	14N	22N
Pokrytí	29/84	32/84	63/84	56/84	57/84	41/84	56/84	56/84	64/84	84/84
Pokrytí [%]	34,5	38,1	75,0	66,7	67,9	48,8	66,7	66,7	76,2	100,0

tab. 7.1 – Chybové pokrytí March testů [4] [19]

Podle tab. 7.1 byly zvoleny pro další analýzu testy MATS+, PMOVI a March SS. MATS+ byl zvolen z důvodu vysoké rychlosti. Tento test může být vhodný u aplikací, kde je kladen důraz na rychlost a malou kapacitu zdrojového kódu. Uvedené klady jsou na úkor malého chybového pokrytí, které je 34,5%. PMOVI patří mezi nepatrně pomalejší testy, ale vzhledem ke svému počtu operací má relativně velké pokrytí poruch 75%. March SS je test, který má 100% pokrytí poruch, což ho řadí k vysoce spolehlivým testům. Avšak doba trvání testu je poměrně dlouhá.

7.2 PROGRAMOVÁNÍ TESTŮ A CHYBOVÝCH MODELŮ

7.2.1 Vývojové prostředky

Pro naprogramování, ladění a následnou analýzu byly použity následující vývojové prostředky:

- AVR Dragon
- AVR Studio
- GCC (GNU Compiler Collection)
- mikrokontrolér ATmega16

AVR Dragon je vývojový nástroj od firmy Atmel, který podporuje programování a ladění programu na mikrokontrolérech rodiny AVR. Obě tyto funkce jsou podporovány přes program AVR Studio. Veškeré programovaní, ladění a testování bylo prováděno pomocí tohoto emulátoru. Komunikace mezi PC a AVR Dragonem byla zprostředkována přes USB linku. Komunikace mezi emulátorem a mikrokontrolérem probíhala přes rozhranní JTAG. Mikrokontrolér ATmega16 byl umístěn v konektoru na univerzální části emulátoru.

AVR Studio je freeware, který nabízí vývojové prostředí pro psaní programu, programování a ladění mikrokontroléru. Program je možné psát v jazyce C pouze za podmínky propojení AVR Studio s programem WinAVR. Ten obsahuje kompilátor GCC a editor pro tvorbu „makefile“. „Makefile“ musí být vytvořen tak, aby byl napsaný zdrojový kód zkompilován do strojového kódu. Dále je nutné nastavit:

- cestu ke zdrojovému souboru obsahující funkci main
- typ mikrokontrolér (např. ATmega16)
- optimalizace, typ výstupního souboru, …

Všechny analýzy byly prováděny na mikrokontroléru ATmega16.

7.2.2 Problematika aplikace testů na mikrokontrolérech [20]

Zásadním problémem při aplikaci March testů na SRAM paměť mikrokontrolérů byl slovně-orientovaný přístup k paměti. To znamená, že není

možné přistupovat k jednotlivým buňkám (bitům), ale pouze k datovým slovům. Mikrokontroléry rodiny AVR mají délku slova 8 bitů.

Řešením tohoto problému je modifikace bitově-orientovaného testu na slovně-orientovaný. Ta byla provedena záměnou jednobitových hodnot za celá datová slova (pozadí) tak, že kde se vyskytla hodnota bitu 0, nahradila se pozadím D_i a kde byla hodnota bitu 1, nahradila se negací D_i (dále označováno D_i'). Tím bylo docíleno všech možných kombinací dat. Pro paměť s délkou slova 8 bitů se používají 4 pozadí D_1=00000000, D_2=01010101, D_3=00110011 a D_4=00001111. Počet různých pozadí určuje počet opakování daného testu na daném slově (1B).

Po provedení modifikace mají testy pro aplikaci na mikrokontroléry s 8-bitovým datovým slovem následující tvar (pro i = 1, 2, 3, 4):

MATS+:

{↕↕(wD_i); ↑↑(rD_i,wD_i'); ↓↓(rD_i',wD_i)}

PMOVI:

{↓↓ (wD_i); ↑↑(rD_i,wD_i',rD_i'); ↑↑(rD_i',wD_i,rDi); ↓↓(rD_i,wD_i',rD_i'); ↓↓(rD_i',wD_i,rD_i)}

March SS:

{↕↕(wD_i); ↑↑(rD_i,rD_i,wD_i,rD_i,wD_i'); ↑↑(rD_i',rD_i',wD_i',rD_i',wD_i);
↓↓(rD_i,rD_i,wD_i,rD_i,wD_i'); ↓↓(rD_i',rD_i',wD_i',rD_i',wD_i); ↕↕(rD_i)}

Testy byly napsány ve zdrojových souborech jako funkce s názvem *IsBlockCorrupted_xxx* (xxx značí název testu MATSplus, PMOVI, MarchSS). I když všechny testy mají stejnou strukturu, liší se použitými March elementy. Ukázka sestavení všech testů je následující.

```
U8BIT IsBlockCorrupted_xxx(void)
{
        U8BIT *addr;
        U8BIT j, i, patt, negpatt;
        addr=(U8BIT *)actual_addr;              //nahrání aktuální adresy
        for(i=0;i<4;i++)                        //4x krát opakovat celý test pro 4 pozadí D1-D4
        {
                patt=__LPM((uint16_t)pattern_array+i);   //načtení vzorku z tabulky ve FLASH
                negpatt=~patt;                  //získaní negovaného vzorku
                wdt_reset();                    //reset watchdogu
                //M1 - nahrání pozadí           //první element
                for(j=0;j<BLOCK;j++)
                { WriteByte(addr+j,patt); }
                //M2                            //druhý element
                :                               :
                //M3                            //třetí element
```

```
                :                                   :
                :                                   :
                //Mn                                //poslední element
        }
        return 0x00;
}
```

Testování se provádí vždy po celých blocích. Takovým blokem mohou být 2B, 4B, 8B, ale i větší. Testování po blocích bylo zvoleno z několika důvodů. Hlavním z nich je maximální využití vlastností March testů. Při testování pouze jednoho bajtu March testy sice odhalí poruchy, ale vlastnost nalézání poruch mezi bajty je nevyužitá. Dalším důvodem je možnost využití testu v online režimu, tedy jeho běhu na pozadí.

Aby bylo docíleno maximální účinnosti hledání poruch, pak se jednotlivé testované bloky musí překrývat. Toho je dosaženo funkcí *InitRamCheck*, která slouží ke kompletnímu otestování paměti po blocích. Ukazatel počátku bloku je po jeho otestování vždy zvětšen o 1.

```
U8BIT InitRamCheck(void)
{
        actual_addr=(U16BIT) &VirtualRAM[V_RAMSTART]; //načtení počáteční adresy
        do      //cyklus pro opakování dokud se neotestuje celá paměť
                if(IsBlockCorrupted_xxx())          //testování jednoho bloku
                        return 0xFF;                //při poruše navrátí hodnotu 0xFF
                else                                //zvětšení aktuální adresy o 1
                        actual_addr++;
        while (actual_addr<=((U16BIT)(&VirtualRAM[V_RAMEND])-BLOCK+1));
        return 0x00; //cely test proběhl bez nalezení poruchy – navrátí  hodnotu 0x00
}
```

Zdrojové kódy všech testů jsou uvedeny v přílohách.

7.2.3 Virtuální paměť s funkčními modely chyb

Pro provedení analýzy musel být vytvořen nástroj, který umožňoval ověření funkčnosti testů. Protože nebyly k dispozici mikrokontroléry s vadnou pamětí, musely být chyby namodelovány.

K tomu byla vytvořena virtuální paměť s funkčními chybovými modely (VirtualSRAM.h). Ta se nadefinovala jako 256B statické pole v SRAM paměti mikrokontroléru ATmega16. K jednotlivým datovým slovům bylo přistupováno pomocí funkcí:

- pro zápis *void WriteByte(U8BIT *addres, U8BIT pattern)*

- pro čtení *U8BIT ReadByte(U8BIT *addres)*

V obou funkcích se vyskytuje datový typ *U8BIT*, což je klasický datový typ jazyka C *unsigned char* v rozsahu 0-255 (1B). Funkce *WriteByte* nemá žádnou návratovou hodnotu. Jejími vstupními parametry jsou ukazatel na 1B (**addres*) a vzorek (*pattern*), který má být na danou adresu uložen. Funkce *ReadByte* vrací uloženou hodnotu z adresy, na kterou ukazuje ukazatel **addres*. Obě funkce ve svém těle obsahují funkční modely chyb, které se pomocí konstant a podmíněného překladu nastavují a spouštějí. Tyto konstanty jsou nadefinovány v horní části zdrojového soubor VirtualSRAM.h. Nastavování konstant, tedy ovládání funkčních chybových modelů, je popsáno níže v kapitole 7.2.5.

Zdrojové kódy funkcí *WriteByte* a *ReadByte* s funkčními chybovými modely jsou uvedeny v přílohách.

7.2.4 Funkční modely chyb [4]

Pro všechny reálné chyby, které mohou vzniknout v SRAM paměti mikrokontroléru se musel vytvořit funkční model. Tyto poruchy jsou uvedeny v kapitole 4.3.1. Každá porucha je charakterizována primitivními chybami a podle jejich počtu má daná chyba množství variant. Primitivní chyby jsou vyjádřeny pomocí závorek <…>. Jejich zápis se pro jednobitové a závislé dvoubitové poruchy od sebe liší.

- Zápis jednobitových poruch má tvar <S/F/R>, kde:

S …vyjadřuje citlivostní operaci nebo hodnotu, která způsobí poruchu daného datového slova. „S“ může být {0, 1, 0w0, 1w1, 0w1, 1w0, 0r0, 1r1}, kde 0 (1) značí nulovou (jedničkovou) hodnotu, 0w0 (1w1) je zápis 0 (1) do buňky, která obsahuje 0 (1), 0w1 (1w0) značí zápis s přechodem z 0 do 1 (z 1 do 0) a 0r0 (1r1) znamená čtení 0 (1).

F …popisuje hodnotu, kterou bude chybová buňka obsahovat po provedení citlivostní operace. „F“ může být {0, 1, ?}, kde 0 (1) značí logickou hodnotu a „?“ znamená nedefinovaný stav.

R …popisuje logickou hodnotu, která bude na výstupu. Tato hodnota se uvádí v případech, kdy je použita citlivostní operace čtení 0r0 (1r1). „R“ může

nabýt hodnot {0, 1, ?, -}, kde 0 (1) značí logickou hodnotu, „?“ nedefinovaný stav a – se používá v případě využití citlivostní operace zápisu.

- Zápis závislých dvoubitový poruch má tvar <S_a;S_v/F/R>, kde:

  S_a…vyjadřuje citlivostní hodnotu nebo operaci, která je aplikována na agresivní buňku (a-bit). Na agresivní buňce je chybová buňka (v-bit) závislá.

  S_v…vyjadřuje citlivostní hodnotu nebo operaci, která je aplikována na chybovou buňku (v-bit).

  S_a a S_v mohou nabýt stejných hodnot/operaci jako S u jednobitových poruch.

  F a R mají stejný význam jako u jednobitových poruch.

Porucha		**Primitivní chyby**	
SF (1) – State Fault		<1/0/-> (0)	<0/1/-> (1)
TF (2) – Transition Fault		<0w1/0/-> (0)	<1w0/1/-> (1)
WDF (3) – Write Destructive Fault		<0w0/1/-> (0)	<1w1/0/-> (1)
RDF (4) – Read Destructive Fault		<0r0/1/1> (0)	<1r1/0/0> (1)
DRDF (5) – Deceptive RDF		<0r0/1/0> (0)	<1r1/0/1> (1)
IRF (6) – Incorrect Read Fault		<0r0/0/1> (0)	<1r1/1/0> (1)
CFst (7) **State Coupling Fault**		<0;0/1/-> (0) <1;0/1/-> (2)	<0;1/0/-> (1) <1;1/0/-> (3)
CFds (8) **Disturb Coupling Fault**	**0w0** **1w1**	<0w0;0/1/-> (0) <1w1;0/1/-> (2)	<0w0;1/0/-> (1) <1w1;1/0/-> (3)
	0w1 **1w0**	<0w1;0/1/-> (4) <1w0;0/1/-> (6)	<0w1;1/0/-> (5) <1w0;1/0/-> (7)
	0r0 **1r1**	<0r0;0/1/-> (8) <1r1;0/1/-> (10)	<0r0;1/0/-> (9) <1r1;1/0/-> (11)
CFtr (9) **Transition CF**		<0;0w1/0/-> (0) <1;0w1/0/-> (2)	<0;1w0/1/-> (1) <1;1w0/1/-> (3)
CFwd (10) **Write destructive CF**		<0;0w0/1/-> (0) <1;0w0/1/-> (2)	<0;1w1/0/-> (1) <1;1w1/0/-> (3)
CFrd (11) **Read destructive CF**		<0;0r0/1/1> (0) <1;0r0/1/1> (2)	<0;1r1/0/0> (1) <1;1r1/0/0> (3)
CFdrd (12) **Deceptive Read Destructive CF**		<0;0r0/1/0> (0) <1;0r0/1/0> (2)	<0;1r1/0/1> (1) <1;1r1/0/1> (3)
Cfir (13) **Incorrect Read CF**		<0;0r0/0/1> (0) <1;0r0/0/1> (2)	<0;1r1/1/0> (1) <1;1r1/1/0> (3)

tab. 7.2 – Primitivní chyby poruch

V tab. 7.2 je přehled primitivních chyb všech reálných jednobitových a závislých dvoubitových poruch. Podle této tabulky byly naprogramovány jednotlivé modely se všemi možnými variantami. Čísla v závorkách u názvů poruch určují hodnotu konstanty ERR_TYPE, při které je daný poruchový model aktivní (hodnota pro podmíněný překlad). Čísla v závorkách u primitivních chyb určují hodnotu konstanty ERR_VAR, při které je daná varianta určité poruchy aktivní.

Každý model byl tvořen jako stavový automat. Níže je uvedena ukázka zdrojového kódu, který simuluje poruchu typu IRF (nekorektní čtení). Uvedený kód je pro tento případ celé tělo funkce *ReadByte*.

```
#if (ERR_TYPE==6)                  //porucha IRF
        if(addres==&VirtualRAM[ERR_ADDR])
        {
                if((ERR_VAR==0) && (!(*addres & (1<<ERR_BIT))))        //<0r0/0/1>
                        return *addres | (1<<ERR_BIT);
                if((ERR_VAR==1) && (*addres & (1<<ERR_BIT)))           //<1r1/1/0>
                        return *addres & (~(1<<ERR_BIT));
                return *addres;
        }
        else
                return *addres;
#endif
```

První podmínka je napsána *#if (ERR_TYPE==6)* pro podmíněný překlad. Další podmínka *if(addres==&VirtualRAM[ERR_ADDR])* zjišťuje, zda se jedná o adresu, kde má být nasimulována porucha. Pokud není splněna, funkce vrací normální (nechybovou) hodnotu *return *addres;*. Avšak pokud je splněna, musí se rozlišit, která varianta poruchy se má provést. To zajišťují následující podmínky, které zároveň testují hodnotu chybového bitu podle primitivní chyby dané poruchy. Když je podmínka splněná, pak je návratová hodnota upravena podle primitivní chyby. Tedy pro ERR_VAR==0 a současné čtení 0 z chybové buňky se návratová hodnota chybového bitu upraví z 0 na 1 *return *addres | (1<<ERR_BIT);*, přičemž v paměti zůstává hodnota nezměněná – přesně podle <0r0/0/1>. To samé platí v případě, kdy ERR_VAR==1 a současné čtení 1 z chybové buňky s tím rozdílem, že návratová hodnota se upraví z 1 na 0 *return *addres & (~(1<<ERR_BIT));* a v paměti opět zůstává hodnota nezměněna podle <1r1/1/0>. Pokud není splněna ani

jedna z podmínek, pak se návratová hodnota nemění, stejně jako při nesplnění první podmínky, kdy se nejedná o adresu chybového bitu.

U závislých dvoubitových chyb se jednotlivé modely pro primitivní chyby dělí pomocí příkazu *switch(ERR_VAR)*, a to kvůli většímu množství variant, kde pouhé použití příkazu *if* je nepraktické.

7.2.5 Nastavování modelů chyb

Jak již bylo uvedeno v kapitole 7.2.3 jsou funkční chybové modely ovládány pomocí konstant. Ty jsou umístěny v horní části zdrojových souboru virtuální paměti (VirtualSRAM.c). Níže je uveden výčet jednotlivých konstant a jejich funkční popis:

- ERR_TYPE
- ERR_VAR
- ERR_ADDR
- ERR_BIT
- ERR_ADDR_OFFS
- ERR_ABIT

Pomocí konstanty ERR_TYPE se nastavuje typ poruchy. Tato konstanta může mít hodnotu od 0 do 13. Pro hodnotu 0 jsou modely poruch neaktivní, tedy virtuální paměť funguje jako normální paměť bez poruchy. Ostatní hodnoty (1-13) označují jednotlivé poruchové modely. Přiřazení jednotlivých číslic je vymezeno v tab. 7.2 ve sloupci s názvy poruch.

Konstanta ERR_VAR definuje variantu poruchy. Každá varianta je definována primitivní chybou dané poruchy. Hodnoty této konstanty pro jednotlivé varianty jsou určeny v tab. 7.2 vždy v závorce za primitivní chybou.

ERR_ADDR je konstanta určující adresu datového slova, ve kterém se vyskytne daná porucha. Je tedy v rozsahu virtuální paměti 0-255 (256B).

ERR_BIT je konstanta, která definuje chybový bit. Může tedy nabýt hodnoty 0 až 7.

Konstanta ERR_ADDR_OFFS definuje offset, který v součtu s konstantou ERR_ADDR určuje adresu slova, ve kterém je agresivní buňka. Konkrétní agresivní buňku (bit) určuje hodnota konstanty ERR_ABIT, která je v rozsahu 0 až 7. Na této

buňce závisí chybový bit. Poslední dvě uvedené konstanty mají účel pouze u závislých dvoubitových poruch.

7.3 ANALÝZA TESTŮ

Cílem analýzy bylo ověřit funkčnost testů a zda odpovídají uváděnému chybovému pokrytí, jakého dosahovaly v tab. 7.1. Protože uváděné výsledky v této tabulce jsou pro bitově-orientovanou paměť, bylo potřeba zjistit, zda je dosaženo alespoň stejných nebo ještě lepších výsledků u modifikovaných testů pro slovně-orientovanou paměť. Dalším účelem bylo zjistit jejich skutečnou dobu trvání a velikost paměti, kterou zabírají v paměti programu.

Poruchy paměti byly simulovány pomocí chybových modelů. Vždy byla snaha maximálně se přiblížit realitě. To bylo zajištěno u závislé chyby například tím, že adresy byly nastavovány tak, že agresivní bity (dále značeny písmenkem *a*) byly sousedy chybové buňky (dále značeny písmenkem *v*). Zároveň také vždy bylo provedeno ověření funkčnosti testu pro vzdálenější vazbu v rámci bloku testovaných dat.

Všechny testy byly prováděny za stejných podmínek:

- stejně velká testovaná data 256B (velikost virtuální paměti)
- 4B velikost bloku testovaných dat
- Stejné adresy poruch stejné (i agresivních bitů)

Zjištěné výsledky jednotlivých testů pomocí analýzy byly použity k porovnání mezi sebou, ale také k porovnání s checker-board testem (nahrávání a následné čtení vzorků 0x55h, 0xAAh – podrobnější popis v kapitole 5.5.5), který je v současnosti velmi často používaný.

7.3.1 Analýza chybového pokrytí

Pro všechny testy bylo nastavení chybových modelů stejné. Pro každou poruchu byl zaznamenán March element testu, který ji odhalil. Zároveň bylo zjištěno pozadí (i = 1, 2, 3, 4), při kterém byla odhalena.

Nastavení poruchových modelů:

a) jednobitové chyby:

ERR_ADDR = 2, ERR_BIT = 3

b) závislé 2-bitové chyby:

závislé poruchy v rámci 1B:

1. ERR_ADDR = 1, ERR_BIT = 5, ERR_ADDR_OFFS = 0, ERR_ABIT = 4 (adresa *a1<v1 – a1* je sousedem *v1*)
2. ERR_ADDR = 1, ERR_BIT = 5, ERR_ADDR_OFFS = 0, ERR_ABIT = 6 (adresa *a2>v1 – a2* je sousedem *v1*)
3. ERR_ADDR = 1, ERR_BIT = 6, ERR_ADDR_OFFS = 0, ERR_ABIT = 1 (adresa *a3<v2 – a3* není sousedem *v2*)

Závislé poruchy v rámci bloku:

4. ERR_ADDR = 1, ERR_BIT = 5, ERR_ADDR_OFFS = -1, ERR_ABIT = 5 (adresa *a4<v1 – a4* je sousedem *v1*)
5. ERR_ADDR = 1, ERR_BIT = 5, ERR_ADDR_OFFS = 1, ERR_ABIT = 5 (adresa *a5>v1 – a5* je sousedem *v1*)
6. ERR_ADDR = 3, ERR_BIT = 2, ERR_ADDR_OFFS = -3, ERR_ABIT = 0 – (adresa *a6<v3 – a6* není sousedem *v3*)
7. ERR_ADDR = 0, ERR_BIT = 0, ERR_ADDR_OFFS = 3, ERR_ABIT = 2 – (adresa *a7>v4 – a7* není sousedem *v4*)

Všechny výsledky analýzy chybového pokrytí byly zapsány do tabulek. Sloupec s názvem „E.V.“ (ERR_VAR) určoval variantu dané poruchy a ve sloupci „V“ jsou zapsaány výsledky dané simulace. Pokud byla chyba nalezena, pak je výsledek „ok“, v opačném případě „fail“. Speciálním výsledkem je ok? (fail?), který určuje, že pro dané nastavení a podmínky byla chyba nalezena (nenalezena), ale zároveň existovalo nastavení a podmínky, pří kterých test poruchu nenalezl (nalezl). To znamená, že test nalezl pouze NĚKTERÉ poruchy tohoto typu, což zaleželo na počátečních podmínkách, anebo na pozici chybového a závislého bitu. Počáteční podmínky byly definovány logickými stavy paměti po resetu mikrokontroléru (nebo

původními uloženými daty). Dále ve sloupci „E“ jsou zapsány elementy, které danou poruchu odhalily (1, 2, 3,…). Ve sloupci „P“ jsou zaznamenána datová pozadí, při nichž byla porucha nalezena.

Analýza 1-bitových závislých poruch

V tab. 7.3 byly zaznamenány výsledky testování funkčnosti modifikovaných March testů na modelech 1-bitových poruch. Testům PMOVI a March SS stačilo k nalezení poruch datové pozadí D1. Modifikovaný test MATS+ dosáhl pomocí druhého pozadí ke zlepšení chybového pokrytí u poruchy TF, kde pro variantu 1 odhalil chybu.

ERR_ADDR = 2, ERR_BIT = 3, E.V. = ERR_VAR										
Porucha	**E.V.**	**MATS+**			**PMOVI**			**MarchSS**		
		V	**E**	**P**	**V**	**E**	**P**	**V**	**E**	**P**
SF (1)	**0**	ok	3	1	ok	2	1	ok	3	1
	1	ok	2	1	ok	2	1	ok	2	1
TF (2)	**0**	ok	3	1	ok	2	1	ok	3	1
	1	ok	2	2	ok	3	1	ok	4	1
WDF (3)	**0**	ok?	2	1	ok?	2	1	ok	2	1
	1	fail?	-	-	fail?	-	-	ok	3	1
RDF (4)	**0**	ok	2	1	ok	2	1	ok	2	1
	1	ok	3	1	ok	2	1	ok	3	1
DRDF (5)	**0**	fail	-	-	ok	4	1	ok	2	1
	1	fail	-	-	ok	3	1	ok	3	1
IRF (6)	**0**	ok	2	1	ok	2	1	ok	2	1
	1	ok	3	1	ok	2	1	ok	3	1

tab. 7.3- Chybové pokrytí 1-bitvých poruch modifikovánými March testy

Analýza 2-bitových závislých poruch v rámci 1B.

Výsledky této analýzy byly zapsány do tab. 7.4. Při analýze v rámci 1B nastal problém u modelů, které obsahovaly operaci zápisu. Jeho podstata byla v tom, že při zápisu vzorku do slovně-orientované paměti mohl být přepisován agresivní i chybový bit současně. Jestliže byla porucha závislá na stavu agresivního bitu, který

byl právě přepisován, pak nebyla jistota, zda ve skutečnosti tím stavem, který určoval chybu, byl stav před zápisem nebo po zápisu vzorku. To je určeno vlastnostmi HW. Naprogramované modely používaly stavy před zápisem. Za podmínky, že byla porucha závislá na zápisu do agresivního bitu a na stavu chybového bitu (CFds), mohl nastat situace, že porucha mohla být maskována vzorkem, což však ve skutečnosti nastat nemusí. Tento problém snižuje důvěryhodnost analýzy v rámci 1B.

První sloupec testu s označením „a1<v1, a2>v1“ obsahuje výsledky pro poruchy mezi přímo sousedícími bity. Pro „a1<v1“ byl model nakonfigurován podle nastavení modelu 1, kde a1-bit je na významově nižší pozici než v1 (ERR_ABIT=4, ERR_BIT=5) a „a2>v1“ byl model nakonfigurován podle nastavení modelu 2, kde a2-bit na významově vyšší pozici (ERR_ABIT=6, ERR_BIT=5). Pro „a3<v2“ byl model nakonfigurován podle nastavení modelu 3 (ERR_BIT=6, ERR_ABIT =1).

Ve všech případech byla pátá, šestá a sedmá varianta poruchy CFds maskována pozadím, a proto nebyly žádným z testů nalezeny. March SS test odhalil všechny poruchy, kromě již zmiňovaných tří variant.

Pro nastavení modelů 1 a 2 byly výsledky stejné, a proto byly zapsány do jednoho sloupce s názvem „a1<v1, a2>v1“. Toto označení bylo použito pro nastavení modelů, kde agresivní a chybový bit byly sousedními bity v jednom datovém slově. Dosažení stejných výsledků vyplývá z charakteristických vlastností slovně-orientovaných testů, protože závislé chyby jsou v rámci slova vyhledávány pomocí změny pozadí (datového vzorku). V tomto případě by měla k nalezení postačit pozadí D1=00000000 a D2=01010101, což tab. 7.4 potvrdila. Jestliže byla porucha nalezena jiným pozadím než D1 a D2, jednalo se o náhodné nalezení v závislosti na pozici chybového a agresivního bitu. V jiné pozici nemusela být vůbec nalezena.

V každém případě tab. 7.4 dokázala, že v rámci 1B záleží na pozici chybového a agresivního bitu. To dokázaly některé různé výsledky u sloupců „a1<v1, a2>v1“ a „a3<v2“ (nastavení modelu 3). Dále také prokázala zlepšení chybového pokrytí testu MATS+ a zhoršení testů PMOVI a March SS.

Porucha	E.V.	MATS+						PMOVI						MarchSS					
		a1<v1, a2>v1			a3<v2			a1<v1. a2>v1			a3<v2			a1<v1, a2>v1			a3<v2		
		V	E	P	V	E	P	V	E	P	V	E	P	V	E	P	V	E	P
CFst (7)	0	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	1	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2	ok	2	2
	2	ok	2	2	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2
	3	ok	3	1	ok	3	1	ok	2	1	ok	2	1	ok	3	1	ok	3	1
CFds (8)	0	ok?	2	1	ok?	2	1	ok?	2	1	ok?	2	1	ok	2	1	ok	2	1
	1	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	3	2	ok	2	2
	2	fail	-	-	ok	2	3	fail	-	-	ok	2	3	ok	2	2	ok	3	2
	3	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	3	1	ok	3	1
	4	ok	2	1	fail	-	-	ok	2	2	fail	-	-	ok	2	2	ok	3	1
	5	fail	-	-	fail	-	-	fail	-	-	fail	-	-	fail	-	-	fail	-	-
	6	fail	-	-	fail	-	-	fail	-	-	fail	-	-	fail	-	-	fail	-	-
	7	fail	-	-	fail	-	-	fail	-	-	fail	-	-	fail	-	-	fail	-	-
	8	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	9	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2	ok	2	2
	10	ok	2	2	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2
	11	ok	3	1	ok	3	1	ok	2	1	ok	2	1	ok	3	1	ok	3	1
CFtr (9)	0	ok	3	1	ok	3	1	ok	2	1	ok	2	1	ok	3	1	ok	3	1
	1	fail	-	-	ok	3	2	ok	3	2	ok	2	2	ok	4	2	ok	3	2
	2	ok	3	2	ok	3	3	ok	2	2	ok	3	2	ok	3	2	ok	4	2
	3	ok	3	3	fail	-	-	ok	3	1	ok	3	1	ok	4	1	ok	4	1
CFwd (10)	0	ok?	2	1	ok?	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	1	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	3	2	ok	2	2
	2	fail	-	-	ok	2	4	fail	-	-	ok	2	3	ok	2	2	ok	3	2
	3	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	3	1	ok	3	1
CFrd (11)	0	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	1	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2	ok	2	2
	2	ok	2	2	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2
	3	ok	3	1	ok	3	1	ok	2	1	ok	2	1	ok	3	1	ok	3	1
CFdrd (12)	0	fail	-	-	fail	-	-	ok	4	1	ok	4	1	ok	2	1	ok	2	1
	1	fail	-	-	fail	-	-	ok	3	2	ok	4	2	ok	3	2	ok	2	2
	2	fail	-	-	fail	-	-	ok	4	2	ok	3	2	ok	2	2	ok	3	2
	3	fail	-	-	fail	-	-	ok	3	1	ok	3	1	ok	3	1	ok	3	1
Cfir (13)	0	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	1	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2	ok	2	2
	2	ok	2	2	ok	3	2	ok	2	2	ok	2	2	ok	2	2	ok	3	2
	3	ok	3	1	ok	3	1	ok	2	1	ok	2	1	ok	3	1	ok	3	1

tab. 7.4 – Chybové pokrytí 2-bitových závislých poruch v rámci 1B (nastavení modelů =1,2,3 – a1 (zleva) a a2 (zprava) jsou sousedy v1, a3 nesousedí s v2)

Analýza 2-bitových závislých poruch v rámci bloku testovaných dat

Výsledky této analýzy byly zapsány do tab. 7.5 (agresivní bity byly shora a zdola sousedy chybového bitu) a tab. 7.6. (agresivní a chybový bit jsou od sebe vzdáleny 3B).

Test March SS v obou případech potvrdil svou spolehlivost pokrytí chyb a všechny chyby odhalil nezávisle na pozici agresivních a chybových bitů a také na jejich adrese v rámci bloku. Naopak testy MATS+ a PMOVI závislost na obou uvedených parametrech potvrdily. Závislost adresy chybového a agresivního bitu prokázaly rozdílné výsledky ve sloupci „V“ pro jednotlivé testy v tab. 7.5, kde pozice bitů byla stejná a pouze se lišily adresy. Závislost na pozici agresivního a chybového bitu ukázaly odpovídající výsledky testů pro „a4<v1“ a „a6<v3“ (kde a-bit měl menší adresu než v-bit a zároveň měly různé pozice) a stejně tak výsledky „a5>v1“ a „a7>v4“ (kde a-bit měl větší adresu než v-bit a zároveň měly různé pozice) z obou tabulek. Důkazem byly opět rozdílné odpovídající výsledky testů MATS+ a PMOVI. Z důvodů závislosti na pozici bitů, byly k výsledkům v tabulkách 7.5 a 7.6 přidány „(?)“, a to pro případ, že se jednotlivé výsledky ve sloupcích a4<v1 a „a6<v3“ nebo „a5>v1“ a „a7>v4“ lišily. Symbol „(?)“ byl použit proto, že test je schopen odhalit pouze NĚKTERÉ poruchy.

V jednotlivých tabulkách modifikované testy PMOVI a MATS+ prokazovaly lepší chybové pokrytí než bitově-orientované testy. To bylo dáno pouze vždy konkrétním nastavením adres a bitů u modelů poruch. V určitých situacích nemusely tyto testy vůbec chybu odhalit.

Porucha	E.V.	MATS+						PMOVI						MarchSS					
		a4<v1			a5>v1			a4<v1			a5>v1			a4<v1			a5>v1		
		V	E	P	V	E	P	V	E	P	V	E	P	V	E	P	V	E	P
CFst (7)	0	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	1	ok	2	3	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	3	1
	2	ok	2	1	ok	3	3	ok	2	1	ok	3	1	ok	2	1	ok	4	1
	3	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	3	1	ok	3	1
CFds (8)	0	ok(?)	2	2	ok(?)	2	1	ok(?)	2	1	ok(?)	2	2	ok	2	1	ok	2	1
	1	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	5	1	ok	3	1
	2	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	6	1	ok	4	1
	3	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	3	1	ok	5	1
	4	ok	2	1	ok	3	3	ok	2	1	ok	4	1	ok	2	1	ok	4	1
	5	fail	-	-	ok	3	1	ok	4	1	ok	2	1	ok	5	1	ok	3	1
	6	fail	-	-	ok	3	3	ok	5	1	ok	3	1	ok	6	1	ok	4	1
	7	ok	2	3	ok	3	1	ok	3	1	ok	5	1	ok	3	1	ok	5	1
	8	ok	2	1	ok	3	3	ok	2	1	ok	4	1	ok	2	1	ok	4	1
	9	fail	-	-	ok	3	1	ok	3	1	ok	3	1	ok	5	1	ok	3	1
	10	fail	-	-	ok	3	3	ok	2	1	ok	4	1	ok	6	1	ok	4	1
	11	ok	2	3	ok	3	1	ok	3	1	ok	3	1	ok	3	1	ok	5	1
CFtr (9)	0	fail	-	-	ok	3	1	ok	4	1	ok	2	1	ok	5	1	ok	3	1
	1	ok	3	3	ok	2	2	ok	3	1	ok	5	1	ok	4	1	ok	6	1
	2	ok	3	1	fail	-	-	ok	2	1	ok	4	1	ok	3	1	ok	5	1
	3	fail	-	-	ok	3	3	ok	5	1	ok	3	1	ok	6	1	ok	4	1
CFwd (10)	0	ok?	2	1	ok?	2	1	ok?	2	1	ok?	2	1	ok	2	1	ok	2	1
	1	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	3	1	ok	5	1
	2	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	2	1	ok	4	1
	3	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	5	1	ok	3	1
CFrd (11)	0	ok	3	3	ok	2	1	ok	3	1	ok	2	1	ok	4	1	ok	2	1
	1	ok	2	3	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	5	1
	2	ok	2	1	ok	3	3	ok	2	1	ok	3	1	ok	2	1	ok	4	1
	3	ok	3	1	ok	2	3	ok	2	1	ok	3	1	ok	5	1	ok	3	1
CFdrd (12)	0	fail	-	-	fail	-	-	ok	4	1	ok	5	3	ok	4	1	ok	2	1
	1	fail	-	-	fail	-	-	ok	5	1	ok	3	1	ok	3	1	ok	5	1
	2	fail	-	-	fail	-	-	ok	5	3	ok	4	1	ok	2	1	ok	4	1
	3	fail	-	-	fail	-	-	ok	3	1	ok	5	1	ok	5	1	ok	3	1
Cfir (13)	0	ok	3	3	ok	2	1	ok	3	1	ok	2	1	ok	4	1	ok	2	1
	1	ok	2	3	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	5	1
	2	ok	2	1	ok	3	3	ok	2	1	ok	3	1	ok	2	1	ok	4	1
	3	ok	3	1	ok	2	3	ok	2	1	ok	3	1	ok	5	1	ok	3	1

tab. 7.5 – Chybové pokrytí 2-bitových závislých poruch v rámci bloku

(nastavení modelů =4, 5 – a4(zdola) a a5(shora) jsou sousedy v1)

Porucha	E.V.	MATS+						PMOVI						MarchSS					
		a6<v3			a7>v4			a6<v3			a7>v4			a6<v3			a7>v4		
		V	E	P	V	E	P	V	E	P	V	E	P	V	E	P	V	E	P
CFst (7)	0	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1	ok	2	1
	1	ok	2	2	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	3	1
	2	ok	2	1	ok	3	2	ok	2	1	ok	3	1	ok	2	1	ok	4	1
	3	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	3	1	ok	3	1
CFds (8)	0	fail	-	-	ok?	2	1	ok?	2	1	fail	-	-	ok	2	1	ok	2	1
	1	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	5	1	ok	3	1
	2	ok(?)	2	4	fail	-	-	ok(?)	2	3	fail	-	-	ok	6	1	ok	4	1
	3	ok(?)	2	3	fail	-	-	ok(?)	2	4	fail	-	-	ok	3	1	ok	5	1
	4	ok	2	1	ok	3	2	ok	2	1	ok	4	1	ok	2	1	ok	4	1
	5	fail	-	-	ok	3	1	ok	4	1	ok	2	1	ok	5	1	ok	3	1
	6	ok(?)	2	3	ok	3	2	ok	5	1	ok	3	1	ok	6	1	ok	4	1
	7	ok	2	2	ok	3	1	ok	3	1	ok	5	1	ok	3	1	ok	5	1
	8	ok	2	1	ok	3	2	ok	2	1	ok	4	1	ok	2	1	ok	4	1
	9	fail	-	-	ok	3	1	ok	3	1	ok	3	1	ok	5	1	ok	3	1
	10	ok(?)	2	3	ok	3	2	ok	2	1	ok	4	1	ok	6	1	ok	4	1
	11	ok	2	2	ok	3	1	ok	3	1	ok	3	1	ok	3	1	ok	5	1
CFtr (9)	0	ok(?)	3	3	ok	3	1	ok	4	1	ok	2	1	ok	5	1	ok	3	1
	1	ok	3	2	ok	3	3	ok	3	1	ok	5	1	ok	4	1	ok	6	1
	2	ok	3	1	ok(?)	2	3	ok	2	1	ok	4	1	ok	3	1	ok	5	1
	3	ok(?)	2	3	ok	3	2	ok	5	1	ok	3	1	ok	6	1	ok	4	1
CFwd (10)	0	ok?	2	1	ok?	2	1	ok?	2	1	ok?	2	1	ok	2	1	ok	2	1
	1	fail	-	-	ok	2	3	fail	-	-	ok(?)	3	3	ok	3	1	ok	5	1
	2	fail	-	-	fail	-	-	fail	-	-	fail	-	-	ok	2	1	ok	4	1
	3	fail	-	-	ok(?)	2	3	fail	-	-	ok(?)	2	4	ok	5	1	ok	3	1
CFrd (11)	0	ok	3	2	ok	2	1	ok	3	1	ok	2	1	ok	4	1	ok	2	1
	1	ok	2	2	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	5	1
	2	ok	2	1	ok	3	2	ok	2	1	ok	3	1	ok	2	1	ok	4	1
	3	ok	3	1	ok	2	2	ok	2	1	ok	3	1	ok	5	1	ok	3	1
CFdrd (12)	0	fail	-	-	fail	-	-	ok	4	1	ok	5	2	ok	4	1	ok	2	1
	1	fail	-	-	fail	-	-	ok	5	1	ok	3	1	ok	3	1	ok	5	1
	2	fail	-	-	fail	-	-	ok	5	2	ok	4	1	ok	2	1	ok	4	1
	3	fail	-	-	fail	-	-	ok	3	1	ok	5	1	ok	5	1	ok	3	1
Cfir (13)	0	ok	3	2	ok	2	1	ok	3	1	ok	2	1	ok	4	1	ok	2	1
	1	ok	2	2	ok	3	1	ok	3	1	ok	2	1	ok	3	1	ok	5	1
	2	ok	2	1	ok	3	2	ok	2	1	ok	3	1	ok	2	1	ok	4	1
	3	ok	3	1	ok	2	2	ok	2	1	ok	3	1	ok	5	1	ok	3	1

tab. 7.6 – Chybové pokrytí 2-bitových závislých poruch v rámci bloku

(nastavení modelů=6, 7 – a6 a v4 i a7 a v5 jsou v krajních slovech bloku)

Shrnutí výsledků chybového pokrytí March testů

Z výše uvedených tabulek 7.3 až 7.6 bylo provedeno shrnutí do přehledových tabulek. Ty byly vytvořeny tak, že pokud test nalezl pouze některé chyby („ok?“), byl ve vyhledávání dané varianty poruchy považován celkově za neúspěšný („fail“). Tímto způsobem byla vytvořena tab. 7.7 z tab. 7.3.

Přehledová tab. 7.8 pro 2-bitové závislé chyby v rámci 1B byla vytvořena tak, že pokud test nalezl danou variantu poruchy v případě sousedícího agresivního a chybového bitu (ve sloupci „a1<v1, a2>v1“ „ok“), ale zároveň jednoznačně nenalezl poruchu pro nesousedící bity (ve sloupci „a3<v2“), byl test ve vyhledávání dané varianty poruchy neúspěšný („fail“).

Přehledová tab. 7.9 pro 2-bitové závislé chyby v rámci bloku dat (a-bit a v-bit nebyly v jednom slově) byla vytvořena následovně. Jestliže varianta poruchy byla jednoznačně nalezena nezávisle na adrese a pozici agresivního a chybného bitu (a4<v1 a a6<v3 nebo a5>v1 a a7>v4 byly „ok“), pak byl March test v jejím odhalení úspěšný.

Výsledná přehledová tab. 7.10 pro 2-bitové závislé chyby byla vytvořena pro celý testovaný blok dat, kde nezáleželo na umístění agresivního a chybového bitu. Jedná se o tabulku, která vznikla kombinací tabulek 7.8 a 7.9.Oba druhy bitů mohly být umístěny kdekoliv v rámci bloku, tedy jak v jednom slově, tak i v jiných datových slovech.

Porucha	SF	TF	WDF	RDF	DRDF	IRF	Pokrytí
MATS+	2/2	2/2	0/2	2/2	0/2	2/2	8/12
PMOVI	2/2	2/2	0/2	2/2	2/2	2/2	10/12
March SS	2/2	2/2	2/2	2/2	2/2	2/2	12/12

tab. 7.7 – Přehled pokrytí 1-bitových poruch paměti

Poruchy	CFst	CFds			CFtr	CFwd	CFrd	CFdrd	Cfir	Pokrytí
		xwx	xwx'	xrx						
MATS+	8/8	0/8	0/8	8/8	4/8	0/8	8/8	0/8	8/8	36/72
PMOVI	8/8	0/8	0/8	8/8	8/8	2/8	8/8	8/8	8/8	50/72
March SS	8/8	8/8	2/8	8/8	8/8	8/8	8/8	8/8	8/8	66/72

tab. 7.8 – Přehled pokrytí 2-bitových poruch paměti v rámci 1B

Poruchy	CFst	CFds			CFtr	CFwd	CFrd	CFdrd	Cfir	Pokrytí
		xwx	xwx'	xrx						
MATS+	8/8	0/8	6/8	6/8	5/8	0/8	8/8	0/8	8/8	41/72
PMOVI	8/8	0/8	8/8	8/8	8/8	0/8	8/8	8/8	8/8	56/72
March SS	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	8/8	72/72

tab. 7.9 – Přehled pokrytí 2-bitových poruch paměti v rámci testovaného bloku dat (a-bit a v-bit nebyly ve stejném datové slově)

Poruchy	CFst	CFds			CFtr	CFwd	CFrd	CFdrd	Cfir	Pokrytí
		xwx	xwx'	xrx						
MATS+	8/8	0/8	0/8	6/8	2/8	0/8	8/8	0/8	8/8	32/72
PMOVI	8/8	0/8	0/8	8/8	8/8	0/8	8/8	8/8	8/8	48/72
March SS	8/8	8/8	2/8	8/8	8/8	8/8	8/8	8/8	8/8	66/72

tab. 7.10 – Přehled celkového pokrytí 2-bitových poruch paměti v rámci bloku

Z tab. 7.7 bylo zjištěno, že chybové pokrytí 1-bitových poruch slovně-orientovaných testů bylo přinejmenším stejné, nebo v případě testu MATS+ ještě lepší, než pro bitově-orientované testy (viz. tab. 7.1). V případě poruchy TF se MATS+ zlepšil natolik, že obě její varianty rozeznal.

Chybové pokrytí 2-bitových závislých poruch testů v rámci 1B bylo uvedeno v tab. 7.8. Z výsledků je patrné, že modifikované testy March SS a PMOVI dosáhly v rámci 1B nižšího pokrytí, než jejich bitově-orientované verze. To bylo způsobeno hlavně nerozpoznáním poruch typu CFds(xwx‘), kde docházelo k maskování poruchy v případech, kdy test chybu vyvolával pomocí citlivostní operace (nahrávání pozadí). Právě zápis vzorku do slova maskoval vyvolanou chybu. Modifikovaný test MATS+ si výrazně polepšil svou bilanci u některých závislých poruch (CFst, CFtr, CFrd, CFir).

Chybové pokrytí 2-bitových závislých poruch testů v rámci testovaného bloku (4B), kde agresivní a chybový bit neležely ve stejném slově, bylo uvedeno v tab. 7.9. Modifikovaný test March SS dosáhl v tomto případě 100% pokrytí. Modifikované testy PMOVI a MATS+ dosáhly lepšího pokrytí, než bitově-orientované testy.

Poslední přehledová tab. 7.10 ukazuje celkové pokrytí 2-bitových závislých poruch v rámci bloku, kde nezáleželo na vzájemném umístění agresivního a

chybového bitu (v rámci 1B, bloku 4B). Jedná se o průnik tab. 7.8tab. 7.9. Modifikované testy March SS a PMOVI měly výsledné krytí nižší než bitově-orientované testy. MATS+ naopak dosáhl lepších výsledků.

Celkové pokrytí chyb slovně-orientovaných testů ukazuje tab. 7.11, kde bylo pro srovnání uvedeno pokrytí bitově-orientovaných testů.

Název testu	**MATS+**		**PMOVI**		**March SS**	
	CH.P.[-]	**CH.P.[%]**	**CH.P.[-]**	**CH.P.[%]**	**CH.P.[-]**	**CH.P.[%]**
Bitově orient.	29/84	34,5	63/84	75,0	84/84	100,0
Slovně orient.	40/84	47,6	58/84	69,0	78/84	92,8

tab. 7.11 – Celkové pokrytí slovně a bitově-orientovaných testů

7.3.2 Analýza doby trvání testů

V případě použití modifikovaného March testu v praxi je důležité znát dobu trvání testování celé paměti (T_C) a bloku dat (T_B). Doba trvání testu byla určena pomocí vygenerovaného impulzu na jednom z výstupu mikrokontroléru. Jeho délka definovala dobu trvání testu a byla změřena pomocí osciloskopu. Impulz pro určení doby testu celé virtuální paměti (256B) se vygeneroval tak, že na začátku funkce *InitRamCheck* se nastavil výstup do logické 1 a na konci této funkce pak do 0. Pro zjištění doby trvání testu jednoho bloku dat (4B) byl před podmínkou, která volá funkci *if(IsBlockCorrupted_xxx)*, nastaven výstup do logické 1 a po návratu z této funkce a nesplnění podmínky byl nastaven do 0 (podmínkového příkazu *else*). Doba testu T_C a T_B byla určena pomocí kurzorů osciloskopu.

Jednotlivé doby trvání jsou zapsány v tab. 7.12. Navíc jsou zde uvedeny doby trvání checkerboard testu, které slouží k pozdějšímu porovnání tohoto testu s March testy.

	MATS+	**PMOVI**	**March SS**	**Checkerboard**
T_C [ms]	16,2	31,8	35,8	0,334
T_B [μs]	62,4	124	140	3,58

tab. 7.12 - Doby trvání otestování celé paměti T_C (256B) a jednoho bloku T_B (4B)

Jelikož je složitost testů lineární, lze zároveň očekávat lineární nárůst doby trvání. Ta by měla být odvozena od počtu prováděných operací nad jednou buňkou.

Při programování testů však musí být použity cykly (pro elementy), které do nich vnáší nelinearitu, ta se přitom promítne do jejich doby trvání. Nelinearita se projeví tím více, čím méně operací je v daném cyklu (elementu). Nelinearita je nejvíce patrná v porovnání doby testů PMOVI a March SS (tab. 7.12). Vykonání třinácti operací PMOVI testu by mělo trvat přibližně polovinu času, než provedení dvaadvaceti operací testu March SS, avšak jejich doby trvání jsou velmi podobné.

7.3.3 Velikost testů v paměti programu

Velikost paměti, kterou testy zabírají ve FLASH paměti programu, byla zjištěna ze souboru s příponou lss, jenž je generován kompilátorem. Tento soubor obsahuje informace o funkcích. Těmito údaji jsou adresy, strojový kód a jemu odpovídající instrukce assembleru. Rozdíl počáteční a koncové adresy funkce, která obsahuje test *IsBlockCorrupted_xxx*, definuje zabrané místo v paměti. Následující tabulka zobrazuje paměťovou náročnost testů. Pro pozdější porovnání je zde i uvedena velikost checkerboard testu.

Druh testu	**MATS+**	**PMOVI**	**March SS**	**checkerboard**
Zabraná paměť [B]	114	180	218	26

tab. 7.13 – Velikost testů v paměti programu

7.3.4 Porovnání March testů s checkerboard testem

Metoda vyhledávání poruch je popsána v kapitole 5.5.5 mezi walk testy. Jedná se o jednoduchý test, který testuje jednotlivá datová slova paměti pomocí uložení a vyčtení vzorku 0x55h a 0xAAh.

Pro porovnání obou druhů testů musela být nejprve provedena analýza pokrytí chyb checkerboard testu. Testování závislých chyb měl smysl pouze v rámci jednoho bajtu, protože tento test nebyl navrhnut pro nalézání poruch v bloku dat. Při analýze byly kladeny stejné požadavky na checkerboard test jako na March testy. To znamená, že test byl úspěšný („ok“) při hledání dané varianty poruchy právě tehdy, když byl nezávislý na počátečních stavech paměti a pozici chybového i agresivního bitu. Jestliže byla daná varianta poruchy nalezena v závislosti na počátečních stavech, pak její výsledek byl „ok?“. Pokud však bylo nalezení poruchy závislé na

pozici v-bitu (1-bitové poruchy), ale i a-bitu (2-bitové poruchy), pak byl výsledek úspěšnosti označen „ok(?)“. Výsledky pokrytí byly uvedeny v tab. 7.14 a tab. 7.15.

1-bitové poruchy pro sudý chybový bit							
	E.V.	**SF (1)**	**TF (2)**	**WDF (3)**	**RDF (4)**	**DRDF (5)**	**IRF (6)**
V	0	ok	ok(?)	ok?	ok	fail	ok
	1	ok	ok?	fail	ok	fail	ok
Přehled		2/2	0/2	0/2	2/2	0/2	2/2
1-bitové poruchy pro lichý chybový bit							
	E.V.	**SF (1)**	**TF (2)**	**WDF (3)**	**RDF (4)**	**DRDF (5)**	**IRF (6)**
V	0	ok	ok?	fail	ok	fail	ok
	1	ok	ok(?)	ok?	ok	fail	ok
Přehled		2/2	0/2	0/2	2/2	0/2	2/2
Pokrytí		**6/12 (50%)**					

tab. 7.14 – Pokrytí 1-bitových chyb testu checkerboard

Z tab. 7.14 je patrné, že checkboard test pokryl přesně polovinu 1-bitových poruch. Při testování 2-bitových poruch u checkerboard testu nezáleží na adrese a-bitu a v-bitu, zda je adresa a-bitu větší nebo menší než v-bitu, ale záleží na jeho pozici v rámci slova, zda je sudá nebo lichá. Podle zadaných kritérií bylo chybové pokrytí 2-bitových závislých poruch nulové. Ve skutečnosti pokrytí těchto poruch nějaké existuje („ok?“, „ok(?)“), avšak je náhodné. Aby nalezení dané varianty nebylo nahodilé, musel by celý řádek výsledků pro danou variantu poruchy v tab. 7.15 být s výsledkem „ok“.

Porucha	E.V.	a-bit a v-bit sousedí		a-bit a v-bit nesousedí		Přehled	Pokrytí
		a-bit sudý v-bit lichý	a-bit lichý v-bit sudý	a-bit sudý v-bit sudý	a-bit lichý v-bit lichý		
CFst (7)	**0**	fail	fail	ok(?)	ok(?)	0/4	**0/36 (0%)**
	1	ok(?)	ok(?)	fail	fail		
	2	ok(?)	ok(?)	fail	fail		
	3	fail	fail	ok(?)	ok(?)		
CFds (8)	**0**	fail	fail	fail	ok?	0/4	
	1	fail	fail	fail	fail		
	2	fail	fail	fail	fail		
	3	fail	fail	ok?	fail		
	4	ok?	fail	fail	fail	0/4	
	5	fail	fail	fail	fail		
	6	fail	fail	fail	fail		
	7	fail	ok?	fail	fail		
	8	fail	fail	ok(?)	ok(?)	0/4	
	9	ok(?)	ok(?)	fail	fail		
	10	ok(?)	ok(?)	fail	fail		
	11	fail	fail	ok(?)	ok(?)		
CFtr (9)	**0**	fail	ok?	ok?	ok(?)	0/4	
	1	fail	ok(?)	fail	fail		
	2	ok(?)	fail	fail	fail		
	3	ok?	fail	ok(?)	ok?		
CFwd (10)	**0**	ok?	fail	fail	ok?	0/4	
	1	fail	fail	fail	fail		
	2	fail	fail	fail	fail		
	3	fail	ok?	ok?	fail		
CFrd (11)	**0**	fail	fail	ok(?)	ok(?)	0/4	
	1	ok(?)	ok(?)	fail	fail		
	2	ok(?)	ok(?)	fail	fail		
	3	fail	fail	ok(?)	ok(?)		
CFdrd (12)	**0**	fail	fail	fail	fail	0/4	
	1	fail	fail	fail	fail		
	2	fail	fail	fail	fail		
	3	fail	fail	fail	fail		
Cfir (13)	**0**	fail	fail	ok(?)	ok(?)	0/4	
	1	ok(?)	ok(?)	fail	fail		
	2	ok(?)	ok(?)	fail	fail		
	3	fail	fail	ok(?)	ok(?)		

tab. 7.15 - Pokrytí 2-bitových závislých chyb v rámci 1B testu checkerboard

Porovnání:

Porovnání March testů bylo provedeno pomocí tab. 7.16, kde jsou uvedeny vlastnosti jednotlivých March testů a checkerboard testu. Je zřejmé, že všechny testy mají svoje kladné, ale i stinné stránky. Výhodou March testů je vyšší pokrytí chyb. Jejich nevýhodou jsou dlouhá doba testování paměti a větší spotřeba paměti programu. Naopak checkerboard test se vyznačuje malou spotřebou paměti programu a rychlým testováním, ale za cenu nižšího pokryti.

	MATS+	**PMOVI**	**March SS**	**checkerboard**
Pokrytí 1-bitových poruch	8/12 (66,7%)	10/12 (83,3%)	12/12 (100%)	6/12 (50%)
Pokrytí 2-bitových poruch v ramci 1B	36/72 (50%)	50/72 (69,4%)	66/72 (91,6%)	0/36 (0%)
Pokrytí všech 2-bitových poruch v rámci bloku	32/72 (44,4%)	48/72 (66,7%)	66/72 (91,6%)	-
Doba trvání testu 1 bloku (4B) T_B[μs]	62,4	124	140	3,58
Doba trvání celého testu (256B) T_C[ms]	16,2	31,8	35,8	0,334
Velikost strojového kódu	114	180	218	26

tab. 7.16 – Porovnání March testů s checkerboard testem

8. ZÁVĚR

S rostoucí složitostí systémů je náročnější vytvářet systémy tak, aby byly bezpečné. Proto jsou vyvíjeny metody, které mají za úkol co nejvíce snížit riziko poruchy a tím zvýšit bezpečnost systému. V této práci jsou popsány dvě metody, které se používají pro návrh bezpečných systémů. Jsou to metody FMEA a FTA. Obě jsou obecné a používají se pro hardware i software.

Práce obsahuje analýzu možných poruch vznikajících na mikrokontroléru. U každé poruchy je obvykle uveden způsob její detekce. V současnosti je vymyšleno mnoho metod pro detekci poruch v mikrokontrolérech. Většina těchto metod je zaměřena na konkrétní periferii. Proto je obvykle využíváno více metod najednou, aby vždy byly odhaleny kritické chyby, které mají velký vliv na bezpečnost. Volba správných metod je obvykle kompromisem mezi požadavky, časovou náročností a náklady. Ve většině případů platí přímá úměrnost mezi spolehlivostí a náklady. Práce obsahuje popis nejčastěji používaných hardwarových a softwarových metod detekce poruch.

Z uvedených metod pro detekci poruch byly vybrány pro aplikaci na mikrokontroléry March testy. Jedná se o skupinu testů, které jsou schopny odhalit jak jednobitové, tak i dvoubitové závislé poruchy, které se mohou vyskytnout v SRAM paměti mikrokontroléru. Ze skupiny March testů byly vybrány testy MATS+, PMOVI a March SS pro následnou analýzu. Tyto testy jsou mezi sebou rozlišeny počtem operací, strukturou elementu a chybovým pokrytím.

Pro aplikaci na mikrokontroléry musely být vybrané testy zmodifikovány z bitově-orientovaných na slovně-orientované. Tato úprava byla nutná z toho důvodu, že SRAM paměť mikrokontroléru je přístupná pouze po slovech (1B). Modifikace se zakládala na záměně logických hodnot za specifická datová slova (pozadí). Pro 8-bitovou šířku slova paměti musí být použity 4 pozadí, pro které je test opakován. Po modifikaci MATS+, PMOVI a March SS byly testy naprogramovány do zdrojových souboru *MATSplus.c*, *PMOVI.c* a *MarchSS.c*.

Protože nebyly k dispozici mikrokontroléry s vadnou SRAM pamětí, musely se jednotlivé poruchy namodelovat. Pro jejich modelaci byla vytvořena virtuální

paměť, do které byl přístup přes funkce zápisu a čtení. Do těla funkcí byly jednotlivé poruchy naprogramovány jako stavové automaty nastavené podle primitivních chyb jednotlivých poruch. Jednotlivé modely byly nastavovány pomocí podmíněného překladu a řídících konstant. Funkce virtuální paměti obsahující modely poruch byly naprogramovány v hlavičkovém souboru *VirtualSRAM.h*.

Na virtuální paměti pak byla provedena analýza chybového pokrytí modifikovaných March testů, a to pro určitá nastavení poruchových modelů. Z naměřených výsledků testů (tab. 7.3 - tab. 7.6) byly vytvořeny přehledové tabulky chybového pokrytí 1-bitových poruch (tab. 7.7) a 2-bitových závislých poruch (tab. 7.10). Také byla vytvořena tab. 7.11 pro porovnaní chybového pokrytí původních bitově-orientovaných a modifikovaných slovně-orientovaných testů. Modifikovaný MATS+ dosáhl lepších výsledků (o 13%) než bitově-orientovaný test. Naopak upravené testy PMOVI a March SS měly menší výsledné pokrytí (PMOVI o -6% a March SS o -7%). V rámci analýzy byly dále změřeny doby otestování bloku dat T_B a celé virtuální paměti T_C a také byla zjištěna velikost potřebného místa v paměti programu.

Pro porovnání modifikovaných March testů s jiným typem vzorkového testu byl zvolen checkerboard test. Pro tento test byla vypracována stejná analýza chybového pokrytí, zjištěna potřebná velikost paměti programu a také byly změřeny časy testování bloku a celé paměti. Z naměřených hodnot pak byla vytvořena tab. 7.16, která dané testy porovnává.

Vytvořené March testy by se mohly dále optimalizovat, aby se docílilo zrychlení testování a zmenšení potřebné paměti programu. Dále by bylo zajímavé provést analýzu možností uplatnění testů v online režimu.

SEZNAM POUŽÍTÉ LITERATURY

[1] STOREY, N.: Safety-critical computer systéme. Addison-Wesley, 1996. ISBN:0-201-42787-7.

[2] JOHNSON, B. W.: Design and analysis of fault-tolerant digital systeme. Addison-Wesley, 1989. ISBN: 0-201-07570-9.

[3] MACHO, T.: Mikroprocesory, elektronická skripta, VUT Brno 2005.

[4] HAMDIOUI, S.: Testing Static Random Access Memory, Defect, Fault Models and Test Patterns. Kluwer Academic Publishers, 2004. ISBN:1-4020-7752-1.

[5] FMEA Software – FMEA Training – FMEA Consulting. Dostupné z URL http://www.harpcosystems.com (duben 2007).

[6] Fault Tree Analysis (FTA), FTA Software, Fault Tree Calculation – Relex Feature Article. Dostupné z URL http://www.relex.com/resources/art/art_fta2.asp

[7] ČSN IEC 1025: leden 1994. Analýza poruchových stavov. Praha: Český normalizační institut, 1993.

[8] UL 1998: 1998. UL Standart for safety for software in programmable components. 2nd edition.

[9] Intel® Telecom produkt: High availability redundant processing architecture FAQs. Dostupné z URL http://www.intel.com/network/csp/products/hafaq.htm (listopad 2007).

[10] DUDÁČEK, K.: Sériová rozhraní SPI, Microwire, I2C a CAN. Dokument dostupný na URL http://home.zcu.cz/~dudacek/NMS/Seriova_rozhrani.pdf (prosinec 2007).

[11] GANSSLE, J.: The firmware handbook. Edited by Jack Ganssle, February 2004, ISBN: 0-7506-7606-X.

[12] FUKSA, M.: Obvody watchdog a obvody hlídající napájení mikroprocesoru. Dostupné z URL http://www.volny.cz/fuksam/povidani/watchdog.htm (leden 2008).

[13] Testování paměti. Dokument dostupný z URL http://www.fit.vutbr.cz/study/courses/DIA/public/testram914.doc (leden 2008).

[14] PETERKA, J.: Zabezpečení dat při přenosech. Dostupné z URL http://www.earchiv.cz/a91/a141c110.php3 (prosinec 2007).

[15] Checksum – Wikipedia, the free encyclopedia. Dostupné z URL http://en.wikipedia.org/wiki/Checksum (prosinec 2007).

[16] PETERKA, J.: CRC. Dostupné z URL www.earchiv.cz/a95/a524k130.php3 (prosinec 2007).

[17] CRC mathematics and teory | Netrino. Dostupné z URL http://www.netrino.com/Embedded-Systems/How-To/CRC-Math-Theory (leden 2008).

[18] Jiří Charvát: Tester paměti RAM ve VHDL. Brno, FIT VUT v Brně 2007

[19] HAMDIOUI, S., GOOR, A.J., RODGERS, M.: March SS:A Static Simple RAM Faults (2002).

[20] JIN-FU, L., TSU-WEI,T., CHIN-LONG, W.: An Efficient Transparent Test Scheme for Embedded Word-Oriented Memories.