Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-13428-64561

Bc. Juraj Šubín

VSTAVANÉ TESTOVANIE A OPRAVA VNORENÝCH PAMÄTÍ RAM

Diplomová práca

Vedúci práce: Ing. Štefan Krištofík

máj 2014

Slovenská technická univerzita v Bratislave
Fakulta informatiky a informačných technológií

FIIT-13428-64561

Bc. Juraj Šubín

VSTAVANÉ TESTOVANIE A OPRAVA VNORENÝCH PAMÄTÍ RAM

Diplomová práca

Študijný program: Počítačové a komunikačné systémy a siete
Študijný odbor: 9.2.4 Počítačové inžinierstvo
Miesto vypracovania: Ústav počítačových systémov a sietí, FIIT STU, Bratislava
Vedúci práce: Ing. Štefan Krištofík

máj 2014

Anotácia

Slovenská technická univerzita v Bratislave
FAKULTA INFORMATIKY A INFORMAČNÝCH TECHNOLÓGIÍ
Študijný program: Počítačové a komunikačné systémy a siete

Autor: Bc. Juraj Šubín
Diplomová práca: Vstavané testovanie a oprava vnorených pamätí RAM
Vedúci diplomovej práce: Ing. Štefan Krištofík
máj, 2014

Diplomová práca sa zaoberá návrhom a implementáciou architektúry pre samočinné testovanie a samočinnú opravu pamätí RAM vnorených v systémoch na čipe. Výsledná architektúra umožňuje samočinné otestovanie a prípadnú opravu pamäte RAM, pričom cieľom pri testovaní pamäte je dosiahnuť čo najvyššie pokrytie porúch a pri oprave pamäte dosiahnuť čo najvyššiu úspešnosť opravy, ak je oprava pamäte možná. Architektúra samočinného testovania je založená na mikrokódovaní, kvôli vysokej flexibilite vo voľbe testovacieho algoritmu. Algoritmus analýzy opravy pozostáva z fázy nevyhnutnej opravy a z finálnej fázy vyčerpávajúceho hľadania. Na ukladanie informácií o detegovaných poruchách sa používa bitmapa. Práca obsahuje analýzu štruktúry pamäte RAM, analýzu modelov porúch, analýzu algoritmov typu march, analýzu blokov vykonávajúcich funkcie samočinného testovania a samočinnej opravy pamätí RAM a analýzu moderných prístupov k riešeniu a implementácii týchto blokov. Ďalej návrh a špecifikáciu požiadaviek jednotlivých častí architektúry, opis implementácie jednotlivých blokov. Práca je ukončená kapitolou k overeniu funkčnosti implementovanej architektúry a vyhodnoteniu plochy potrebnej na čipe pre rôzne kombinácie počtov záložných prvkov a veľkostí hlavnej pamäte.

Annotation

Slovak University of Technology Bratislava
FACULTY OF INFORMATICS AND INFORMATION TECHNOLOGIES
Degree Course: Computer and Communication Systems and Networks

Author: Bc. Juraj Šubín
Master's Thesis: Built-in self-test and self-repair for embedded RAM memories
Supervisor: Ing. Štefan Krištofík
2014, may

This master's thesis deals with the design and implementation of built-in self-test architecture and built-in repair analysis architecture for embedded RAM memories. The final architecture allows self testing and repairing of RAM memory. The built-in self-test and built-in repair analysis architecture designs are focused on obtaining high fault coverage and high repair rate, respectively. The built-in self-test architecture is based on micro-codes technique, because the micro-coded built-in self-test is the most flexible of all self-test structures. Redundancy analysis algorithm consists of a must-repair phase and a final-repair phase, in which an exhaustive search algorithm is performed. The information about detected faults is stored in a bitmap. This project contains the analysis of a structure of RAM memory, analysis of fault models, analysis of march algorithms, analysis of built-in self-test and built-in repair analysis blocks for RAM memories and analysis of modern approaches to desing and implementation of these blocks. It also contains design and specification requirements of architecture elements and the implementation description of parts of the architecture. The project is concluded with a chapter dealing with testing of implemented architecture and the evaluation of the area overhead for various numbers of spares and various memory sizes.

Poďakovanie

Týmto chcem poďakovať Ing. Štefanovi Krištofíkovi za ústretový prístup, odborné rady a pripomienky.

Obsah

1 Úvod 1
2 Analýza 3
2.1 Pamäte RAM 3
2.2 Modely porúch 4
2.3 Algoritmy march 7
2.4 Architektúra samočinného testovania 9
2.5 Architektúra analýzy opravy 10
2.6 Analýza existujúceho riešenia (Moťovský E., 2010) 11
2.7 Analýza moderných prístupov k riešeniu BIRA 13
2.7.1 BIRA s 2D redundanciou a optimálnou úspešnosťou opravy 13
2.7.2 BIRA s 3D redundanciou 18
2.7.3 BIRA s programovateľnými zálohami 23
2.7.4 BIRA s identifikáciou poruchových vzorov 31
2.7.5 BIRA s 2D redundanciou a nízkou cenou 36
2.8 Analýza moderných prístupov k riešeniu BIST 42
2.8.1 BIST s neobmedzeným počtom operácii v elemente march 42
2.8.2 BIST s testom vo forme VHDL skriptu 44
2.8.3 BIST s podporou viacerých typov algoritmov 45
2.8.4 BIST založený na kombinácii FSM a mikrokódovania 48
3 Špecifikácia požiadaviek a návrh riešenia 51
3.1 Celková architektúra 51
3.2 Architektúra BIST 52
3.3 Architektúra BIRA 53
3.4 Prepojenie architektúr BIST a BIRA 57
3.5 Testovaná pamäť a architektúra 58
4 Implementácia riešenia 59
4.1 Architektúra BIST 59
4.1.1 Architektúra BIST všeobecne 59
4.1.2 Pamäť inštrukcií 59
4.1.3 Formát mikroinštrukcie 60
4.1.4 Riadiaca jednotka 61
4.1.5 Komparátor 64

4.2 Architektúra BIRA 65
4.2.1 Architektúra BIRA všeobecne 65
4.2.2 Generátor stratégie 66
4.2.3 Bitmapa (MLB) 67
4.2.4 Register vykonaných opráv 68
4.2.5 Konečný stavový automat 70
4.3 Prepojenie architektúr BIST a BIRA 73
4.4 Testovaná pamäť 74
5 Overenie riešenia a experimentálne výsledky 75
5.1 Overenie riešenia 75
5.1.1 Príklad 1 75
5.1.2 Príklad 2 77
5.1.3 Príklad 3 79
5.1.4 Príklad 4 80
5.2 Experimentálne výsledky 82
5.2.1 Plocha potrebná pre architektúru opravy 82
5.2.2 Porovnanie s existujúcimi riešeniami 87
5.2.3 Výsledky z časového hľadiska 89
6 Zhodnotenie 91
Príloha A: Technická dokumentácia 92
A.1 Testovaná pamäť 92
A.2 Architektúra BIST 94
A.2.1 Funkčný a testovací režim pamäte 94
A.2.2 BIST obvod 95
A.2.3 Komparátor 98
A.3 Architektúra BIRA 99
A.3.1 Základná bunka poľa bitmapa 99
A.3.2 Počítadlá porúch 101
A.3.3 Bunka adresného registra 102
A.4 Prepojenie architektúr BIST a BIRA 104
A.5 Generátor stratégie 105
A.6 Register vykonaných opráv 108
Príloha B: Používateľská príručka 111
B.1 Nahratie testu 111

B.2 Spustenie simulácie 113
Príloha C: Obsah elektronického nosiča 114
Použité skratky 115
Použitá literatúra 116

Zoznam obrázkov

Obr. 2.1: Štruktúra buniek pamätí DRAM a SRAM 4
Obr. 2.2: Základná architektúra BIST jednotky 9
Obr. 2.3: Blokový diagram analyzovanej architektúry BISR 11
Obr. 2.4: Organizácia slovne orientovanej pamäte RAM s 1 záložným riadkom (R) a stĺpcom (C) 13
Obr. 2.5: Blokový diagram BISR obvodu 14
Obr. 2.6: Blokový diagram lokálnej bitmapy o veľkosti 6x6 16
Obr. 2.7: Príklad fungovania analyzovaného algoritmu 18
Obr. 2.8: Organizácia slovne orientovanej pamäte RAM s 3D zálohami 19
Obr. 2.9: Blokový diagram BISR obvodu s 3D redundanciou 20
Obr. 2.10: Blokový diagram BIRA obvodu s 3D redundanciou 21
Obr. 2.11: Schéma pamäte s vyznačenými poruchami 23
Obr. 2.12: Bitmapa s informáciami o detegovaných poruchách. (a), (b) kroky algoritmu 23
Obr. 2.13: Blokový diagram BISR obvodu s programovateľnými zálohami 25
Obr. 2.14: Modelová situácia premapovania adries č. 1 26
Obr. 2.15: Modelová situácia premapovania adries č. 2 27
Obr. 2.16: Modelová situácia premapovania adries č. 3 27
Obr. 2.17: Poruchová mapa pamäte 28
Obr. 2.18: Príklad priebehu algoritmu s identifikáciou vodiacich prvkov 29
Obr. 2.19: Príklad priebehu rozšíreného algoritmu s identifikáciou vodiacich prvkov 29
Obr. 2.20: Príklad priebehu algoritmu pre alokáciu konfigurovateľných záloh 30
Obr. 2.21: Blokový diagram BISR obvodu s identifikáciou poruchových vzorov 32
Obr. 2.22: Formát poruchových syndrómov 32
Obr. 2.23: Stavový diagram BISR jednotky s identifikáciou poruchových vzorov 35
Obr. 2.24: Modelová situácia pri testovaní algoritmom 36
Obr. 2.25: Možnosti organizácie záložných prvkov 37
Obr. 2.26: Štruktúra 1-D lokálnej bitmapy 37
Obr. 2.27: Blokový diagram BISR obvodu s 2D redundanciou a nízkou cenou 38
Obr. 2.28: Príklad priebehu algoritmu (napĺňanie a alokácia) 40
Obr. 2.29: Príklad priebehu algoritmu (po ukončení testovania) 41

Obr. 2.30: Bloková schéma architektúry BIST s neobmedzeným počtom operácií v elemente 42
Obr. 2.31: Formát mikroinštrukcie architektúry BIST s neobmedzeným počtom operácií v elemente 43
Obr. 2.32: Bloková schéma architektúry BIST s testom vo forme VHDL skriptu 44
Obr. 2.33: Bloková schéma architektúry BIST s podporou viacerých typov algoritmov 46
Obr. 2.34: Formát mikroinštrukcie architektúry BIST s podporou viacerých typov algoritmov 47
Obr. 2.35: Bloková schéma architektúry BIST založenej na kombinácii FSM a mikrokódovania 48
Obr. 2.36: Formát mikroinštrukcie architektúry BIST založenej na kombinácii FSM a mikrokódovania 50
Obr. 3.1: Blokový návrh riešenia 51
Obr. 3.2: Blokový návrh architektúry BIST 52
Obr. 3.3: Bloková schéma riadiacej BIST jednotky 53
Obr. 3.4: Blokový návrh architektúry BIRA 54
Obr. 3.5: Bloková schéma bloku bitmapa 56
Obr. 3.6: Blokový návrh bloku Prepojenie architektúr BIST a BIRA 58
Obr. 4.1: BIST obvod 59
Obr. 4.2: Bloková schéma pamäte inštrukcií 60
Obr. 4.3: Formát mikroinštrukcie 60
Obr. 4.4: Riadiaca BIST jednotka 62
Obr. 4.5: Vývojový diagram riadiacej BIST jednotky 63
Obr. 4.6: Bloková schéma obvodu komparátor 64
Obr. 4.7: BIRA obvod 65
Obr. 4.8: Základná schéma obvodu bitmapa 67
Obr. 4.9: Základná schéma obvodu register vykonaných opráv 69
Obr. 4.10: Stavový diagram konečného stavového automatu 72
Obr. 4.11: Základná schéma testovanej pamäte 74
Obr. 5.1: Pr. 1 – rozloženie porúch 76
Obr. 5.2: Pr. 1 – možnosti opravy 76
Obr. 5.3: Pr. 1 – obsah stĺpcovej CAM v RSR 77
Obr. 5.4: Pr. 1 – obsah riadkovej CAM v RSR 77
Obr. 5.5: Pr. 2 – rozloženie porúch 77

Obr. 5.6: Pr. 2 – možnosti opravy....78
Obr. 5.7: Pr. 2 – obsah stĺpcovej CAM v RSR....78
Obr. 5.8: Pr. 2 – obsah riadkovej CAM v RSR....78
Obr. 5.9: Pr. 3 – rozloženie porúch....79
Obr. 5.10: Pr. 3 – možnosti opravy....79
Obr. 5.11: Pr. 3 – obsah stĺpcovej CAM v RSR....80
Obr. 5.12: Pr. 3 – obsah riadkovej CAM v RSR....80
Obr. 5.13: Pr. 4 – rozloženie porúch....80
Obr. 5.14: Pr. 4 – možnosti opravy....81
Obr. 5.15: Pr. 4 – Neúspešné ukončenie opravy pamäte....81
Obr. 5.16: Pr. 4 – Závislosť počtu tranzistorov potrebných na realizáciu architektúry BISR od počtu záložných prvkov....84
Obr. 5.17: Percentuálny podiel plochy pridanej na čip v závislosti od veľkosti hlavnej pamäte (4–64 kb)....86
Obr. 5.18: Percentuálny podiel plochy pridanej na čip v závislosti od veľkosti hlavnej pamäte (256–4096 kb)....86
Obr. A.1: Komponent RAM(synthesizable) knižnice Moduleware....92
Obr. A.2: Princíp zapojenia komponentov RAM....93
Obr. A.3: Zapojenie komponentov RAM....93
Obr. A.4: Zapojenie výstupu pamäte RAM....94
Obr. A.5: Zapojenie BIST obvodu a testovanej pamäte....95
Obr. A.6: Logické rozdelenie BIST obvodu na 3 bloky....95
Obr. A.7: Zapojenie vstupov pamäte inštrukcií....96
Obr. A.8: Zapojenie výstupu pamäte inštrukcií....97
Obr. A.9: Zapojenie generátora adries....98
Obr. A.10: Štruktúra a zapojenie bloku komparátor....99
Obr. A.11: Zapojenie jednej bunky poľa....100
Obr. A.12: Štruktúra sčítačky....102
Obr. A.13: Princíp zapojenia bunky adresných registrov....104
Obr. A.14: Zapojenie počítadiel a adresných dekóderov....105
Obr. A.15: 1. blok obvodu generátora stratégie....106
Obr. A.16: 2. blok obvodu generátora stratégie....107
Obr. A.17: Princíp zapojenia CAM pamätí v registri vykonaných opráv....108
Obr. A.18: Zapojenie log. členov pre generovanie signálov selektor....110

Obr. A.19: Princíp prešírenia hodnoty zo záložnej pamäte 110

Zoznam tabuliek

Tab. 2.1: Vybrané typy march testov 7
Tab. 4.1: Správanie kombinačného obvodu 66
Tab. 4.2: Obsah pamäte 66
Tab. 4.3: Konečný stavový automat v bloku prepojenie architektúr BIST a BIRA 73
Tab. 5.1: Počty tranzistorov potrebných na realizáciu architektúry samočinnej opravy 82
Tab. 5.2: Počty tranzistorov potrebných na realizáciu logických členov a prekl. obvodov 83
Tab. 5.3: Počty tranzistorov potrebných na realizáciu hlavnej pamäte 85
Tab. 5.4: Percentuálne podiely plochy architektúry BISR k ploche hlavnej pamäte 85
Tab. 5.5: Porovnanie výsledkov s riešením analyzovaným v kapitole 2.6, 5 záložných riadkov aj stĺpcov 87
Tab. 5.6: Porovnanie základných vlastností s riešením analyzovaným v kapitole 2.6 87
Tab. 5.7: Porovnanie výsledkov s riešením analyzovaným v kapitole 2.7.4, 4 záložné riadky aj stĺpce 88
Tab. 5.8: Porovnanie základných vlastností s riešením analyzovaným v kapitole 2.7.4 88
Tab. 5.9: Porovnanie výsledkov s riešením analyzovaným v kapitole 2.7.5, 2 záložné riadky aj stĺpce 89
Tab. 5.10: Porovnanie základných vlastností s riešením analyzovaným v kapitole 2.7.5 89
Tab. A.1: Pravdivostná tabuľka sčítačky 102

1 Úvod

V novodobých systémoch na čipe zaberá pamäť viac plochy čipu ako riadiaca alebo kontrolná logika. Priemerne je to cca 86 % celkovej plochy čipu. Hlavným dôvodom sú náročné pamäťové požiadavky aplikácií. Dôsledkom tohto faktu je, že celková výťažnosť systémov na čipe závisí vo veľkej miere od výťažnosti pamätí. Navyše, zväčšovaním kapacity pamätí klesá ich výťažnosť, a tak klesá celková výťažnosť systémov na čipe. Aby sa zabránilo tomuto trendu, bolo potrebné podniknúť určité opatrenia. Výsledkom týchto opatrení je pridanie bloku pre opravu pamäte na čip. Oprava pamäte na rozdiel od samotnej detekcie porúch zvyšuje výťažnosť pamätí, preto sú bloky pre detekciu porúch a opravu pamätí často vyžadované, až nutné. [1]

Ako už bolo spomenuté, pridanie bloku opravy pamäte má pozitívny vplyv na výťažnosť pamätí, a tým aj na výťažnosť systémov na čipe. Ak by blok opravy pamätí nebol do pamätí pridávaný, každý vyrobený systém na čipe s poruchovou pamäťou by bol nepoužiteľný. Existuje viacero druhov porúch pamätí, porucha môže nastať v jednom bite, riadku, stĺpci alebo v adresnom dekóderi. Pamäť po analýze výsledkov testu môže byť označená ako bezporuchová, poruchová opraviteľná alebo poruchová neopraviteľná. To záleží od konkrétnej pamäte a konkrétnej realizácie jej opravného bloku. Pod pomenovaním bloku „oprava pamäte" sa na prvý pohľad rozumie reálna oprava poruchového bitu/riadku/stĺpca. V skutočnosti, ak je niektorá časť pamäte poruchová, nedá sa opraviť do funkčného stavu. Oprava pamäte v tomto ponímaní znamená nahradenie poruchovej časti pamäte bezporuchovou. Toto je realizované preadresovaním poruchových blokov pamäte na bezporuchové (pamäť obsahuje určitý počet pamäťových blokov navyše, slúžiacich ako zálohy). Pre opravu vnorených pamätí sa najčastejšie používajú metódy BISR (Built-in Self-Repair, vstavaná samočinná oprava), ktoré sú založené na algoritmoch BIRA (Built-in Repair Analysis, vstavaná analýza opravy). Rôzne implementácie týchto metód sa líšia hlavne použitím rôznych typov záložných prvkov (napr. 1D zálohy – len riadky alebo len stĺpce, 2D zálohy – aj riadky, aj stĺpce, blokové zálohy a pod.). Z použitého typu záloh a spôsobu ich využitia na opravu pamäte potom vyplýva úspešnosť týchto implementácií. [1]

Testovanie je potrebné pre odhalenie porúch na vyrobenom čipe, či už pamäťovom alebo hociakom inom. Kvalitné testy pamätí musia garantovať veľké pokrytie porúch. Kvalita

testov, či už z hľadiska pokrytia porúch alebo dĺžky testu, veľmi závisí od použitého modelu porúch. Samotný test sa musí vyvíjať a zlepšovať svoj algoritmus pre odhaľovanie porúch, aby v prípade objavenia nového druhu poruchy bol schopný ju v čipe detegovať. U vnorených pamätí sa najčastejšie pre generovanie testovacích vzoriek používa metóda BIST (Built-in Self-Test, vstavané samočinné testovanie). Pre túto metódu je charakteristická nízka ovládateľnosť vstupných testovacích vzoriek, ale má ľahko pozorovateľné výstupy. Vstavané samočinné testovanie pre testovanie pamätí RAM najčastejšie využíva testy typu march, tieto testy sú však vhodné aj pre použitie na externé testovanie pamäte. Dôvodom obľúbenosti a častého využitia testov march je ich lineárna zložitosť, pravidelnosť, symetrickosť a dobré pokrytie porúch. [1], [3]

Cieľom diplomovej práce je oboznámiť sa s fungovaním blokov vykonávajúcich funkcie samočinného testovania a samočinnej opravy pamätí RAM vnorených v systémoch na čipe a na základe vykonanej analýzy vybrať jeden z moderných prístupov pre implementáciu každého bloku, navrhnúť celkovú architektúru digitálneho systému realizujúceho samočinné testovanie a opravu pamätí RAM, navrhnúť prepojenie jednotlivých funkčných blokov systému, a nakoniec tento systém implementovať a otestovať. Výstupom práce bude implementácia takéhoto systému na úrovni prepojení logických blokov použitím dostupného softvérového návrhového prostriedku.

V analýze problému sa budem zaoberať opisom typov porúch detegovaných v pamätiach RAM, opisom algoritmov pre detekciu porúch a opisom moderných prístupov k riešeniu blokov vykonávajúcich funkcie samočinného testovania a samočinnej opravy pamätí RAM.

V časti Špecifikácia požiadaviek a návrh riešenia uvediem návrh riešenia jednotlivých častí architektúry aj s požiadavkami na ich funkcionalitu, rozhranie a pod.

V časti Implementácia riešenia opíšem spôsob implementácie jednotlivých častí architektúry.

V časti Overenie riešenia a experimentálne výsledky uvediem niekoľko reprezentatívnych príkladov, ktorými demonštrujem funkčnosť implementovanej architektúry. Experimentálne výsledky porovnám s výsledkami existujúcich riešení.

Dokument je ukončený zhodnotením práce a dosiahnutých výsledkov.

2 Analýza

2.1 Pamäte RAM

Pamäť je vo všeobecnosti prostriedok na uchovávanie informácií, v počítačovom svete vo forme jednotiek a núl. Väčšina pamätí dokáže uložiť vstupné údaje na miesta, ktoré sú jednoznačne dané adresou (adresným vstupom). Použitím tej istej adresy sa následne dajú uložené údaje z pamäte prečítať. Pamäť je tvorená poľom pamäťových buniek. Ako sú realizované jednotlivé pamäťové bunky, závisí od konkrétneho typu pamäte. [2]

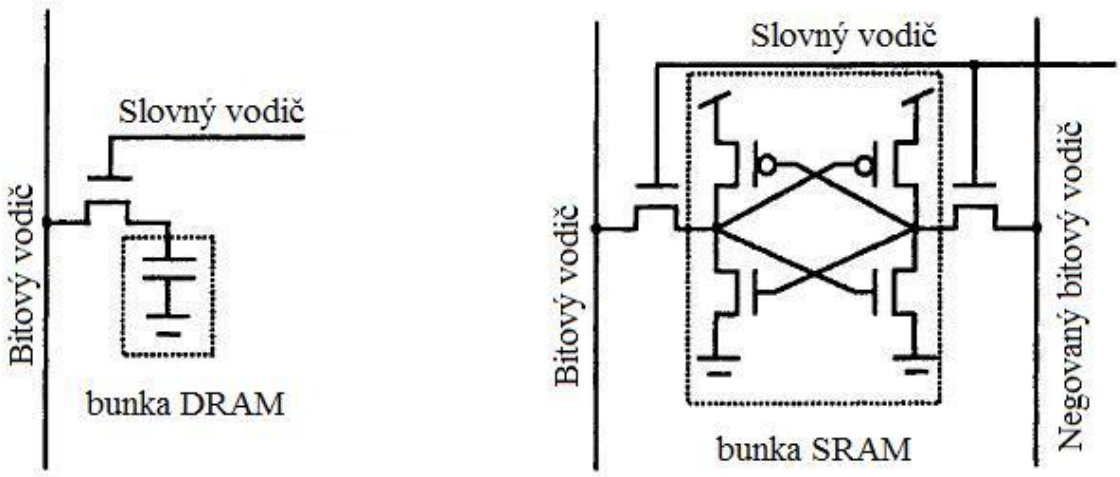
V systémoch na čipe sa používajú v prevažnej väčšine pamäte typu RAM (Random access memory). Tieto pamäte majú výhodu v rovnakej dobe prístupu ku každej ich bunke a vďaka svojej pravidelnej štruktúre sú vhodné na veľký stupeň integrácie a osadenie na čip. Delenie pamätí typu RAM z pohľadu funkcionality je na [2]:

- ROM (Read only memory) – pamäť RAM, z ktorej je možné údaje len čítať,
- RWM (Read write memory) – pamäť RAM, z ktorej je možné údaje čítať, aj do nej údaje zapisovať.

Iné delenie pamätí typu RAM je podľa spôsobu relizácie pamäťovej bunky na [2]:

- SRAM (Static RAM) – statické pamäte RAM,
- DRAM (Dynamic RAM) – dynamické pamäte RAM.

Na obrázku Obr. 2.1 je zobrazená štruktúra buniek pamätí DRAM a SRAM. Ako vidieť, bunka pamäte SRAM pozostáva zo šiestich tranzistorov. Takáto bunka, pokiaľ je bezporuchová, si uchová uložené údaje po celú dobu pripojenia k napätiu. Bunka pamäte DRAM pozostáva z jedného tranzistora a jedného kondenzátora. Obsah takejto bunky musí byť pravidelne obnovovaný, pretože nie je prítomné priame prepojenie pre dobitie kondenzátora, v ktorom je uložený údaj. Z porovnania štruktúry buniek je zrejmé, že pamäť SRAM bude mať vyššiu energetickú náročnosť a cenu, ale kratšiu dobu prístupu v porovnaní s pamäťou DRAM a tiež je lepšia pre použitie ako vnorená. Pamäť DRAM je vďaka svojim vlastnostiam (vysoká hustota, nízka cena, nižšia energetická náročnosť) vhodná pre použitie ako veľkokapacitná pamäť. [2]



Obr. 2.1: Štruktúra buniek pamätí DRAM a SRAM [2]

Zo spôsobu realizácie pamäťovej bunky pamäte ďalej vyplývajú modely porúch pamäte, ktoré slúžia ako nejaký „odrazový mostík“ pre vytváranie testovacích stratégií na odhalenie porúch v pamäti. Nakoľko štruktúra buniek SRAM a DRAM je odlišná, musia byť na ich testovanie použité rôzne testovacie stratégie. [2]

2.2 Modely porúch

Model poruchy reprezentuje jednu z možností, akým spôsobom môže pamäťová bunka funkcionálne zlyhať. Znalosť modelov porúch pamätí, ktoré chceme testovať, je kľúčová pre správne zostavenie postupnosti testovacích operácií (resp. stratégie testovania), a tak byť schopný poruchu v pamäti detegovať.

Základné rozdelenie modelov porúch je na statické, dynamické, modely porúch n spárených buniek, modely viacnásobných porúch. Nakoľko testy march implementované v rámci riešenia zadania diplomového projektu budú schopné detegovať statické modely porúch (v skutočnosti aj dynamické modely porúch, ktoré sa od statických líšia len tým, že sú vyvolané bezprostredne za sebou vykonávanými operáciami nad pamäťovou bunkou, no majú rovnaké charakteristiky prejavu poruchy), uvediem ďalej len tieto modely porúch. [4]

Modely porúch jednej bunky (single-cell faults) [4]:

- Trvalá porucha (State Fault, SF_x) – pamäťová bunka obsahuje stále rovnakú hodnotu, bez ohľadu na operácie zápisu a čítania, ktoré sú nad ňou vykonávané. Exitujú 2 typy trvalej poruchy – porucha trvalej nuly (SF_0) a porucha trvalej jednotky (SF_1).
- Porucha prechodu (Transition fault, TF_x) – obsah pamäťovej bunky nie je možné zmeniť buď z hodnoty nula na jedna (TF_1) alebo z hodnoty jedna na nula (TF_0). Ak sa do danej bunky zapíše hodnota, z ktorej už nie je schopná prechodu na inú hodnotu, ostáva táto hodnota v bunke po celý čas.
- Porucha deštrukčného čítania (Read destructive fault, RDF_x) – operácia čítania vykonaná nad poruchovou bunkou preklopí jej obsah a vráti nesprávnu hodnotu na výstup pamäte. Existujú 2 typy tejto poruchy – pamäťová bunka obsahuje hodnotu jedna, po jej prečítaní sa jej obsah zmení na hodnotu nula a na výstup pamäte sa vráti hodnota nula (RDF_1) alebo naopak (RDF_0).
- Porucha deštrukčného zápisu (Write destructive fault, WDF_x) – operácia zápisu rovnakej hodnoty, akú momentálne má poruchová bunka, spôsobí, že obsah pamäťovej bunky sa preklopí (bude obsahovať opačnú hodnotu). Existujú 2 typy tejto poruchy – pamäťová bunka obsahuje hodnotu jedna, vykoná sa nad ňou operácia zápisu hodnoty jedna, no obsah pamäťovej bunky sa preklopí na hodnotu nula (WDF_1) alebo naopak (WDF_0).
- Porucha nesprávneho čítania (Incorrect read fault, IRF_x) – operácia čítania vykonaná nad poruchovou bunkou vráti nesprávnu hodnotu na výstup pamäte, pričom obsah bunky sa nemení. Exitujú 2 typy tejto poruchy – pamäťová bunka obsahuje hodnotu jedna, po jej prečítaní sa na výstup pamäte vráti hodnota nula (IRF_1) alebo naopak (IRF_0).
- Klamná porucha deštrukčného čítania (Deceptive read destructive fault, $DRDF_x$) – operácia čítania vykonaná nad poruchovou bunkou preklopí jej obsah, ale vráti správnu hodnotu (očakávanú) na výstup pamäte. Exitujú 2 typy tejto poruchy - pamäťová bunka obsahuje hodnotu jedna, po jej prečítaní sa jej obsah zmení na hodnotu nula a na výstup pamäte sa vráti hodnota jedna ($DRDF_1$) alebo naopak ($DRDF_0$).

Modely porúch spárených buniek (coupling faults). Pri týchto modeloch majú pamäťové bunky označenie agresor (keď sa vykoná operácia nad touto pamäťovou bunkou alebo je bunka v určitom stave, ovplyvní to nejakým spôsobom hodnotu v druhej bunke, väčšinou susednej) a obeť (bunka ovplyvnená operáciou vykonanou na bunkou agresor alebo stavom agresora). V opise jednotlivých modelov porúch použijem skrátené označenia agresor (pamäťová bunka agresor) a obeť (pamäťová bunka obeť). [4]

- Rušivá porucha spárených buniek (Disturb coupling fault, CF_{ds}) – operácia čítania alebo zápisu vykonaná nad agresorom spôsobí zmenu obsahu obete na danú hodnotu. Exituje 16 typov tejto poruchy – ak agresor má hodnotu nula a obeť má hodnotu nula, vykoná sa operácia zápisu hodnoty nula nad agresorom, hodnota obete sa zmení na jedna ($CF_{ds(00,w0)}$) a všetky možné kombinácie s počiatočnými hodnotami buniek a operáciami čítania a zápisu.

Nakoľko zvyšné modely porúch spárených buniek sú odvodené od vyššie popísaných modelov porúch jednej bunky, uvediem len jednu poruchu ako príklad [4]:

- Trvalá porucha spárených buniek (State coupling fault, CF_{st}) – hodnota obete sa zmení na danú logickú hodnotu, ak agresor má určitú danú hodnotu, bez potreby vykonania operácie nad týmito bunkami. Existujú 4 typy tejto poruchy – ak agresor má hodnotu nula a obeť má hodnotu nula, hodnota obete sa zmení na jedna ($CF_{st(00)}$) a naopak ($CF_{st(11)}$), a ak agresor má hodnotu nula a obeť má hodnotu jedna, hodnota obete sa zmení na nula ($CF_{st(01)}$) a naopak ($CF_{st(10)}$).

Medzi modely porúch, ktoré sú testy typu march sú schopné detegovať, patria aj tieto [4]:

- Porucha prístupu (Stuck open fault, SOF) – obsah bunky je neprístupný.
- Porucha adresácie (Address decoder fault, AF) – bunka nie je adresovaná žiadnou adresou alebo je adresovaná viacerými adresami.

2.3 Algoritmy march

Algoritmy testovania pamätí typu march sú veľmi vhodné pre použitie v jednotke samočinného testovania pamäte, najmä pre ich lineárnu zložitosť (*n* – počet testovaných buniek pamäte), pravidelnosť a dobré pokrytie porúch. Z toho dôvodu sa aj v dnešnej dobe na ich vývoji stále pracuje, so snahou o pokrytie čo najväčšieho množstva modelov porúch vrátane nových modelov porúch (modely porúch, ktoré prichádzajú s novými technológiami a čoraz väčšou integráciou) s čo možno najmenšou časovou zložitosťou testu. [1]

Testy typu march pozostávajú z postupnosti operácií zápisu a čítania, vykonávaných na bunkách pamäte vo vzostupnom alebo zostupnom poradí adries. Rôznymi kombináciami týchto operácií sa dá dosiahnuť rôzne pokrytie modelov porúch. Operácie sú zoskupené do elementov. Element testu march je teda skupina operácií, ktoré sa vykonajú na jednej pamäťovej bunke za sebou a až potom sa prechádza na testovanie ďalšej pamäťovej bunky.

V tabuľke Tab. 2.1 je uvedených niekoľko príkladov algoritmov typu march. Použitá notácia operácií:

- ⇑ – smer adresácie od najnižšej adresy po najvyššiu
- ⇓ – smer adresácie od najvyššej adresy po najnižšiu
- ⇳ – na smere adresácie nezáleží
- w0 – operácia zápisu log. 0,
- w1 – operácia zápisu log. 1,
- r0 – operácia čítania log. 0,
- r1 – operácia čítania log. 1.

Tab. 2.1: Vybrané typy march testov [5], [11], [12], [14]

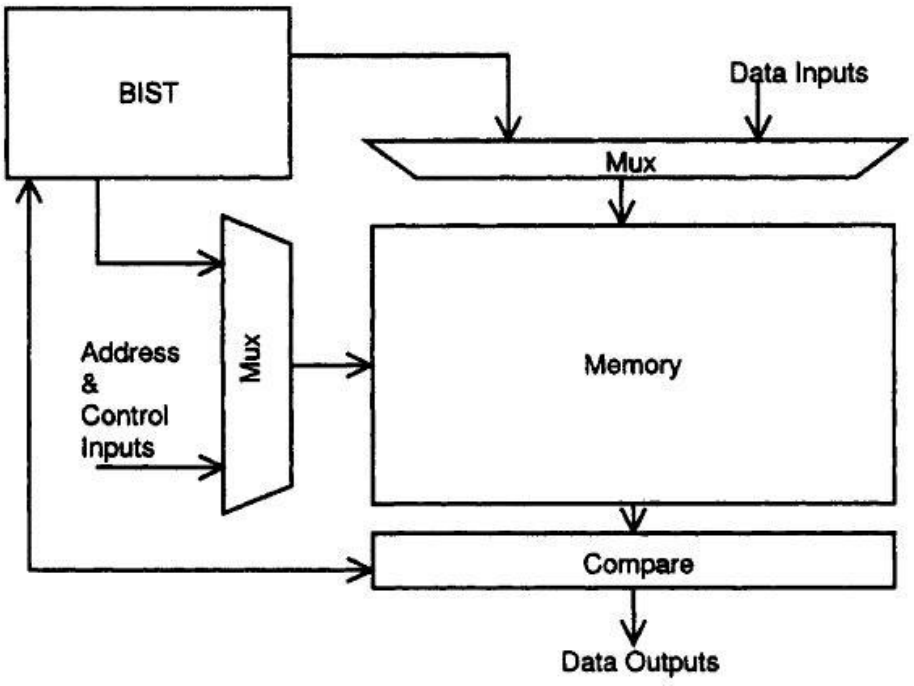
Algoritmus	Operácie
MATS (4n)	⇳ (w0); ⇳ (r0, w1); ⇳ (r1)
MATS+ (5n)	⇳ (w0); ⇑ (r0, w1); ⇓ (r1, w0)
MATS++ (6n)	⇳ (w0); ⇑ (r0, w1); ⇓ (r1, w0, r0)
March X (6n)	⇳ (w0); ⇑ (r0, w1); ⇓ (r1, w0); ⇳ (r0)

March Y (8n)	⇳ (w0); ⇑ (r0, w1, r1); ⇓ (r1, w0, r0); ⇳ (r0)
March C- (10n)	⇳ (w0); ⇑ (r0, w1); ⇑ (r1, w0); ⇓ (r0, w1); ⇓ (r1, w0); ⇳ (r0)
March U (12n)	⇳ (w0); ⇑ (r0, w1, r1, w0); ⇑ (r0, w1); ⇑ (r1, w0, r0, w1); ⇳ (r0)
March LR (14n)	⇳ (w0); ⇓ (r0, w1); ⇑ (r1, w0, r0, w1); ⇑ (r1, w0); ⇑ (r0, w1, r1, w0); ⇳ (r0)
March SR (14n)	⇓ (w0); ⇑ (r0, w1, r1, w0); ⇓ (r0, r0); ⇑ (w1); ⇓ (r1, w0, r0, w1); ⇑ (r1, r1)
Marching 1/0 (14n)	⇑ (w0); ⇑ (r0, w1, r1); ⇓ (r1, w0, r0); ⇑ (w1); ⇑ (r1, w0, r0); ⇓ (r0, w1, r1)
March SRD (14n + oneskorenia)	⇓ (w0); ⇑ (r0, w1, r1, w0); oneskorenie; ⇓ (r0, r0); ⇑ (w1); ⇓ (r1, w0, r0, w1); oneskorenie; ⇑ (r1, r1)
March A (15n)	⇳ (w0); ⇑ (r0, w1, w0, w1); ⇑ (r1, w0, w1); ⇓ (r1, w0, w1, w0); ⇓ (r0, w1, w0)
IFA-13 (16n + oneskorenia)	⇑ (w0); ⇑ (r0, w1, r1); ⇑ (r1, w0, r0); ⇓ (r0, w1, r1); ⇓ (r1, w0, r0); oneskorenie; ⇑ (r0, w1); oneskorenie; ⇑ (r1)
March B (17n)	⇳ (w0); ⇑ (r0, w1, r1, w0, r0, w1); ⇑ (r1, w0, w1); ⇓ (r1, w0, w1, w0); ⇓ (r0, w1, w0)
Enhanced March C- (18n)	⇳ (w0); ⇑ (r0, w1, r1, w1); ⇑ (r1, w0, r0, w0); ⇓ (r0, w1, r1, w1); ⇓ (r1, w0, r0, w0) ; ⇳ (r0)
March LA (22n)	⇳ (w0); ⇑ (r0, w1, w0, w1, r1); ⇑ (r1, w0, w1, w0, r0); ⇓ (r0, w1, w0, w1, r1); ⇓ (r1, w0, w1, w0, r0); ⇓ (r0)
March SS (22n)	⇳ (w0); ⇑ (r0, r0, w0, r0, w1); ⇑ (r1, r1, w1, r1, w0); ⇓ (r0, r0, w0, r0, w1); ⇓ (r1, r1, w1, r1, w0); ⇳ (r0)
March G (23n)	⇳ (w0); ⇑ (r0, w1, r1, w0, r0, w1); ⇑ (r1, w0, w1); ⇓ (r1, w0, w1, w0); ⇓ (r0, w1, w0); ⇓ (r0); oneskorenie; ⇳ (r0, w1, r1); oneskorenie; ⇳ (r1, w0, r0)
March MSL (23n)	⇳ (w0); ⇑ (r0, w1, w1, r1, r1, w0); ⇑ (r0, w0); ⇑ (r0); ⇑ (r0, w1); ⇑ (r1, w0, w0, r0, r0, w1); ⇑ (r1, w1); ⇑ (r1); ⇓ (r1, w0)
March SL (41n)	⇳ (w0); ⇑ (r0, r0, w1, w1, r1, r1, w0, w0, r0, w1); ⇑ (r1, r1, w0, w0, r0, r0, w1, w1, r1, w0); ⇓ (r0, r0, w1, w1, r1, r1, w0, w0, r0, w1); ⇓ (r1, r1, w0, w0, r0, r0, w1, w1, r1, w0)
March BLC (46n)	⇳ (w0); ⇑ (r0, r0,w0, r0,w1,w1, r1); ⇑ (r1, r1,w1, r1,w0, w1); ⇑ (r1, r1, w0, w0, r0); ⇑ (r0, r0, w0, r0, w1, w1, w0); ⇓ (r0, r0, w0, w1, w1, r1); ⇓ (r1, r1, w0, w1); ⇓ (r1, r1, w0, w0, r0); ⇓ (r0, r0, w1, w1, w0)

2.4 Architektúra samočinného testovania

Jednotka vstavaného samočinného testovania slúži na otestovanie pamäte a poskytnutie informácií o detegovaných poruchách. Je realizovaná ako digitálna logika, ktorá je súčasťou pamäťového čipu. Je veľmi užitočná pre testovanie pamätí vnorených v systémoch na čipe, pretože takéto pamäte nemajú žiadne prepojenia s „vonkajším svetom“, tým pádom sa nedajú otestovať externými testovacími zariadeniami. Pri návrhu jednotky vstavaného samočinného testovania je potrebné vychádzať zo štruktúry pamäte a z nej vyplývajúcich modelov porúch. Taktiež je dôležitá potreba malého percenta plochy na čipe v porovnaní s testovanou pamäťou a schopnosť testovať pamäť na jej pracovnej frekvencii. [2]

Na obrázku Obr. 2.2 je zobrazená základná architektúra BIST jednotky a jednotlivé prepojenia s testovanou pamäťou (*Memory*). Signály pre údajové vstupy (*Data Inputs*), adresné a ovládacie vstupy (*Address & Control Inputs*) sú multiplexované. Toto riešenie zabezpečí, že vo funkčnom režime bude pamäť ovládaná perifériami na čipe a v testovacom režime bude ovládaná jednotkou *BIST*. Údajové výstupy (*Data Outputs*) vstupujú najskôr do komparátora, ktorý v testovacom režime deteguje poruchy porovnávaním výstupných údajov z pamäte s očakávanými údajmi. [2]



Obr. 2.2: Základná architektúra BIST jednotky [2]

Architektúry BIST možno rozdeliť na 2 základné typy: architektúra BIST založená na konečnom stavovom automate (BIST s pevnou logikou) a architektúra BIST založená na mikrokódovaní. [3]

Architektúra BIST založená na konečnom stavovom automate môže generovať jeden typ testu alebo niekoľko typov testu. Táto architektúra je v praxi využívaná pre generovanie len jedného typu testu (napr. 1 test typu march). Keď chceme otestovať pamäť komplexnejšie, je potrebných niekoľko typov testu, čo robí návrh konečných stavových automatov pre reprezentáciu viacerých typov testu komplexným a netriviálnym. Z reprezentácie testov stavovým automatom vyplýva výrazné obmedzenie vo flexibilite generovaných testov. Ak chceme zmeniť testovacie vzorky, je potrebné znova navrhnúť konečný stavový automat. Hlavnou výhodou tejto architektúry je menšia plocha na čipe v porovnaní s architektúrou založenou na mikrokódovaní. [3]

Architektúra BIST založená na mikrokódovaní je programovateľná, tým pádom poskytuje oveľa väčšiu flexibilitu vo voľbe testovacích vzoriek aplikovaných na testovanú pamäť. Testovacie vzorky vo forme mikroinštrukcií bývajú uložené v pamäti inštrukcií a pred samotným testovaním (resp. počas testovania) sa musia nahrať do pamäte riadiacej jednotky, ktorá ich dekóduje a následne generuje zodpovedajúce testovacie vzorky. Berúc do úvahy fakt, že niektoré typy porúch sa odhalia až počas výroby pamätí v danej technológii, možnosť modifikovať testovacie vzorky pri tejto architektúre je obrovská výhoda. Po navhrnutí testovacích vzoriek tak, aby pokrývali aj novo odhalené poruchy pamätí, je možné ich do tejto architektúry naprogramovať a testovať pamäte aj na tieto nové poruchy bez potreby nového návrhu architektúry BIST. Najväčšou nevýhodou tejto architektúry je väčšia plocha na čipe v porovnaní s architektúrou založenou na konečnom stavovom automate. [3]

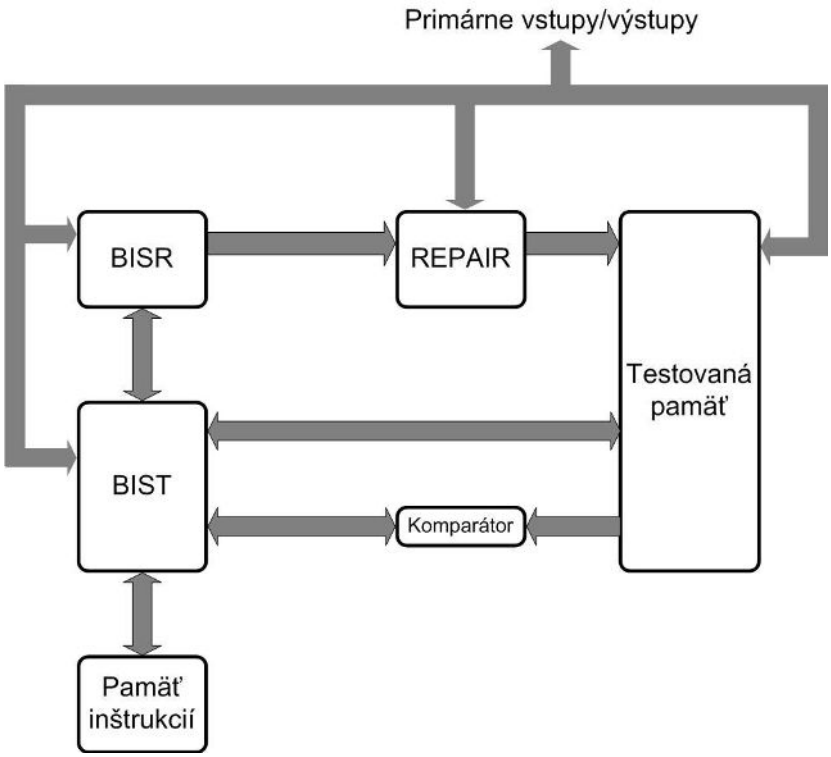
2.5 Architektúra analýzy opravy

Jednotka vstavanej analýzy opravy (BIRA) slúži na alokáciu záložných prvkov pamäte namiesto poruchových buniek. Informácie o poruchových miestach pamäte dostáva spravidla z jednotky samočinného testovania BIST. Prítomnosť tejto jednotky na čipe pamätí výrazne zvyšuje ich výťažnosť, vďaka jej schopnosti opraviť poruchovú pamäť. Či bude oprava úspešná alebo nie, závisí od stratégie alokácie záložných prvkov a od zvoleného algoritmu, ktorý túto alokáciu vykonáva. Každá pamäť, ktorá obsahuje jednotku

BIRA, obsahuje spravidla niekoľko záložných prvkov (pamäťové bunky navyše), ktoré môžu nahradiť poruchové miesta pamäte, a tým pamäť opraviť. Nahradenie poruchových prvkov pamäte za bezporuchové sa vykonáva readresáciou. Podľa toho, aké typy záložných prvkov sa v pamäti nachádzajú, môžme hovoriť o 1D redundancii (pamäť obsahuje len záložné stĺpce alebo len záložné riadky), 2D redundacii (pamäť obsahuje záložné stĺpce aj záložné riadky), 3D redundacii (pamäť obsahuje záložné riadky, stĺpce a pamäťové bloky) a pod. Ak je použitá 1D redundancia, záložný prvok sa aplikuje prakticky hneď po detegovaní poruchy. Z toho vyplýva výrazné obmedzenie – ak sa v pamäti nachádzajú poruchy na viacerých riadkov/stĺpcoch, ako je počet záložných prvkov, pamäť je neopraviteľná. Pri použití 2D a 3D redundancie už nie je alokácia záložných prvkov triviálna úloha, preto sa neustále pracuje na algoritmoch, ktoré dosiahnu čo najlepšiu pravdepodobnosť opravy pamäte, pokiaľ je oprava uskutočniteľná použitím dostupných záložných prvkov. [2], [3]

2.6 Analýza existujúceho riešenia (Moťovský E., 2010)

Analyzované riešenie [5] predstavuje celkové riešenie architektúry pre samočinné testovanie a opravu bitovo orientovaných pamätí RAM. Na obrázku Obr. 2.3 je zobrazený blokový diagram architektúry.



Obr. 2.3: Blokový diagram analyzovanej architektúry BISR [5]

Význam jednotlivých blokov [5]:

- *Testovaná pamäť* – predstavuje opraviteľnú pamäť RAM pozostávajúcu zo samotnej pamäte RAM a záloh.
- *BIST* – predstavuje blok samočinného testovania, ktorý dokáže v pamäti RAM detegovať poruchy a odoslať zodpovedajúce informácie o poruchách do bloku *BISR*.
- *BISR* – predstavuje blok samočinnej analýzy opravy.
- *REPAIR* – vykonáva funkciu adresného rekonfigurátora.
- *Pamäť inštrukcií* – pamäť inštrukcií pre algoritmy march, čítaná jednotkou *BIST*.
- *Komparátor* – slúži na kontrolu správnosti prečítaných údajov z pamäte pri testovaní pamäte.

Jednotka vstavaného samočinného testovania (BIST) je založená na mikrokódovaní. Podporuje testy typu march. Pred spustením testovania je možné do pamäti inštrukcií nahrať postupnosť inštrukcií a tak ovlyvniť, aký test march sa vykoná. Výhodou tohto riešenia je poskytnutá flexibilita vo voľbe march testu. Zásadným obmedzením implementácie tohto riešenia je fakt, že vzhľadom na zvolený formát zápisu inštrukcií pomocou mikrokódov BIST jednotka nepodporuje march testy, ktoré obsahujú 2x rovnakú operáciu ihneď za sebou. Stráca sa tým možnosť použiť napr. algoritmus March SR, ktorý je schopný detegovať aj dynamické poruchy v pamäti RAM typu klamná porucha deštruktívneho čítania (ang. deceptive read destructive fault) a pod. Analyzovaná architektúra bola testovaná použitím testov MATS+, MATS++ a March C-. [5]

BIRA jednotka vykonáva analýzu opravy algoritmom SFCC (Selected fail count comparison). Informácie o poruchách si ukladá do 3-och registrov (Parent, Child, Must-Repair). Podľa obsahu v týchto registroch sa následne alokujú záložné prvky, ktoré môžu byť buď záložný riadok, alebo záložný stĺpec. Je použitá 2D redundancia. [5]

Model pamäte použitej pre testovanie algoritmu nie je známy, v dokumentácii je spomenutá len veľkosť pamäte. Konkrétne sa testovanie vykonávalo na pamätiach veľkosti 1024 x 1024 bitov, 512 x 512 bitov a 256 x 256 bitov. Vo všetkých prípadoch bolo k dispozícii 5 záložných riadkov a 5 záložných stĺpcov. [5]

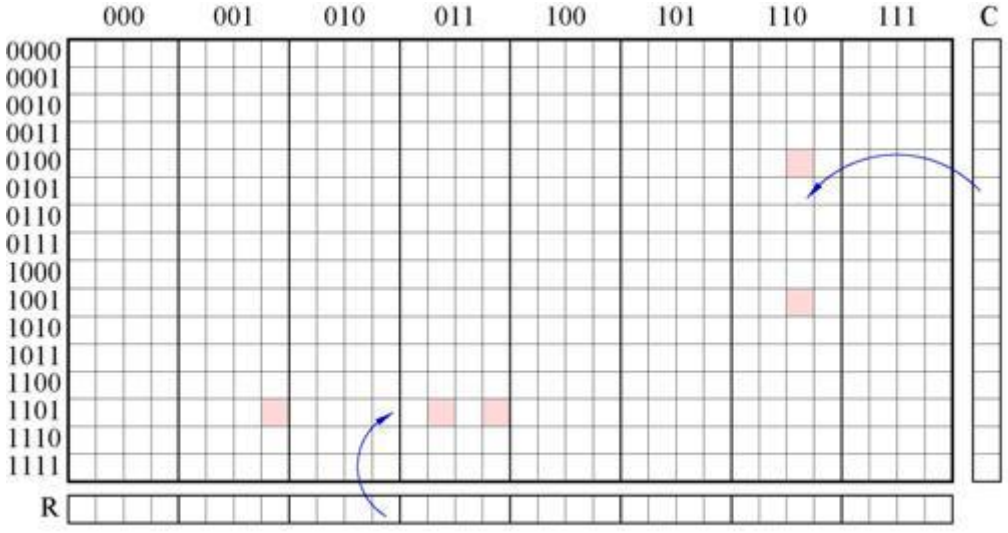
2.7 Analýza moderných prístupov k riešeniu BIRA

2.7.1 BIRA s 2D redundanciou a optimálnou úspešnosťou opravy

Všeobecne

Analyzované riešenie predstavuje algoritmus opravy slovne orientovaných pamätí RAM. Algoritmus analýzy opravy BIRA pozostáva z dvoch fáz – fáza nevyhnutnej opravy [6] (ang. must-repair) a finálna fáza. Pri alokácii záložných prvkov sa vychádza z lokálnej bitmapy [6] najmenšej možnej veľkosti vzhľadom na počet záloh. Vo finálnej fáze algoritmu sa vykonáva algoritmus vyčerpávajúceho hľadania riešenia opravy. Na tento účel je použitý sekvenčný analyzátor, ktorý prechádza postupne všetkými možnými riešeniami aplikácie záloh na poruchové miesta, kým nenájde úspešné riešenie opravy pamäte.

Schéma BIRA je navrhnutá pre *N* x *M* x *W* - bitovú pamäť RAM s lokálnymi záložnými riadkami a stĺpcami, ktoré môžu byť použité ako náhrada za ktorýkoľvek riadok resp. stĺpec pamäte. Jedná sa o 2D redundanciu. Písmeno *N* predstavuje počet riadkov, písmeno *M* predstavuje počet stĺpcov a písmeno *W* predstavuje počet bitov jedného slova pamäte. Organizácia takejto pamäte je zobrazená na obrázku Obr. 2.4. [6]

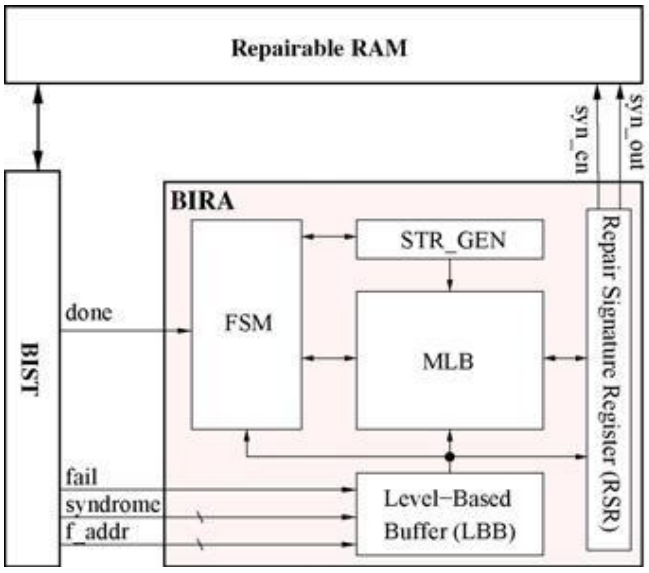


Obr. 2.4: Organizácia slovne orientovanej pamäte RAM s 1 záložným riadkom (R) a stĺpcom (C) [6]

Na obrázku Obr. 2.5 je zobrazený koncepčný diagram analyzovaného BISR obvodu.

Význam jednotlivých blokov [6]:

- *Repairable RAM* – predstavuje opraviteľnú pamäť RAM pozostávajúcu zo samotnej pamäte RAM obsahujúcej zálohy a rekonfiguračného mechanizmu. Tento mechanizmus dokáže na základe príznaku opravy nahradiť poruchové časti pamäte záložnými.
- *BIST* – predstavuje blok samočinného testovania, ktorý dokáže v pamäti RAM detegovať poruchy a odoslať zodpovedajúce informácie o poruchách do bloku *BIRA*.
- *BIRA* – predstavuje blok samočinnej analýzy opravy, ktorý na základe informácií o poruchách pamäte z bloku *BIST* posiela príznak opravy do opraviteľnej pamäte RAM. Skladá sa z nasledovných častí:
    - *FSM* – konečný stavový automat, vykonáva samotnú analýzu opravy a pracuje paralelne s blokom *BIST*.
    - *STR_GEN* – generátor stratégie opravy (vysvetlené v kapitole 3.3, str. 54), generuje postupne všetky možné stratégie opravy na základe voľných záloh.
    - *MLB* – lokálna bitmapa, používa sa pre ukladanie informácií o poruchách pamäte a vykonávanie algoritmu analýzy opravy.
    - *Level-Based Buffer (LBB)* – zásobník, zbiera informácie o poruchách pamäte odoslané z bloku *BIST* a zisťuje adresu poruchového bitu.
    - *Repair Signature Register (RSR)* – register príznaku opravy, používa sa na ukladanie informácií o oprave odoslaných z bloku *MLB* a taktiež na kontrolu úspešnosti vykonanej opravy.



Obr. 2.5: Blokový diagram BISR obvodu [6]

Princíp fungovania

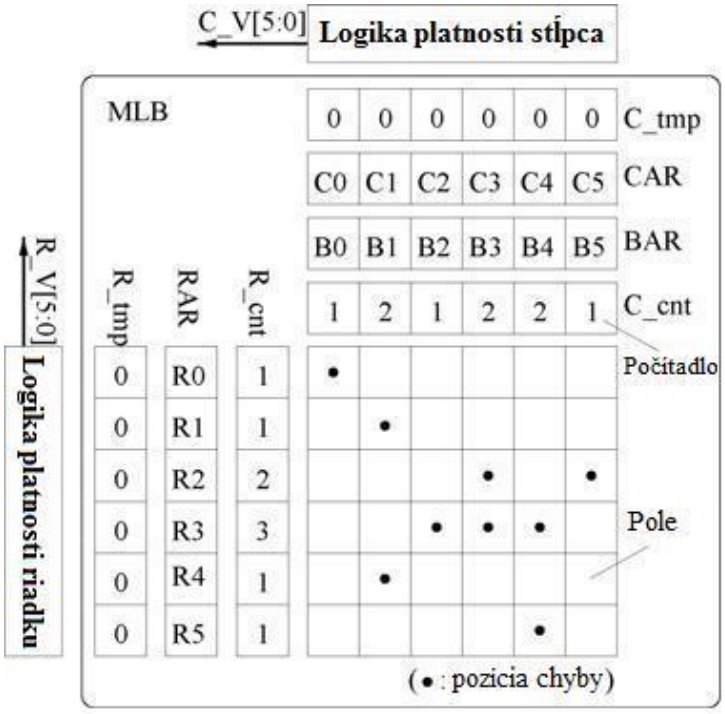
Keď blok *BIST* deteguje poruchu v pamäti, aktivuje sa signál *fail*. Následne sa odošlú informácie o poruche do bloku *BIRA* prostredníctvom signálov *syndrome* a *f_addr*. Tieto informácie o poruche obsahujú adresu poruchovej bunky (väčšinou adresa riadka a adresa stĺpca) a syndróm, čo je vlastne výsledok operácie XOR vykonanej nad slovom pamäte (porovnáva sa prečítaná hodnota s očakávanou). Syndróm obsahuje hodnotu log. 1 na pozícii poruchového bitu slova. [6]

Po prijatí signálov sa v zásobníku (*LBB*) určí adresa poruchového bitu v slove a prepošlú sa informácie o poruche pamäte (adresa riadka, adresa stĺpca a adresa bitu) do lokálnej bitmapy. Tieto informácie sa porovnajú s informáciami o opravených poruchách v registri príznaku opravy a pokiaľ sa porovnaním zistí, že daná porucha je už pokrytá, nezapisuje sa do lokálnej bitmapy. Ak daná porucha pokrytá nie je, informácie sa zapíšu do lokálnej bitmapy. [6]

Na obrázku Obr. 2.6 je zobrazený blokový diagram použitej lokálnej bitmapy o veľkosti 6x6. Význam jednotlivých registrov [6]:

- *RAR* – register riadkových adries porúch,
- *CAR* – register stĺpcových adries porúch,
- *BAR* – register bitových adries porúch,
- *R_cnt* – počítadlo porúch v riadku, využitie najmä vo fáze nutnej opravy – ak jeho hodnota presiahne počet voľných záložných stĺpcov, je splnená podmienka nutnej opravy,
- *C_cnt* – počítadlo porúch v stĺpci, využitie najmä vo fáze nutnej opravy – ak jeho hodnota presiahne počet voľných záložných riadkov, je splnená podmienka nutnej opravy,
- *R_tmp* – register pre indikáciu opravených riadkov počas finálnej fázy algoritmu,
- *C_tmp* – register pre indikáciu opravených stĺpcov počas finálnej fázy algoritmu,
- *R_V* – register indikujúci platnosť záznamu v danom riadku lokálnej bitmapy počas fázy nutnej opravy, počas finálnej fázy sa používa na kontrolu úspešnosti opravy. Jeho obsah je generovaný logikou platnosti riadku,

- *C_V* – register indikujúci platnosť záznamu v danom stĺpci lokálnej bitmapy počas fázy nutnej opravy, počas finálnej fázy sa používa na kontrolu úspešnosti opravy. Jeho obsah je generovaný logikou platnosti stĺpca.



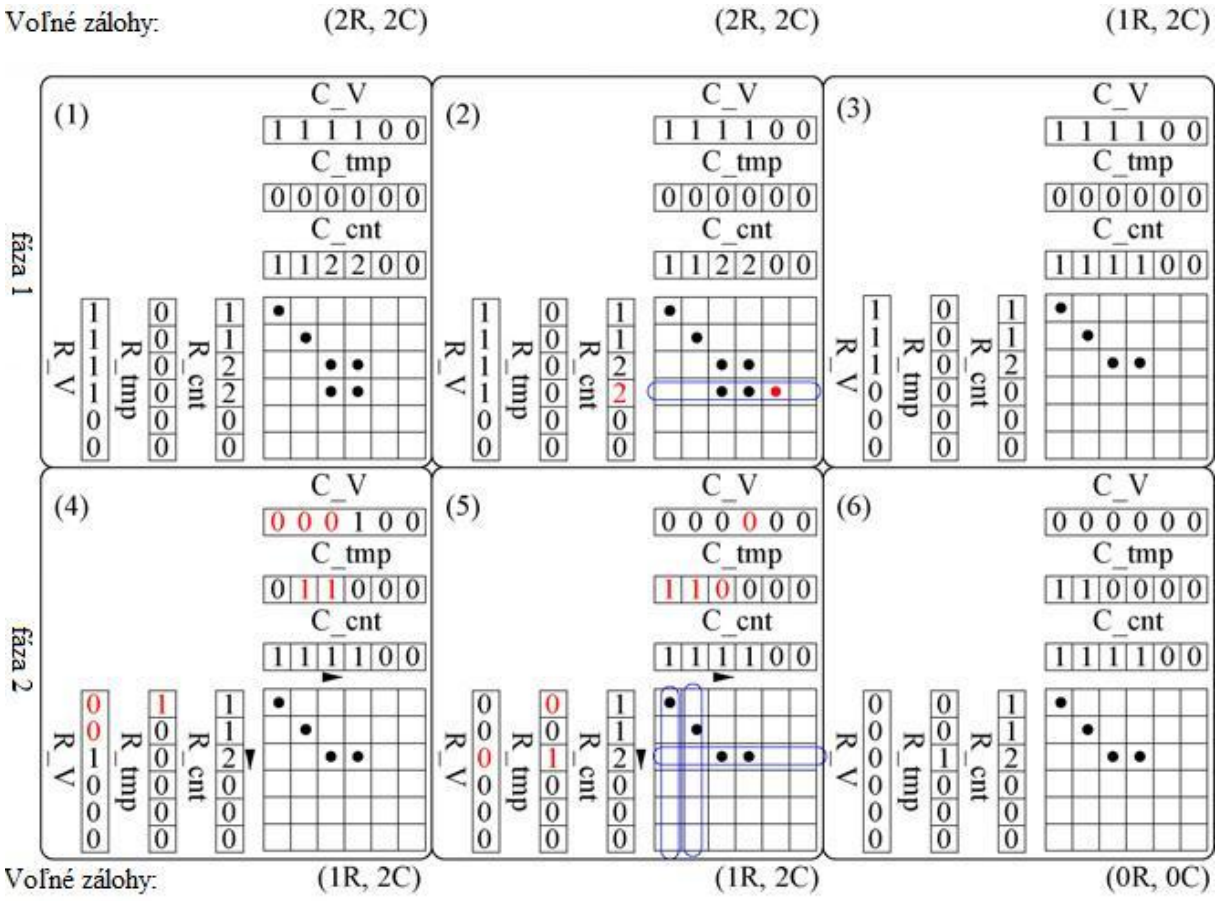
Obr. 2.6: Blokový diagram lokálnej bitmapy o veľkosti 6x6 [6]

Obsah registrov sa pravidelne aktualizuje v závislosti na priebehu testovania a algoritmu analýzy opravy. Na začiatku obsahujú všetky registre samé hodnoty log. 0. Keď sa zistí porucha a ešte nie je pokrytá, registre *RAR*, *CAR* a *BAR* sú naplnené adresami riadka, stĺpca a bitu. Taktiež sú aktualizované hodnoty počítadiel *R_cnt* a *C_cnt*, *R_V* a *C_V*. Ako je spomenuté vyššie, registre *R_tmp* a *C_tmp* indikujú opravené riadky resp. stĺpce vo finálnej fáze, kedy sa môže skúšať vykonať oprava viacerými stratégiami pokrytia porúch záložnými riadkami a stĺpcami a v prípade, že daná stratégia nie je úspešná, ide sa na ďalšiu. [6]

Príklad

Na obrázku Obr. 2.7 je zobrazený príklad, ako sa menia stavy lokálnej bitmapy a jednotlivých registrov počas behu algoritmu opravy. Pre lepšiu prehľadnosť sú vynechané registre RAR, CAR a BAR, nakoľko tie obsahujú vždy len adresy a z pohľadu fungovania algoritmu nie sú zaujímavé. Predpokladá sa počet záloh 2 riadky a 2 stĺpce. Popis jednotlivých krokov algoritmu (častí obrázku pomocou čísla v ľavom rohu v zátvorke) [6]:

- (1) – Predpokladá sa zobrazený stav lokálnej bitmapy, kedy je detegovaných 6 porúch, ktoré sú zapísané do lokálnej bitmapy a zodpovedajúce registre sú nastavené na správne hodnoty. Algoritmus sa nachádza vo fáze nevyhnutnej opravy, testovanie pamäte stále prebieha.
- (2) – Našla sa tretia porucha na jednom riadku (červená bodka), zodpovedajúce pole počítadla *R_cnt* by následne malo hodnotu 3, čo je ale viac, ako je dostupných záložných stĺpcov. Tým pádom je splnená podmienka nevyhnutnej opravy a daný riadok je nahradený záložným riadkom.
- (3) – V tomto okne je zobrazený stav po alokácii záložného riadku a aktualizovaní hodnôt v registroch. Predpokladá sa ukončenie testovania a prechod do finálnej fázy algoritmu.
- (4) – 1. pokus vo finálnej fáze algoritmu o pokrytie porúch stratégiou RCC. Aká stratégia sa práve aplikuje je možné vidieť na hodnotách registrov *R_tmp* a *C_tmp* – na mieste, kde sa nachádza hodnota log. 1, sa aplikuje záloha. Následne sa vypočítajú hodnoty v registroch *R_V* a *C_V* a pretože neobsahujú samé log. 0, oprava touto stratégiou nie je úspešná. Porucha v 3. riadku a 4. stĺpci lokálnej bitmapy ostane nepokrytá.
- (5) – 2. pokus vo finálnej fáze algoritmu o pokrytie porúch stratégiou CCR. V tomto prípade je oprava úspešná, všetky poruchy sú pokryté.
- (6) – V tomto okne je len zobrazený finálny stav so statusom, že pamäť je opraviteľná.



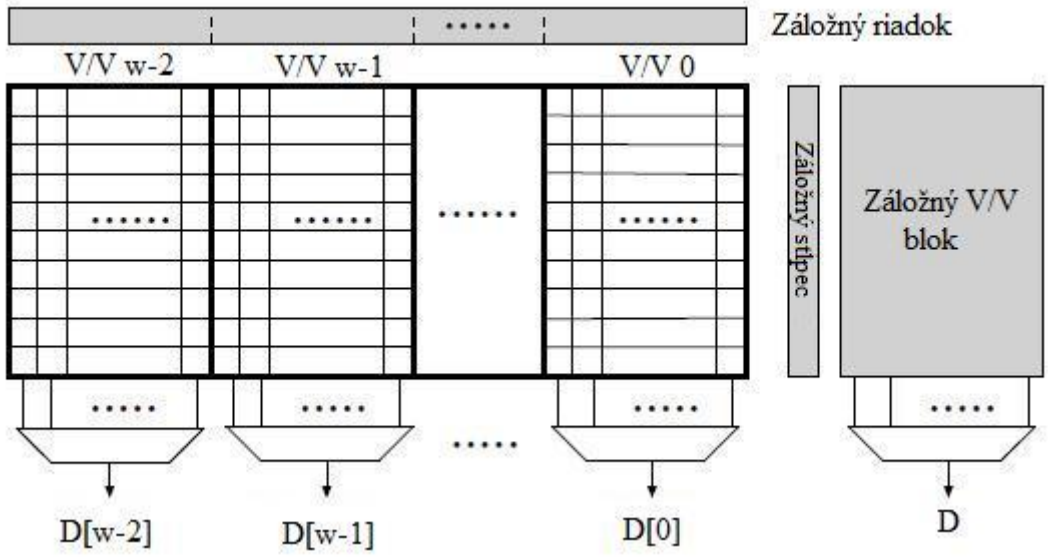
Obr. 2.7: Príklad fungovania analyzovaného algoritmu [6]

2.7.2 BIRA s 3D redundanciou

Všeobecne

Analyzované riešenie predstavuje algoritmus opravy slovne orientovaných pamätí RAM s použitím 3D redundancie. Záložné prvky sú záložný riadok, stĺpec a vstupno-výstupný blok. Pri alokácii záložných prkov sa vychádza z lokálnej bitmapy najmenšej možnej veľkosti vzhľadom na počet záloh a nastavenie prahovej hodnoty pre alokáciu záložného vstupno-výstupného bloku. Pokiaľ nie je splnená podmienka pre alokáciu záložného vstupno-výstupného bloku, alokácia záložných riadkov a stĺpcov sa vykonáva použitím algoritmu väčšinovej opravy (ang. repair most). [7]

Schéma BIRA je navrhnutá pre organizáciu pamäte ako je na obrázku Obr. 2.8. Jednotlivé bity slova pamäte sú zoskupené do vstupno-výstupných blokov. Ak je šírka slova (w) pamäte 8 bitov, pamäť bude obsahovať 8 vstupno-výstupných blokov. [7]

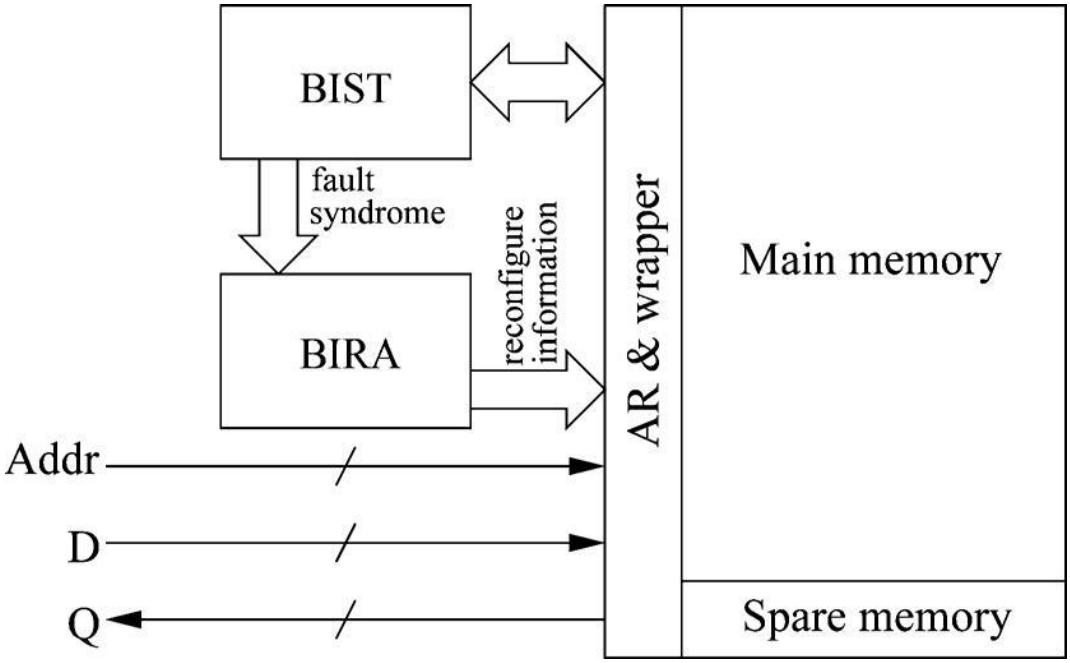


Obr. 2.8: Organizácia slovne orientovanej pamäte RAM s 3D zálohami [7]

Na obrázku Obr. 2.9 je zobrazený koncepčný diagram analyzovaného BISR obvodu, ktorý sa nijak zásadné nelíši od bežne používaných schém zapojení jednotlivých blokov.

Význam jednotlivých blokov [7]:

- *Main Memory* – predstavuje opraviteľnú pamäť RAM pozostávajúcu zo samotnej pamäte RAM obsahujúcej zálohy (*Spare Memory*) a rekonfiguračného mechanizmu (*AR & Wrapper*).
- *BIST* – predstavuje blok samočinného testovania, ktorý dokáže v pamäti RAM detegovať poruchy a odoslať zodpovedajúce informácie o poruchách do bloku *BIRA*.
- *BIRA* – predstavuje blok samočinnej analýzy opravy, ktorý na základe informácií o poruchách pamäte z bloku *BIST* vykoná alokáciu záložných prvkov.



Obr. 2.9: Blokový diagram BISR obvodu s 3D redundanciou [9]

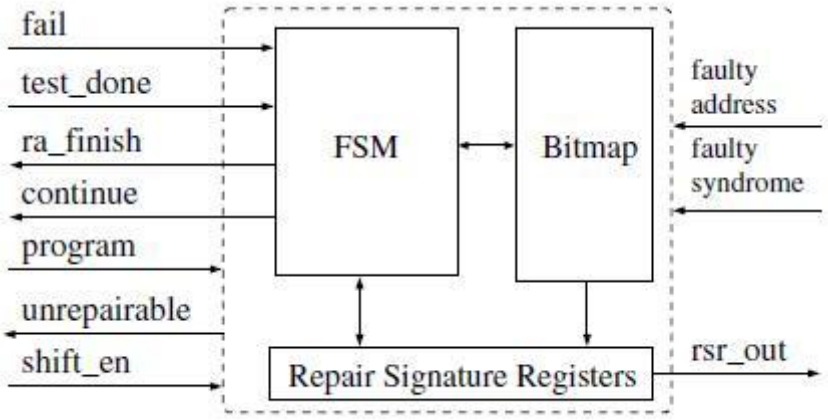
Na obrázku Obr. 2.10 je zobrazený koncepčný diagram BIRA obvodu. Význam jednotlivých blokov [7]:

- *FSM* – konečný stavový automat, ktorý realizuje algoritmus analýzy opravy.
- *Bitmap* – lokálna bitmapa pre uchovávanie informácií o poruchových miestach pamäte.
- *Repair Signature Registers (RSR)* – registre pre uchovávanie opravených adries.

Význam jednotlivých signálov [7]:

- *fail* – signál z BIST jednotky informujúci o detekcii poruchy,
- *test_done* – signál informujúci o ukončení testovania,
- *ra_finish* – signál informujúci o ukončení testovania a analýzy opravy,
- *continue* – signál pre jednotku *BIST* pre obnovenie testovania,
- *program* – pomocou tohto signálu je možné meniť prahovú hodnotu pre alokáciu záložného vstupno-výstupného pamäťového bloku,
- *unrepairable* – signál je aktivovaný, ak je pamäť neopraviteľná,
- *shift_en* – signál pre odoslanie informácií z *RSR*,
- *faulty address* – signál nesúci adresu detegovanej poruchy,
- *faulty syndrome* – signál nesúci syndróm detegovanej poruchy,

- *rsr_out* – signál nesúci informácie o opravených adresách z *RSR* do adresného rekonfigurátora.



Obr. 2.10: Blokový diagram BIRA obvodu s 3D redundanciou [7]

Princíp fungovania

BIST jednotka počas testovania ukladá informácie o detegovaných poruchách pamäte do lokálnej bitmapy. Algoritmus BIRA sa vykoná, ak sa počas testovania lokálna bitmapa naplní alebo je testovanie ukončené a bitmapa nie je prázdna. Algoritmus BIRA pri alokácii záložných prvkov najprv skontroluje, či je k dispozícii voľný záložný vstupno-výstupný blok. Môžu nastať nasledovné prípady [7]:

- Je k dispozícii voľný záložný vstupno-výstupný blok a počet porúch v niektorom poruchovom vstupno-výstupnom bloku pamäte dosahuje (alebo presahuje) prahovú hodnotu pre alokáciu záložného vstupno-výstupného bloku. V tomto prípade je alokovaný záložný vstupno-výstupný blok pre opravu poruchového vstupno-výstupného bloku pamäte.
- Je k dispozícii voľný záložný vstupno-výstupný blok, ale počet porúch v žiadnom poruchovom bloku pamäte nedosahuje prahovú hodnotu pre alokáciu záložného vstupno-výstupného bloku. V tomto prípade algoritmus BIRA alokuje záložný riadok, resp. stĺpec na opravu niektorých porúch použitím algoritmu väčšinovej opravy (ang. repair most).
- Ak nie je k dispozícii voľný záložný vstupno-výstupný blok, alokujú sa záložné prvky (riadok a stĺpec) použitím algoritmu väčšinovej opravy (ang. repair most).

Pokiaľ sa nepodarí pokryť detegované poruchy použitím dostupných záloh, pamäť je označená ako neopraviteľná.

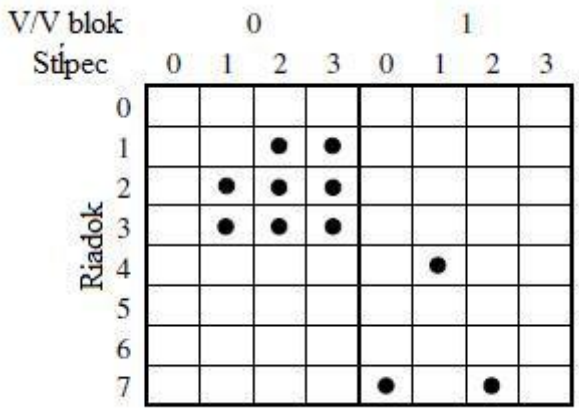
Príklad

Na obrázku Obr. 2.11 je zobrazený príklad poruchovej pamäte. Pamäť má rozmery 8 x 4 x 2 (8 riadkov, 4 stĺpce, slovo o šírke 2 bity) a je rozdelená na 2 vstupno-výstupné bloky (2-bitové slovo). K dispozícii sú záložné prvky 1 riadok, 1 stĺpec a 1 vstupno-výstupný blok. Predpokladaná prahová hodnota pre alokáciu záložného vstupno-výstupného bloku je 3. Pre uchovávanie informácií o poruchách pamäte je použitá lokálna bitmapa o rozmeroch 3x3. Táto bitmapa je zobrazená na obrázku Obr. 2.12, obsahuje registre [7]:

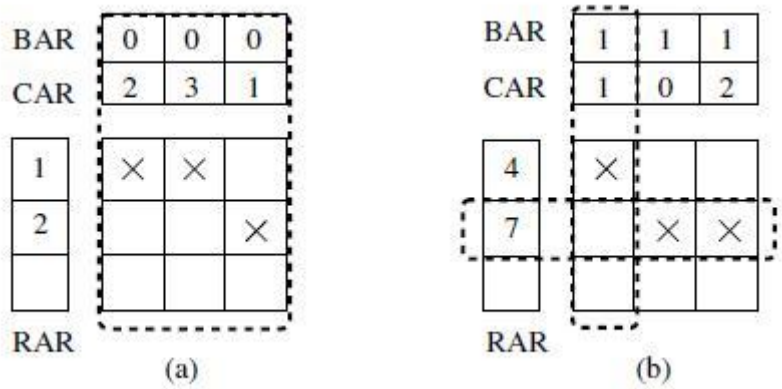
- RAR (row address register) – register obsahujúci riadkovú adresu poruchy,
- CAR (column address register) – register obsahujúci stĺpcovú adresu poruchy,
- BAR (bit address register) – register identifikujúci konkrétny poruchový bit v slove, pri danej architektúre to predstavuje číslo vstupno-výstupného bloku.

Počas testovania pamäte jednotkou BIST sú prvé 3 detegované poruchy zaznamenané v bitmape. V tomto okamihu (obrázok Obr. 2.12 a) je bitmapa plná [7], testovanie je pozastavené a vykoná sa analýza opravy za účelom opravy niektorých porúch a uvoľnenia miesta v bitmape pre zaznamenanie ďalších porúch. Nakoľko všetky poruchy majú bitovú adresu 0, je splnená prahová podmienka pre alokáciu záložného vstupno-výstupného bloku a ten je použitú na opravu detegovaných porúch. Bitmapa je po alokácii prázdna a pokračuje sa v testovaní pamäte. Chyby vo vstupno-výstupnom bloku 0 už boli pokryté práve vykonanou opravou, nie sú teda zaznamenávané do bitmapy. [7]

Bitmapa sa opäť naplní až po detekcii všetkých 3-och chýb vo vstupno-výstupnom bloku číslo 1 (obrázok Obr. 2.12 b). V tomto okamihu už nie je k dispozícii záložný vstupno-výstupný blok, iba riadok a stĺpec. BIRA jednotka vykoná analýzu opravy algoritmom väčšinovej opravy (and. repair most), ktorej výsledkom je alokácia záložného riadku na pokrytie porúch v riadku číslo 7 a alokácia záložného stĺpca na pokrytie poruchy v stĺpci číslo 1. Oprava pamäte je v tomto prípade úspešná. [7]



Obr. 2.11: Schéma pamäte s vyznačenými poruchami [7]



Obr. 2.12: Bitmapa s informáciami o detegovaných poruchách [7]. (a), (b) kroky algoritmu

2.7.3 BIRA s programovateľnými zálohami

Všeobecne

Analyzované riešenie predstavuje algoritmus opravy slovne orientovaných pamätí RAM založený na používaní konfigurovateľných záložných prvkov. Vychádza z algoritmu s identifikáciou vodiacich prvkov (ang. essential spare pivoting) s flexibilnou architektúrou záloh, kedy každý záložný prvok môže byť nakonfigurovaný ako riadok, stĺpec alebo pravouholník (obdĺžnik, štvorec) podľa potreby. Pre vykonanie algoritmu opravy pamäte postačuje jedno zbehnutie testu pamäte, a v prípade bezporuchovej pamäte sa čas testu nepredlžuje. [8]

Na obrázku Obr. 2.13 je zobrazený koncepčný diagram analyzovaného BISR obvodu.

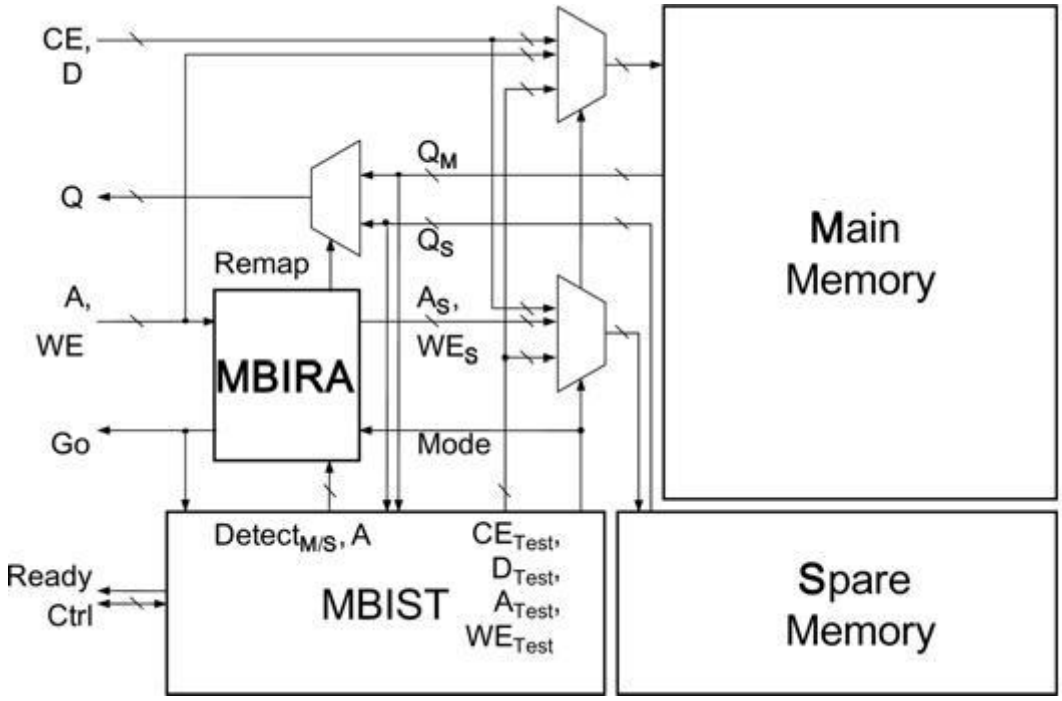
Popis jednotlivých signálov [8]:

- *CE* (Chip-enable) – aktivuje danú pamäť, vykonanie operácie čítania z pamäte,
- *D* (data-in) – vstupné údaje do pamäte,
- *Q* (data-out) – výstupné údaje z pamäte,
- *A* (address) – adresa,
- *WE* (write-enable) – signál pre vykonanie zápisu do pamäte,
- *Go* – ak je aktívny, pamäť je v poriadku, v opačnom prípade je pamäť poruchová a neopraviteľná,
- *Ready* – signalizuje, že sa ukončil test pamäte,
- *Ctrl* (control pins) – piny pre pripojenie k napájaniu,
- *Mode* – ak je aktívny, prebieha testovanie pamäte.

Ako je vidieť, všetky vstupy do pamäte sú multiplexované podľa toho, či pamäť pracuje vo funkčnom režime (*CE*, *D*, *A*, *WE*) alebo testovacom režime (CE_{Test}, D_{Test}, A_{Test}, WE_{Test}). [8]

Význam jednotlivých blokov [8]:

- *Main Memory* – predstavuje opraviteľnú pamäť RAM pozostávajúcu zo samotnej pamäte RAM a záloh (*Spare Memory*).
- *MBIST* – predstavuje blok samočinného testovania, ktorý dokáže v pamäti RAM detegovať poruchy a odoslať zodpovedajúce informácie o poruchách do bloku *MBIRA*.
- *MBIRA* – predstavuje blok samočinnej analýzy opravy, ktorý na základe informácií o poruchách pamäte z bloku *BIST* vykoná alokáciu záložných prvkov. Tento blok vykonáva aj funkciu rekonfiguračného mechanizmu - premapovanie adries.



Obr. 2.13: Blokový diagram BISR obvodu s programovateľnými zálohami [8]

Princíp fungovania

Blok *MBIST* je ovládaný testerom a vykonáva testovanie a diagnostiku hlavnej a záložnej pamäte. Po každom resete alebo nábehu napájania táto jednotka vykoná test March-CW. Vždy sa najprv testuje záložná pamäť, až potom hlavná pamäť. Je to z toho dôvodu, aby jednotka *MBIRA* mala záznam o tom, koľko má použiteľných bezporuchových záloh. [8]

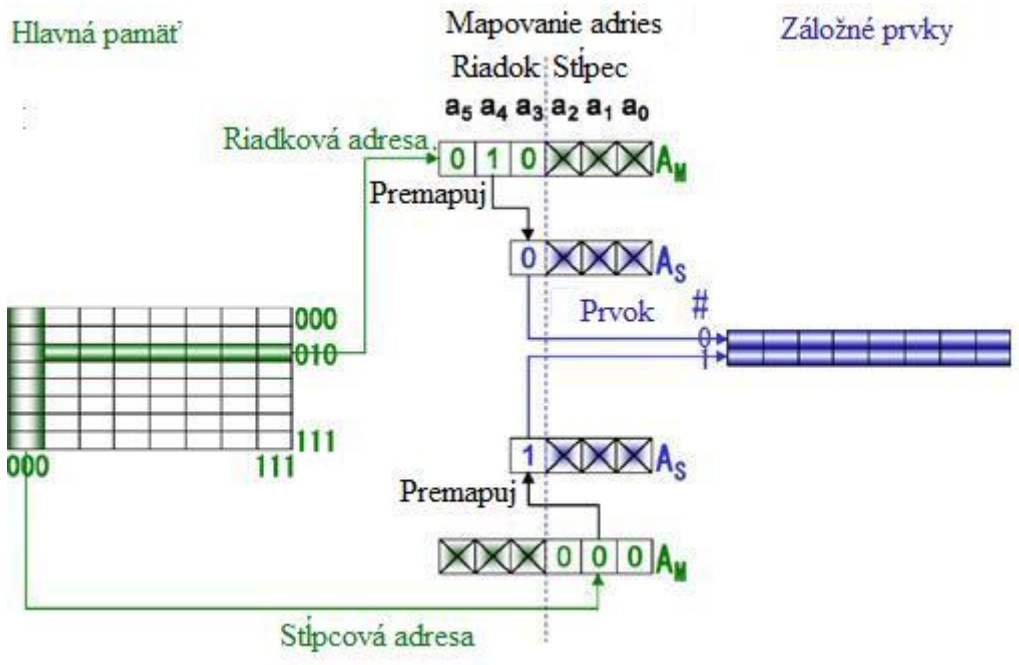
Blok *MBIRA* si v testovacom režime najprv zaznamená nepoužiteľné prvky záložnej pamäte a následne vykonáva alokáciu bezporuchových záložných prvkov podľa výsledkov testovania. Testovanie pamäte a alokácia záložných prvkov prebiehajú súčasne. [8]

Blok *MBIRA* vykonáva aj adresnú rekonfiguráciu. Vo funkčnom režime vykonáva premapovanie poruchových adries hlavnej pamäte na adresy záložnej pamäte podľa toho, ako boli pridelené záložné prvky počas vykonania opravy. Preto má záložná pamäť vstupné signály WE_S a A_S ovládané blokom *MBIRA*. Blok *MBIRA* takisto ovláda multiplexor pre výstup údajov (buď z hlavnej pamäte – Q, alebo záložnej pamäte – Q_S). [8]

Príklad

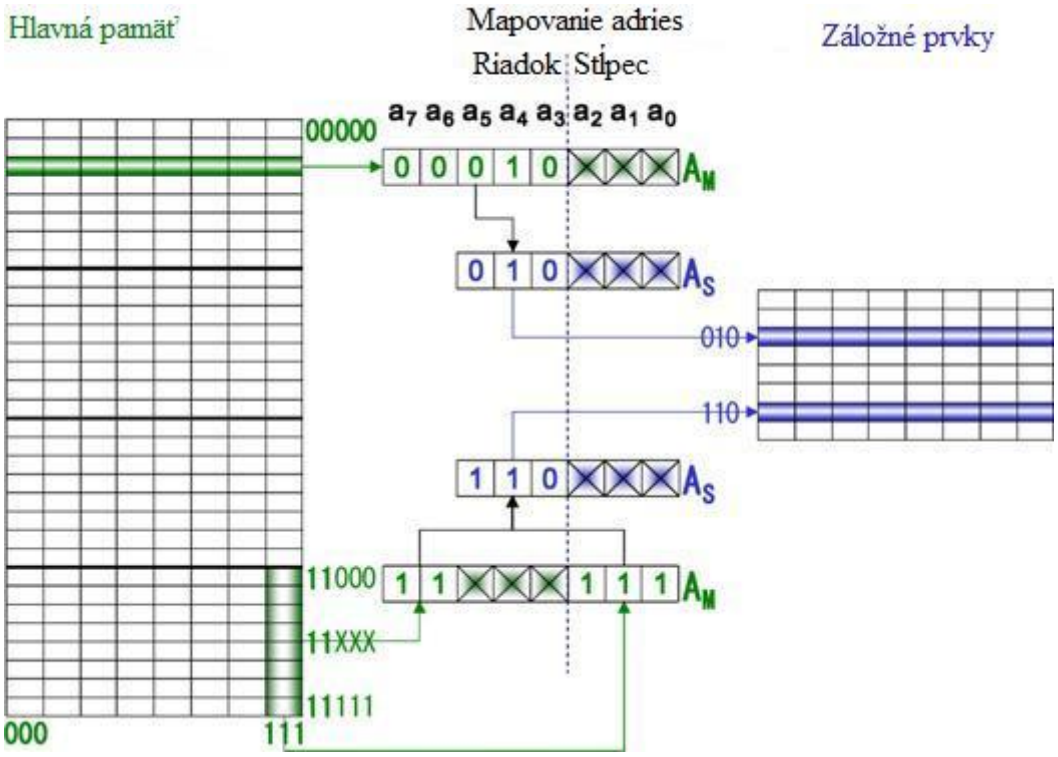
Na obrázku Obr. 2.14 je zobrazená modelová situácia, kedy riadková a stĺpcová adresa slova pamäte majú rovnaký počet bitov. Záložné prvky sa dajú teda nakonfigurovať ako

záloha za riadok alebo za stĺpec. Poruchový riadok hlavnej pamäte s riadkovou adresou 010 sa premapuje na záložný prvok s adresou 0 (v záložnej pamäti). Pri preklade adresy sa preloží len prvá polovica adresy – adresa riadka na hodnotu 0, čím sa adresuje záložný prvok. Zvyšné 3 bity (adresa stĺpca) ostávajú ponechané pre adresáciu v záložnej pamäti. Obdobne sa vykoná premapovanie aj pre chybný stĺpec hlavnej pamäte s adresou 000. [8]



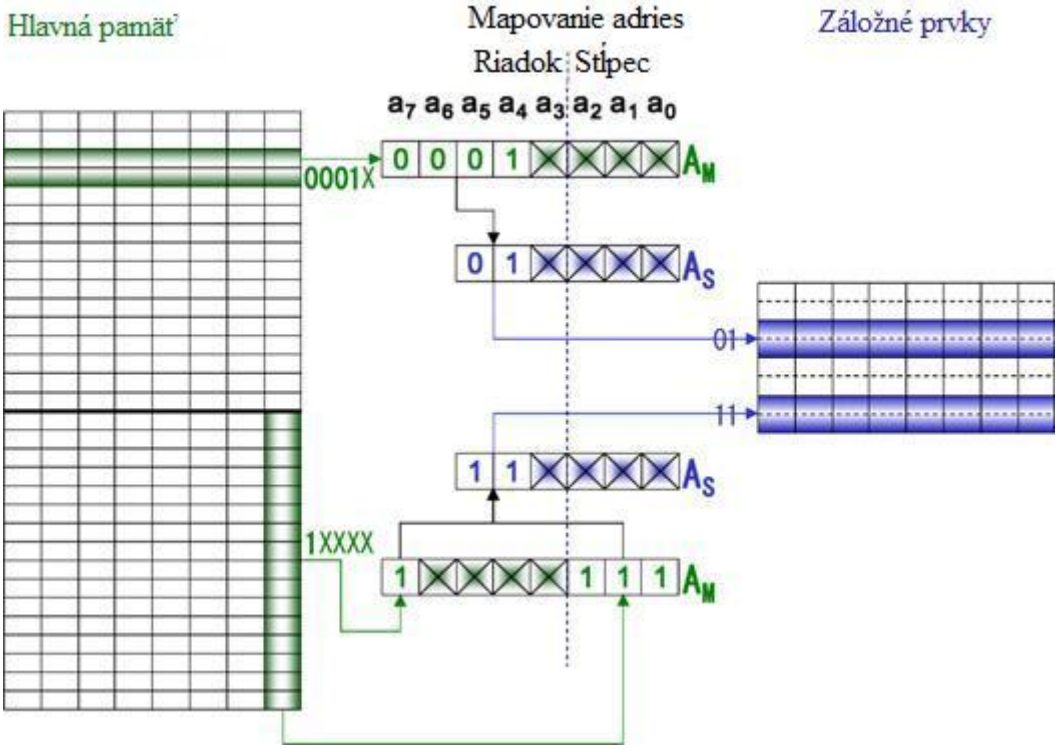
Obr. 2.14: Modelová situácia premapovania adries č. 1 [8]

Na obrázku Obr. 2.15 je zobrazená modelová situácia, kedy stĺpcová adresa slova pamäte je dlhšia ako riadková adresa. Záložné prvky sa dajú nakonfigurovať ako záloha za hociktorý riadok pamäte alebo záloha za stĺpec jedného zo 4-och segmentov pamäte. Poruchový riadok s riadkovou adresou 00010 sa premapuje na záložný prvok s adresou 010 (v záložnej pamäti), pričom sa prekladajú všetky bity riadkovej adresy. Poruchový stĺpec štvrtého segmentu hlavnej pamäte so stĺpcovou adresou 111 a riadkovými adresami v rozsahu 11000-11111 sa premapuje na záložný prvok s adresou 110 (v záložnej pamäti). Pri preklade adresy sa preložia prvé 2 bity riadkovej adresy (ktoré sú spoločné pre všetky slová v tomto stĺpci) a všetky bity stĺpcovej adresy na hodnotu 110. Zvyšné 3 bity sa používajú na adresovanie konkrétneho slova v rámci záložného prvku. [8]



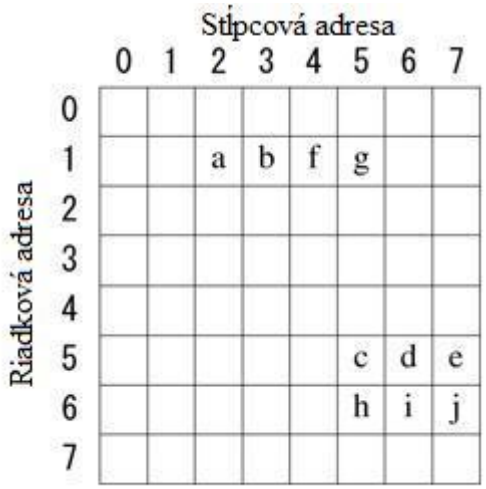
Obr. 2.15: Modelová situácia premapovania adries č. 2 [8]

Využitím rovnakých princípov sa vykonáva premapovanie adries aj pri modelovej situácii na obrázku Obr. 2.16, kedy sa záložné prvky zdvojujú pre opravu napr. 2-och riadkov hlavnej pamäte.



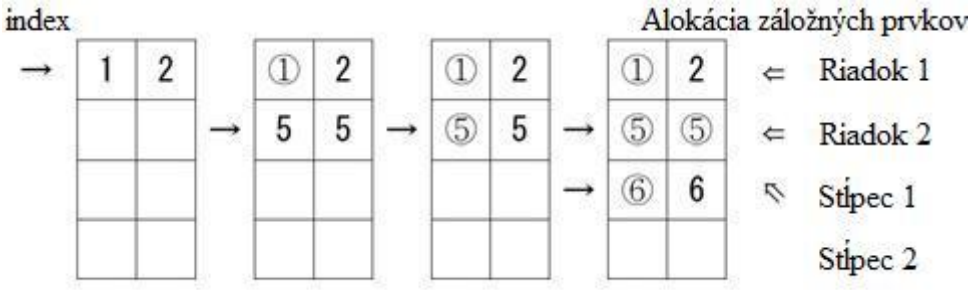
Obr. 2.16: Modelová situácia premapovania adries č. 3 [8]

Na obrázku Obr. 2.17 je zobrazený príklad poruchovej bitmapy pre ilustráciu rozšíreného algoritmu s identifikáciou vodiacich prvkov a jeho porovnanie s pôvodným algoritmom s identifikáciou vodiacich prvkov algoritmom. Poruchy sú detegované v abecednom poradí.



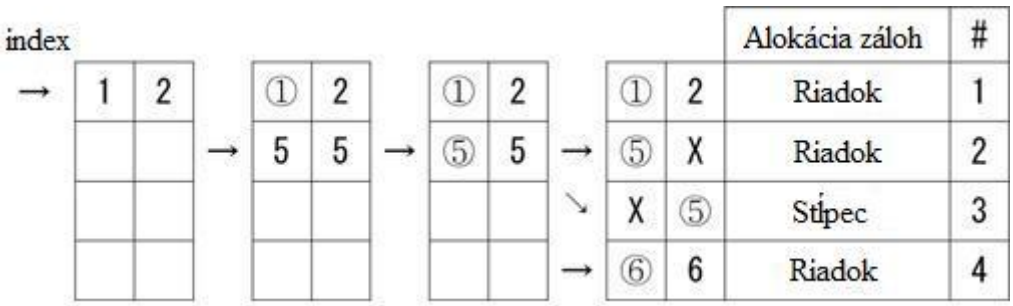
Obr. 2.17: Poruchová mapa pamäte [8]

Na obrázku Obr. 2.18 je zobrazený príklad priebehu algoritmu s identifikáciou vodiacich prvkov s prahom hodnoty 2, kedy sú k dispozícii 2 záložné riadky a 2 záložné stĺpce. Po detegovaní poruchy a sa zapíšu do pamäte adresy riadka a stĺpca daného slova – 1 a 2. Následne sa deteguje porucha b, ktorá je druhá na tom istom riadku, preto sa na opravu použije prvý záložný riadok, čo je naznačené zakrúžkovaním na obrázku. Po detegovaní poruchy c sa zapíšu adresy slova do pamäte. Po detegovaní poruchy d, ktorá je opäť druhá na tom istom riadku, sa na opravu použije druhý záložný riadok. Poruchy e, f, g sú už pokryté, preto sa obsah pamäte nemení. Po detegovaní poruchy h, ktorá je druhá v tom istom stĺpci ako porucha c, sa na opravu použije prvý záložný stĺpec. Po detekcii poruchy i sa zapíšu jej adresy do pamäte. Po detekcii poruchy j, ktorá je na tom istom riadku ako porucha i, sa snaží algoritmus vykonať opravu riadkom. Nakoľko všetky záložné riadky už boli použité, oprava zlyhá. Môže použiť len zvyšný záložný stĺpec, čím pokryje poruchu i, ale porucha j ostane nepokrytá. Oprava nie je úspešná. [8]



Obr. 2.18: Príklad priebehu algoritmu s identifikáciou vodiacich prvkov [8]

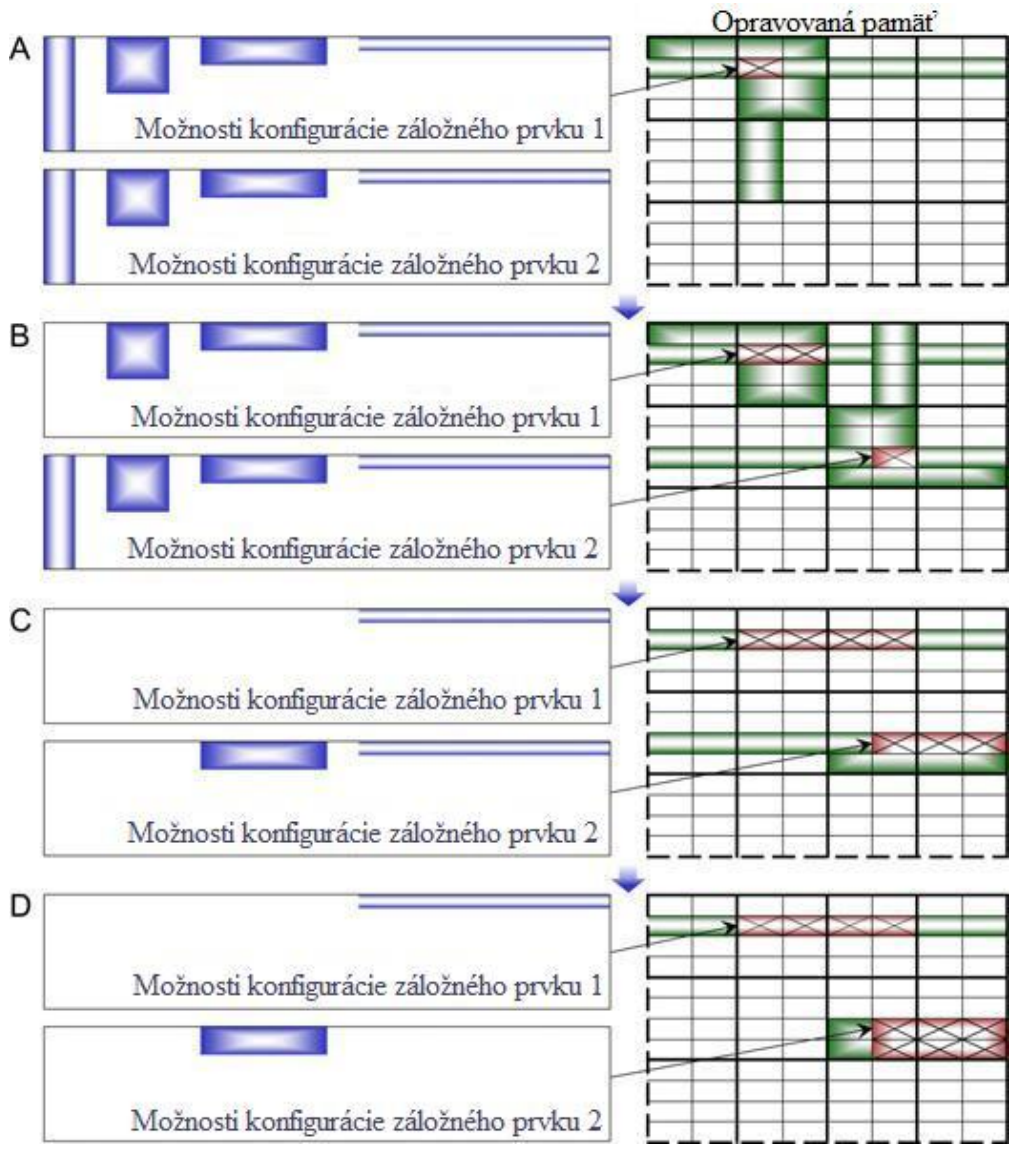
Na obrázku Obr. 2.19 je zobrazený príklad priebehu rozšíreného algoritmu s identifikáciou vodiacich prvkov s prahom hodnoty 2, kedy sú použité 4 záložné prvky, ktoré môžu byť nakonfigurované ako záložný riadok alebo záložný stĺpec. Priebeh algoritmu je prakticky rovnaký až po detekciu poruchy j, kedy sa záložný prvok nakonfiguruje ako záložný riadok a oprava prebehne úspešne. Ak sa v tom istom riadku pamäte pre vykonávanie algoritmu s identifikáciou vodiacich prvkov nachádza adresa, ktorej riadok aj stĺpec boli opravené, rozdeľuje sa tento záznam na 2 riadky s vložením hodnoty X, čo znamená, že na nej nezáleží. Predchádza sa tak konfliktom pri konfigurácii záloh. Keďže pamäť pre algoritmus s identifikáciou vodiacich prvkov má toľko riadkov pre zaznamenanie poruchových adries, koľko má záložných prvkov, nepredstavuje tento krok žiadne obmedzenie alebo nežiadúce chovanie algoritmu. [8]



Obr. 2.19: Príklad priebehu rozšíreného algoritmu s identifikáciou vodiacich prvkov [8]

Na obrázku Obr. 2.20 je zobrazený príklad postupu algoritmu pre alokáciu konfigurovateľných záloh, ak je algoritmus schopný konfigurovať zálohy nielen ako riadky a stĺpce, ale aj ako obdĺžnik. Ak sa použije takýto postup pri rozšírenom algoritme s identifikáciou vodiacich prvkov, opäť stúpne úspešnosť opravy, pretože algoritmus dokáže zálohy používať ešte efektívnejšie. Pri danej modelovej situácii sú k dispozícii 2 konfigurovateľné zálohy. Pamäť je pre zjednodušenie bitovo orientovaná o veľkosti 12 x 8 bitov. Každý záložný prvok má veľkosť 8 bitov. Možno ho nakonfigurovať ako 8-bitový

stĺpec, obdĺžnik 4 x 2 bity, obdĺžnik 2 x 4 bity alebo 8-bitový riadok. Na obrázku červené políčko predstavuje detegovanú poruchu a zelené políčka predstavujú aktuálne možnosti pokrytia poruchy / porúch so záložnými prvkami. V kroku A je detegovaná prvá porucha a na jej opravu je možné použiť každý typ konfigurácie záložného prvku. V kroku B sú detegované ďalšie 2 poruchy, z možností konfigurácie prvého záložného prvku vypadáva stĺpec, pretože by dané poruchy nepokryl. V kroku C sú detegované ďalšie 4 poruchy, z možností konfigurácie prvého záložného prvku ostal už len riadok, z možností konfigurácie druhého záložného prvku vypadol stĺpec a obdĺžnik 4 x 2. V kroku D sú detegované ďalšie 3 poruchy a z možností konfigurácie druhého záložného prvku ostal už len obdĺžnik 2 x 4. V momente, keď ostane len jeden typ konfigurácie záložného prvku pre pokrytie detegovaných porúch, vykoná sa oprava. Tým pádom sa záložné prvky využijú najefektívnejšie a neplytvá sa nimi. [8]



Obr. 2.20: Príklad priebehu algoritmu pre alokáciu konfigurovateľných záloh [8]

2.7.4 BIRA s identifikáciou poruchových vzorov

Všeobecne

Analyzované riešenie predstavuje algoritmus opravy slovne orientovaných pamätí RAM založený na identifikácii tzv. poruchového vzoru. Poruchový vzor predstavuje výskyt porúch v pamäti s určitým rozmiestnením. V tomto riešení sú identifikované 3 typy poruchových vzorov [9]:

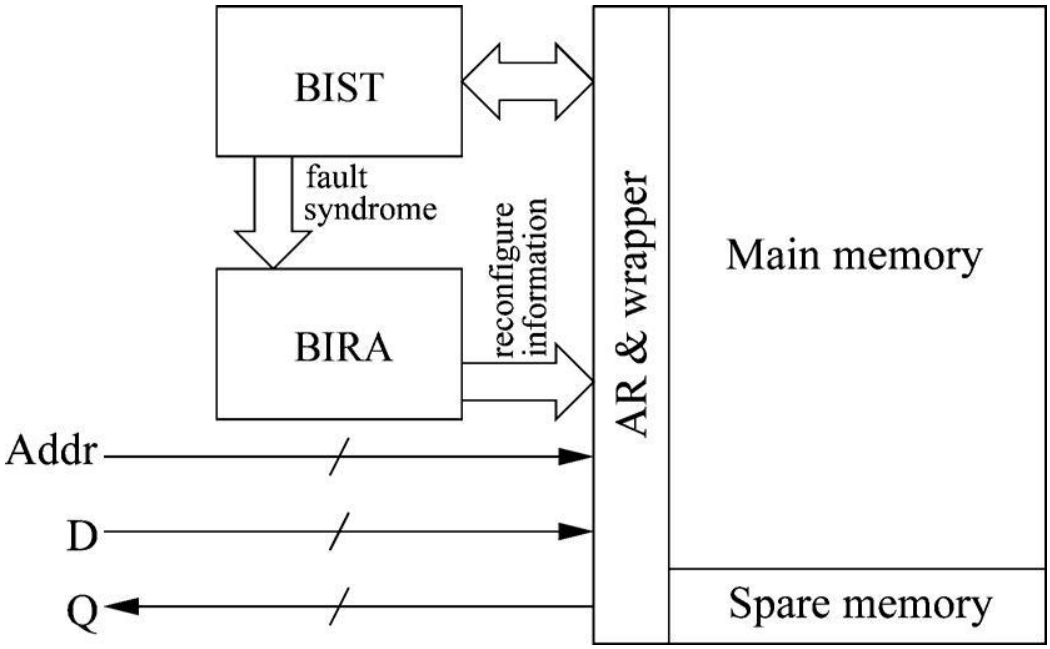
- poruchový riadok - nadväzujúce poruchy v rovnakom riadku,
- poruchový stĺpec – nadväzujúce poruchy v rovnakom stĺpci,
- poruchové slovo – všetky ostatné poruchy (ortogonálne poruchy).

Pri využití tejto metodiky musia prejsť zodpovedajúcimi úpravami BIST jednotka aj BIRA jednotka. Prispôsobenie BIST jednotky predstavuje implementáciu metodiky detekcie poruchových vzorov v pamäti a spôsob odovzdávania informácií o detegovaných poruchách do BIRA jednotky. BIRA jednotka musí vedieť spracovať informácie daného formátu a vykonať príslušnú alokáciu záložných prvkov. Pre vykonanie algoritmu opravy pamäte postačuje jedno zbehnutie testu pamäte, a v prípade bezporuchovej pamäte sa čas testu nepredlžuje. Algoritmus analýzy opravy BIRA pozostáva z dvoch fáz – fáza nevyhnutnej opravy (ang. must-repair) a finálna fáza. Fáza nevyhnutnej opravy pri tomto algoritme predstavuje samotnú schému identifikácie poruchových vzorov. Finálna fáza predstavuje riešenie opravy ortogonálnych chýb. Spomínané rozdelenie na fázy je len logické, v skutočnosti sa obidve fázy vykonávajú súčasne v závislosti od identifikovaného poruchového vzoru. Pri alokácii záložných prkov sa neukladajú informácie do lokálnej bitmapy alebo iného pamäťového miesta, algoritmus BIRA vykonáva opravu ihneď po prijatí informácie o detegovaných poruchách. Použité záložné prvky sú záložný riadok, záložný stĺpec a záložné slovo (rozdelenie záložného riadku na jednotlivé slová a možnosť alokácie samotného záložného slova). [9]

Na obrázku Obr. 2.21 je zobrazený koncepčný diagram analyzovaného BISR obvodu, ktorý sa nijak zásadné nelíši od bežne používaných schém zapojení jednotlivých blokov. Význam jednotlivých blokov [9]:

- *Main Memory* – predstavuje opraviteľnú pamäť RAM pozostávajúcu zo samotnej pamäte RAM obsahujúcej zálohy (*Spare Memory*) a rekonfiguračného mechanizmu (*AR & Wrapper*).

- *BIST* – predstavuje blok samočinného testovania, ktorý dokáže v pamäti RAM detegovať poruchy a odoslať zodpovedajúce informácie o poruchách do bloku *BIRA*.
- *BIRA* – predstavuje blok samočinnej analýzy opravy, ktorý na základe informácií o poruchách pamäte z bloku *BIST* vykoná alokáciu záložných prvkov.



Obr. 2.21: Blokový diagram BISR obvodu s identifikáciou poruchových vzorov [9]

Na obrázku Obr. 2.22 je zobrazený formát poruchových syndrómov (informácia, ktorú blok BIST odovzdáva bloku BIRA) pre používané poruchové vzory a takisto pôvodný formát poruchového syndrómu, ktorý pozostáva z troch polí [9]:

- operácia,
- adresa,
- poruchový syndróm slova.

Obr. 2.22: Formát poruchových syndrómov [9]

V nasledovných formátoch poruchových syndrómov sa nachádza pole identifikátor syndrómu, ktoré slúži na rozlíšenie medzi jednotlivými poruchovými vzormi (00 – poruchové slovo, 11 – poruchový riadok, 01 – poruchový stĺpec). Komprimácia syndrómu sa vykonáva Huffmanovým kódom. [9]

Poruchový syndróm používaný pre vzor poruchové slovo pozostáva zo štyroch polí [9]:

- identifikátor syndrómu,
- operácia,
- adresa,
- skomprimovaný poruchový syndróm slova.

Poruchový syndróm používaný pre vzor poruchový riadok pozostáva zo štyroch polí [9]:

- identifikátor syndrómu,
- operácia,
- adresa,
- stĺpcová adresa posledného poruchového slova.

Poruchový syndróm používaný pre vzor poruchový stĺpec pozostáva zo štyroch polí [9]:

- identifikátor syndrómu,
- operácia,
- adresa,
- riadková adresa posledného poruchového slova,
- skomprimovaný poruchový syndróm slova.

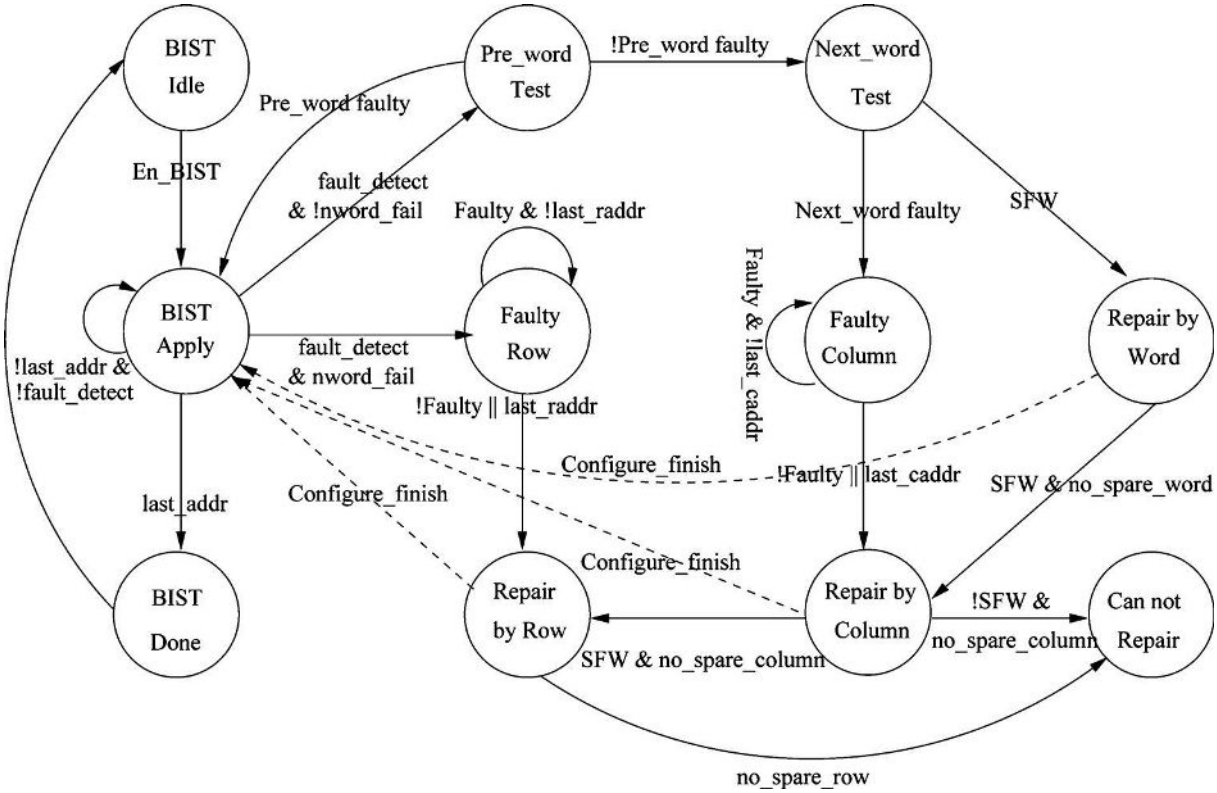
Princíp fungovania

Stavový diagram je zobrazený na obrázku Obr. 2.23. Počiatočný stav je *BIST Idle*, následne po spustení BIST jednotky sa prechádza do stavu *BIST Apply*, v ktorom prebieha testovanie pamäte, až kým neprebehne test na celej pamäti. Nasleduje popis jednotlivých možností prechodov medzi stavmi [9]:

- Prvý prípad predstavuje správanie v prípade detekcie vzoru poruchový riadok. Ak je detegovaná porucha v pamäti, testuje sa ďalšie slovo v tom istom riadku. Pokiaľ je poruchové, jednotka prejde do stavu *Faulty Row* a testujú sa ďalšie slová pamäte v tom istom riadku, až kým sa nenarazí na bezporuchové slovo alebo sa nedôjde na

koniec riadku. Následne sa detegované poruchy opravujú záložným riadkom (stav *Repair by Row*) a pokračuje sa v testovaní pamäte (stav *BIST Apply*). Pokiaľ už nie sú k dispozícii voľné záložné riadky, pamäť je označená ako neopraviteľná (stav *Can not Repair*).

- V druhom prípade sa jedná o detekciu vzoru poruchový stĺpec. Ak je detegovaná porucha v pamäti a ďalšie slovo v tom istom riadku sa vyhodnotí ako bezporuchové, prechádza sa do stavu *Pre_word Test*. V tomto stave sa otestuje slovo pamäte v tom istom stĺpci o riadok vyššie. Ak je poruchové, ide sa naspäť do stavu *BIST Apply* a pokračuje testovanie pamäte, pretože v tom prípade už bola táto porucha pokrytá predošlým výskytom vzoru poruchový stĺpec. Ak je testované slovo bezporuchové, otestuje sa slovo pamäte v tom istom stĺpci o riadok nižšie (stav *Next_word Test*). Ak je toto slovo poruchové, jedná sa o vzor poruchový stĺpec (stav *Faulty Column*) a pokračuje sa testovaním slov v tom istom stĺpci pamäte, až kým sa nenarazí na bezporuchové slovo alebo sa nedojde na koniec stĺpca. Následne sa prejde do stavu *Repair by Column*, v ktorom je snaha opraviť detegované poruchy použitím záložného stĺpca a v prípade úspechu nasleduje návrat do stavu *BIST Apply* a pokračuje testovanie pamäte. Ak nie sú k dispozícii voľné záložné stĺpce, pamäť je označená ako neopraviteľná (stav *Can not Repair*).
- Tretí prípad nadväzuje na predošlý prípad v momente, ak je v stave *Next_word Test* testované slovo vyhodnotené ako bezporuchové. Potom sa jedná o vzor poruchové slovo. Nasleduje priamo prechod do stavu *Repair by Word*, kde je snaha opraviť detegovanú poruchu záložným slovom. Ak nie sú k dispozícii voľné záložné slová, prechádza sa do stavu *Repair by Column*, v ktorom je snaha opraviť detegovanú poruchu použitím záložného stĺpca. Ak nie sú k dispozícii voľné záložné stĺpce, prechádza sa do stavu *Repair by Word*, v ktorom je snaha opraviť detegovanú poruchu použitím záložného riadku. Ak nie sú k dispozícii ani voľné záložné riadky, pamäť je označená ako neopraviteľná (stav *Can not Repair*). V prípade úspešnej opravy v ktoromkoľvek spomenutom stave sa prechádza naspäť do stavu *BIST Apply* a pokračuje testovanie pamäte. Po ukončení testovania pamäte sa prechádza cez stav *BIST Done* do stavu *BIST Idle*, v ktorom sa čaká na spustenie nového testovania.



Obr. 2.23: Stavový diagram BISR jednotky s identifikáciou poruchových vzorov [9]

Príklad

Na obrázku Obr. 2.24 je zobrazená modelová situácia, na ktorej sa dá demonštrovať fungovanie analyzovaného algoritmu. Vyčiarkovaný štvorček predstavuje testované slovo.

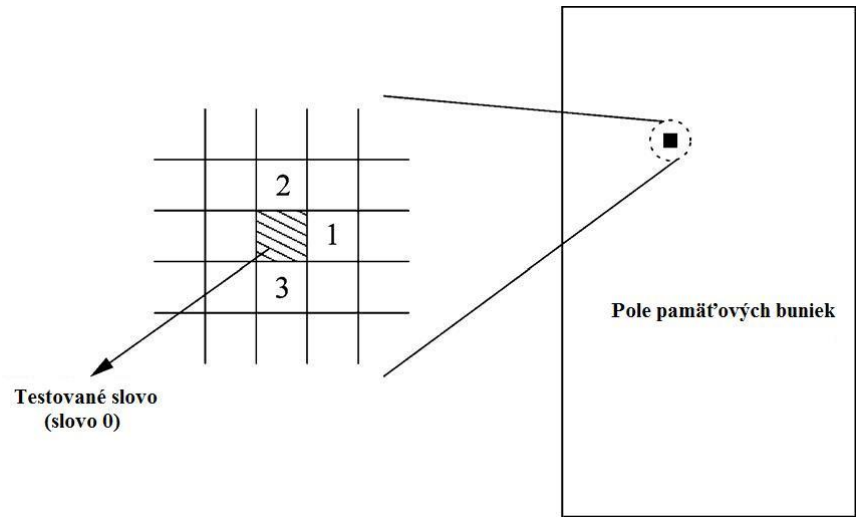
- Identifikácia vzoru poruchový riadok: ak je testované slovo pamäte poruchové, testuje sa slovo s číslom 1, pokiaľ je aj to chybné, testujú sa ďalšie slová v tom istom riadku, až kým sa nenarazí na bezporuchové slovo alebo koniec riadku. Následne BIRA jednotka obdrží poruchový syndróm pre vzor poruchový riadok a vykoná opravu záložným riadkom. [9]
- Identifikácia vzoru poruchový stĺpec: ak je testované slovo pamäte poruchové, postupuje sa nasledovne [9]:
    a. pamäťové slovo s číslom 1 je bezporuchové, môže sa vylúčiť vzor poruchové slovo,
    b. pamäťové slovo nad testovaným slovom v rovnakom stĺpci (číslo 2) je bezporuchové, v opačnom prípade bolo testované poruchové slovo pokryté predošlou identifikáciou vzoru poruchový stĺpec,

c. ak je pamäťové slovo pod testovaným slovom v rovnakom stĺpci (číslo 3) poruchové, testujú sa ďalšie slová v tom istom stĺpci, až kým sa nenarazí na bezporuchové slovo alebo koniec stĺpca.

Následne BIRA jednotka obdrží poruchový syndróm pre vzor poruchový stĺpec a vykoná opravu záložným stĺpcom. [9]

Pri tomto vzore je dôležité, že ak majú testované slová, ktoré by mohli byť identifikované ako vzor poruchový stĺpec odlišný syndróm (poruchu v odlišnom bite slova), nemôžu sa tieto slová identifikovať ako poruchový stĺpec. [9]

- Identifikácia vzoru poruchové slovo: pokiaľ je testované slovo poruchové a slová číslo 1, 2 a 3 sú bezporuchové, jedná sa o tento vzor. Následne BIRA jednotka obdrží poruchový syndróm pre vzor poruchové slovo a vykoná opravu záložným slovom (poprípade stĺpcom alebo riadkom, podľa postupu popísaného vyššie v princípe fungovania). [9]



Obr. 2.24: Modelová situácia pri testovaní algoritmom [9]

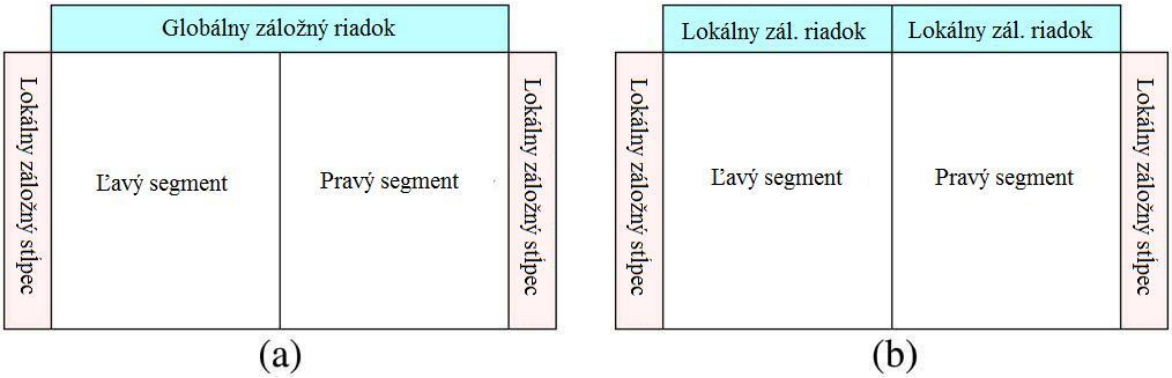
2.7.5 BIRA s 2D redundanciou a nízkou cenou

Všeobecne

Analyzované riešenie predstavuje algoritmus opravy slovne orientovaných pamätí RAM . Algoritmus analýzy opravy BIRA pozostáva z dvoch fáz – fáza nevyhnutnej opravy (ang. must-repair) a finálna fáza. Vo finálnej fáze algoritmu sa vykonáva algoritmus väčšinovej

opravy (ang. repair most). Pri alokácii záložných prkov sa vychádza z 1-D lokálnej bitmapy najmenšej možnej veľkosti vzhľadom na počet záloh. [10]

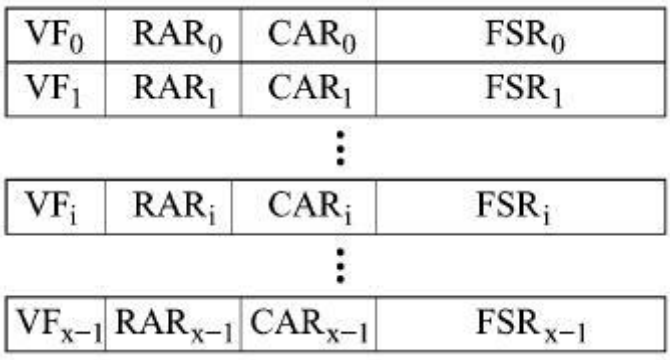
Analyzované riešenie algoritmu BIRA môže byť použité pri dvoch typoch organizácie záložných prvkov (obrázok Obr. 2.25) – buď globálne záložné riadky a lokálne záložné stĺpce (a), alebo lokálne záložné riadky a lokálne záložné stĺpce (b). Pamäť je rozdelená na 2 segmenty, pričom globálne záložné prvky môžu byť použité na opravu porúch v ktoromkoľvek segmente pamäte, lokálne záložné prvky len v danom segmente, ktorému sú určené. [10]



Obr. 2.25: Možnosti organizácie záložných prvkov [10]

Na obrázku Obr. 2.26 je zobrazená štruktúra 1-D lokálnej bitmapy. Bitmapa môže mať X záznamov a každý záznam pozostáva zo štyroch polí [10]:

- VF (valid flag) – identifikátor platnosti záznamu,
- RAR (row address register) – register riadkovej adresy,
- CAR (column address register) – register stĺpcovej adresy,
- FSR (fault syndrome register) – register pre poruchový syndróm.

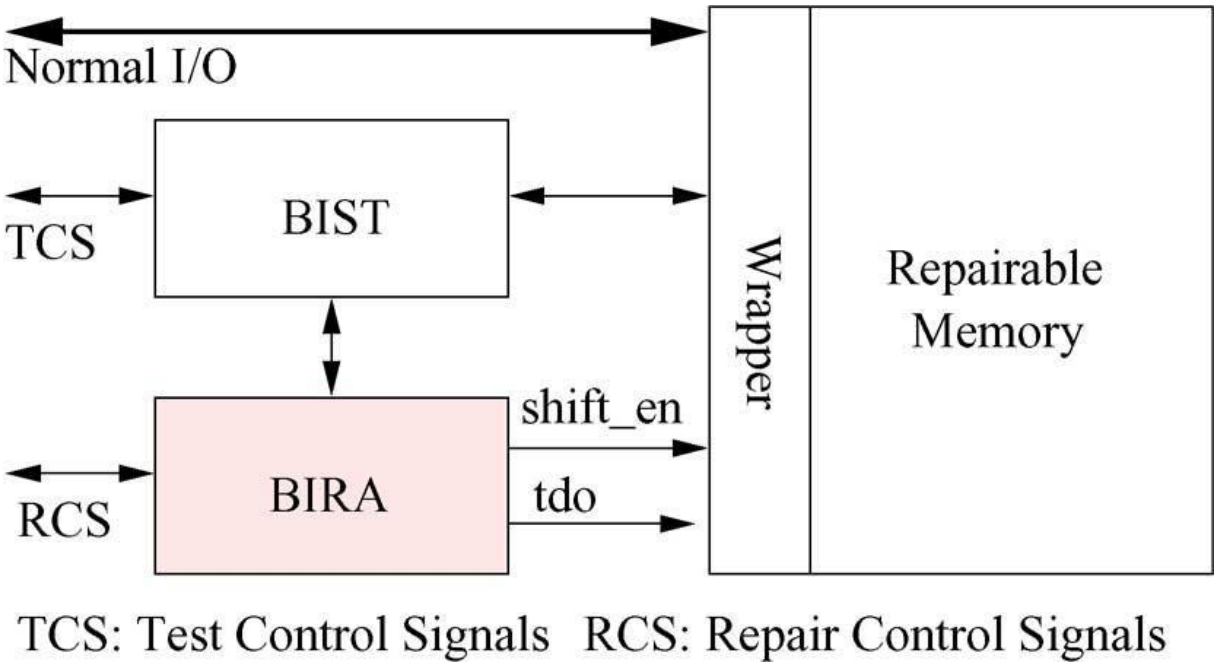


Obr. 2.26: Štruktúra 1-D lokálnej bitmapy [10]

Ak daný riadok bitmapy obsahuje platný záznam, identifikátor platnosti má hodnotu 1, registre RAR a CAR obsahujú riadkovú, resp. stĺpcovú adresu poruchového slova pamäte a register FSR obsahuje poruchový syndróm, z ktorého je možné určiť konkrétny poruchový bit slova. Počet záznamov lokálnej bitmapy sa dá obmedziť na maximum, ktoré sa rovná súčtu počtu záložných riadkov a záložných stĺpcov. [10]

Na obrázku Obr. 2.27 je zobrazený koncepčný diagram analyzovaného BISR obvodu. Význam jednotlivých blokov [10]:

- *Repairable Memory* – predstavuje opraviteľnú pamäť RAM pozostávajúcu zo samotnej pamäte RAM obsahujúcej zálohy a rekonfiguračného mechanizmu (*Wrapper*). Tento mechanizmus prepína režim pamäte medzi funkčným a testovacím a tiež vykonáva rekonfiguráciu pamäte.
- *BIST* – predstavuje blok samočinného testovania, ktorý dokáže v pamäti RAM detegovať poruchy a odoslať zodpovedajúce informácie o poruchách do bloku *BIRA*.
- *BIRA* – predstavuje blok samočinnej analýzy opravy, ktorý na základe informácií o poruchách pamäte z bloku *BIST* posiela príznak opravy do bloku *Wrapper*.



Obr. 2.27: Blokový diagram BISR obvodu s 2D redundanciou a nízkou cenou [10]

Princíp fungovania

Uvažujme pamäť rozdelenú na 2 segmenty (pravý a ľavý). BIRA jednotka vykonáva analýzu opravy v spolupráci s jednotkou BIST, ktorá deteguje poruchy v pamäti. Pri každej detekcii poruchy je BIST jednotka pozastavená a vykoná sa jedna z nasledovných troch možností [10]:

- Ak je adresa detegovanej poruchy zhodná s niektorou z adries v platných záznamoch bitmapy, aktualizuje sa hodnota registra FSR daného záznamu.
- Ak je riadková adresa detegovanej poruchy zhodná s niektorou z riadkových adries v platných záznamoch bitmapy, ale stĺpcová adresa nie je, algoritmus alokuje záložný riadok pre opravu poruchového riadku a aktualizuje obsah bitmapy. Ak v tomto momente nie sú k dispozícii voľné záložné riadky, pamäť je neopraviteľná.
- Ak riadková ani stĺpcová adresa detegovanej poruchy nie je zhodná so žiadnou riadkovou, resp. stĺpcovou adresou v platných záznamoch bitmapy, je táto adresa uložená do voľného záznamu. V prípade, že bitmapa sa pridaním záznamu naplnila, BIRA jednotka musí vykonať analýzu opravy, a tým uvoľniť miesto v bitmape pre ďalšie záznamy. Ak po pridaní záznamu bitmapa ešte nie je plná, BIST jednotka dostane signál pre obnovenie testovania.

Ak je testovanie kompletné, vykonáva sa nasledovná postupnosť krokov, až kým nie je bitmapa prázdna (oprava úspešná) alebo sa pamäť označí za neopraviteľnú [10]:

Skontroluje sa, či sú k dispozícii lokálne záložné stĺpce v oboch segmentoch.

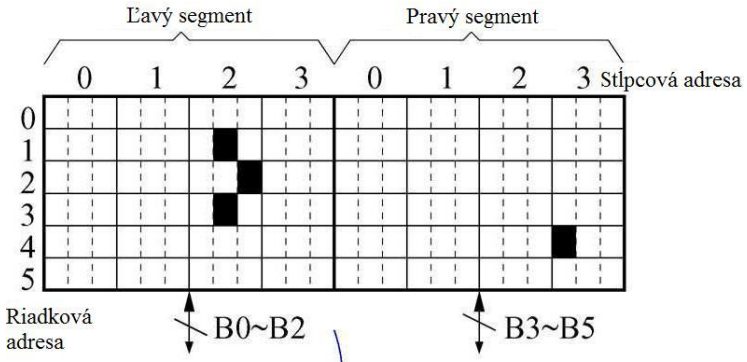
- Ak nie sú voľné záložné stĺpce v niektorom zo segmentov, skontroluje sa výskyt porúch v danom segmente. Ak sa v danom segmente vyskytujú poruchy, porovná sa počet poruchových riadkov v danom segmente s počtom voľných záložných riadkov. Ak je počet porúch menší ako počet voľných záložných riadkov, opravia sa všetky poruchy daného segmentu alokáciou záložných riadkov. Ak je počet porúch väčší ako počet voľných záložných riadkov, pamäť je neopraviteľná.
- Ak sú voľné záložné stĺpce, algoritmus hľadá čo najväčšiu skupinu záznamov, ktoré majú rovnakú stĺpcovú adresu. Ďalej v rámci tejto skupiny hľadá konkrétny bit, v ktorom je najväčší počet porúch. Týmto postupom identifikovaný stĺpec je opravený použitím záložného stĺpca.

- Ak nie sú voľné záložné stĺpce v žiadnom zo segmentov a počet poruchových riadkov je menší ako počet voľných záložných riadkov, opravia sa všetky poruchy alokáciou záložných riadkov. V opačnom prípade je pamäť neopraviteľná.

Popísaný postup sa vykoná aj v prípade potreby uvoľniť miesto pre ďalší záznam v bitmape počas testovania.

Príklad

Na obrázku Obr. 2.28 je zobrazený príklad, akým spôsobom sa lokálna bitmapa napĺňa informáciami a následne alokujú záložné prvky (1 globálny riadok, 1 lokálny stĺpec pre každý segment). Testovaná pamäť má veľkosť 6 x 4 x 6 (6 riadkov, 4 stĺpce, slovo o šírke 6 bitov). Navyše, pamäť je rozdelená na 2 segmenty – bity B0 – B2 sú z ľavého segmentu, bity B3 – B5 sú z pravého segmentu. Ako je vidieť na obrázku, bitmapa obsahuje 4 platné záznamy. Algoritmus sa nachádza v štádiu testovania pamäte a potrebuje uvoľniť miesto v bitmape pre ďalšie záznamy. Algoritmus analýzy opravy musí alokovať nejaký záložný prvok. Ešte sú voľné záložné stĺpce, takže algoritmus hľadá čo najväčšiu skupinu záznamov, ktoré majú rovnakú stĺpcovú adresu. Táto skupina je na obrázku orámovaná (rovnaká adresa stĺpca hodnoty 2). Ďalej v rámci tejto skupiny hľadá konkrétny bit, v ktorom je najväčší počet porúch. Je to B1. Tento stĺpec pamäte sa opraví záložným stĺpcom, aktualizuje sa obsah bitmapy a pokračuje sa ďalej v testovaní. [10]



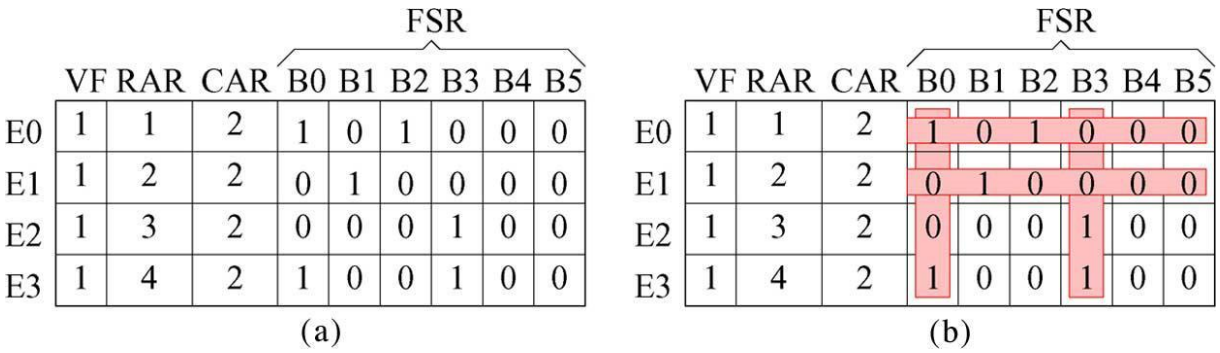
FSR

	VF	RAR	CAR	B0	B1	B2	B3	B4	B5
E0	1	1	2	0	1	0	0	0	0
E1	1	2	2	0	0	1	0	0	0
E2	1	3	2	0	1	0	0	0	0
E3	1	4	3	0	0	0	1	0	0

Obr. 2.28: Príklad priebehu algoritmu (napĺňanie a alokácia) [10]

Na obrázku Obr. 2.29 je zobrazený príklad vykonávania algoritmu po ukončení testovania s obsahom bitmapy ako je na obrázku. Organizácia pamäte je rovnaká ako v predošlom príklade. K dispozícii sú 2 globálne záložné riadky a 1 lokálny záložný stĺpec pre každý segment. Ako je vidieť na obrázku, bitmapa obsahuje 4 platné záznamy, každá porucha na inom riadku. Algoritmus analýzy opravy a alokácia záložných prvkov sa vykoná nasledovne [10]:

Záložné stĺpce sú ešte voľné, takže algoritmus hľadá čo najväčšiu skupinu záznamov, ktoré majú rovnakú stĺpcovú adresu. V tomto prípade majú všetky záznamy rovnakú stĺpcovú adresu. Ďalej v rámci tejto skupiny hľadá konkrétny bit, v ktorom je najväčší počet porúch. Sú to bity B0 a B3 so zhodným počtom porúch 2. Pri určovaní, ktorý bit sa opraví záložným stĺpcom, má prednosť bit s menšou váhou, čiže B0. Bit B0 sa opraví záložným stĺpcom v ľavom segmente pamäte. Nakoľko sa v ľavom segmente pamäte stále nachádzajú poruchy a ich počet je menší, resp. rovný ako počet voľných záložných riadkov, algoritmus opraví tieto poruchy použitím záložných riadkov. V tomto momente sú v bitmape platné záznamy E2 a E3 a je k dispozícii 1 záložný stĺpec v pravom segmente pamäte. Rovnakým postupom ako je popísané vyššie algoritmus alokuje záložný stĺpec na opravu bitu B3. Tým pádom je bitmapa prázdna a pamäť úspešne opravená. [10]



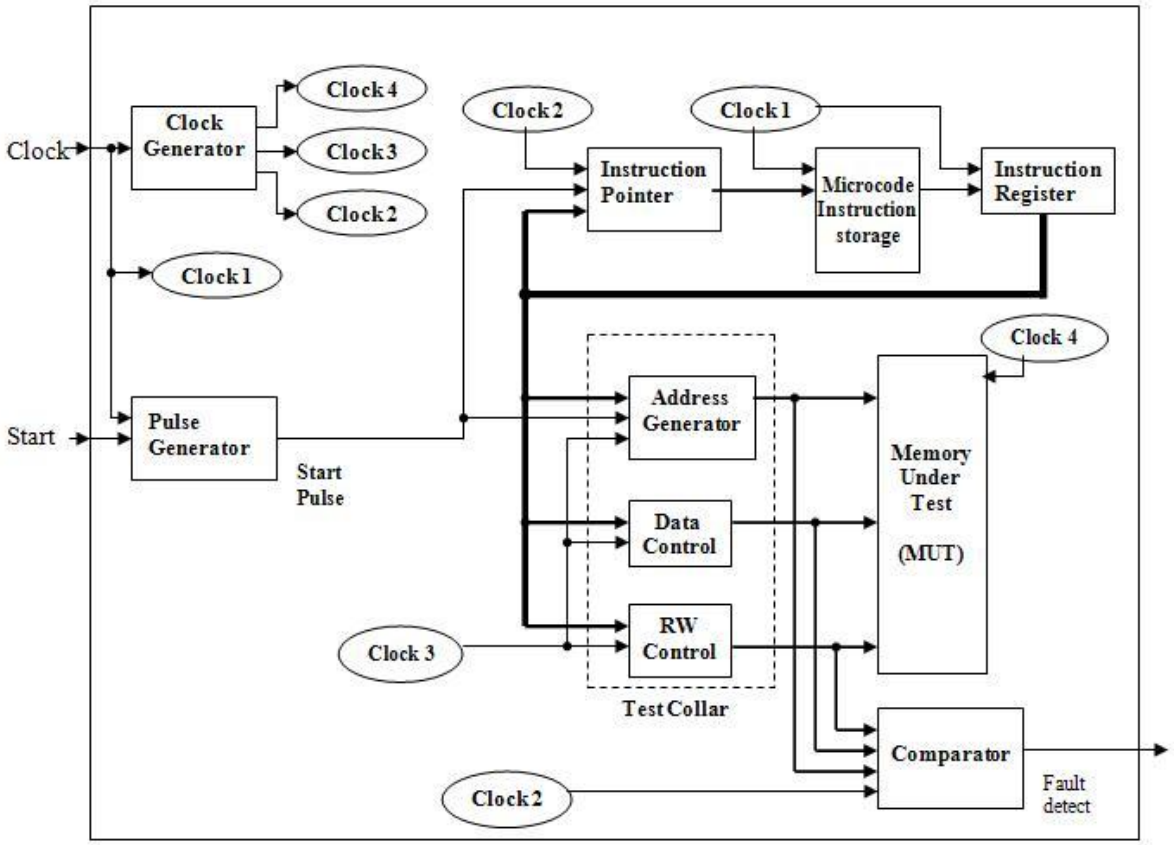
Obr. 2.29: Príklad priebehu algoritmu (po ukončení testovania) [10]

2.8 Analýza moderných prístupov k riešeniu BIST

2.8.1 BIST s neobmedzeným počtom operácii v elemente march

Analyzované riešenie predstavuje riešenie vstavaného samočinného testovania (BIST) založeného na mikrokódovaní. Hlavným prínosom tohto riešenia je podpora všetkých algoritmov typu march bez obmedzenia počtu operácií v ich elementoch. [11]

Bloková schéma analyzovaného riešenia je zobrazená na obrázku Obr. 2.30.



Obr. 2.30: Bloková schéma architektúry BIST s neobmedzeným počtom operácií v elemente [11]

Význam jednotlivých blokov [11]:

- *Clock Generator* – generátor hodín pre jednotlivé bloky architektúry.
- *Pulse Generator* – generuje impulz pre spustenie testovania pri nábežnej hrane signálu *Start*.
- *Instruction Pointer* – ukazuje na ďalšie inštrukčné slovo predstavujúce nasledujúcu operáciu testu march, ktorá sa vykoná.

- *Microcode Instruction storage* – pamäť obsahujúca celý test vo forme inštrukčných slov (mikroinštrukcií).
- *Instruction register* – obsahuje inštrukčné slovo predstavujúce operáciu testu march, ktorá sa momentálne vykonáva.
- *Address Generator* – generuje adresy pre testovanie.
- *Data Control* – generuje údaje na zápis, resp. údaje, ktoré sa očakávajú na výstupe z pamäte pri operácii čítania.
- *RW Control* – generuje signály pre ovládanie operácií zápisu a čítania z pamäte.
- *Memory Under Test* – testovaná pamäť.
- *Comparator* – porovnáva očakávanú hodnotu s prečítanou hodnotou z pamäte a informuje o detegovanej poruche zmenou hodnoty signálu *Fault detect*.

Na obrázku Obr. 2.31 je zobrazený formát inštrukčného slova (mikroinštrukcie) použitého v analyzovanej architektúre. Inštrukčné slovo reprezentuje jednu operáciu testu march.

#1	#2	#3	#4	#5	#6	#7
Valid	Fo	Io	Lo	I/D	R/W	Data

Obr. 2.31: Formát mikroinštrukcie architektúry BIST s neobmedzeným počtom operácií v elemente [11]

Význam jednotlivých polí [11]:

- *Valid* – slúži na označenie platnosti mikroinštrukcie, ak sa narazí na neplatnú mikroinštrukciu, končí sa testovanie.
- *Fo (first operation)* – označenie prvej operácie v rámci elementu march.
- *Io (in-between operation)* – označenie operácie v rámci elementu march.
- *Lo (last operation)* – označenie poslednej operácie v rámci elementu march.
- *I/D* – určuje, či má byť pamäť adresovaná vzostupne alebo zostupne.
- *R/W* – operácia čítania alebo zápisu.
- *Data* – údaj pre zápis do pamäte alebo údaj očakávaný na výstupe z pamäte.

Hodnoty bitov *Fo*, *Io* a *Lo* môžu nadobúdať tieto štyri kombinácie [11]:

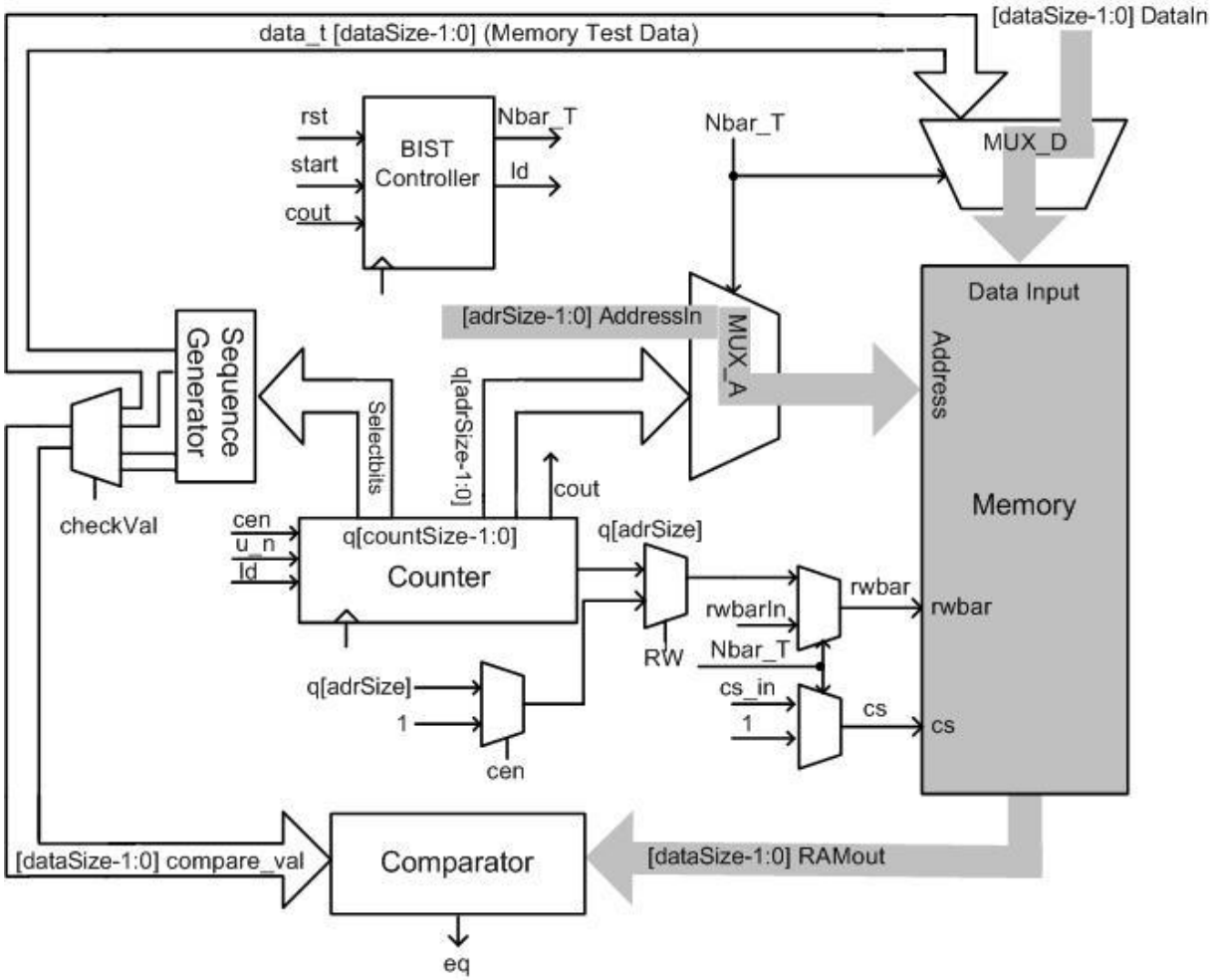
- 000 – jedná sa o element march s jednou operáciou,
- 100 – jedná sa o prvú operáciu elementu march pozostávajúceho z viacerých operácií,

- 010 – jedná sa o operáciu v rámci elementu march pozostávajúceho z viacerých operácií,
- 001 - jedná sa o poslednú operáciu elementu march pozostávajúceho z viacerých operácií.

2.8.2 BIST s testom vo forme VHDL skriptu

Analyzované riešenie predstavuje programovateľnú architektúru vstavaného samočinného testovania (BIST). Hlavným prínosom tohto riešenia je možnosť prispôsobenia architektúry testovania pamäte ľubovoľnej veľkosti pamäte a podpora algoritmov typu march a algoritmu Disturb. [12]

Bloková schéma analyzovaného riešenia je zobrazená na obrázku Obr. 2.32.



RW = (TestMethod = 0) | (TestMethod = 1)

checkVal = (TestMethod = 1) | (TestMethod = 2)

Selectbits = q[countSize-1:memSize]

Obr. 2.32: Bloková schéma architektúry BIST s testom vo forme VHDL skriptu [12]

Označenia *dataSize*, *adrSize* a *countSize* predstavujú počet bitov potrebných na reprezentáciu daných informácií, čiže určujú šírku jednotlivých zberníc podľa použitej pamäte. Multiplexory sa využívajú na prepínanie medzi zdrojmi údajov (funkčný a testovací režim) a takisto na prepínanie medzi zdrojmi údajov pri testovaní testom typu march alebo testom Disturb (nie všetky sú vždy využité). [12]

Význam jednotlivých blokov [12]:

- *BIST Controller* – riadiaca jednotka testovania.
- *Memory* – testovaná pamäť.
- *Comparator* – porovnáva očakávanú hodnotu s prečítanou hodnotou z pamäte a informuje o detegovanej poruche zmenou hodnoty signálu *eq*.
- *Counter* – počítadlo adries.
- *Sequence Generator* – generuje postupnosť operácií, obsahuje test uložený vo forme skriptu napísaného v jazyku VHDL. Ktorý test sa aplikuje, určuje signál *checkVal*.

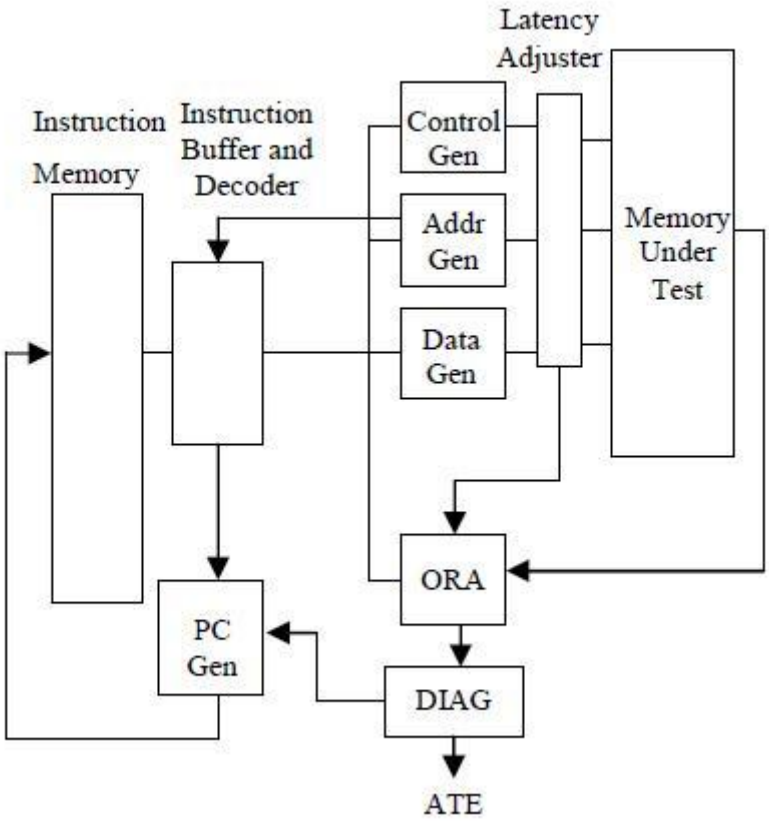
Všetky bloky architektúry sú popísané v jazyku VHDL, takisto jednotlivé testy sú uložené vo forme skriptu, resp. pseudogenerátora v jazyku VHDL. Používateľ môže tak rozšíriť pôvodne obsiahnuté testy (MATS, March C-, Disturb) o ľubovoľný test, pokiaľ je schopný daný test opísať v zodpovedajúcom formáte. Toto riešenie ale značným spôsobom uberá z jednoduchosti použitia. [12]

2.8.3 BIST s podporou viacerých typov algoritmov

Analyzované riešenie predstavuje riešenie vstavaného samočinného testovania (BIST) založeného na mikrokódovaní. Hlavným prínosom tohto riešenia je podpora rôznych algoritmov používajúcich rôzne adresovacie schémy a testovacie údaje. Medzi podporované algoritmy patria [13]:

- algoritmy march,
- algoritmy Galloping/Walking,
- algoritmy pre testovanie adresného dekódera,
- Butterfly algoritmy,
- Sliding Diagonal algoritmy.

Bloková schéma analyzovaného riešenia je zobrazená na obrázku Obr. 2.33.



Obr. 2.33: Bloková schéma architektúry BIST s podporou viacerých typov algoritmov [13]

Význam jednotlivých blokov [13]:

- *Instruction Memory* – pamäť inštrukcií.
- *Instruction Buffer and Decoder* – zásobník inštrukcií a dekóder inštrukcií. Dekóduje konfiguračné inštrukcie čítané z pamäte inštrukcií a testovacie mikroinštrukcie čítané zo zásobníka inštrukcií.
- *PC Gen (Program Counter Generator)* – generátor programového počítadla. Generuje 3 adresy pre prístup do inštrukčnej pamäte, pre prístup do zásobníkov pre základné a lokálne slučky.
- *Control Gen (Control Signal Generator)* – generátor kontrolných signálov. Generuje signály pre ovládanie operácií zápisu a čítania z pamäte.
- *Addr Gen (Address Generator)* – generátor adries. Generuje adresy počas testovania podľa príslušnej adresovacej schémy.
- *Data Gen (Data Generator)* – generátor údajov. Generuje údaj určený na zápis do pamäte alebo údaj očakávaný ako výstup z pamäte.

- *Latency Adjuster* – synchronizačná jednotka. Synchronizuje do neho vstupujúce signály, je to z dôvodu, že môžu byť generované pod iným hodinovým cyklom.
- *Memory Under Test* – testovaná pamäť.
- *ORA (Output response analyzer)* – komparátor. Porovnáva hodnoty údajov čítaných z pamäte s očakávanými hodnotami.
- *DIAG (Diagnostic Monitor)* – diagnostický monitor. Zachytáva a posiela informácie o detegovaných poruchách.

Nakoľko analyzovaná architektúra podporuje aj algoritmy, ktoré obsahujú vnorené slučky (max. 1 vnorenie v rámci daného kroku algoritmu), vonkajšia alebo hlavná slučka sa označuje „základná“ slučka a vnorená slučka sa označuje „lokálna“ slučka. [13]

Na obrázku Obr. 2.34 je zobrazený formát inštrukčného slova (mikroinštrukcie) použitého v analyzovanej architektúre. Mikroinštrukcia môže byť inštrukčné slovo testovacieho algoritmu alebo konfiguračná inštrukcia. Inštrukčné slovo testovacieho algoritmu reprezentuje operáciu, ktorá sa má vykonať v testovanej pamäti. Pomocou konfiguračnej inštrukcie možno meniť testovací algoritmus, adresovaciu schému, schému testovacích údajov alebo spôsob diagnostiky a odosielania údajov o detegovaných poruchách. [13]

[8]	[7]	[6]	[5]	[4:3]				[2]	[1]	[0]
EBL	BBL	ELL	BLL	B	L	B+1	B-1	0/1	R/W	A
End of Base Loop	Begin of Base Loop	End of Local Loop	Begin of Local Loop	Base	Local	Base + 1	Base - 1	Data or Data Inverse	Read or Write	Algorithm Instruction (=1)

Obr. 2.34: Formát mikroinštrukcie architektúry BIST s podporou viacerých typov algoritmov [13]

Význam jednotlivých polí [13]:

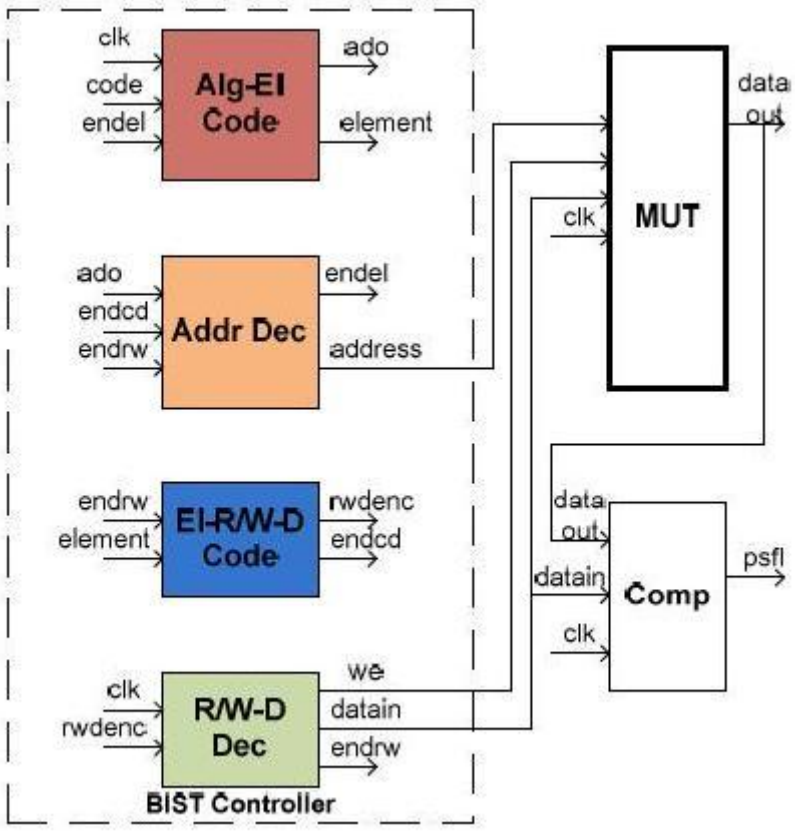
- *EBL (End of Base Loop)* – označenie koncového bodu základnej slučky.
- *BBL (Begin of Base Loop)* – označenie začiatočného bodu základnej slučky.
- *ELL (End of Local Loop)* – označenie koncového bodu lokálnej slučky.
- *BLL (Begin of Local Loop)* – označenie začiatočného bodu lokálnej slučky.
- *B (Base)* – označuje vykonanie operácie na slove základnej slučky.
- *L (Local)* – označuje vykonanie operácie na slove lokálnej slučky.
- *B+1 (Base + 1)* – označuje vykonanie operácie na ďalšom slove základnej slučky.
- *B-1 (Base - 1)* – označuje vykonanie operácie na predošlom slove základnej slučky.
- *0/1* – údaj pre zápis do pamäte alebo údaj očakávaný na výstupe z pamäte.

- *R/W* – operácia čítania alebo zápisu.
- *A* – slúži na rozlíšenie medzi inštrukčným slovom testovacieho algoritmu a konfiguračnou inštrukciou.

2.8.4 BIST založený na kombinácii FSM a mikrokódovania

Analyzované riešenie predstavuje riešenie programovateľnej architektúry vstavaného samočinného testovania (BIST) založenej na kombinácii techník konečného stavového automatu a mikrokódovania. Hlavným prínosom tohto riešenia je menšia plocha na čipe oproti predošlým riešeniam rovnakého typu pri podpore viacerých algoritmov typu march. [14]

Bloková schéma analyzovaného riešenia je zobrazená na obrázku Obr. 2.35.



Obr. 2.35: Bloková schéma architektúry BIST založenej na kombinácii FSM a mikrokódovania [14]

Význam jednotlivých blokov [14]:

- *Alg-El Code* – kóder elementov algoritmu.
- *Addr Dec* – adresný dekóder.
- *El-R/W-D Code* – kóder operácie čítania/zápisu z pamäte a testovacích údajov.
- *R/W-D Dec* – dekóder čítacej/zapisovacej sekvencie testovacieho algoritmu.
- *MUT* – testovaná pamäť.
- *Comp* – porovnáva očakávanú hodnotu s prečítanou hodnotou z pamäte a informuje o detegovanej poruche zmenou hodnoty signálu *psfl*.

Základné fungovanie: Blok *Alg-El Code* posiela kód elementu a adresu mikrokódu do blokov *El-R/W-D Code* a *Addr Dec*. Blok *El-R/W-D Code* začne ihneď po prijatí kódu elementu kódovať operácie čítania a zápisu daného elementu a testovacie údaje. Následne ich vo forme klastrov (*Cluster*) mikroinštrukcií pošle bloku *R/W-D Dec*, ktorý vykoná zakódované operácie s príslušnými testovacími údajmi. Jednotlivé signály zobrazené v blokovej schéme slúžia na zasielanie informácie medzi blokmi napr. o prebehnutí operácií elementu na celej pamäti a pod. (stavové signály). [14]

Analyzovaná architektúra podporuje 8 rôznych algoritmov march [14]:

- MATS+,
- March X,
- March C-,
- March A,
- March B,
- March U,
- March LR,
- March SS.

Tieto algoritmy obsahujú 12 rôznych elementov march, ktoré sú reprezentované použitím klastrov (tabuľka na obrázku Obr. 2.36). [14]

Operácie elementov (*R/W Operation in the Element*) march sú teda rozdelené do klastrov mikrokódu. Na odlíšenie medzi jednotlivými elementmi march sa používa 4-bitová hodnota kód elementu (*Element Code*). Klaster môže obsahovať 2-bitový alebo 4-bitový mikrokód v závislosti od toho, či sú v ňom zakódované 1 alebo 2 operácie. Nepárne bity

klastra identifikujú operáciu, ktorá sa má vykonať (1 – čítanie, 0 – zápis), párne bity klastra identifikujú údaj, s ktorým sa má daná operácia vykonať (1/0). Či sa má element march vykonávať vo vzostupnom alebo zostupnom poradí adries, je zakódované samostatne. Z reprezentácie mikroinštrukcií je zrejmé, že táto architektúra nie je schopná vykonávať march testy, ktoré majú v jednom elemente viac ako 6 operácií. [14]

4 bits MARCH Element Code	R/W Operation in the Element	No of Clusters	Cluster 1	Cluster 2	Cluster 3
0000	w0	1	w0	-	-
0001	r0,w1	1	r0w1	-	-
0010	r1,w0	1	r1w0	-	-
0011	r0	1	r0	-	-
0100	r0,w1,w0,w1	2	r0w1	w0w1	-
0101	r1,w0,w1	2	r1	w0w1	-
0110	r1,w0,w1,w0	2	r1w0	w1w0	-
0111	r0,w1,w0	2	r0	w1w0	-
1000	r0,w1,r1,w0,r0,w1	3	r0w1	r1w0	r0w1
1001	r0,w1,r1,w0	3	r0w1	r1w0	-
1010	r1,w0,r0,w1	2	r1w0	r0w1	-
1011	r0,r0,w0,r0,w1	3	r0	r0w0	r0w1
1100	r1,r1,w1,r1,w0	3	r1	r1w1	r1w0

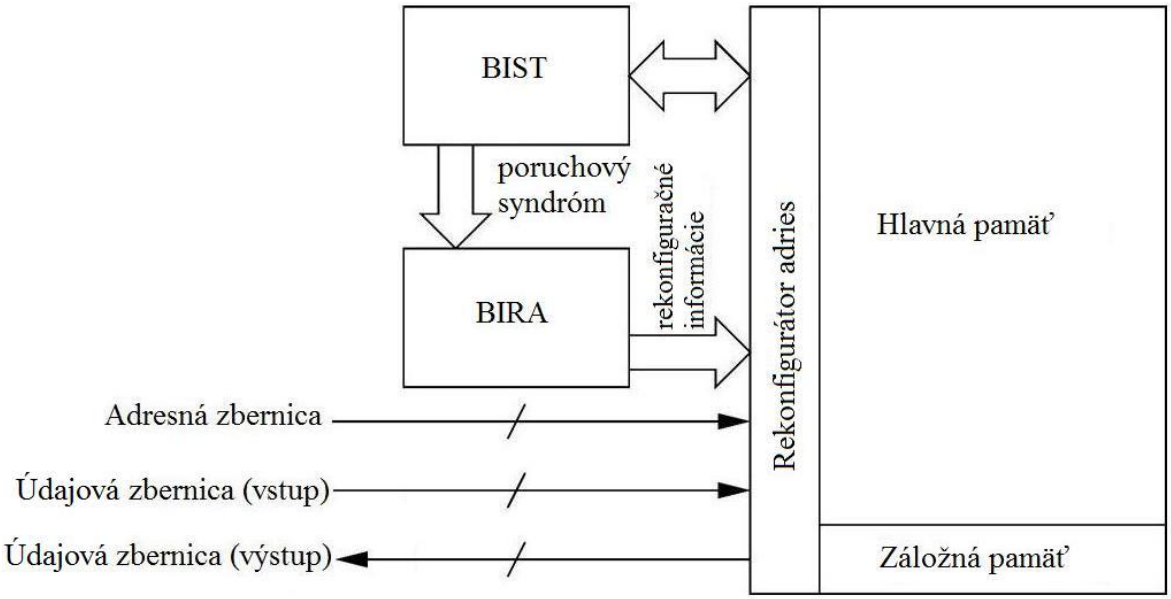
Obr. 2.36: Formát mikroinštrukcie architektúry BIST založenej na kombinácii FSM a mikrokódovania [14]

3 Špecifikácia požiadaviek a návrh riešenia

3.1 Celková architektúra

Na vykonanie opravy hlavnej pamäte má architektúre stačiť jedno zbehnutie testu a jedno zbehnutie algoritmu pre alokáciu záložných prvkov. Ďalšou požiadavkou pre architektúru je čo najmenšia plocha potrebná na čipe.

Bloková schéma navrhovanej architektúry je zobrazená na obrázku Obr. 3.1. Hlavná pamäť je slovne orientovaná. Blok *BIST* je blok samočinného testovania hlavnej pamäte a bude realizovaný prístupom analyzovaným v kapitole 2.8.1. Tento prístup bol zvolený, pretože je moderný a umožňuje použitie ľubovoľného testu march s neobmedzeným počtom operácii v jeho elemente. Blok *BIRA* je blok samočinnej opravy pamäte a bude realizovaný prístupom analyzovaným v kapitole 2.7.1. Tento prístup bol zvolený, pretože poskytuje vysokú úspešnosť opravy a podporuje slovne orientované pamäte. Blok *BIST* posiela bloku *BIRA* informácie o detegovaných poruchách v hlavnej pamäti. Blok *BIRA* na základe týchto informácií alokuje záložné prvky (predpoklad použitia 2D redundancie – záložné riadky a záložné stĺpce) zo záložnej pamäte namiesto poruchových miest v hlavnej pamäti, a tým hlavnú pamäť opraví. Jednotlivé zbernice (adresná, údajová) a riadiace signály (pre ovládanie operácií čítania a zápisu) budú multiplexované za účelom prepínať medzi funkčným a testovacím režimom hlavnej pamäte. *Rekonfigurátor adries* bude na základe informácií prijatých z bloku *BIRA* vykonávať readresáciu pamäťových buniek.

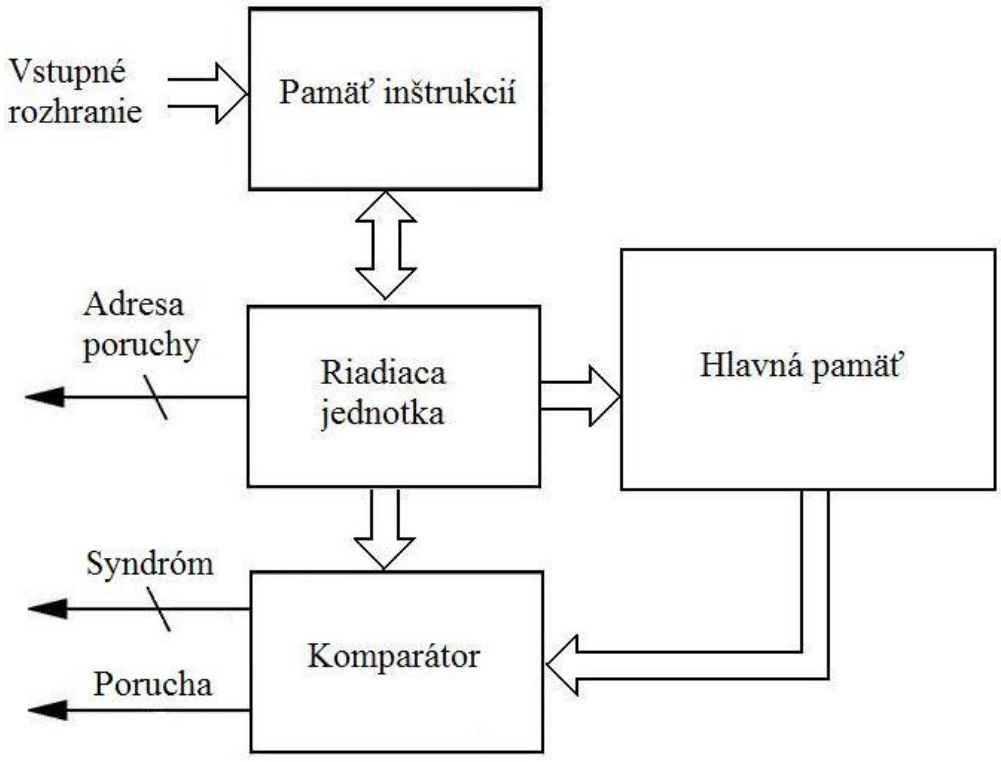


Obr. 3.1: Blokový návrh riešenia

3.2 Architektúra BIST

Architektúra BIST má poskytovať možnosť voľby testu pamäte. Testovanie pamäte má prebiehať na operačnej frekvencii pamäte (architektúra BIST pripojená na rovnaký hodinový signál) bez pozastavenia a omeškania v dôsledku dekódovania mikroinštrukcií alebo generovania testovacích signálov. Architektúra musí poskytovať informáciu o detegovanej poruche a to: informácia o výskyte poruchy, adresa poruchovej bunky pamäte a syndróm určujúci poruchové bity poruchového slova pamäte.

Na obrázku Obr. 3.2 je zobrazená bloková schéma navrhovanej architektúry BIST. Blok *Pamäť inštrukcií* bude uchovávať test typu march v podobe mikroinštrukcií, ktorý sa má vykonať. Blok *Riadiaca jednotka* bude čítať mikroinštrukcie z bloku *Pamäť inštrukcií*, dekódovať ich a vykonávať príslušné operácie nad hlavnou pamäťou. Blok *Komparátor* bude slúžiť na detekciu poruchy v pamäti porovnávaním očakávaného a reálneho výstupu z pamäte.



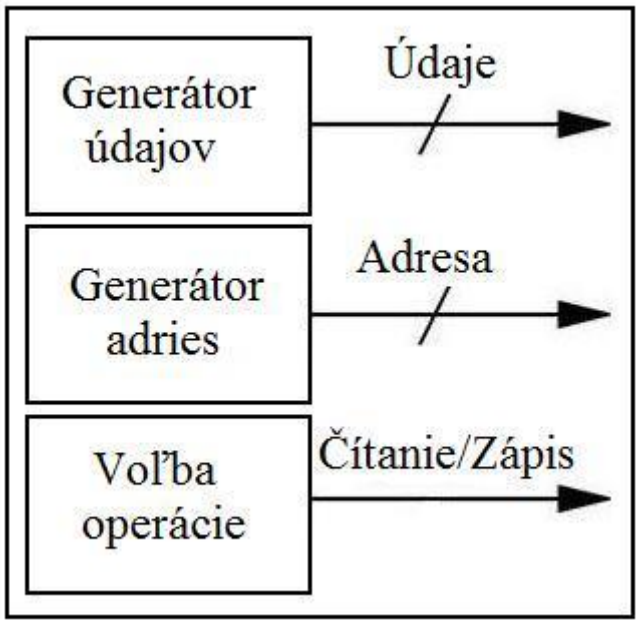
Obr. 3.2: Blokový návrh architektúry BIST

Architektúra BIST bude programovateľná v zmysle možnosti voľby testu pamäte, resp. nahrania požadovaného testu do pamäte inštrukcií. Túto možnosť bude poskytovať vstupné rozhranie pamäte inštrukcií.

Výstupom architektúry BIST budú signály:

- *Adresa poruchy* – adresa pamäťovej bunky, v ktorej bola detegovaná porucha, pozostáva zo zreťazenej riadkovej a stĺpcovej adresy danej pamäťovej bunky.
- *Syndróm* – údaj o šírke n bitov (n – dĺžka slova pamäte), ktorý obsahuje hodnoty log. 1 na pozíciách tých bitov slova, ktoré sú poruchové.
- *Porucha* – signál indikujúci výskyt poruchy.

Blok *Riadiaca jednotka* sa dá ďalej logicky rozdeliť na bloky *Generátor údajov*, *Generátor adries*, *Voľba operácie* (obrázok Obr. 3.3).



Obr. 3.3: Bloková schéma riadiacej BIST jednotky

Na základe načítanej mikroinštrukcie jednotlivé bloky generujú:

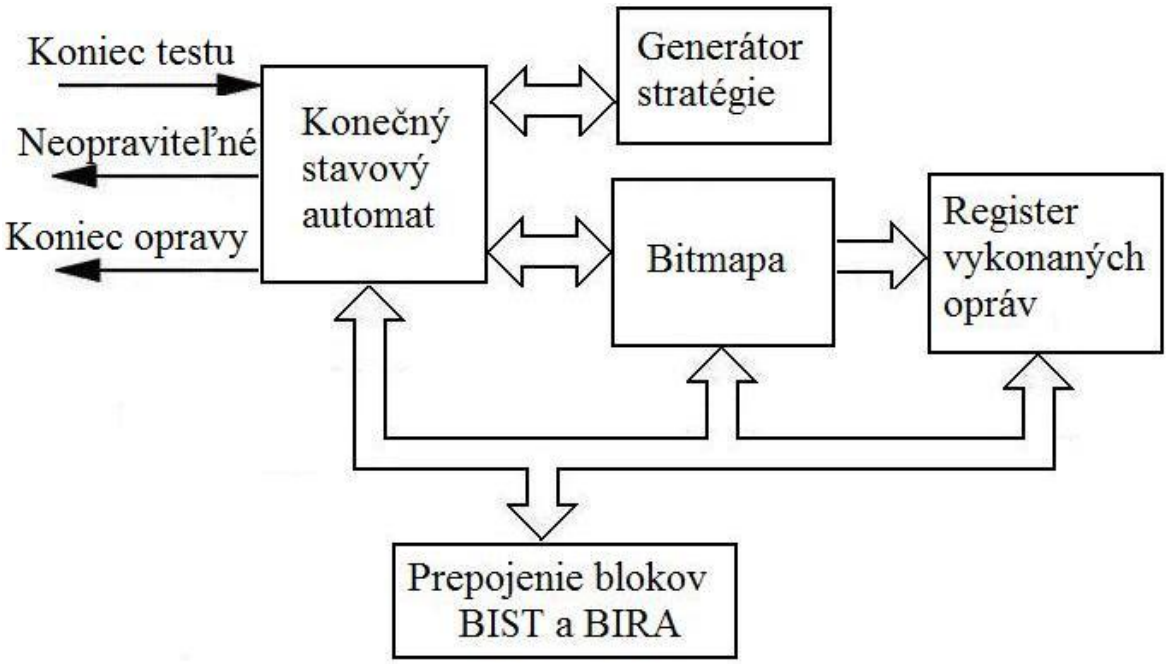
- *Generátor údajov* – testovacie údaje,
- *Generátor adries* – adresa testovanej pamäťovej bunky,
- *Voľba operácie* – nastavenie testovacej operácie pamäte na čítanie alebo zápis.

3.3 Architektúra BIRA

Základnou požiadavkou na každú architektúru BIRA by malo byť nájdenie riešenia opravy poruchových buniek pamäte, pokiaľ také riešenie existuje vzhľadom na počet dostupných záložných prvkov. Ďalšou požiadavkou môže byť činnost architektúry BIRA počas doby

testovania pamäte bez nutnosti pozastaviť samotné testovanie, túto požiadavku však splním návrhom vhodného prepojenia medzi blokmi BIST a BIRA.

Na obrázku Obr. 3.4 je zobrazený blokový návrh architektúry BIRA. Riadenie algoritmu bude vykonávané blokom *Konečný stavový automat*. Blok *Generátor stratégie* bude obsahovať všetky stratégie opravy – poradie aplikovania záložných riadkov a stĺpcov na poruchové bunky pamäte. Blok *Bitmapa* je logická štruktúra, ktorá sa bude používať na uchovávanie informácií o detegovaných poruchách a vyhodnocovanie úspešnosti aplikovania stratégie opravy. Blok *Register vykonaných opráv* bude uchovávať údaje o vykonaných opravách pamäte (tento istý blok používa adresný rekonfigurátor vo funkčnom režime pamäte). Blok *Prepojenie blokov BIST a BIRA* bude prepájať jednotlivé bloky vhodným spôsobom (návrh tohto bloku v samostatnej kapitole).



Obr. 3.4: Blokový návrh architektúry BIRA

Blok *Generátor stratégie* bude realizovaný ako pamäť určená len na čítanie. Blok *Register vykonaných opráv* bude realizovaný ako pamäť typu CAM (pamäť adresovaná obsahom). Blok *Konečný stavový automat* bude realizovaný ako logický obvod opísaný konečným stavovým automatom. Vybraný algoritmus analýzy opravy pozostáva z dvoch fáz – fáza nevyhnutnej opravy a finálna fáza. Fáza nevyhnutnej opravy sa vykonáva počas behu testovania. Finálna fáza predstavuje algoritmus vyčerpávajúceho hľadania riešenia opravy. Vstupný signál *Koniec testu* je signál z architektúry BIST informujúci o ukončení

testovania. Výstupný signál *Neopraviteľné* indikuje, že pamäť nie je opraviteľná. Výstupný signál *Koniec opravy* indikuje ukončenie behu algoritmu analýzy opravy.

Realizácia bloku *Bitmapa* bude zložitejšia, pretože okrem poľa pamäťových buniek obsahuje rôzne registre, ktorých obsah má rôznu šírku bitov a napĺňa sa rôznymi spôsobmi. Bloková schéma bloku *Bitmapa* je zobrazená na obrázku Obr. 3.5. Pole o rozmeroch m x n označené ako *bitmapa* bude slúžiť na indikáciu detegovanej poruchy. K príslušnej bunke poľa *bitmapa* potom prislúchajú príslušné bunky jednotlivých registrov:

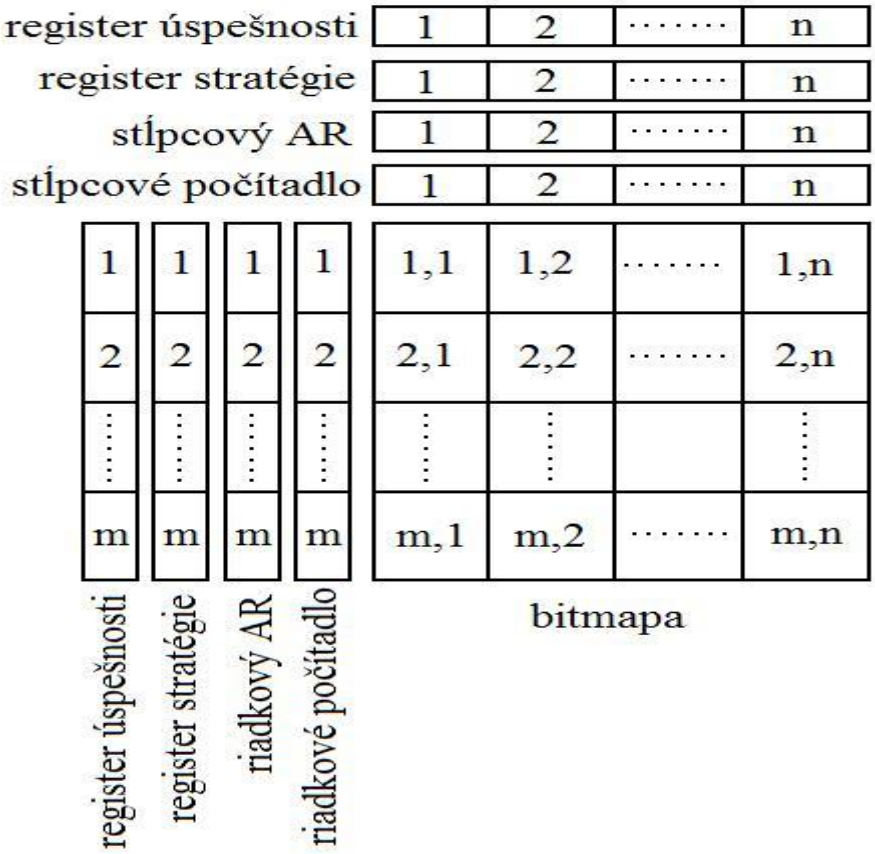
- *register úspešnosti* – register indikujúci platnosť záznamu v danom riadku / stĺpci lokálnej bitmapy počas fázy nutnej opravy, počas finálnej fázy sa používa na kontrolu úspešnosti opravy. Jeho obsah je generovaný logikou platnosti riadku / stĺpca,
- *register stratégie* – register pre indikáciu opravených riadkov / stĺpcov počas finálnej fázy algoritmu,
- *riadkový / stĺpcový AR* (adresný register) – register riadkovej / stĺpcovej adresy detegovanej poruchy,
- *riadkové / stĺpcové počítadlo* - počítadlo porúch v riadku / stĺpci, využitie vo fáze nutnej opravy – ak jeho hodnota presiahne počet voľných záložných riadkov / stĺpcov, je splnená podmienka nevyhnutnej opravy.

Rozmery poľa *bitmapa* m x n sú dané nasledovnými vzťahmi (za predpokladu, že sa vykonáva fáza nevyhnutnej opravy, čo je v tomto prípade splnené, r – počet záložných riadkov, c – počet záložných stĺpcov) [6]:

$m = (r.c + r)$

$n = (c.r + c)$

Následne platí pravidlo, že ak sa detegovaná porucha už nedá zapísať do poľa bitmapa po fáze nevyhnutnej opravy, pamäť je neopraviteľná [6]. Pre implementáciu sme sa rozhodli použiť architektúru s tromi záložnými riadkami a stĺpcami, veľkosť poľa bitmapa bude preto 12 x 12.



Obr. 3.5: Bloková schéma bloku bitmapa

Kľúčovým prvkom celej architektúry BIRA je práve *Bitmapa*. Pomocou tejto logickej štruktúry (ďalej len bitmapa) sa vykonáva fáza nevyhnutnej opravy a takisto sa vyhodnocuje úspešnosť aplikovania opravnej stratégie vo finálnej fáze algoritmu analýzy opravy. Nefunkčnosť bitmapy by znamenala nefunkčnosť celej architektúry BIRA. Z pohľadu konečného stavového automatu predstavujúceho riadiacu jednotku BIRA obvodu možno definovať požiadavky na funkčnosť bitmapy:

1. Bitmapa musí umožňovať nasledovné vstupy:
   - samostatný zápis hodnoty do ľubovoľnej bunky poľa,
   - samostatný zápis do ľubovoľného poľa registrov stratégie,
   - samostatný zápis do ľubovoľného poľa oboch adresných registrov,
   - reset celého obvodu, ďalej reset buniek poľa po stĺpcoch a riadkoch (využitie vo fáze nevyhnutnej opravy),
   - samostatný reset registrov stratégie.

2. Bitmapa musí poskyťovať nasledovné výstupy:

   - informácia o úspešnosti opravy po aplikovaní opravnej stratégie,
   - údajové výstupy z adresných registrov, aby bolo možné zapísať opravy do registra vykonaných opráv,
   - informácia o nájdenej zhode pri kontrole adries novej poruchy, či už je v poli záznam s rovnakou riadkovou a/alebo stĺpcovou adresou,
   - adresné výstupy z adresných registrov, aby bolo možné zapísať novú poruchu s rovnakou riadkovou alebo stĺpcovou adresou do rovnakého riadka / stĺpca poľa, ďalej aby bolo možné zapísať úplne novú poruchu,
   - informácia o počte porúch v danom riadku alebo stĺpci poľa, pre kontrolu splnenia podmienky nevyhnutnej opravy.

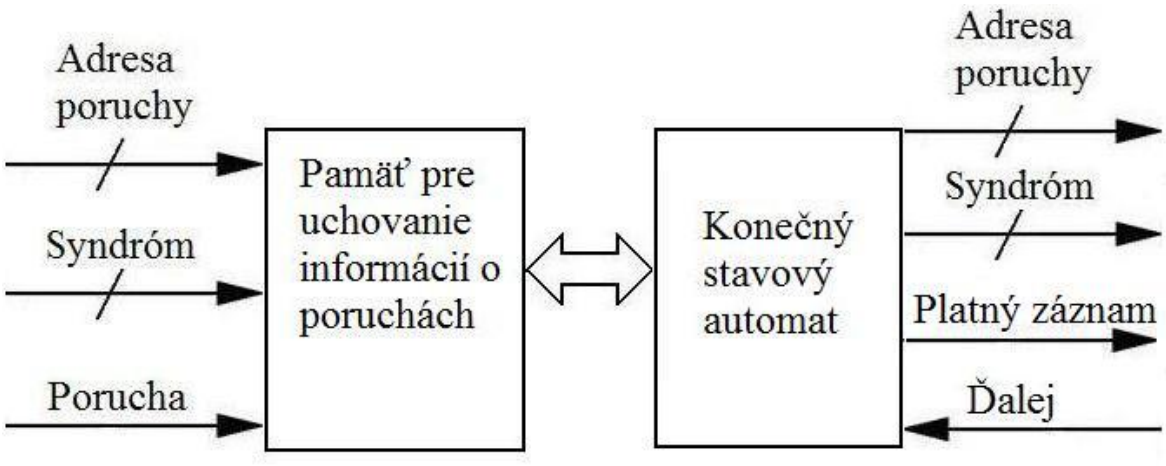
3.4 Prepojenie architektúr BIST a BIRA

Blok, ktorý bude slúžiť na prepojenie architektúr BIST a BIRA sa dá prirovnať k zásobníku, resp. FIFO radu. Jeho kapacita sa určí kombináciou teoretického výpočtu a testovania tak, aby dokázal uchovávať informácie o dostatočnom počte detegovaných porúch, ktoré príjme z architektúry BIST, kým si tieto informácie architektúra BIRA prevezme a spracuje. Samozrejme, strata informácií o čo i len jednej detegovanej poruche nie je prípustná.

Navyše, tento blok musí realizovať funkciu analýzy syndrómu danej poruchy. Je to z toho dôvodu, že stĺpcové zálohy majú šírku 1 bit a nie šírku celého slova. To znamená, že z pohľadu algoritmu analýzy opravy predstavuje syndróm „1100“ (pri šírke slova 4 bity) 2 poruchy. Blok teda musí architektúre BIRA poskytnúť najprv informácie v tvare *adresa_poruchy* + *syndróm „1000“* a následne *adresa_poruchy* + *syndróm „0100“*.

Na obrázku Obr. 3.6 je zobrazený blokový návrh prepojenia architektúr BIST a BIRA. Blok *Pamäť pre uchovanie informácií o poruchách* bude realizovaný ako FIFO zásobník. Vstupom tohto bloku budú výstupné signály architektúry BIST. Blok *Konečný stavový automat* bude logický obvod opísaný konečným stavovým automatom, ktorého funkciou bude extrakcia jednotlivých poruchových bitov z prijatého poruchového syndrómu a poskytovanie tejto informácie architektúre BIRA. Výstupnými signálmi sú teda *Adresa poruchy* a nový *Syndróm*, ďalej riadiaci signál *Platný záznam* informujúci o platnosti

údajov na výstupoch, vstupný signál *Ďalej* bude riadiaci signál z architektúry BIRA, ktorým si vyžiada ďalší záznam.



Obr. 3.6: Blokový návrh bloku Prepojenie architektúr BIST a BIRA

3.5 Testovaná pamäť a architektúra

Veľkosť testovanej pamäte pre potreby testovania funkčnosti riešenia zvolíme 32 x 32 bitov s pamäťovým slovom o šírke 4 bity. Štruktúra pamäte bude 32 riadkov x 8 stĺpcov. Slovne orientovaná pamäť bola zvolená z toho dôvodu, že dnes sa bitovo orientované pamäte používajú veľmi zriedkavo. Testovanú pamäť a celkovú architektúru implementujeme pomocou dostupných prostriedkov v programe HDL Designer. Pre účely injekcie a simulácie porúch pamäte budeme využívať príkaz *Force* v programe Modelsim.

Počet záložných prvkov bude 3 riadky a 3 stĺpce. Tento počet bol zvolený vzhľadom na veľkosť bitmapy, ktorá je v tomto prípade 12 x 12 bitov, čo je z pohľadu časovej zložitosti jej implementácie prijateľné. 2-D zálohy (riadkové a stĺpcové) boli zvolené z toho dôvodu, že sú to najpoužívanejšie zálohy v architektúrach opravy pamäte a algoritmy vďaka nim dosahujú vysokú úspešnosť v porovnaní s algoritmami využívajúcimi iba 1-D zálohy.

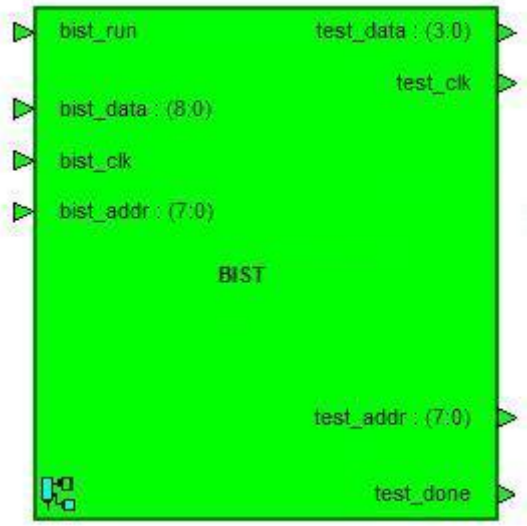
4 Implementácia riešenia

4.1 Architektúra BIST

4.1.1 Architektúra BIST všeobecne

Na obrázku Obr. 4.1 je zobrazený BIST obvod z najvyššej úrovne abstrakcie. Bloková schéma BIST obvodu sa nelíši od návrhu (kapitola 3.2). Obsahuje nasledovné vstupy a výstupy:

- *bist_run* – signál pre spustenie testovania pamäte,
- *bist_data*, *bist_clk*, *bist_addr* – signály slúžiace na zápis mikroinštrukcií do pamäte inštrukcií BIST obvodu,
- *test_data* – testovacie údaje pre pamäť,
- *test_we* – ovládanie operácie čítania a zápisu počas testovania,
- *test_addr* – testovací adresný vstup pamäte,
- *test_done* – signál indikujúci ukončenie testovania (vtedy *test_done* = '1').

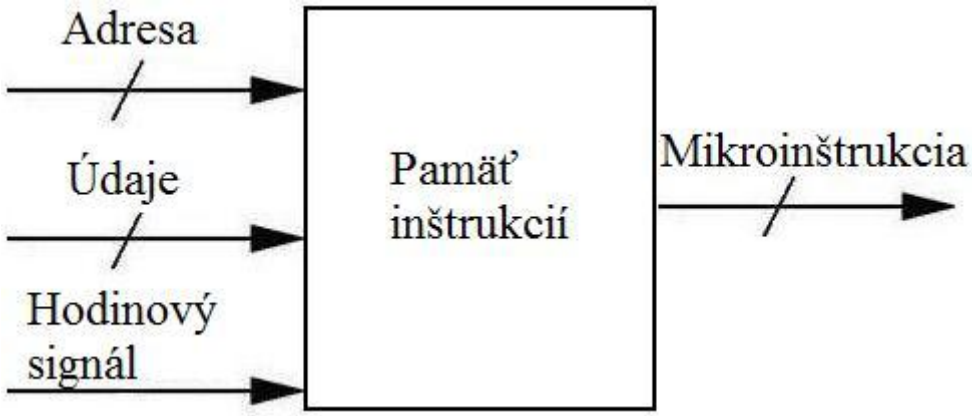


Obr. 4.1: BIST obvod

4.1.2 Pamäť inštrukcií

Pamäť inštrukcií slúži na uloženie testu typu march vo formáte mikroinštrukcií. Dôsledkom prítomnosti tejto pamäte v architektúre je pridanie rozhrania pre prístup používateľa do tejto pamäte. Používateľ si môže do pamäte inštrukcií nahrať ľubovoľný

test typu march, ktorý má maximálnu zložitosť 255N (v závislosti na konkrétnej implementácii, kapacita pamäte sa dá rozšíriť) a nevyužíva prvky oneskorenia. Na obrázku Obr. 4.2 je zobrazená bloková schéma pamäte inštrukcií.



Obr. 4.2: Bloková schéma pamäte inštrukcií

Význam jednotlivých signálov rozhrania:

- *Adresa (bist_addr)* – 8-bitová adresa do pamäte, nahrávanie inštrukcií do pamäte začína od adresy „00000000“,
- *Údaje (bist_data)* – údajový vstup do pamäte, sem používateľ pripravuje konkrétnu mikroinštrukciu, ktorá má byť uložená na zvolenú adresu,
- *Hodinový signál (bist_clk)* – signál pre ovládanie vykonania operácie zápisu do pamäte, zápis sa vykoná pri nábežnej hrane tohto signálu,
- *Mikroinštrukcia* – výstupné údaje z pamäte, ktoré pri testovaní používa riadiaca jednotka architektúry BIST pre generovanie testovacích vektorov a vykonávanie testu.

4.1.3 Formát mikroinštrukcie

Na obrázku Obr. 4.3 je zobrazený formát mikroinštrukcie.

platnosť záznamu	druh operácie		vzostupné / zostupné poradie adries	typ operácie	údaje			
0	1	2	3	4	5	6	7	8

Obr. 4.3: Formát mikroinštrukcie

Mikroinštrukcia pozostáva z 9-tich bitov, ktoré majú nasledovný význam a nadobúdajú nasledovné hodnoty:

- *platnosť záznamu* – indikuje, či je daný záznam platný alebo nie:
  - ‘0’ – neplatný záznam,
  - ‘1’ – platný záznam,
- *druh operácie* – definuje druh operácie v rámci elementu march:
  - „00“ – samostatná operácia, daný element má len jednu operáciu,
  - „10“ – prvá operácia elementu, ktorý obsahuje 2 a viac operácií,
  - „01“ – prostredná / priebežná operácia elementu, ktorý obsahuje 3 a viac operácií,
  - „11“ – posledná operácia elementu, ktorý obsahuje 2 a viac operácií,
- *vzostupné / zostupné poradie adries* – definuje smer vykonávania operácie:
  - ‘0‘ – smer od vyšších adries k nižším adresám (dekrementácia adries),
  - ‘1‘ – smer od nižších adries k vyšším adresám (inkrementácia adries),
- *typ operácie* – definuje konkrétnu operáciu nad pamäťou:
  - ‘0‘ – operácia čítania,
  - ‘1‘ – operácia zápisu,
- *údaje* – definuje konkrétnu 4-bitovú hodnotu (testovaciu vzorku), s ktorou operácia pracuje (ľubovoľná 4-bitová kombinácia hodnôt log. 1 a log. 0).

Posledný záznam, ktorý používateľ nahrá do pamäte inštrukcií musí mať bit indikujúci platnosť záznamu nastavený na hodnotu log. 0.

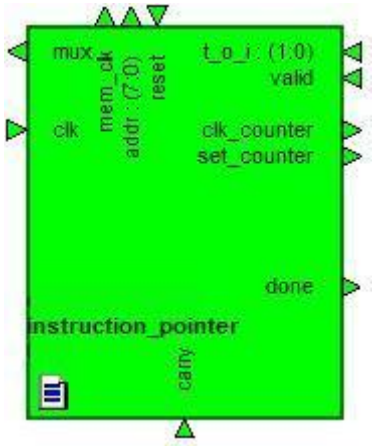
4.1.4 Riadiaca jednotka

Na obrázku Obr. 4.4 je zobrazená riadiaca jednotka testovania (*instruction_pointer*).

Význam jednotlivých signálov:

- *clk* – hodinový signál (spoločný s pamäťou RAM),
- *mux* – riadiaci signál, ktorý ovláda pripojenie hodinového signálu pamäte RAM na riadiacu logiku BIST,
- *mem_clk*, *addr* – riadiace signály pre ovládanie pamäte inštrukcií, hodinový a adresný signál,
- *reset* – slúži na počiatočnú inicializáciu BIST obvodu,

- *t_o_i* – na tento vstup sú pripojené 2 výstupné údajové bity z pamäte inštrukcií, v ktorých je zakódovaný druh operácie,
- *valid* – na tento vstup je pripojený 1 výstupný údajový bit z pamäte inštrukcií, ktorý indikuje platnosť prečítaného záznamu,
- *clk_counter, set_counter* – signály slúžia na ovládanie generátora adries, prvý je hodinový signál pre inkrementáciu / dekrementáciu generovanej adresy, druhý signál slúži na reset generátora adries,
- *done* – signál indikujúci ukončenie testovania,
- *carry* – signál indikujúci pretečenie počítadla adresy v generátore adries.



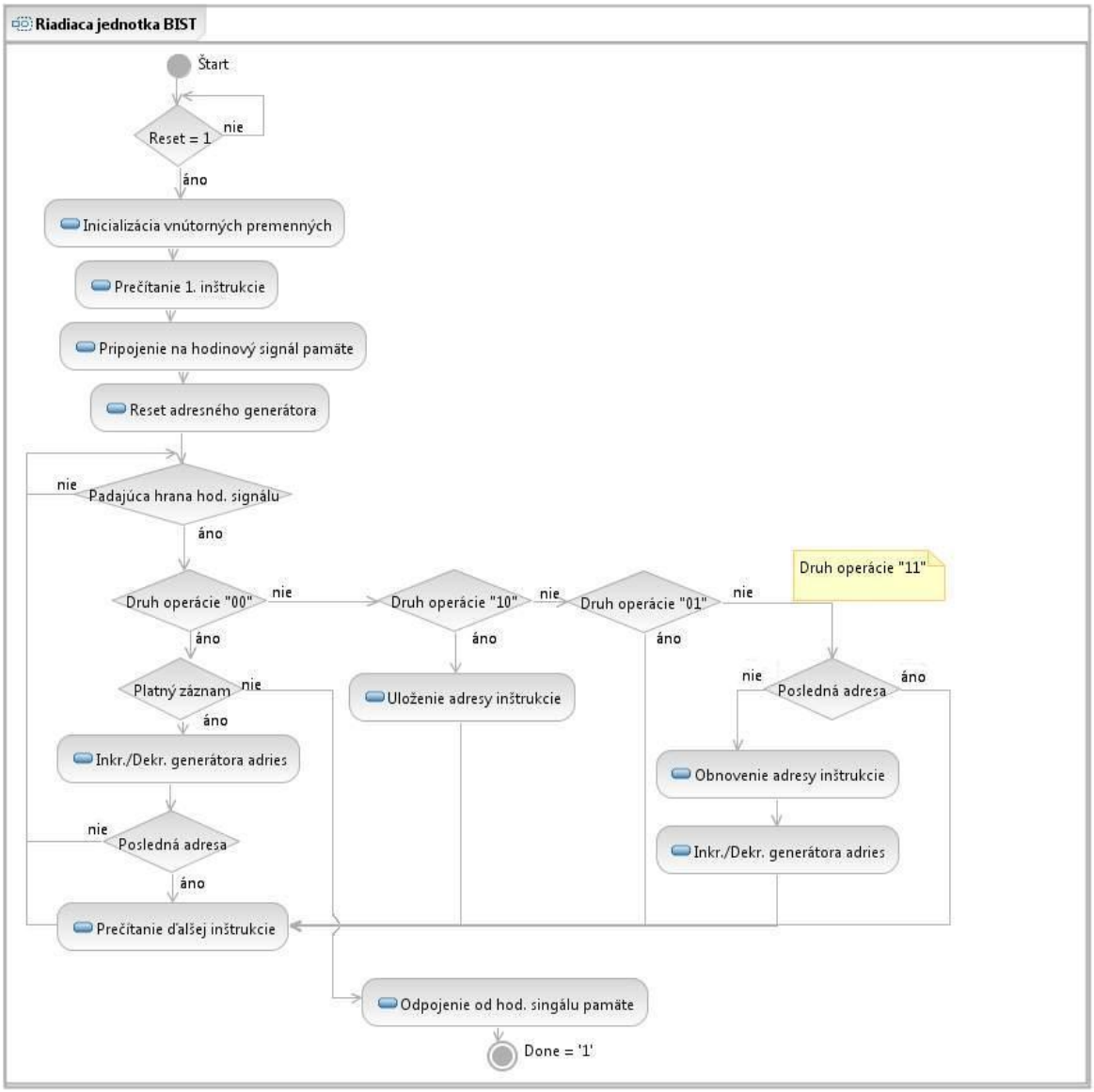
Obr. 4.4: Riadiaca BIST jednotka

Na obrázku Obr. 4.5 je zobrazený vývojový diagram riadiacej BIST jednotky. Po štarte systému, resp. nábehu napájacieho napätia sa čaká na signál *Reset*, aby sa mohlo začať testovanie. Po príchode nábežnej hrany signálu reset nasleduje inicializačná procedúra, ktorá pozostáva z inicializácie vnútorných premenných (registrov obsahujúcich adresy do pamäte inštrukcií), prečítania prvej inštrukcie z pamäte inštrukcií, pripojenia sa na hodinový signál pamäte a vynulovania počítadiel v generátore adries.

Ďalšia činnosť riadiacej BIST jednotky je synchronizovaná na padajúcu hranu hodinového signálu. Je to z toho dôvodu, že pamäť je synchronizovaná na stúpacúju hranu hodinového signálu a v tomto čase je už potrebné mať pripravené všetky riadiace signály. Čo sa stane po detekcii padajúcej hrany hodinového signálu, závisí od druhu operácie.

Ak je druh operácie „00“ (samostatná operácia, daný element má len jednu operáciu), kontroluje sa platnosť záznamu, pretože len v tomto prípade sa môže jednať o posledný záznam obsahujúci samé hodnoty log. 0. Ak je prečítaná inštrukcia posledný záznam,

jednotka sa odpojí od hodinového signálu pamäte a nastaví signál *done (test_done)* na hodnotu log. 1 – testovania ukončené. Ak sa nejedná o posledný záznam z pamäte inštrukcií, vykoná sa inkrementácia alebo dekrementácia generovanej adresy (podľa smeru adresovania). Ak sa vykonala operácia na poslednej pamäťovej bunke adresného priestoru, načíta sa ďalšia inštrukcia z pamäte inštrukcií. Ak sa ešte operácia nevykonala na všetkých bunkách testovanej pamäte, čaká sa na hodinový signál.



Obr. 4.5: Vývojový diagram riadiacej BIST jednotky

Ak je druh operácie „10“ (prvá operácia elementu, ktorý obsahuje 2 a viac operácií), uloží sa adresa prečítanej inštrukcie do vnútornej premennej, pretože sem sa neskôr po vykonaní poslednej operácie elementu jednotka vráti. Prečíta sa ďalšia inštrukcia z pamäte inštrukcií a čaká sa na hodinový signál.

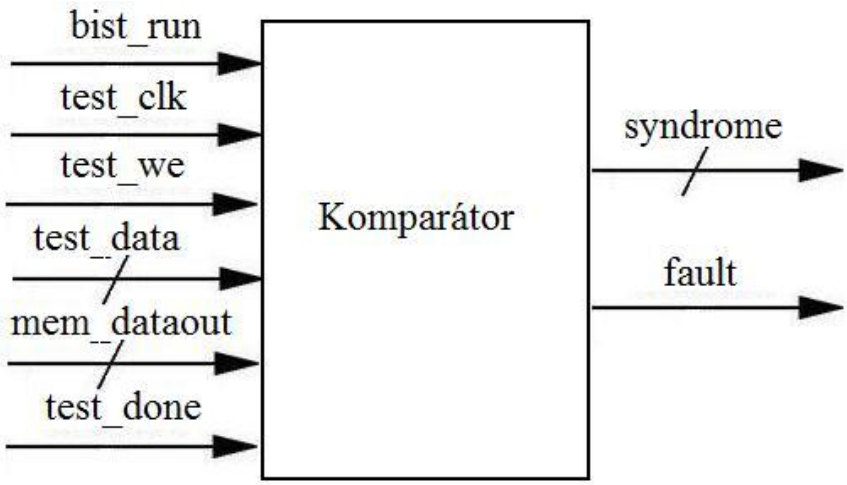
Ak je druh operácie „01“ (prostredná / priebežná operácia elementu, ktorý obsahuje 3 a viac operácií), tak sa len prečíta ďalšia inštrukcia z pamäte inštrukcií a čaká sa na hodinový signál.

Ak je druh operácie „11“ (posledná operácia elementu, ktorý obsahuje 2 a viac operácií) a operácia sa vykonala nad poslednou bunkou pamäte, prečíta sa ďalšia inštrukcia z pamäte inštrukcií. V opačnom prípade sa ešte netestovali všetky bunky pamäťového priestoru, obnoví sa adresa prvej operácie elementu march testu, z ktorej sa prečíta inštrukcia a inkrementuje alebo dekrementuje sa generátor adries. Čaká sa na hodinový signál.

4.1.5 Komparátor

Na obrázku Obr. 4.6 je zobrazená bloková schéma obvodu komparátor. Komparátor má 2 výstupné signály:

- *syndrome* – údaj o šírke n bitov (n – dĺžka slova pamäte), ktorý obsahuje hodnoty log. 1 na pozíciách tých bitov slova, ktoré sú poruchové.
- *fault* – signál indikujúci výskyt poruchy. Aby bol tento signál aktívny (hodnota log. 1), musia byť splnené nasledovné podmienky:
  - prebieha test pamäte (signál *bist_run* = ‘1’),
  - prebieha operácia čítania nad pamäťou (signál *test_clk* = ‘1’ a signál *test_we* = ‘0’),
  - údaje prečítané z pamäte sa nezhodujú s očakávanými údajmi (údaje na zbernici *test_data* sa líšia od údajov na zbernici *mem_dataout*).
  - test ešte neskončil (signál *test_done* = ‘0’).



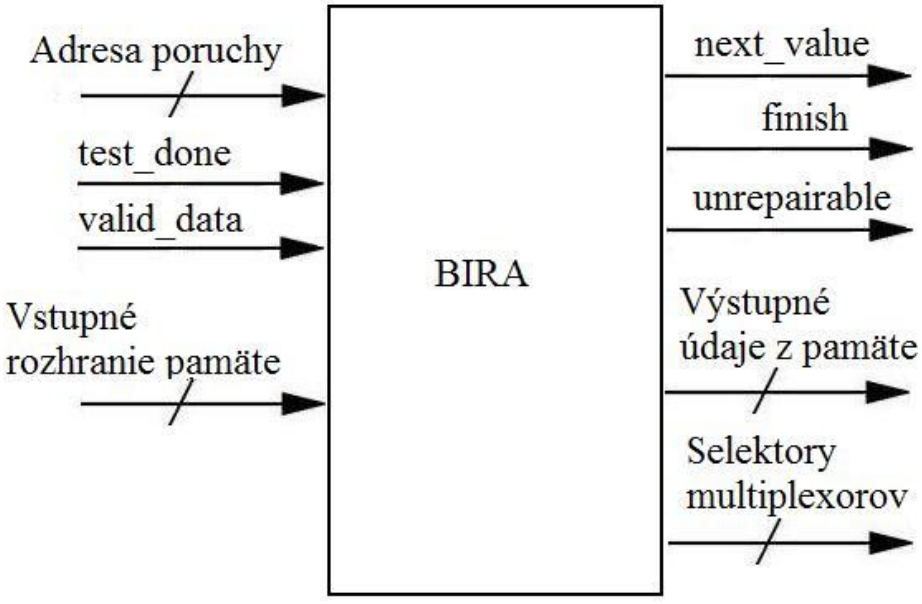
Obr. 4.6: Bloková schéma obvodu komparátor

4.2 Architektúra BIRA

4.2.1 Architektúra BIRA všeobecne

Na obrázku Obr. 4.7 je zobrazený BIRA obvod z najvyššej úrovne abstrakcie. Bloková schéma BIRA obvodu sa nelíši od návrhu (kapitola 3.3). Obsahuje nasledovné vstupy a výstupy:

- *Adresa poruchy* – signál z bloku prepojenia obvodov BIST a BIRA obsahujúci adresu detegovanej poruchy,
- *test_done* – signál z BIST obvodu informujúci o ukončení testovania pamäte,
- *valid_data* – signál z bloku prepojenia obvodov BIST a BIRA informujúci o platnosti údajov na vstupe *Adresa poruchy*,
- *Vstupné rozhranie pamäte* – zahŕňa hodinový signál, údajovú zbernicu, adresnú zbernicu a signál voľby operácie hlavnej pamäte,
- *next_value* – signál do bloku prepojenia obvodov BIST a BIRA pre vyžiadanie ďalšieho záznamu,
- *finish* – signál indikujúci ukončenie opravy pamäte,
- *unrepairable* – signál indikujúci neopraviteľnosť pamäte,
- *Výstupné rozhranie z pamäte* – zahŕňa údajové výstupy zo záložnej pamäte,
- *Selektory multiplexorov* – signály ovládajúce multiplexory, ktoré zabezpečujú prešírenie správnej hodnoty z pamäte (hlavnej alebo záložnej) na údajový výstup.



Obr. 4.7: BIRA obvod

4.2.2 Generátor stratégie

Generátor stratégie je pamäť určená len na čítanie (ROM), ktorá obsahuje všetky možnosti aplikovania záloh. Z ktorej adresy tejto pamäte sa číta a koľko bitov sa berie do úvahy, určuje pomocný kombinačný obvod, ktorý má na svojom vstupe počet záloh (*R* – riadková, *C* – stĺpcová) a na výstupe poskytuje základnú adresu do generátora stratégie (*Adresa do STR*, tam sa nachádza prvá aplikovateľná stratégia), celkový počet stratégií (*Kombinácií*) a počet záloh (*Bitov*), koľko sa môže aplikovať. Správanie kombinačného obvodu je zobrazené v tabuľke Tab. 4.1. Pre zvolený počet záloh 3 riadky a 3 stĺpce existuje 20 možností, v akom poradí možno zálohy aplikovať. Pamäť obsahujúca jednotlivé stratégie má teda 20 záznamov a jej obsah je zobrazený v tabuľke Tab 4.2. Hodnota ‘0’ znamená oprava stĺpcom, hodnota ‘1’ znamená oprava riadkom.

Tab. 4.1: Správanie kombinačného obvodu

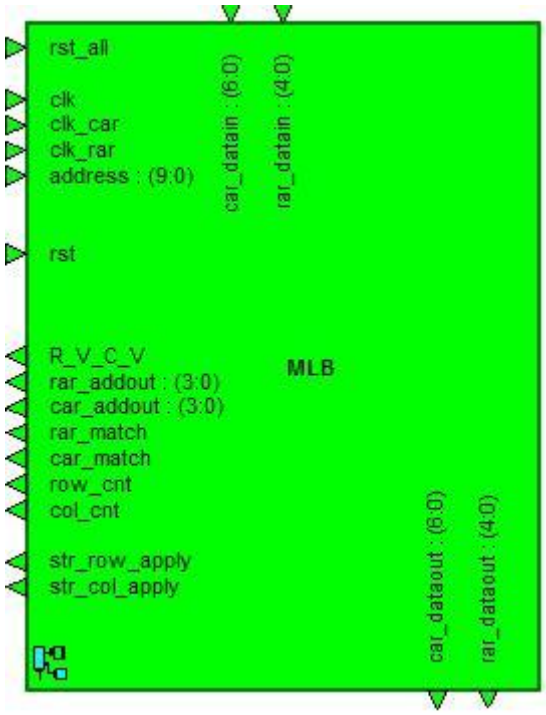
Ostáva záloh		Adresa do STR	Kombinácií	Bitov
R	C			
3	3	0	20	6
3	2	10	10	5
3	1	16	4	4
3	0	16	1	3
2	3	0	10	5
2	2	0	6	4
2	1	0	3	3
2	0	16	1	2
1	3	6	4	4
1	2	3	3	3
1	1	0	2	2
1	0	16	1	1
0	3	6	1	3
0	2	6	1	2
0	1	6	1	1
0	0	xxxxx	xxxxx	xxxxx

Tab. 4.2: Obsah pamäte

Adresa	Stratégia
0	0 1 1 0 0 1
1	1 0 1 0 0 1
2	1 1 0 0 0 1
3	0 0 1 1 0 1
4	0 1 0 1 0 1
5	1 0 0 1 0 1
6	0 0 0 1 1 1
7	0 0 1 0 1 1
8	0 1 0 0 1 1
9	1 0 0 0 1 1
10	1 0 0 1 1 0
11	0 1 0 1 1 0
12	0 0 1 1 1 0
13	1 1 0 0 1 0
14	1 0 1 0 1 0
15	0 1 1 0 1 0
16	1 1 1 0 0 0
17	1 1 0 1 0 0
18	1 0 1 1 0 0
19	0 1 1 1 0 0

4.2.3 Bitmapa (MLB)

Na obrázku Obr. 4.8 je zobrazená logická štruktúra bitmapa (ďalej len bitmapa, resp. MLB) z najvyššieho pohľadu abstrakcie. Rozmer poľa je 12 x 12 (vyplýva z použitia 3 záložných riadkov a 3 záložných stĺpcov).



Obr. 4.8: Základná schéma obvodu bitmapa

Význam jednotlivých signálov:

- *rst_all* – celkový reset obvodu,
- *clk* – hodinový signál, slúžiaci na vykonanie zápisu do poľa bitmapy alebo do registrov stratégie,
- *clk_rar*, *clk_car* – hodinový signál pre zápis do riadkového a stĺpcového adresného registra,
- *address* – 10-bitová adresná zbernica:
    - bity 0-3: stĺpcová adresa v rámci poľa,
    - bity 4-7: riadková adresa v rámci poľa,
    - bity 8-9: slúžia na určenie adresovaných prvkov:
        - „00“ – riadkový register stratégie,
        - „01“ – stĺpcový register stratégie,
        - „10“ – bunky poľa,
        - „11“ – nemá určenie,

- *rst* – reset, slúži na reset poľa bitmapy, registrov stratégie alebo celého obvodu,
- *rar_datain*, *car_datain* – údajový vstup do riadkového a stĺpcového adresného registra,
- *R_V_C_V* – signál indikujúci úspešnosť opravy (využitie vo finálnej fáze algoritmu opravy),
- *rar_dataout*, *car_dataout* – údajový výstup riadkového a stĺpcového adresného registra (využitie pri alokácii záložných prvkov a na zápis opravy do registra vykonaných opráv),
- *rar_addout*, *car_addout* – adresný výstup riadkového a stĺpcového adresného registra (využitie pri zápise poruchy do bitmapy a kontrolu zhody v adrese detegovanej poruchy),
- *rar_match*, *car_match* – signál indikujúci nájdenú zhodu adresy poruchy v riadkovom alebo stĺpcovom adresnom registri (využitie pri zápise poruchy, ktorá má rovnakú riadkovú alebo stĺpcovú adresu s už zaznamenanou poruchou),
- *row_cnt, col_cnt* – signál indikujúci výskyt troch porúch pri zápise do riadku alebo stĺpca poľa, jeho aktívna hodnota (log. 0) znamená splnenie podmienky nevyhnutnej opravy – je snaha zapísať 4. poruchu, no k dispozícii sú len 3 zálohy,
- *str_row_apply, str_col_apply* – signál indikujúci potrebu opraviť vybraný riadok alebo stĺpec (využitie v druhej fáze algoritmu opravy).

Väčšina vyššie uvedených signálov tvorí rozhranie medzi bitmapou a konečným stavovým automatom, ktorý na základe týchto signálov vykonáva opravu pamäte.

4.2.4 Register vykonaných opráv

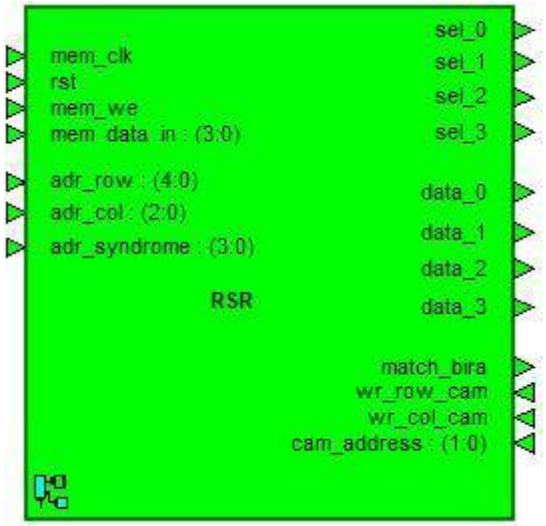
Na obrázku Obr. 4.9 je zobrazený obvod register vykonaných opráv (RSR) z najvyššieho pohľadu abstrakcie. Tento obvod obsahuje pamäte typu CAM pre zaznamenanie opráv použitím záložného riadka alebo stĺpca, záložnú pamäť a obvody generujúce niektoré riadiace signály. Kým prebieha testovanie a oprava pamäte, používajú sa nasledovné signály a majú nasledovný význam:

- *adr_row* – na tento vstup je multiplexovaná buď adresa riadka hlavnej pamäte, ktorý bol opravený, alebo riadková adresa detegovanej poruchy pre overenie, či je daný riadok už opravený,

- *adr_col, adr_syndrome* – na tieto vstupy sú multiplexované buď adresa stĺpca hlavnej pamäte a syndróm, ktorý bol opravený, alebo stĺpcová adresa a syndróm detegovanej poruchy pre overenie, či je daný stĺpec už opravený,
- *match_bira* – signál indikujúci zhodu s porovnávanou adresou – porucha na danej adrese je už opravená,
- *rst* – reset celého obvodu,
- *wr_row_cam* – signál ovládajúci operáciu zápisu do pamäte CAM, v ktorej sú zaznamenané opravy vykonané použitím záložného riadka,
- *wr_col_cam* – signál ovládajúci operáciu zápisu do pamäte CAM, v ktorej sú zaznamenané opravy vykonané použitím záložného stĺpca,
- *cam_address* – na tento vstup je multiplexovaná adresa do pamäte CAM pre opravené riadky alebo do pamäte CAM pre opravené stĺpce.

Keď je pamäť používaná vo funkčnom režime, používajú sa nasledovné signály:

- *mem_clk, mem_data, adr_row + adr_col, mem_we* – klasické rozhranie hlavnej pamäte – hodinový signál, údajová zbernica, adresná zbernica a signál voľby operácie hlavnej pamäte,
- *sel_0, sel_1, sel_2, sel_3* – signály ovládajúce multiplexory, ktoré zabezpečujú prešírenie správnej hodnoty z pamäte (hlavnej alebo záložnej) na údajový výstup,
- *data_0, data_1, data_2, data_3* – signály obsahujúce výstupné údaje zo záložnej pamäte.



Obr. 4.9: Základná schéma obvodu register vykonaných opráv

4.2.5 Konečný stavový automat

Konečný stavový automat predstavuje riadiacu jednotku celej architektúry BIRA. Má vstupno/výstupné rozhranie s každým vyššie uvedeným blokom v architektúre BIRA, aby na základe informácií poskytnutých z týchto blokov mohol vykonávať prílušné akcie. Konečný stavový automat má voči ostatným blokom v architektúre BIRA nasledovné postavenie/funkciu:

- z *generátora stratégie* získava opravné stratégie,
- riadi celý proces zápisu, čítania, mazania hodnôt a aplikácie stratégie v *bitmape*,
- riadi celý proces zápisu a čítania hodnôt v *registri vykonaných opráv*,
- riadi spracovanie informácií o poruchách z bloku prepojenia obvodov BIST a BIRA.

Na obrázku Obr. 4.10 je zobrazený stavový diagram konečného stavového automatu. Diagram je logicky rozdelený na 2 polovice (hornú a dolnú). Horná polovica stavového diagramu opisuje činnosť konečného stavového automatu v čase, keď sa vykonáva testovanie pamäte (fáza nevyhnutnej opravy). Dolná polovica stavového diagramu opisuje činnosť konečného stavového automatu v čase, keď sa vykonáva fáza vyčerpávajúceho hľadania.

Po resete obvodu sa automat dostane do stavu *start*. Pokiaľ beží testovanie pamäte, automat čaká na platné údaje z bloku prepojenia blokov BIST a BIRA. Keď dostane adresu poruchy, prechádza do stavu *valid_data*.

V stave *valid_data* ako prvé automat skontroluje, či je daná porucha už opravená alebo zaznamenaná v bitmape. Ak áno, vracia sa do stavu *start* a čaká na ďalšiu poruchu. Ak nie, vykonajú sa nasledovné kontroly a akcie.

Ak sa jedná o novú poruchu (v bitmape nie je porucha s rovnakou riadkovou alebo stĺpcovou adresou) alebo o poruchu, ktorá má zhodnú riadkovú alebo stĺpcovú adresu s už zaznamenanou poruchou a nie je splnená podmienka nevyhnutnej opravy, automat prechádza do stavu *write*, kde túto poruchu zapíše do bitmapy. Následne sa vráti do stavu *start*.

Ak sa jedná o poruchu, ktorá má zhodnú riadkovú alebo stĺpcovú adresu s už zaznamenanou poruchou a je splnená podmienka nevyhnutnej opravy: V prípade, že sa

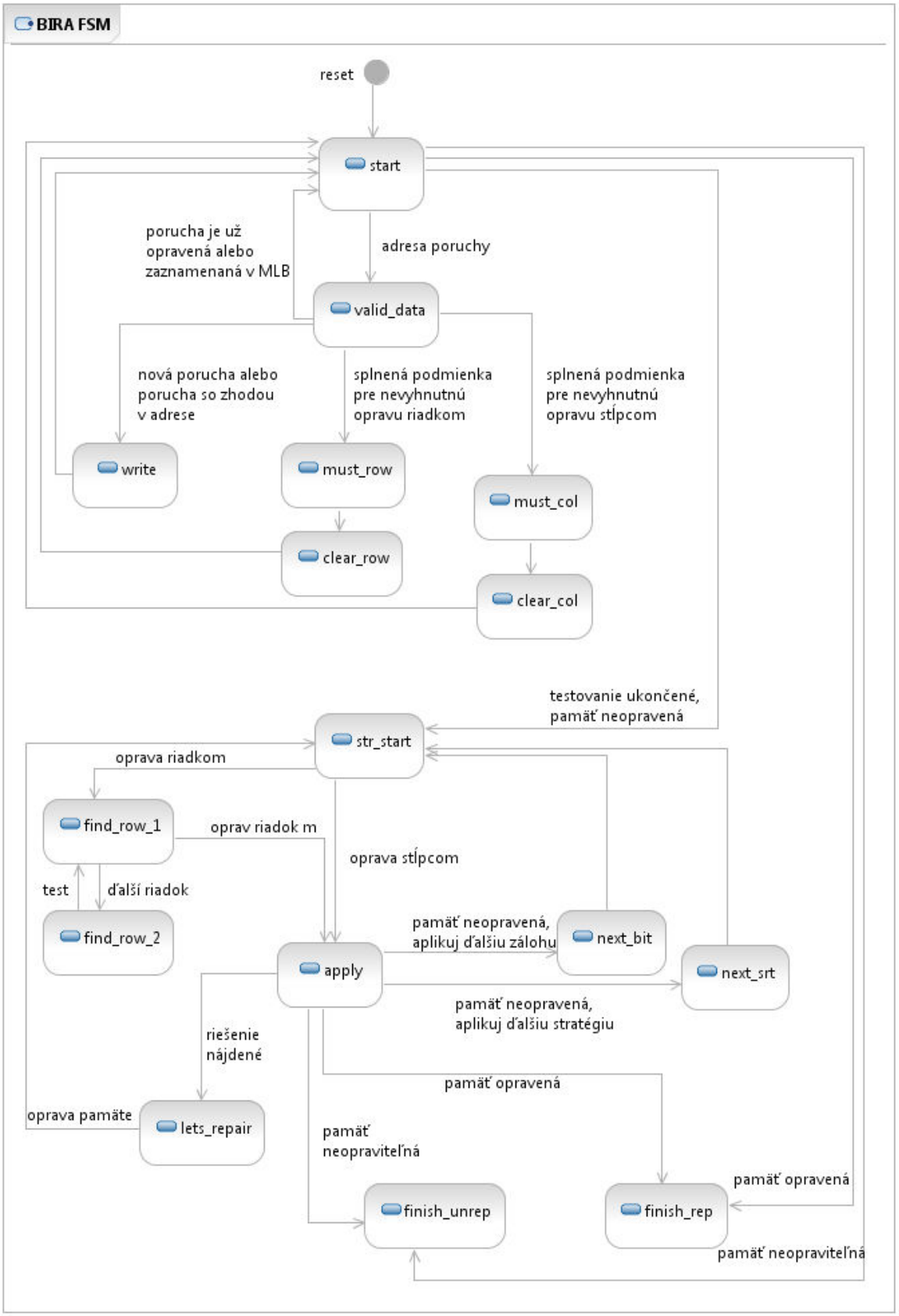
jedná o riadok, automat prechádza do stavu *must_row*, kde vykoná opravu záložným riadkom, následne prejde do stavu *clear_row*, kde vymaže záznamy z bitmapy na danom riadku a vráti sa do stavu *start*. V prípade, že sa jedná o stĺpec, vykoná sa obdobná akcia, ale s prechodmi cez stavy *must_col* a *clear_col*.

Ak sa počas testovania pamäte stane, že je detegovaný taký počet porúch, že pamäť nie je opraviteľná, prechádza automat do konečného stavu *finish_unrep*. Ak po ukončení testovania bitmapa neobsahuje záznamy o poruchách, prechádza automat do konečného stavu *finish_rep*, pamät je opravená. Ak po ukončení testovania bitmapa obsahuje záznamy o poruchách, automat prechádza do stavu *str_start*, v ktorom bude skúšať aplikovať opravné stratégie.

Pri prechode do stavu *str_start* má už automat k dispozícii prvú opravnú stratégiu. Začne ju aplikovať. Ak sa má použiť na opravu stĺpec, prechádza do stavu *apply*, v ktorom nastaví príslušné pole stĺpcového regista stratégie v bitmape. Ak sa má použiť na opravu riadok, nájde riadok v bitmape obsahujúci záznamy o danej poruche a prechádza do stavu *apply*, v ktorom nastaví príslušné pole riadkového regista stratégie v bitmape.

Zo stavu *apply* sú prechody nasledovné: Ak pamäť nie je opravená a nie sú ešte použité všetky dostupné zálohy, prechádza automat do stavu *next_bit*, kde zistí typ ďalšej zálohy a vráti sa do stavu *str_start*, z ktorého následne túto zálohu aplikuje spôsobom opísaným vyššie. Ak pamäť nie je opravená a sú použité všetky dostupné zálohy, prechádza automat do stavu *next_str*, kde prečíta ďalšiu stratégiu a vráti sa do stavu *str_start*, z ktorého následne túto stratégiu aplikuje spôsobom opísaným vyššie. Ak pamäť nie je opravená a sú použité všetky dostupné zálohy a vyskúšané všetky stratégie, prechádza automat do konečného stavu *finish_unrep*, pamäť je neopraviteľná.

Ak je pamäť po aplikácii zálohy v stave *apply* opravená, našlo sa riešenie, automat prechádza do stavu *lets_repair*, v ktorom nastaví príznak, aby sa táto oprava zapísala do registra vykonaných opráv a vráti sa do stavu *str_start*. Zo stavu *str_start* sa opakovaním rovnakého postupu aplikácie záloh zapíše úspešná oprava do registra vykonaných opráv. Ak je oprava vykonaná a zapísaná, prechádza automat do konečného stavu *finish_rep*, pamäť je opravená.



Obr. 4.10: Stavový diagram konečného stavového automatu

4.3 Prepojenie architektúr BIST a BIRA

Bloková schéma prepojenia architektúr BIST a BIRA sa nelíši od návrhu (kapitola 3.4). Pozostáva z pamäte pre uchovanie informácií o detegovaných poruchách, ktorá je realizovaná ako FIFO zásobník – údaje, ktoré sa zapíšu ako prvé, sa ako prvé aj spracujú. Kapacita bola zvolená na 16 záznamov. Nakoľko stĺpcová záloha má šírku iba 1 bit, obsahuje tento obvod aj konečný stavový automat, ktorý vykonáva funkciu analýzy syndrómu danej poruchy.

V tabuľke Tab. 4.3 je zapísaný konečný stavový automat typu mealy umiestnený v bloku prepojenia architektúr BIST a BIRA. Pre jednoduchosť sú zapísané len tie kombinácie vstupných signálov, pri ktorých dochádza k zmene stavu a tým aj hodnôt výstupných signálov. Výstupné signály nadobúdajú svoje hodnoty hneď pri splnení podmienky prechodu do príslušného stavu. Hodnota X znamená, že na hodnote daného bitu nezáleží.

Tab. 4.3: Konečný stavový automat v bloku prepojenie architektúr BIST a BIRA

	vstupné signály				výstupné signály		
stav	data_in	valid_bit	next_value	nasledujúci stav	data_out	valid_data	inc_buffer
start	1XXX	1	1	prvy	1000	1	0
	01XX	1	1	druhy	0100	1	0
	001X	1	1	treti	0010	1	0
	0001	1	1	stvrty	0001	1	0
prvy	1000	X	0	start	1000	0	1
	XXXX	X	0	M1	1000	0	0
M1	11XX	X	1	druhy	0100	1	0
	101X	X	1	treti	0010	1	0
	1001	X	1	stvrty	0001	1	0
druhy	X100	X	0	start	0100	0	1
	XXXX	X	0	M2	0100	0	0
M2	X11X	X	1	treti	0010	1	0
	X101	X	1	stvrty	0001	1	0
treti	XX10	X	0	start	0010	0	1
	XXXX	X	0	M3	0010	0	0
M3	XX11	X	1	stvrty	0001	1	0
stvrty	XXXX	X	0	start	0001	0	1

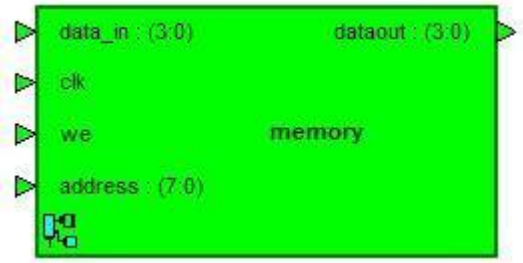
Význam jednotlivých signálov:

- *data_in* – signál obsahujúci syndróm z pamäte pre uchovanie informácií o poruchách (môže obsahovať hodnotu log. 1 na viacerých bitoch),
- *valid_bit* – signál indikujúci platnosť údajov prečítaných z pamäte pre uchovanie informácií o poruchách,
- *next_value* – signál z riadiacej BIRA jednotky indikujúci požiadavku na ďalší údaj,
- *data_out* – signál obsahujúci syndróm, ktorý má hodnotu log. 1 len na jednom bite,
- *valid_data* – signál indikujúci platnosť údajov pre riadiacu jednotku BIRA,
- *inc_buffer* – signál indikujúci požiadavku na ďalší údaj z pamäte pre uchovanie informácií o poruchách.

4.4 Testovaná pamäť

Testovaná pamäť má veľkosť 32 riadkov x 8 stĺpcov, šírka slova je 4 bity. Na obrázku Obr. 4.11 je zobrazený obvod pamäte z najvyššej úrovne abstrakcie. Obsahuje nasledovné vstupy a výstupy:

- *data_in* – hodnota na tomto vstupe bude počas operácie zápisu zapísaná do pamäte na adresu špecifikovanú vstupom *address*, šírka 4 bity – 4 bitové slovo pamäte,
- *clk* – hodinový signál pamäte,
- *we* – signál indikujúci operáciu zápisu (hodnota log. 1) alebo operáciu čítania (hodnota log. 0),
- *address* – určuje adresu, z ktorej sa bude čítať alebo na ktorú sa bude zapisovať, šírka 8 bitov – 32 riadkov (5 bitov) a 8 stĺpcov (3 bity),
- *dataout* – tento signál obsahuje prečítané údaje z pamäte, šírka 4 bity – 4 bitové slovo pamäte.



Obr. 4.11: Základná schéma testovanej pamäte

5 Overenie riešenia a experimentálne výsledky

5.1 Overenie riešenia

Pre overenie riešenia som zvolil 4 rôzne príklady:

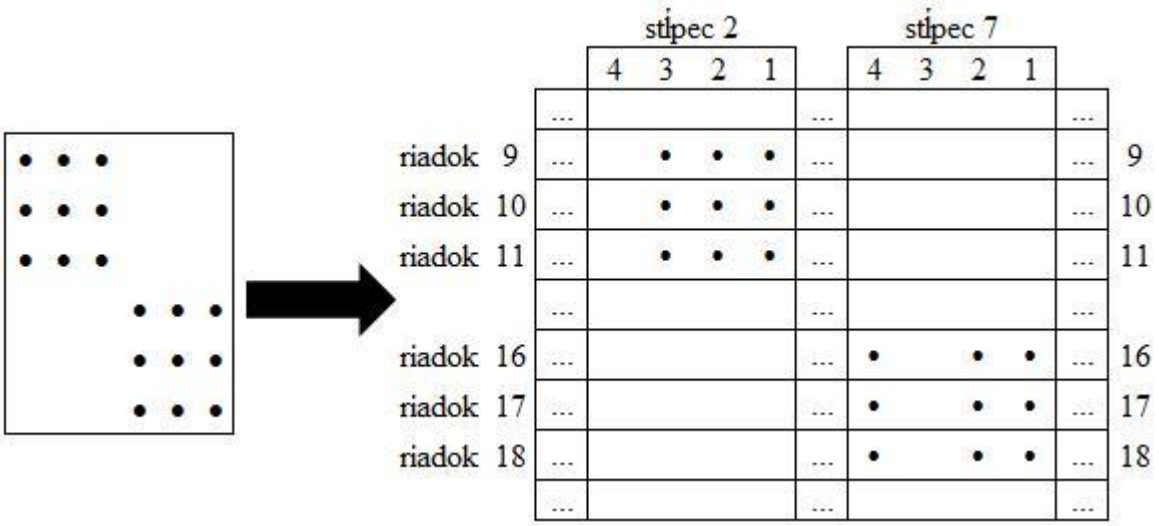
1. Príklad 1 – vo fáze nevyhnutnej opravy sa oprava nevykoná, na opravu je potrebných všetkých 6 záloh.
2. Príklad 2 – vo fáze nevyhnutnej opravy sa oprava nevykoná, existuje viacero možných riešení opravy.
3. Príklad 3 – vo fáze nevyhnutnej opravy sa oprava vykoná, no existuje len jedno riešenie opravy, ktoré je nutné nájsť vo fáze vyčerpávajúceho hľadania.
4. Príklad 4 – neexistuje riešenie opravy.

Pre obrázky zo simulácie v tejto kapitole platí:

- *col_00, col_01, col_10* – pamäte CAM, v ktorých sú zaznamenané opravy záložnými stĺpcami v registri vykonaných opráv,
- *row_00, row_01, row_10* – pamäte CAM, v ktorých sú zaznamenané opravy záložnými riadkami v registri vykonaných opráv,
- písmeno *A/N* pri signáli *valid_bit* každej CAM pamäte znamená platnosť záznamu, resp. použitie/nepoužitie riadkovej alebo stĺpcovej zálohy,
- číslo vedľa signálov *col_adr_x* a *q_x* je adresa zobrazená v desiatkovej sústave,
- hodnota signálu *syn_x* označuje bit slova, ktorý bol danou zálohou opravený (*syn_3* najviac významový bit, *syn_0* je najmenej významový bit).

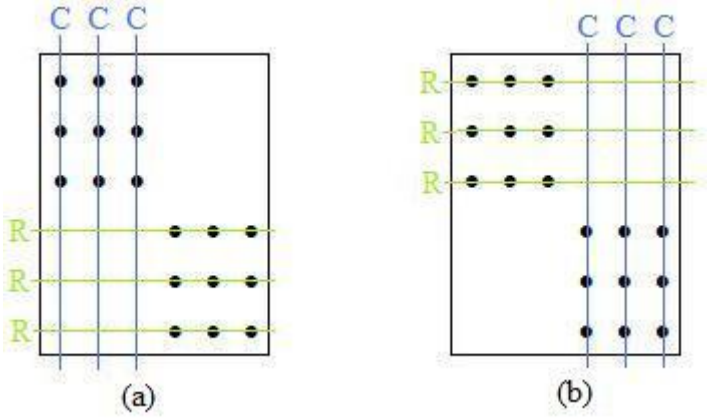
5.1.1 Príklad 1

Rozloženie porúch v pamäti spolu s ich adresami je schématicky znázornené na obrázku Obr. 5.1. Naľavo je zobrazený vzorec, ktorý vytvoria poruchy v bitmape. Toto rozloženie je zaujímavé z toho dôvodu, že niektoré z algoritmov opravy, napr. [15], ho pokladajú za neopraviteľné, aj keď v skutočnosti opraviteľné je.



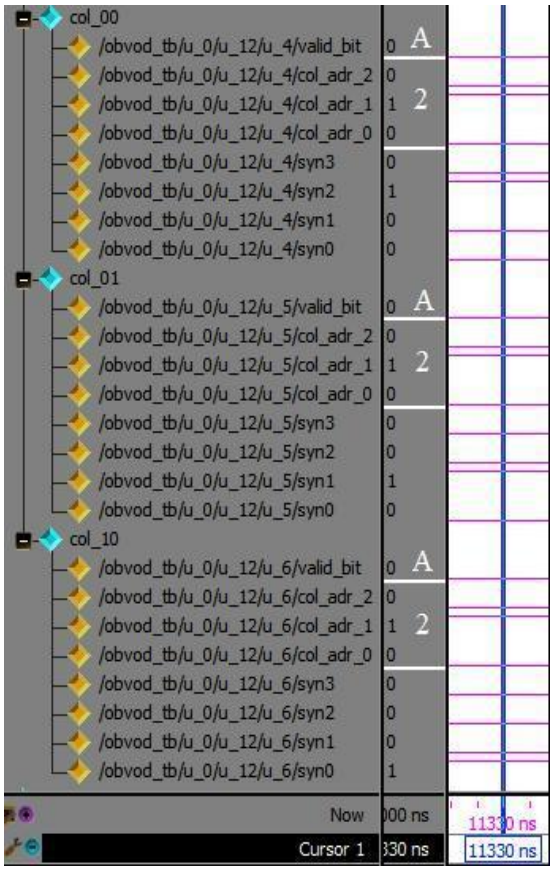
Obr. 5.1: Pr. 1 – rozloženie porúch

Existujú len 2 riešenia (zobrazené na obrázku Obr. 5.2), ktorými je možné tieto poruchy opraviť. Rozdiel je len v tom, či sa ako prvé použijú riadkové alebo stĺpcové zálohy. Výsledok možno ľahko predpovedať pri pohľade do obsahu generátora stratégie, kde je stratégia CCCRRR analyzovaná skôr, preto bude použitá pre opravu.

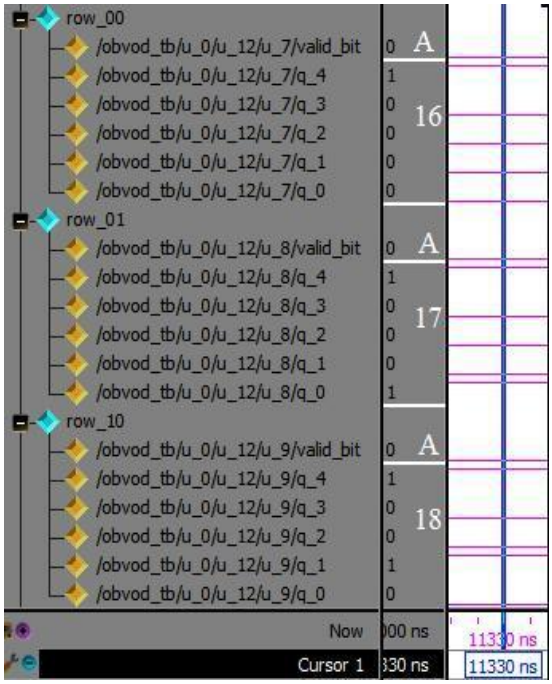


Obr. 5.2: Pr. 1 – možnosti opravy

Oprava pamäte bude vykonaná podľa možnosti (a) na obrázku Obr. 5.2. Na obrázkoch Obr. 5.3 a Obr. 5.4 je možné vidieť výsledok z registra vykonaných opráv. Sú použité všetky dostupné záložné prvky, poruchy v stĺpci 2 boli opravené použitím stĺpcových záloh a poruchy v riadkoch 16, 17 a 18 boli opravené použitím riadkových záloh. Oprava pamäte je v tomto prípade úspešná.



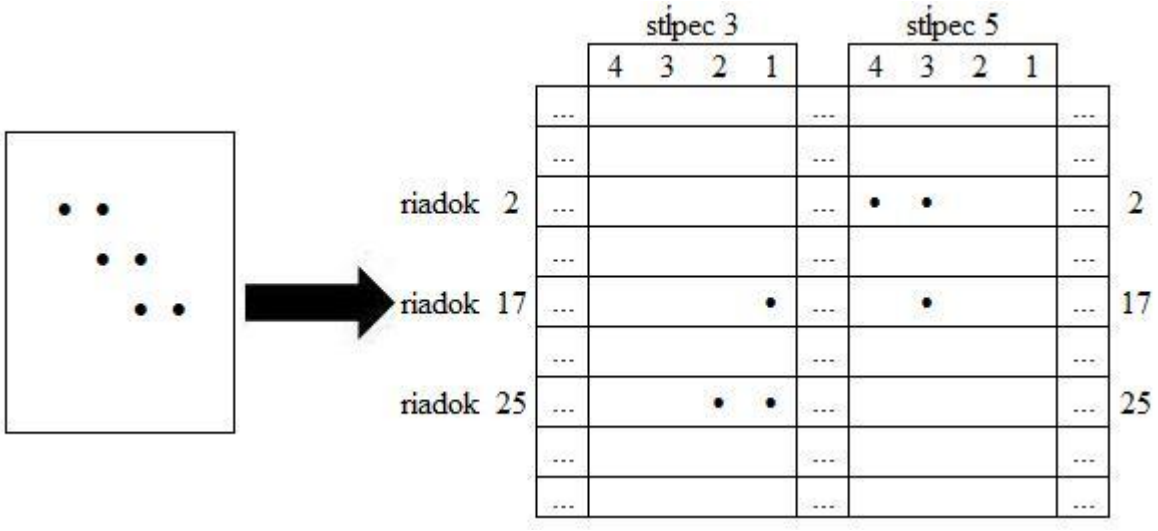
Obr. 5.3: Pr. 1 – obsah stĺpcovej CAM v RSR



Obr 5.4: Pr. 1 – obsah riadkovej CAM v RSR

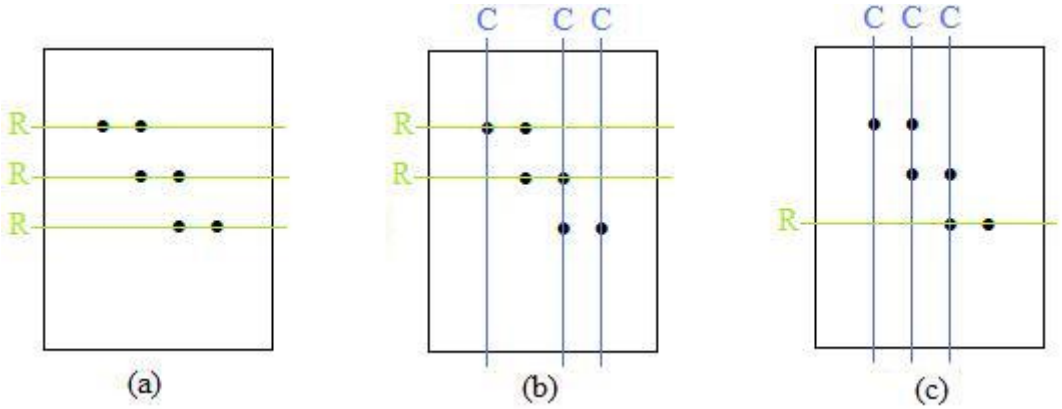
5.1.2 Príklad 2

Rozloženie porúch v pamäti spolu s ich adresami je schématicky znázornené na obrázku Obr. 5.5. Naľavo je zobrazený vzorec, ktorý vytvoria poruchy v bitmape. Toto rozloženie je zaujímavé z toho dôvodu, že niektoré algoritmy, napr. [15], na opravu pamäte s rozložením porúch tohto typu musia uplatniť pokročilé postupy opravy .



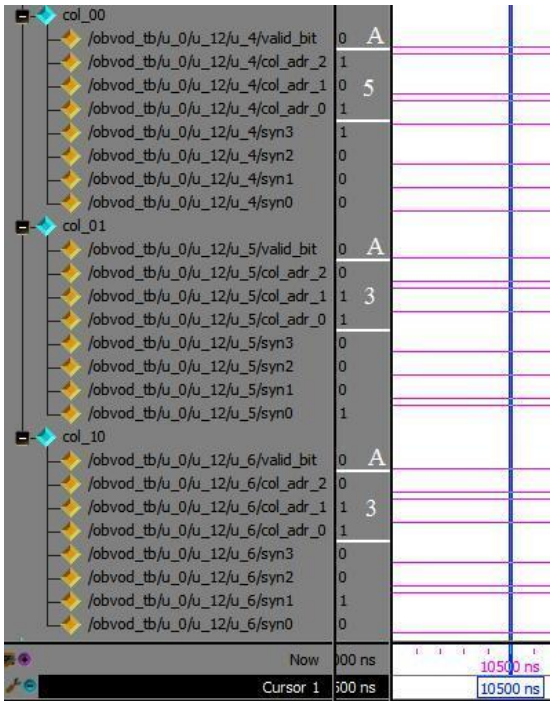
Obr. 5.5: Pr. 2 – rozloženie porúch

Existuje viacero riešení (niektoré sú zobrazené na obrázku Obr. 5.6), ktorými je možné tieto poruchy opraviť. V tomto prípade záleží len od toho, ktorá stratégia bude úspešná ako prvá, zhodou okolností je úspešná hneď prvá testovaná stratégia CRRCCR.

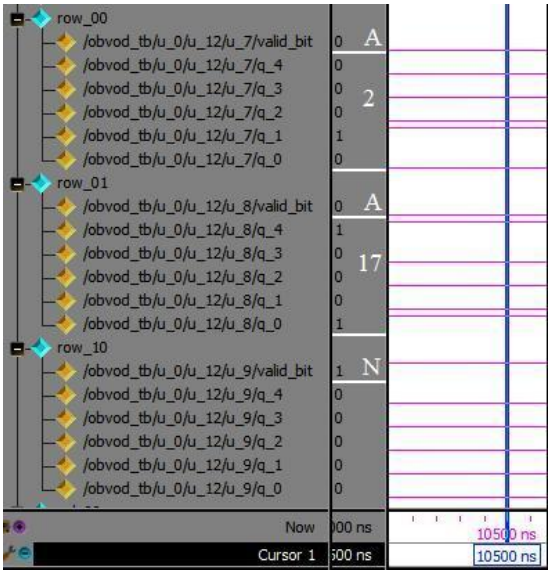


Obr. 5.6: Pr. 2 – možnosti opravy

Oprava pamäte bude vykonaná podľa možnosti (b) na obrázku Obr. 5.6. Na obrázkoch Obr. 5.7 a Obr. 5.8 je možné vidieť výsledok z registra vykonaných opráv. Sú použité 3 záložné stĺpce a 2 záložné riadky. Poruchy na adresách (2,5,S4), (25,3,S1) a (25,3,S2) boli opravené použitím stĺpcových záloh a poruchy v riadkoch 2 a 17 boli opravené použitím riadkových záloh. Oprava pamäte je v tomto prípade úspešná.



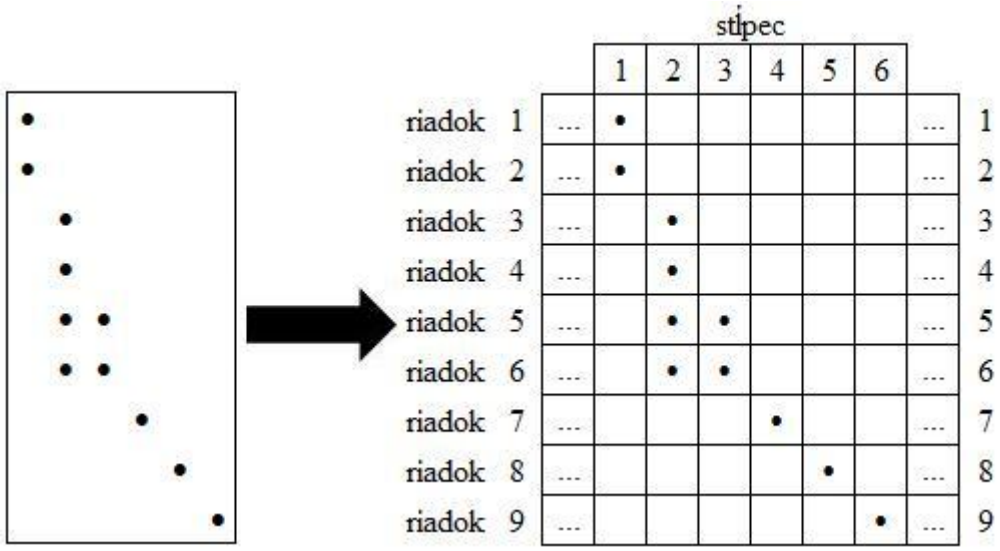
Obr. 5.7: Pr. 2 – obsah stĺpcovej CAM v RSR



Obr 5.8: Pr. 2 – obsah riadkovej CAM v RSR

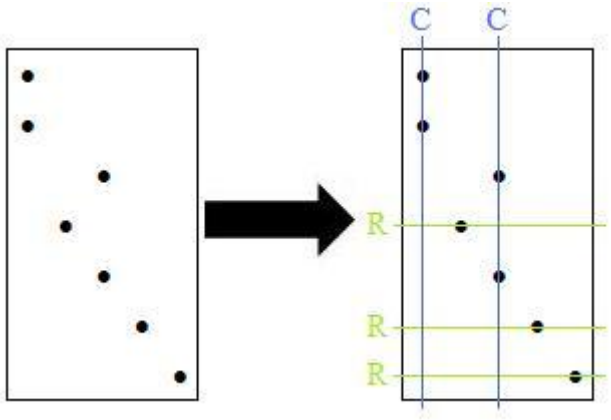
5.1.3 Príklad 3

Rozloženie porúch v pamäti spolu s ich adresami je schématicky znázornené na obrázku Obr. 5.9. Naľavo je zobrazený vzorec, ktorý by vytvorili poruchy v bitmape, pre stĺpec 2 je ale splnená podmienka nevyhnutnej opravy, preto pri testovaní stratégií v bitmape už tieto poruchy nebudú. Toto rozloženie je zaujímavé z toho dôvodu, že existuje len jedno riešenie.



Obr. 5.9: Pr. 3 – rozloženie porúch

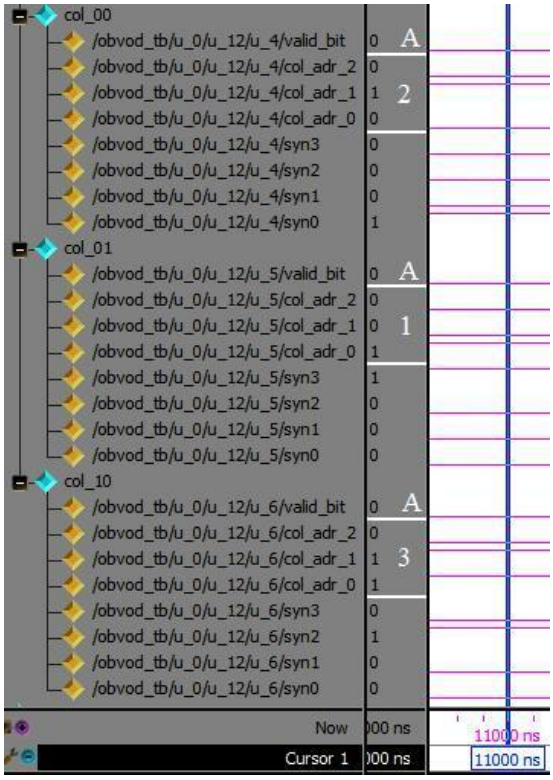
Existuje len jedno riešenie (zobrazené vpravo na obrázku Obr. 5.10). Naľavo na obrázku je zobrazené konečné rozloženie porúch v bitmape po ukončení testovania. Ako vidieť, je potrebné použiť stratégiu CRCRR.

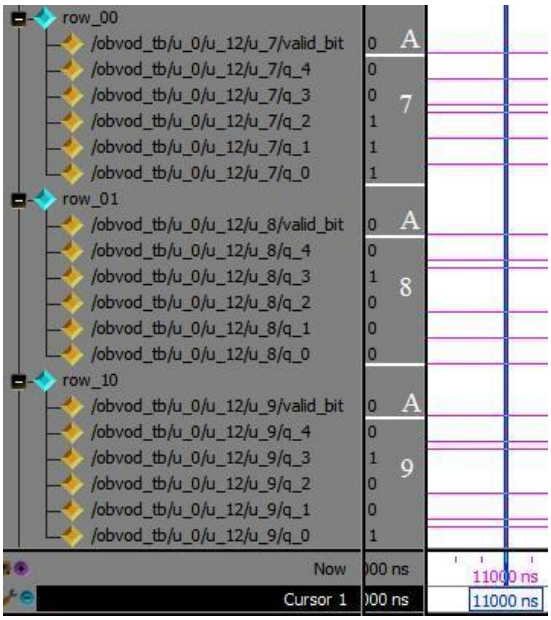


Obr. 5.10: Pr. 3 – možnosť opravy

Oprava pamäte bude vykonaná podľa zobrazenia na obrázku Obr. 5.10. Na obrázkoch Obr. 5.11 a Obr. 5.12 je možné vidieť výsledok z registra vykonaných opráv. Sú použité všetky dostupné záložné prvky. Poruchy v stĺpcoch 1, 2 a 3 boli opravené použitím stĺpcových

záloh a poruchy v riadkoch 7, 8 a 9 boli opravené použitím riadkových záloh. Oprava pamäte je v tomto prípade úspešná.



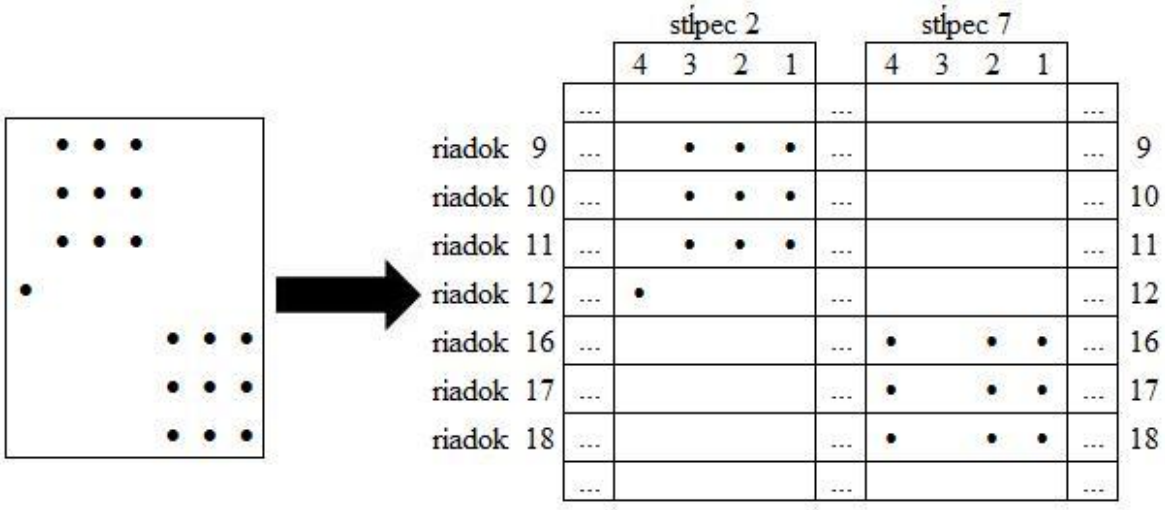


Obr 5.12: Pr. 3 – obsah riadkovej CAM v RSR

Obr. 5.11: Pr. 3 – obsah stĺpcovej CAM v RSR

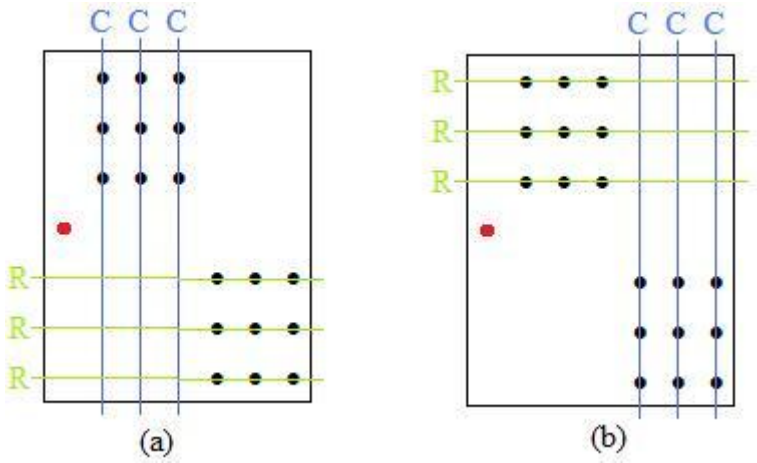
5.1.4 Príklad 4

Rozloženie porúch v pamäti spolu s ich adresami je schématicky znázornené na obrázku Obr. 5.13. Naľavo je zobrazený vzorec, ktorý vytvoria poruchy v bitmape. Toto rozloženie je zaujímavé z toho dôvodu, že riešenie opravy neexistuje.



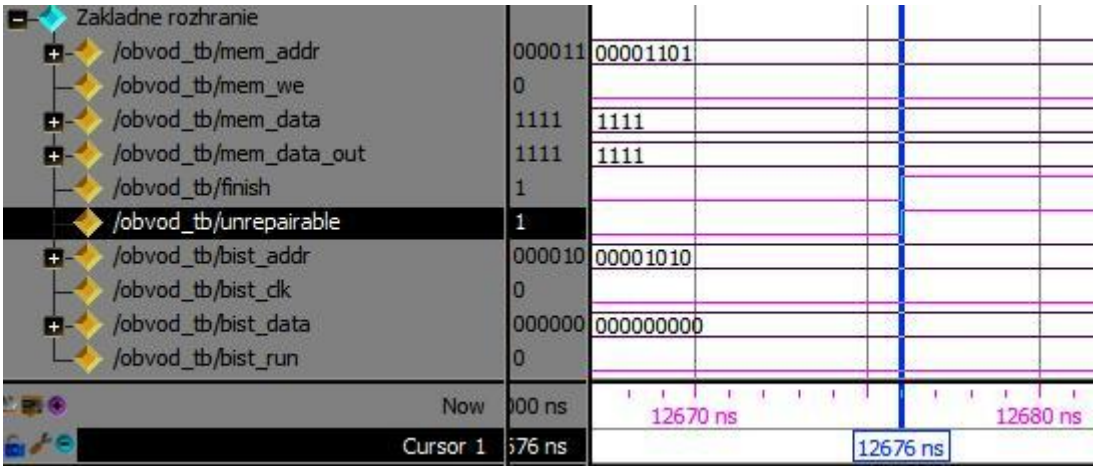
Obr. 5.13: Pr. 4 – rozloženie porúch

Skutočnosť, že riešenie opravy neexistuje, je demonštrovaná na obrázku Obr. 5.14. Aj pri maximálne efektívnom využití dostupných záložných prvkov nie je možné opraviť poruchu na adrese (12,2,S4) – zobrazená na obrázku červenou farbou.



Obr. 5.14: Pr. 4 – možnosti opravy

Po otestovaní všetkých stratégií a zistení skutočnosti, že oprava neexistuje, obvod nastaví spolu so signálom *finish* aj signál *unrepairable* na hodnotu log. 1, ako je zobrazené na obrázku Obr. 5.15. Pamäť nie je opraviteľná.



Obr. 5.15: Pr. 4 – Neúspešné ukončenie opravy pamäte

5.2 Experimentálne výsledky

5.2.1 Plocha potrebná pre architektúru opravy

V tabuľke Tab. 5.1 sa nachádzajú hodnoty počtov tranzistorov, ktoré sú potrebné na realizáciu architektúry samočinnej opravy pamäte určitej veľkosti pre daný počet záložných riadkov a záložných stĺpcov. Notácia *2R + 2C* znamená 2 záložné riadky a 2 záložné stĺpce. Hodnoty sú uvedené pre architektúru podporujúcu testy march do zložitosti 127N.

Tab. 5.1: Počty tranzistorov potrebných na realizáciu architektúry samočinnej opravy

pamäť / počet záloh	**1 kb**	**4 kb**	**16 kb**	**64 kb**	**256 kb**	**1024 kb**	**4096 kb**
2R + 2C	38574	42390	47190	54086	64662	83910	117718
2R + 3C	46228	50584	56116	64128	76588	99256	139556
2R + 4C	54975	59871	66135	75263	89607	115695	162487
2R + 5C	64383	69819	76815	87059	103287	132795	186079
3R + 3C	57715	62727	69107	78351	92811	119015	165923
3R + 4C	71578	77246	84474	94950	111410	141150	194666
3R + 5C	92026	98350	106426	118134	136594	169870	229994
4R + 4C	97072	103512	111704	123528	142104	175496	235736
4R + 5C	124005	131217	140373	153545	174237	211281	278245
5R + 5C	164910	173010	183246	197882	220806	261618	335422

Nižšie je uvedený príklad vypočítania hodnoty v tabuľke Tab. 5.1 pre pamäť o veľkosti 16 kb so 4 záložnými riadkami a 4 záložnými stĺpcami. Pre každý blok architektúry je uvedený počet tranzistorov, konečná hodnota je výsledkom ich súčtu:

- BIST blok: 10921
- Konečný stavový automat: 3178
- Generátor stratégie: 5261
- Bitmapa: 64838
- Register vykonaných opráv: 5492
- Prepojenie blokov BIST a BIRA: 14798
- Záložná pamäť: 7216

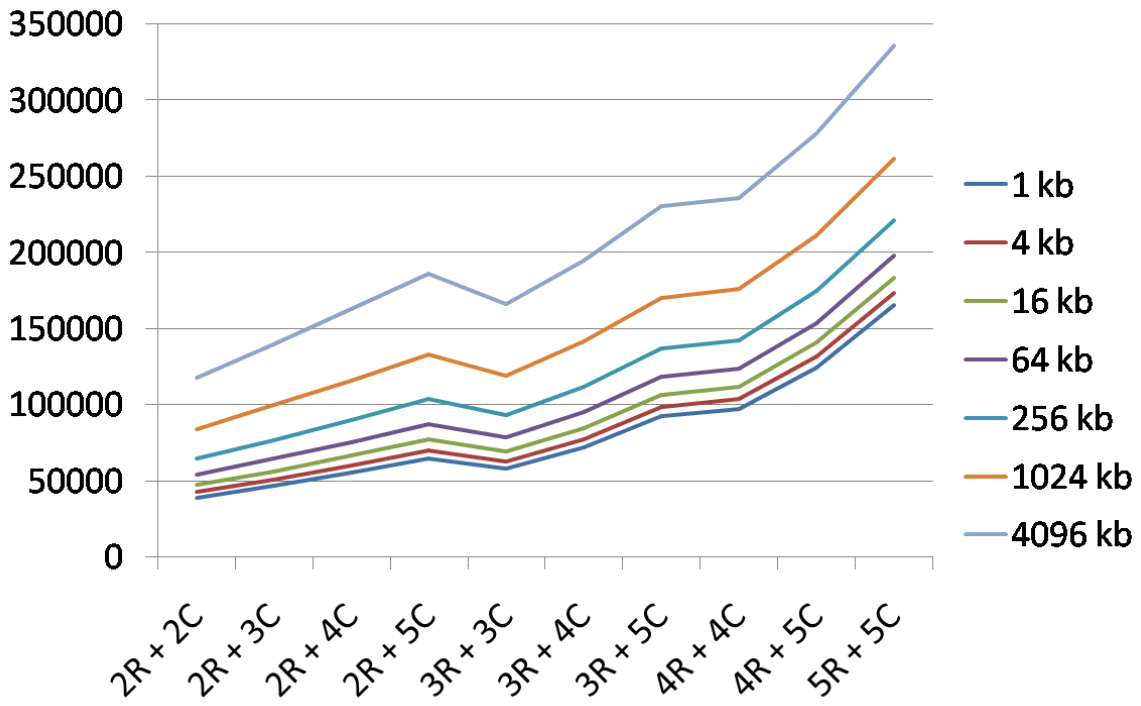
V tabuľke Tab. 5.2 sú uvedené všeobecne známe hodnoty počtov tranzistorov, ktoré sú potrebné na realizáciu jednotlivých logických členov a preklápacích obvodov. Všetky hodnoty sú uvedené pre 2-vstupové logické členy. Ak sa v obvode nachádzal logický člen s 3 a viac vstupmi, počet tranzistor bol vypočítaný nasledovne:

(počet vstupov – 1) * počet tranzistorov potrebných na realizáciu 2-vstup. log. člena

Tab. 5.2: Počty tranzistorov potrebných na realizáciu logických členov a prekl. obvodov

logický člen	**počet tranzistorov**
NOT	2
NAND	4
AND	6
NOR	4
OR	6
XOR	6
XNOR	8
2-vstup. MUX	6
D-FF	48
Latch D	16

Na obrázku Obr. 5.16 sú graficky zobrazené výsledky z tabuľky Tab. 5.1. Na osi y sa nachádzajú hodnoty počtov tranzistorov, na osi x sa nachádzajú konfigurácie záložných prvkov. V grafe je zobrazená závislosť pre každú veľkosť pamäte. Na grafe je možné pozorovať takmer lineárnu závislosť počtu tranzistorov potrebných na realizáciu obvodu samočinnej opravy od počtu záložných prvkov.



Obr. 5.16: Závislosť počtu tranzistorov potrebných na realizáciu architektúry BISR od počtu záložných prvkov

Nižšie je uvedený spôsob získania hodnoty počtu tranzistorov potrebných na realizáciu hlavnej pamäte (uvažuje sa pamäť typu SRAM). Táto hodnota je pre potreby porovnania plochy pridanej na čip architektúrou BISR dôležitá, nakoľko plocha na čipe potrebná na realizáciu architektúry BISR sa vyjadruje v percentuálnom pomere ku veľkosti hlavnej pamäte. Počet tranzistorov potrebný na realizáciu hlavnej pamäte je súčtom týchto hodnôt:

- počet bitov pamäte * 6 (pre realizáciu jednej bunky typu SRAM je potrebných 6 tranzistorov),
- riadkový adresný dekóder (počet tranzistorov záleží od počtu riadkov pamäte),
- stĺpcový adresný dekóder (počet tranzistorov záleží od počtu stĺpcov pamäte),
- zosilňovač signálu * 5 (zosilňovač signálu je potrebný pre každý stĺpec pamäte, pričom jeden zosilňovač je možné realizovať piatimi tranzistormi).

V tabuľke Tab 5.3 sú uvedené hodnoty počtov tranzistorov potrebných na realizáciu hlavných pamätí jednotlivých veľkostí spolu s organizáciou daných pamätí.

Tab. 5.3: Počty tranzistorov potrebných na realizáciu hlavnej pamäte

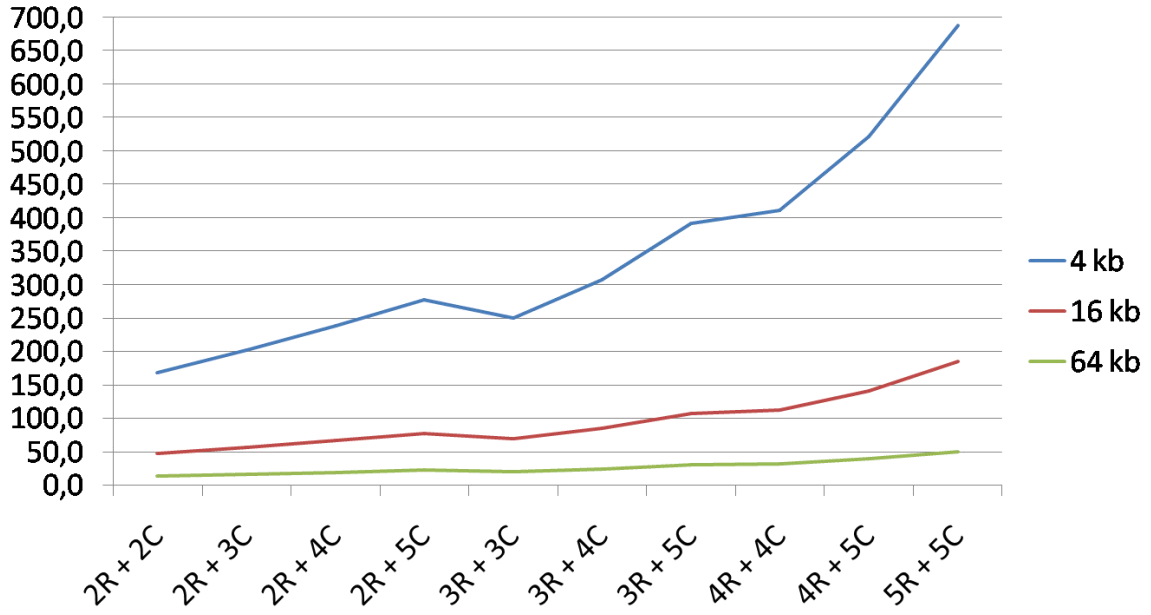
pamäť [kB]	**pamäť [kb]**	**počet bitov**	**počet riadkov**	**počet stĺpcov**	**šírka slova**	**počet tranzistorov**
0,125	1	1024	32	8	4	6456
0,5	4	4096	64	16	4	25140
2	16	16384	128	32	4	99336
8	64	65536	256	64	4	395276
32	256	262144	512	128	4	1576560
128	1024	1048576	1024	256	4	6299316
512	4096	4194304	2048	512	4	25180120

V tabuľke Tab. 5.4 sú uvedené hodnoty percentuálnych podielov plochy architektúry BISR ku ploche hlavnej pamäte. Inými slovami je to hodnota, o koľko percent sa zväčší plocha pamäte po pridaní architektúry BISR na čip. Ako možno vidieť v tabuľke, navrhnutá architektúra je z hľadiska plochy pridanej na čip efektívna pre pamäte väčších kapacít, najmä 256 kb, 1024 kb a 4096 kb. Všeobecne platí, čím väčšia pamäť, tým menšie percento plochy čipu je potrebné pre architektúru BISR pre jej opravu.

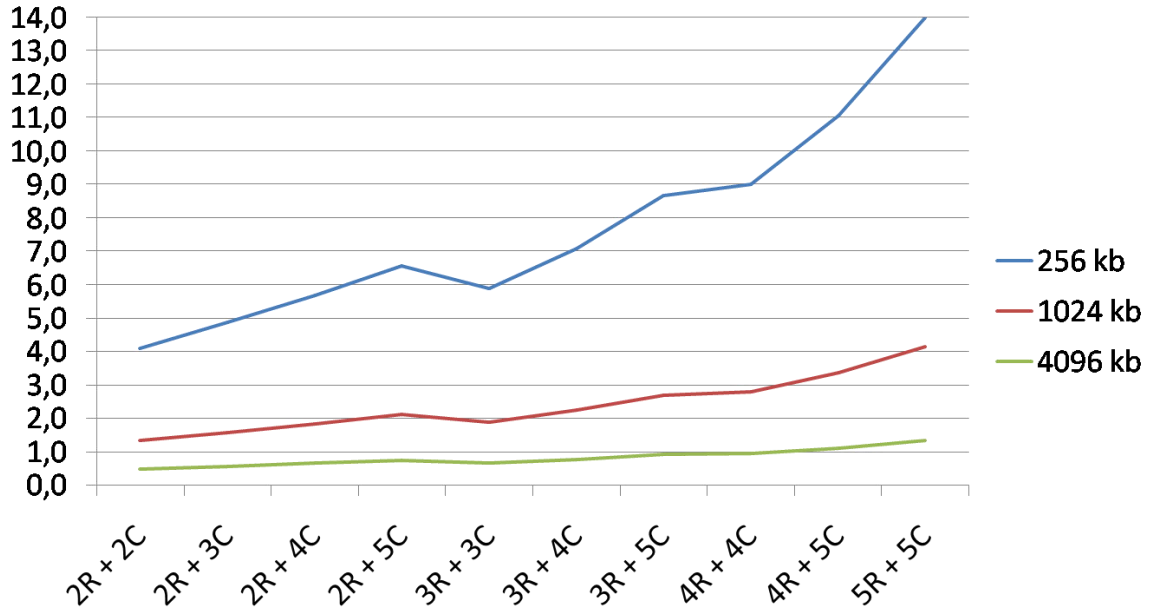
Tab. 5.4: Percentuálne podiely plochy architektúry BISR k ploche hlavnej pamäte

pamäť	**1 kb**	**4 kb**	**16 kb**	**64 kb**	**256 kb**	**1024 kb**	**4096 kb**
2R + 2C	597,5	168,6	47,5	13,7	4,1	1,3	0,5
2R + 3C	716,0	201,2	56,5	16,2	4,9	1,6	0,6
2R + 4C	851,5	238,2	66,6	19,0	5,7	1,8	0,6
2R + 5C	997,3	277,7	77,3	22,0	6,6	2,1	0,7
3R + 3C	894,0	249,5	69,6	19,8	5,9	1,9	0,7
3R + 4C	1108,7	307,3	85,0	24,0	7,1	2,2	0,8
3R + 5C	1425,4	391,2	107,1	29,9	8,7	2,7	0,9
4R + 4C	1503,6	411,7	112,5	31,3	9,0	2,8	0,9
4R + 5C	1920,8	521,9	141,3	38,8	11,1	3,4	1,1
5R + 5C	2554,4	688,2	184,5	50,1	14,0	4,2	1,3

V grafoch na obrázkoch 5.17 a 5.18 sú zobrazené výsledky uvedené v tabuľke Tab. 5.4 v grafickej podobe. Údaje z tabuľky boli rozdelené do dvoch grafov, prvý graf pre pamäte o veľkostiach 4 - 64 kb a druhý graf pre pamäte o veľkostiach 256 - 4096 kb, dôvodom rozdelenia je prehľadnosť grafov. Graf pre pamäť o veľkosti 1 kb nemá zmysel robiť, nakoľko pre pamäť tejto veľkosti nie je navrhnutá architektúra efektívna (rozsah približne 600 - 2500 percent pridanej plochy).



Obr. 5.17: Percentuálny podiel plochy pridanej na čip v závislosti od veľkosti hlavnej pamäte (4–64 kb)



Obr. 5.18: Percentuálny podiel plochy pridanej na čip v závislosti od veľkosti hlavnej pamäte (256–4096 kb)

5.2.2 Porovnanie s existujúcimi riešeniami

V tabuľke Tab. 5.5 je uvedené porovnanie percent pridanej plochy pre architektúru BISR nášeho riešenia s existujúcim riešením (Moťovský E., 2010) analyzovaným v kapitole 2.6. Dostupné sú výsledky pre hlavnú pamäť o veľkostiach 64, 256 a 1024 kb pri použití 5 záložných riadkov a 5 záložných stĺpcov. Na prvý pohľad sú výsledky druhého riešenia lepšie. Je však nutné podotknúť, že tieto výsledky boli získané spočítaním logických členov (a nie tranzistorov) potrebných na realizáciu danej architektúry BISR. Pre realizáciu logického člena však môže byť potrebných 6 tranzistorov (napr. log. člen AND), ale aj 48 tranzistorov (napr. D preklápací obvod).

Tab. 5.5: Porovnanie pridanej plochy s riešením analyzovaným v kapitole 2.6, 5 záložných riadkov aj stĺpcov

pamäť [kb]	**Moťovský E., 2010**	**naše výsledky**
64	28,92 %	50,1 %
256	8,21 %	14 %
1024	2,55 %	4,2 %

V tabuľke Tab. 5.6 sa nachádza porovnanie základných vlastností oboch architektúr BISR. Okrem vlastností zobrazených v tabuľke je ďalším obmedzením existujúceho riešenia aj fakt, že nepodporuje testy march, ktoré obsahujú vo svojich elementoch dvakrát tú istú operáciu bezprostredne za sebou. Toto obmedzenie vyplýva zo samotnej implementácie riešenia. Maximálna zložitosť podporovaného testu march je teoreticky 217N. Je to v prípade, že každý element testu march obsahuje práve 7 operácií. Počet elementov testu march je obmedzený na 31.

Tab. 5.6: Porovnanie základných vlastností s riešením analyzovaným v kapitole 2.6

	Moťovský E., 2010	**naše výsledky**
testovanie pamäte na jej operačnej frekvencii	nie	áno
max. počet operácií v elemente testu march	7	neobmedzený
max. zložitosť testu march	teoreticky 217N	127N
vždy nájde riešenie, pokiaľ existuje	nie	áno
podpora slovne orientovaných pamätí	nie	áno

V tabuľke Tab. 5.7 je uvedené porovnanie percent pridanej plochy pre architektúru BISR nášho riešenia s existujúcim riešením analyzovaným v kapitole 2.7.4. Dostupné sú výsledky pre hlavnú pamäť o veľkosti 512 kb pri použití 4 záložných riadkov a 4 záložných stĺpcov. Výsledky porovnávaného riešenia sú lepšie. Opäť je nutné podotknúť, že tieto výsledky boli získané spočítaním logických členov (a nie tranzistorov). Navyše, plocha porovnávaného riešenia je vypočítaná bez implementácie druhej fázy algotritmu – fázy vyčerpávajúceho hľadania.

Tab. 5.7: Porovnanie pridanej plochy s riešením analyzovaným v kapitole 2.7.4, 4 záložné riadky aj stĺpce

pamäť [kb]	**kapitola 2.7.4**	**naše výsledky**
512	4,1 %	5,0 %

V tabuľke Tab. 5.8 sa nachádza porovnanie základných vlastností oboch architektúr BISR. Porovnávaná architektúra je jednoznačne lepšia z hľadiska podpory pamätí so šírkou slova až 64 bitov. Naša architektúra je implementovaná pre pamäte so šírkou slova 4 bity a každé rozšírenie slova by znamenalo značný nárast na ploche nášho riešenia. Porovnávaná architektúra navyše neuvádza bližšiu špecifikáciu testovacieho algoritmu pamäte, nie je teda možné povedať, či sa jedná o algoritmus, ktorý možno modifikovať alebo o pevne daný algoritmus. Takisto je otázne, či je porovnávaná architektúra schopná nájsť riešenie opravy pamäte, ak toto riešenie existuje. Z opisu a princípu fungovania algoritmu vyplýva, že architektúra by mala byť vždy schopná opraviť pamäť, ktorá je opraviteľná s daným počtom záloh, táto skutočnosť však v článku jednoznačne uvedená nie je.

Tab. 5.8: Porovnanie základných vlastností s riešením analyzovaným v kapitole 2.7.4

	kapitola 2.7.4	**naše výsledky**
testovanie pamäte na jej operačnej frekvencii	nie	áno
max. počet operácií v elemente testu march	neznámy	neobmedzený
max. zložitosť testu march	neznáma	127N
vždy nájde riešenie, pokiaľ existuje	pravdepodobne áno	áno
podpora slovne orientovaných pamätí	áno	áno
šírka slova pamäte [bit]	64	4

V tabuľke Tab. 5.9 je uvedené porovnanie percent pridanej plochy pre architektúru BISR nášeho riešenia a architektúru BIRA existujúceho riešenia analyzovaného v kapitole 2.7.5. Dostupné sú výsledky pre hlavnú pamäť o veľkosti 512 kb pri použití 2 záložných riadkov a 2 záložných stĺpcov. Výsledky nášeho riešenia sú lepšie. Navyše, plocha porovnávaného riešenia je vypočítaná bez implementácie BIST bloku.

Tab. 5.9: Porovnanie pridanej plochy s riešením analyzovaným v kapitole 2.7.5, 2 záložné riadky aj stĺpce

pamäť [kb]	**kapitola 2.7.5**	**naše výsledky**
512	2,9 %	2,3 %

V tabuľke Tab. 5.10 sa nachádza porovnanie základných vlastností oboch architektúr. Nakoľko porovnávaná architektúra je len BIRA a nie celková architektúra BISR, nemá zmysel porovnávať vlastnosti BIST bloku. Jednoznačným kladom porovnávanej architektúry je podpora pamätí so šírkou slova až 64 bitov. Naša architektúra je implementovaná pre pamäte so šírkou slova 4 bity. Podstatným nedostatkom porovnávanej architektúry je, že nie vždy nájde riešenie opravy pamäte, aj keď toto riešenie existuje.

Tab. 5.10: Porovnanie základných vlastností s riešením analyzovaným v kapitole 2.7.5

	kapitola 2.7.5	**naše výsledky**
počet zbehnutí testu	1	1
počet analyzátorov opravy	1	1
bitmapa	1-D	2-D
vždy nájde riešenie, pokiaľ existuje	nie	áno
podpora slovne orientovaných pamätí	áno	áno
šírka slova pamäte [bit]	64	4

5.2.3 Výsledky z časového hľadiska

Čas potrebný na otestovanie pamäte je vzhľadom na použitie testu march lineárne závislý od veľkosti pamäte a závisí teda od zložitosti zvoleného testu march. Testovanie pamäte vždy trvá <*počet_bitov_pamäte*> hodinových cyklov plus 2 hodinové cykly potrebné na reset architektúry BISR.

Časová náročnosť opravy pamäte je v prípade našej implementácie premenlivá veličina a čas potrebný na analýzu opravy pamäte a samotnú opravu pamäte nie je možné vyjadriť vzorcom. Je to z toho dôvodu, že algoritmus pri analýze opravy pamäte používa na detekciu nutnosti opravy signál indikujúci chybu v danom stĺpci bitmapy. Ak sa podľa stratégie opravy má na opravu poruchy použiť záložný riadok, musí algoritmus nájsť v danom stĺpci bitmapy riadok, v ktorom je zapísaná porucha. Tento úkon môže trvať pri rozmere bitmapy 12 riadkov od jedného hodinového cyklu (porucha je zapísaná hneď v prvom riadku bitmapy v danom stĺpci) až po 23 hodinových cyklov (porucha je zapísaná v poslednom riadku bitmapy v danom stĺpci). Celkový čas opravy nezávisí len od počtu porúch v bitmape a počtu stratégií, ktoré sa majú odskúšať, ale aj od konkrétneho rozloženia porúch v bitmape.

V prípade, že všetky detegované poruchy sa opravia ešte počas testovania tak, že sa splní podmienka nevyhnutnej opravy, po skončení testovania je bitmapa prázdna, už nie je čo opravovať. V tom prípade je potrebných 0 hodinových cyklov, pamäť je okamžite označená ako opravená. Oprava pamäte si vyžaduje čas len pri nutnosti vykonať druhú fázu algoritmu opravy – fázu vyčerpávajúceho hľadania. Pre ilustráciu uvádzame počet hodinových cyklov potrebných na opravu pamäte podľa jednotlivých príkladov uvedených v podkapitolách kapitoly 5.1:

- príklad 1: 263 hodinových cyklov, 18 porúch v bitmape, úspešná stratégia č. 7,
- príklad 2: 37 hodinových cyklov, 6 porúch v bitmape, úspešná stratégia č. 1,
- príklad 3: 121 hodinových cyklov, 7 porúch v bitmape, úspešná stratégia č. 2,
- príklad 4: 600 hodinových cyklov, 18 porúch v bitmape, neopraviteľné.

6 Zhodnotenie

V prvom semestri sme sa v rámci analýzy zamerali na testovanie pamätí, modely porúch v pamätiach a moderné prístupy k riešeniu implementácie architektúry BISR. Analýzou vybraných prístupov k riešeniam architektúr BIST a BIRA sme si urobili podrobný prehľad o dnešných požiadavkách na tieto architektúry. Na základe získaných vedomostí sme urobili prvotný návrh nášho riešenia.

Počas druhého semestra sme sa venovali špecifikácii požiadaviek a podrobnejšiemu návrhu celkovej architektúry opravy pamätí RAM. Pre návrh a následnú implementáciu architektúry BIST bol zvolený prístup, ktorý umožňuje voľbu testu march takým spôsobom, že používateľ si je schopný tento test uložiť do pamäte inštrukcií v predpísanom formáte a následne týmto testom otestovať hlavnú pamäť. Veľkou výhodou je neobmedzený počet operácii v elemente testu march. Pre návrh a následnú implementáciu architektúry BIRA bol zvolený prístup, ktorý bol navrhnutý pre slovne orientované pamäte, používa 2-D zálohy a pokiaľ existuje riešenie opravy pamäte s daným počtom záloh, zaručene ho nájde.

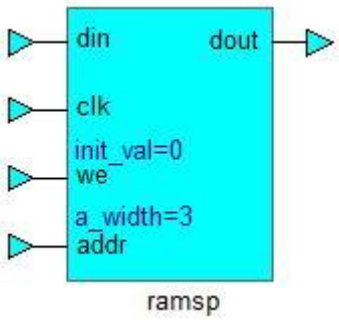
Funkčnosť riešenia bola overená na pamäti o veľkosti 32 x 32 bitov (organizácia 8 riadkov, 4 stĺpce, slovo o šírke 4 bity), s použitím 3 záložných riadkov a 3 záložných stĺpcov. Tento počet záložných prvkov bol zvolený vzhľadom na implementáciu bitmapy, ktorej veľkosť je v tomto prípade 12 x 12 bitov, čo bolo z pohľadu časovej zložitosti jej implementácie prijateľné. 2-D zálohy (riadkové a stĺpcové) boli zvolené z toho dôvodu, že sú to najpoužívanejšie zálohy v architektúrach opravy pamäte a algoritmy vďaka nim dosahujú vysokú úspešnosť v porovnaní s algoritmami využívajúcimi iba 1-D zálohy.

Výsledkom práce je plne funkčná architektúra pre samočinné testovanie a opravu vnorených pamätí RAM. Spĺňa všetky moderné požiadavky, najdôležitejšie sú jedno zbehnutie testu, jeden analyzátor možných riešení opravy pamäte a schopnosť nájsť riešenie opravy, pokiaľ toto riešenie existuje. Plocha pridaná na čipe bola vypočítaná ako počet tranzistorov potrebných na implementáciu architektúry. Následne bola táto plocha vyhodnotená ako podiel z plochy hlavnej pamäte v percentách a tieto výsledky sa porovnali s existujúcimi riešeniami, kde naše riešenie obstálo veľmi dobre. Pridanie architektúry opravy pamäte na čip si vyžaduje pridanie dvoch vstupov pre zbernice (8-bitovú a 9-bitovú), jeden vstup pre 1-bitový signál a 2 výstupy pre 1-bitový signál.

Príloha A: Technická dokumentácia

A.1 Testovaná pamäť

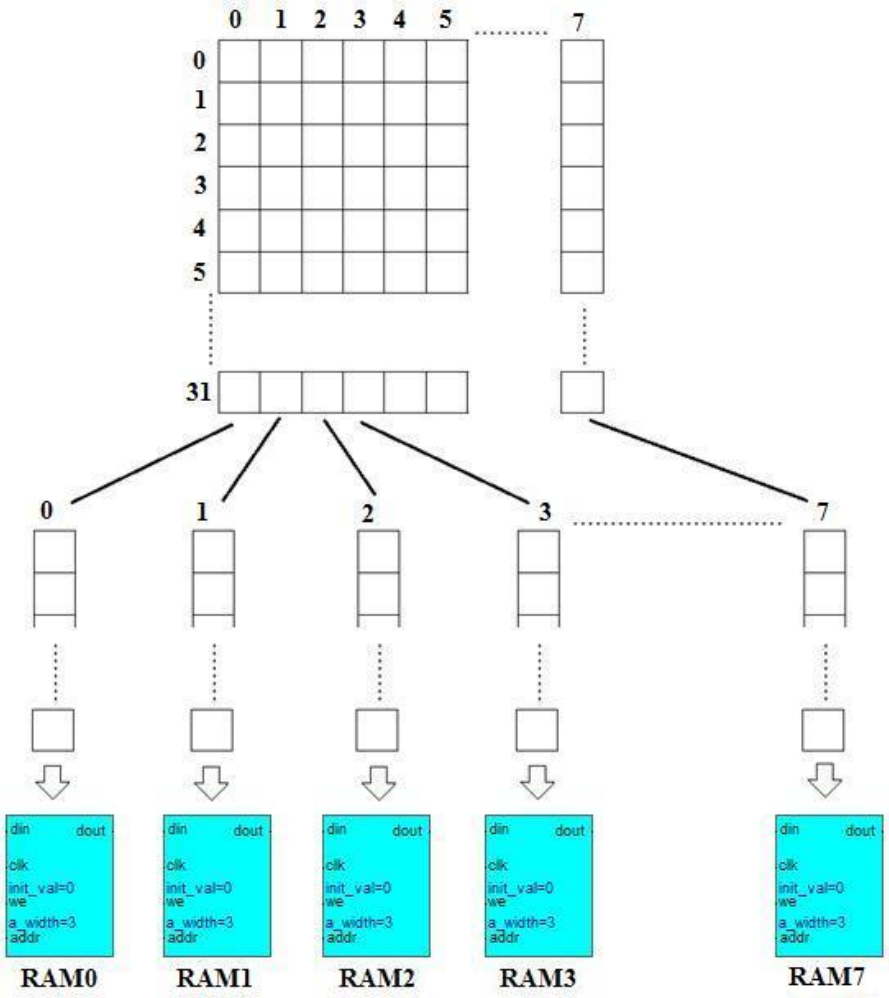
Základné rozhranie je opísané v kapitole 4.4. Pre implementáciu bol vybraný komponent knižnice Moduleware s názvom *RAM(synthesizable)*, teda syntetizovateľná pamäť RAM. Tento komponent je zobrazený na obrázku Obr. A.1. Vstupné a výstupné signály sú identické s pamäťou, signál *din* je ekvivalentom signálu *data_in*, signál *dout* je ekvivalentom signálu *dataout* a signál *addr* je ekvivalentom signálu *address*.



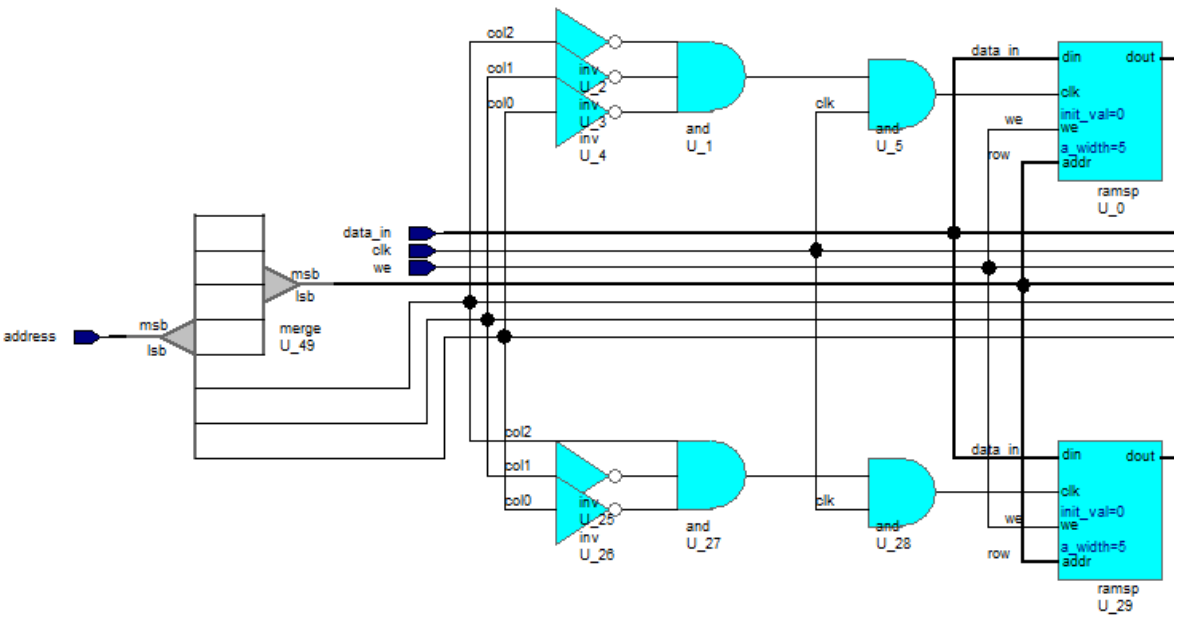
Obr. A.1: Komponent *RAM(synthesizable)* knižnice Moduleware

Týmto komponentom sa dá modelovať jeden stĺpec pamäte o ľubovoľnom počte riadkov, nie však pamäť obsahujúca viacero stĺpcov. Preto bolo potrebné tieto komponenty správne zapojiť, aby sme získali želanú pamäť o veľkosti 32 riadkov x 8 stĺpcov. Principiálne zapojenie týchto komponentov je zobrazené na obrázku Obr. A.2.

Spôsob, akým som dosiahol správne adresovanie jednotlivých komponentov a tým správne fungovanie celej pamäte je zobrazený na obrázku Obr. A.3.

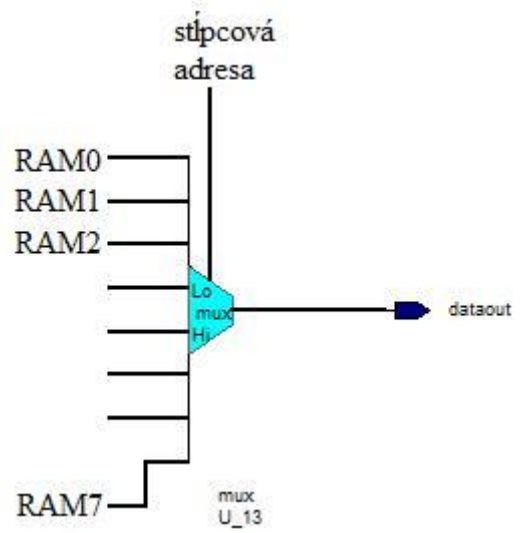


Obr. A.2: Princíp zapojenia komponentov RAM



Obr. A.3: Zapojenie komponentov RAM

Vstupné signály *data_in*, *clk* a *we* sú privedené priamo na vstupy každého komponentu RAM. Takisto je privedená na vstup každého komponentu aj 5-bitová adresná zbernica určujúca riadkovú adresu (na vstup *addr* komponentu). Na obrázku je vidieť 2 komponenty RAM, pred každým je kombinačná logika. Táto logika slúži ako aktivácia (ang. chip-select) adresovaného komponentu. Na výstupe prvého logického člena AND bude signál log. 1 len v prípade, že stĺpcová adresa adresuje daný komponent. Napríklad horná pamäť RAM je aktivovaná pri stĺpcovej adrese „000“, je to 0. stĺpec pamäte. Dolná pamäť RAM je aktivovaná pri stĺpcovej adrese „100“, je to 4. stĺpec pamäte. Druhý AND slúži na prešírenie hodinového signálu a vykonanie požadovanej operácie len na adresovanom komponente. Výstupný signál každého komponentu (*dout*) je privedený na multiplexor, ktorý podľa stĺpcovej adresy prešíri na celkový výstup *dataout* signál zo správneho komponentu RAM (obrázok Obr. A.4)



Obr. A.4: Zapojenie výstupu pamäte RAM

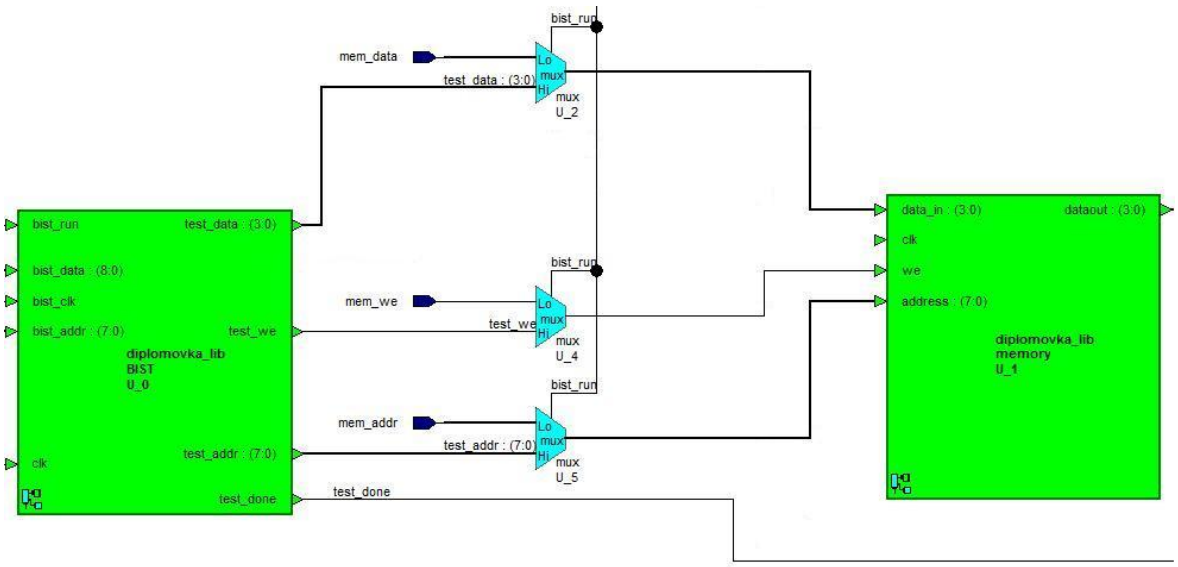
A.2 Architektúra BIST

A.2.1 Funkčný a testovací režim pamäte

Na obrázku Obr. A.5 je zobrazené použité zapojenie BIST obvodu s testovanou pamäťou, ktoré umožňuje používať pamäť v testovacom alebo funkčnom režime.

Vľavo je blok BIST, vpravo je blok pamäte. 3 multiplexory medzi týmito blokmi vyberajú vstupné signály pre pamäť podľa toho, či sa nachádza v testovacom režime (signál *bist_run* = ‘1’) alebo vo funkčnom režime (signál *bist_run* = ‘0’). V testovacom režime je pamäť

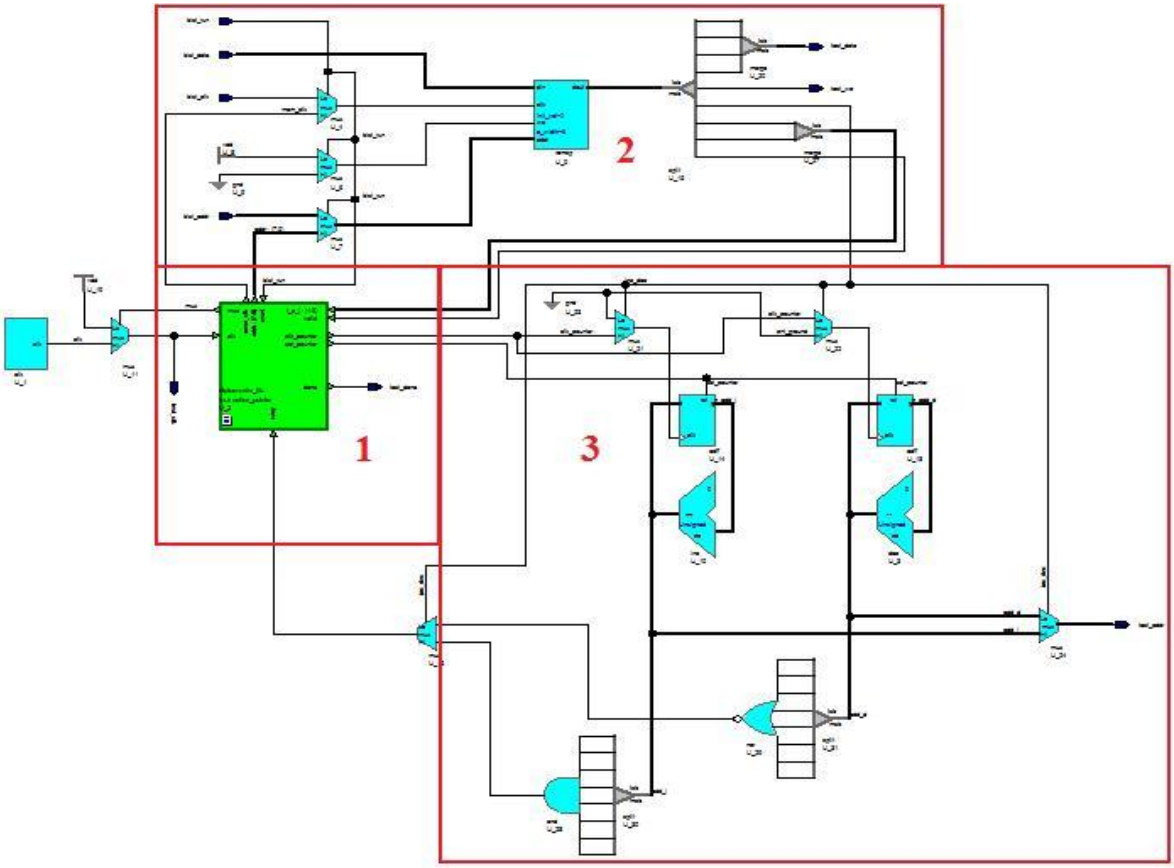
ovládaná signálmi, ktoré generuje BIST obvod. Vo funkčnom režime je pamäť ovládaná používateľskými vstupnými signálmi.



Obr. A.5: Zapojenie BIST obvodu a testovanej pamäte

A.2.2 BIST obvod

Samotný BIST obvod možno logicky rozdeliť na 3 bloky (obrázok Obr. A.6).



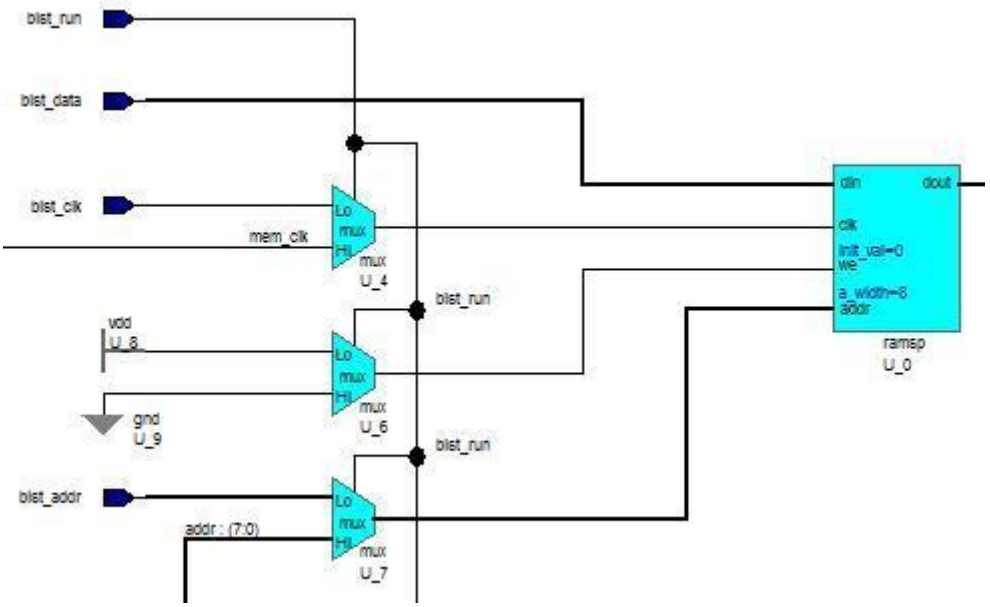
Obr. A.6: Logické rozdelenie BIST obvodu na 3 bloky

Jednotlivé bloky BIST obvodu:

- 1 – Riadiaca jednotka testovania,
- 2 – Pamäť inštrukcií,
- 3 – Generátor adries.

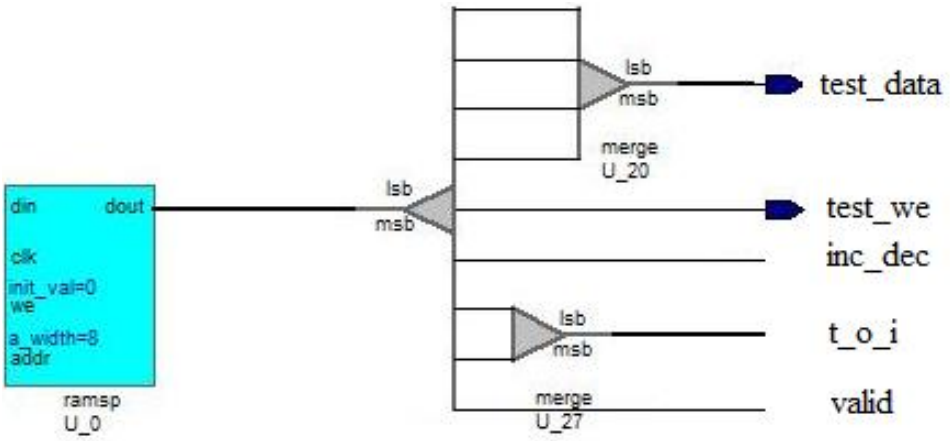
Pamäť inštrukcií

Pamäť inštrukcií je modelovaná komponentom knižnice Moduleware *RAM(synthesizable)*, rovnako ako stĺpce testovanej pamäte. Na obrázku Obr. A.7 je zobrazené jej zapojenie.



Obr. A.7: Zapojenie vstupov pamäte inštrukcií

Obsah pamäte inštrukcií možno meniť pomocou externých signálov. Na vstup *din* pamäte inštrukcií je priamo privedený externý signál *bist_data*. Ďalšie signály sú už multiplexované, buď sú riadené z externých vstupov (v prípade, že sa netestuje, *bist_run* = ‘0’) alebo riadiacou jednotkou BIST (v prípade, že prebieha testovanie, *bist_run* = ‘1’). Adresná zbernica je momentálne nastavená na šírku 8 bitov – maximálne 256 záznamov. Zapojenie výstupu pamäte inštrukcií je zobrazené na obrázku Obr. A.8.

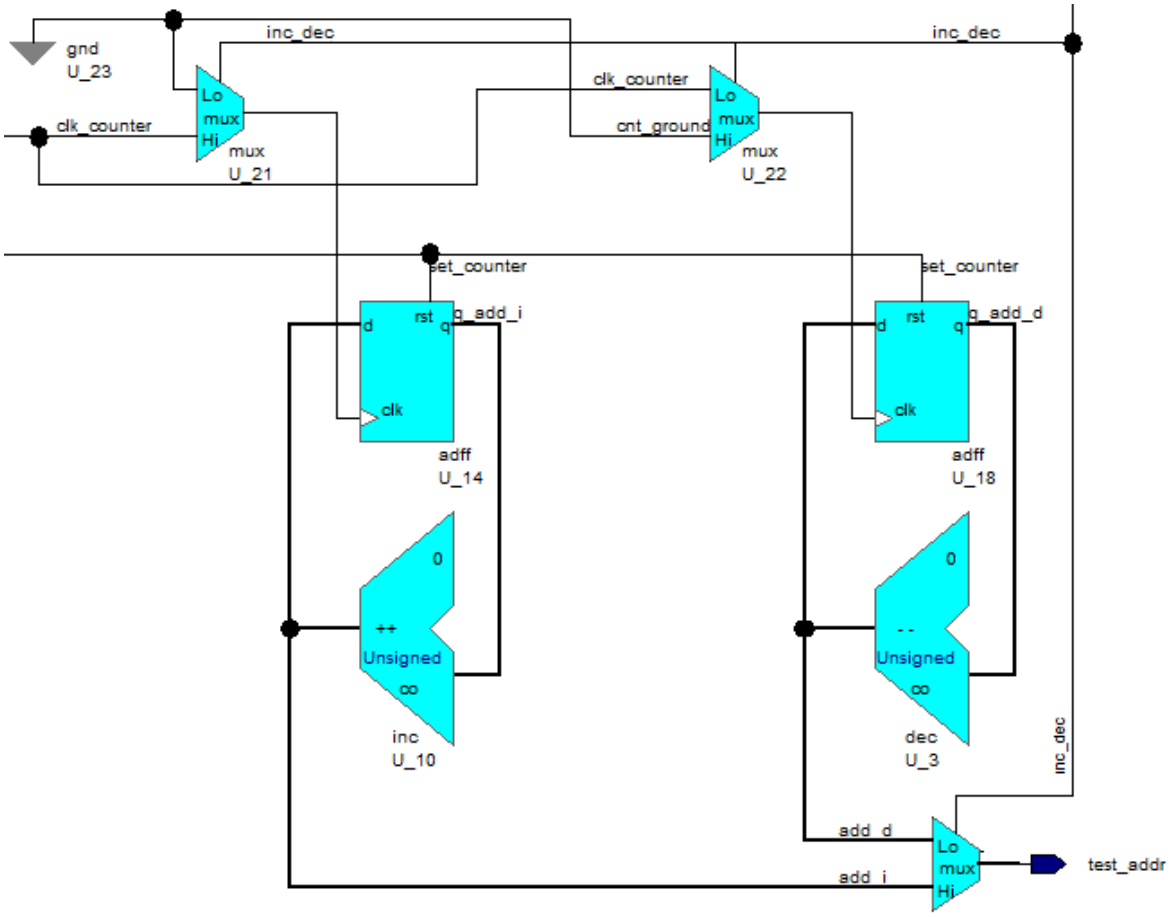


Obr. A.8: Zapojenie výstupu pamäte inštrukcií

Ako vidieť, 9-bitová údajová zbernica je rozdelená na jednotlivé bity. Horné 4 bity predstavujú testovaciu vzorku, netreba ich extra spracovávať, sú spojené do 4-bitovej zbernice a vyvedené priamo na výstup BIST obvodu. Takisto piaty bit zhora nie je potrebné zvlášť spracovávať, predstavuje nastavenie signálu *we* testovanej pamäte. Je vyvedený priamo na výstup BIST obvodu. Šiesty bit zhora (signál *inc_dec*) určuje smer adresovania pamäte, tento signál je použitý na ovládanie multiplexorov v generátore adries. Siedmy a ôsmy bit kódujú druh elementu, sú spojené do zbernice a privedené na vstup *t_o_i* riadiacej BIST jednotky. Deviaty bit indikuje platnosť prečítaného záznamu a je privedený na vstup *valid* riadiacej BIST jednotky.

Generátor adries

Schéma zapojenia generátora adries je zobrazená na obrázku Obr. A.9. Funkciu generátora adries vykonáva obvod inkrementer (vľavo) alebo dekrementer (vpravo) spojený s logickým D preklápacím obvodom do slučky. Ovládané sú z riadiacej BIST jednotky pomocou signálov *set_counter* (reset obvodov) a *clk_counter* (inkrementácia alebo dekrementácia adresy). Signál *clk_counter* sa prešíri do správneho obvodu pomocou multiplexorov ovládaných bitom mikroinštrukcie, ktorý údava smer adresovania. Adresný signál (*test_addr*), ktorý sa prešíri na výstup BIST obvodu je určený multiplexorom ovládaným rovnako ako zvyšné dva.



Obr. A.9: Zapojenie generátora adries

Dôvod, prečo musí generátor adries obsahovať inkrementačný aj dekrementačný obvod je nasledovný. Inicializáciou akéhokoľvek obvodu riadiaca jednotka stráca jeden hodinový cyklus. Ak by generátor adries pozostával len z jedného kombinovaného obvodu schopného inkrementovať aj dekrementovať, pri zmene smeru adresovania by bolo potrebné najprv obvod inicializovať na správnu hodnotu (buď počiatočná alebo koncová adresa pamäte), čím by sa stratil jeden cyklus. Požiadavkou je, aby testovanie prebiehalo na operačnej frekvencii pamäte. Riešením sú teda dva obvody, jeden pre vzostupný smer adresovania, druhý pre zostupný smer adresovania.

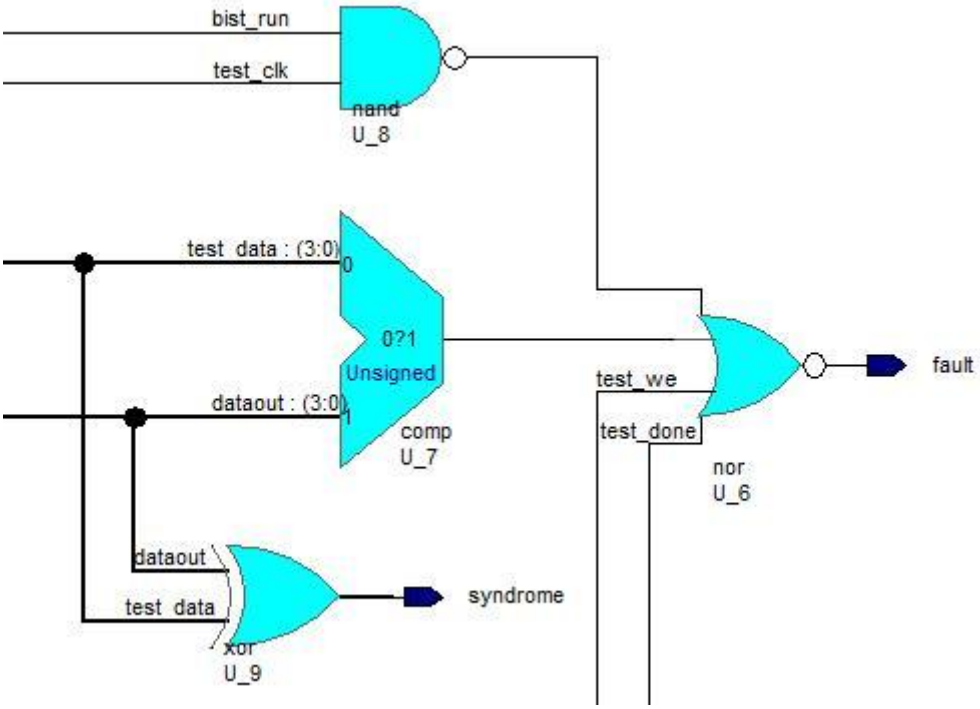
A.2.3 Komparátor

Na obrázku Obr. A.10 je zobrazená štruktúra a zapojenie bloku komparátor. Komparátor má 2 výstupné signály:

- *syndrome* – údaj o šírke n bitov (n – šírka slova pamäte), ktorý obsahuje hodnoty log. 1 na pozíciách tých bitov slova, ktoré sú poruchové. Tento signál je výstupom

log. člena XOR, ktorý porovnáva prečítané údaje z pamäte s očakávanými bit po bite.

- *fault* – signál indikujúci výskyt poruchy. Aby bol tento signál aktívny (hodnota log. 1), musia byť splnené nasledovné podmienky:
    - Výstup log. člena NAND musí byť log. 0, čo znamená, že signál *bist_run* musí mať hodnotu log. 1 (beží test), aj signál *test_clk* musí mať hodnotu log. 1, prebieha nejaká operácia nad pamäťou.
    - Výstup člena s označením *comp U_7* musí byť log. 0, to je v prípade, že sa údaje prečítané z pamäte nezhodujú s očakávanými údajmi.
    - Signál *test_we* musí mať hodnotu log. 0 – prebieha operácia čítania z pamäte.
    - Signál *test_done* musí mať hodnotu log. 0 – test ešte neskončil.

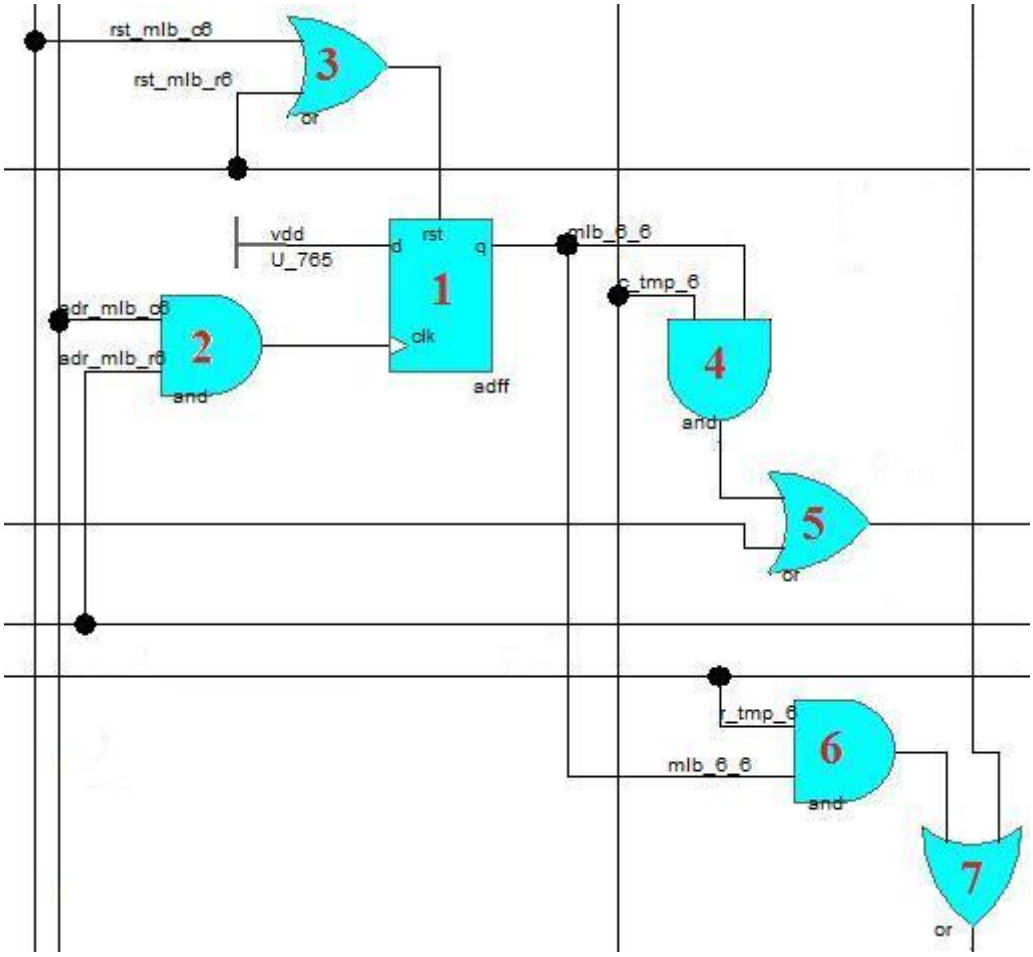


Obr. A.10: Štruktúra a zapojenie bloku komparátor

A.3 Architektúra BIRA

A.3.1 Základná bunka poľa bitmapa

Na obrázku Obr. A.11 je zobrazená štruktúra a zapojenie jednej bunky poľa bitmapa.



Obr. A.11: Zapojenie jednej bunky poľa

Význam jednotlivých komponentov:

- 1 – D preklápací obvod, reprezentuje jednu bunky poľa (konkrétne 6,6 – riadok s adresou 6 a stĺpec s adresou 6), na výstupe *q* drží hodnotu log. 0, pokiaľ je bunka voľná a hodnotu log. 1, pokiaľ je bunka použitá na reprezentáciu poruchy. Vstup *rst* slúži na vynulovanie, vstup *clk* na zápis, na vstup *d* je pripojená trvalá log. 1, pretože do registra sa zapisuje len táto hodnota.
- 2 – logický člen AND, cez tento logický člen sa prešíri hodnota log. 1 (vykoná sa zápis do bunky) len v prípade, že je adresovaná táto konkrétna bunka a aktívny hodinový signál. Vstupné signály *adr_mlb_c6* a *adr_mlb_r6* sú aktívne vtedy, keď sa adresuje riadok poľa s adresou 6, resp. stĺpec poľa s adresou 6.
- 3 – logický člen OR, cez tento logický člen sa prešíri hodnota log. 1 (vykoná sa reset bunky) len v prípade, že je aktívny signál reset a adresuje sa riadok alebo stĺpec, ktorého je bunka súčasťou. Výstupný signál tohto člena bude mať hodnotu

log. 1 aj v prípade, že sa vykonáva reset celého obvodu bitmapa (táto možnosť je zahrnutá pomocou kombinačnej logiky do signálu pre reset stĺpca poľa).

- 4 a 5 – logické členy AND a OR, tieto logické členy slúžia na výpočet bunky riadkového registra úspešnosti, tento výpočet možno reprezentovať vzorcom:

$$RRU[i] = \overline{RRS[i]} \,\&\, ((pole[i,0] \,\&\, \overline{SRS[0]}) \,|\, ... \,|\, (pole[i,11] \,\&\, \overline{SRS[11]}))$$

(RRU – riadkový register úspešnosti, RRS – riadkový register stratégie, SRS – stĺpcový register stratégie, i – riadkový index v rámci poľa)

Na vstupy log. člena AND sú privedené signály z D preklápacieho obvodu a bunky stĺpcového registra stratégie. Na vstupy log. člena OR je privedený výstup log. člena AND a výstup log. člena OR predchádzajúcej bunky v riadku.

- 6 a 7 – logické členy AND a OR, tieto logické členy slúžia na výpočet bunky stĺpcového registra úspešnosti, tento výpočet možno reprezentovať vzorcom:

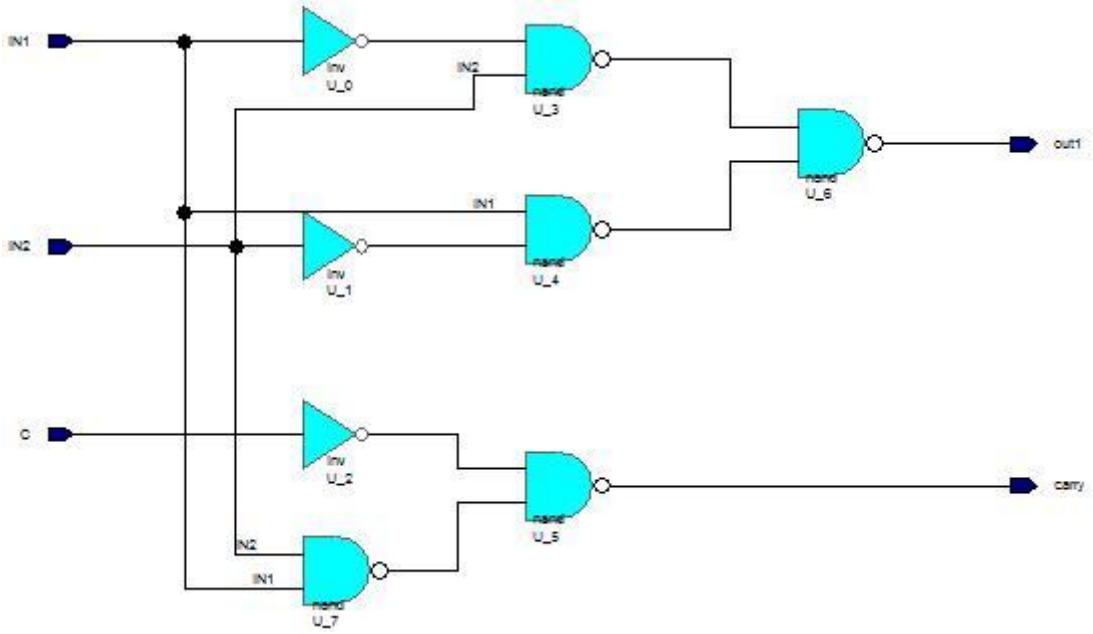
$$SRU[i] = \overline{SRS[i]} \,\&\, ((pole[0,i] \,\&\, \overline{RRS[0]}) \,|\, ... \,|\, (pole[11,i] \,\&\, \overline{RRS[11]}))$$

(SRU – stĺpcový register úspešnosti, RRS – riadkový register stratégie, SRS – stĺpcový register stratégie, i – stĺpcový index v rámci poľa)

Na vstupy log. člena AND sú privedené signály z D preklápacieho obvodu a bunky riadkového registra stratégie. Na vstupy log. člena OR je privedený výstup log. člena AND a výstup log. člena OR predchádzajúcej bunky v stĺpci.

A.3.2 Počítadlá porúch

Počítadla porúch alebo sčítačky sú obvody, ktorých výstupom je signál *row_cnt* alebo *col_cnt* bitmapy. Sú zapojené za sebou tak, že na vstup *IN2* a *C* sú privedené výstupy *out1* a *carry* predošlého obvodu, na vstup *IN1* sú postupne pripojené výstupy D preklápacích obvodov (*q*) riadka alebo stĺpca, v ktorom je potrebné spočítať zaznamenané poruchy. Štruktúra jednej takejto sčítačky je zobrazená na obrázku Obr. A.12.



Obr. A.12: Štruktúra sčítačky

V tabuľke Tab. A.1 je zobrazená pravdivostná tabuľka sčítačky. Vstupy *IN1* a *IN2* majú váhu 1, vstup *C* (prenos do vyššieho rádu) má váhu 2. Výstup *out1* má váhu 1, *carry* má váhu 2. Posledný prípad nemôže nastať, pretože pri kontrole počtu porúch zaznamenaných v danom riadku alebo stĺpci sa zistí splnenie podmienky nevyhnutnej opravy, ktorá sa hneď aj vykoná. Bitmapa teda nebude obsahovať vo svojom poli viac ako 3 záznamy v jednom riadku, resp. jednom stĺpci.

Tab. A.1: Pravdivostná tabuľka sčítačky

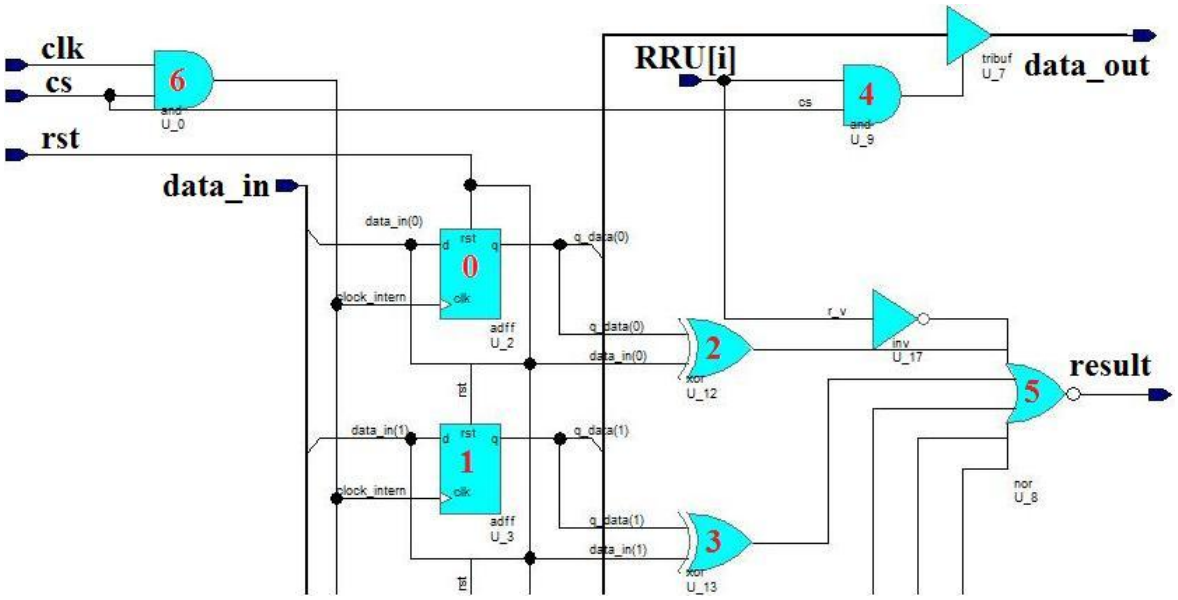
vstupy			výstupy	
C	IN1	IN2	out1	carry
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	0	1
1	0	1	1	1
1	1	0	1	1
1	1	1	X	X

A.3.3 Bunka adresného registra

Adresný register (riadkový aj stĺpcový) funguje principiálne ako pamäť typu CAM. To znamená, že umožňuje zápis do pamäte, čítanie z pamäte, vymazanie z pamäte

a porovnanie obsahu pamäte s indikáciou zhody. Každá bunka adresného registra predstavuje bunku pamäte CAM (pre riadkové alebo stĺpcové adresy). Rozdiel medzi nimi je len v tom, že CAM pamäť pre riadkové adresy uchováva údaj o šírke 5 bitov a CAM pamäť pre stĺpcové adresy uchováva údaj o šírke 7 bitov (3 bity adresa stĺpca + 4 bity syndróm). Na obrázku Obr. A.13 je zobrazený princíp zapojenia jednej bunky adresného registra, teda pamäte CAM pre adresy. Význam jednotlivých komponentov:

- 0 a 1 – D preklápacie obvody, uchovávajú zapísaný údaj, ich počet závisí od konkrétnej pamäte (pre riadkový register adries 5 kusov, pre stĺpcový 7 kusov).
- 2 a 3 – Logické členy XOR, slúžia na okamžité porovnanie signálu *q* D preklápacích obvodov s jednotlivými bitmi signálu *data_in* (vstupné údaje).
- 4 – Logiký člen AND, ktorý prešíri hodnotu log. 1 na trojstavové hradlo v prípade, že je zvolená daná bunka (aktívny signál *cs*) a že uchovávaný záznam je platný (aktívny bit riadkového alebo stĺpcového registra úspešnosti). Aktiváciou trojstavového hradla sa výstup z obvodov prešíri na výstup CAM pamäte.
- 5 – Logický člen NOR – výstupom tohto log. člena je výsledok porovnania, pokiaľ majú všetky jeho vstupné signály hodnotu log. 0, prešíri na výstup *result* hodnotu log. 1. Podľa hodnoty signálu *result* vie potom ďalšia logika určiť, v ktorej bunke pamäte nastala zhoda.
- 6 – Logický člen AND – na výstup prešíri hodnotu log. 1 v prípade, že je aktívny hodinový signál (signál *clk*) a je adresovaná daná bunka pamäte (signál *cs*). Týmto logickým členom sa ovláda zápis hodnoty zo vstupu *data_in* do D preklápacích obvodov.



Obr. A.13: Princíp zapojenia bunky adresných registrov

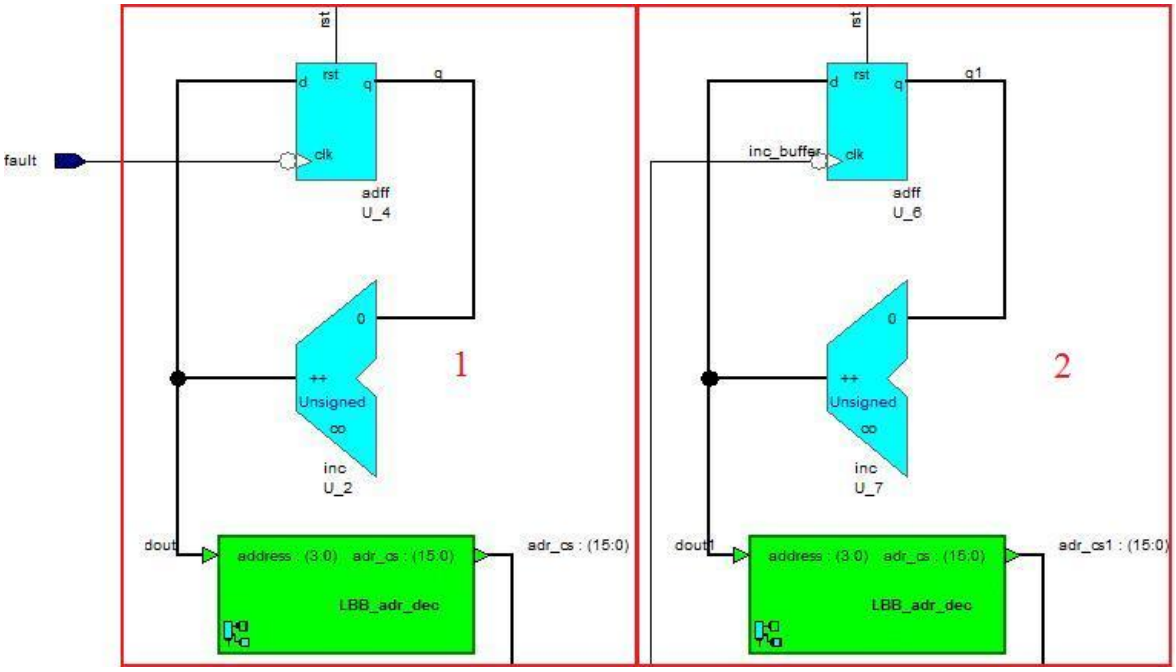
A.4 Prepojenie architektúr BIST a BIRA

Pamäť pre uchovanie informácií o poruchách pozostáva zo 16-tich buniek podobnej štruktúry ako bunka adresných registrov v bitmape (obrázok Obr. A.13), obsahuje 13 D preklápacích obvodov, neobsahuje však nič z logiky potrebnej pre vykonávanie operácie porovnávania. 12 D preklápacích obvodov je potrebných pre uchovanie adresy (8 bitov) a syndrómu (4 bity) a 1 D preklápací obvod je potrebný pre indikáciu platnosti záznamu.

Pamäť pre uchovanie informácií o poruchách je adresovaná dvomi ukazovateľmi. Jeden ukazovateľ adresuje tú bunku pamäte, do ktorej sa zapíšu údaje o najbližšie detegovanej poruche, druhý ukazovateľ adresu tú bunku pamäte, z ktorej sa čítajú údaje o poruche a spracúvajú sa obvodom BIRA.

Tieto ukazovatele sú výstupom adresných dekóderov, ktoré majú na vstupe výstupy z počítadiel. Realizácia a zapojenie počítadiel je známe zo zapojenia generátora adries v obvode BIST, prvé počítadlo sa inkrementuje po zapísaní údajov o detegovanej poruche, druhé počítadlo sa inkrementuje po spracovaní údajov o detegovanej poruche.

Výstupom adresného dekódera je teda 16 signálov, z ktorých každý aktivuje určitú bunku v pamäti. Zapojenie počítadiel a adresných dekóderov je zobrazené na obrázku Obr. A.14

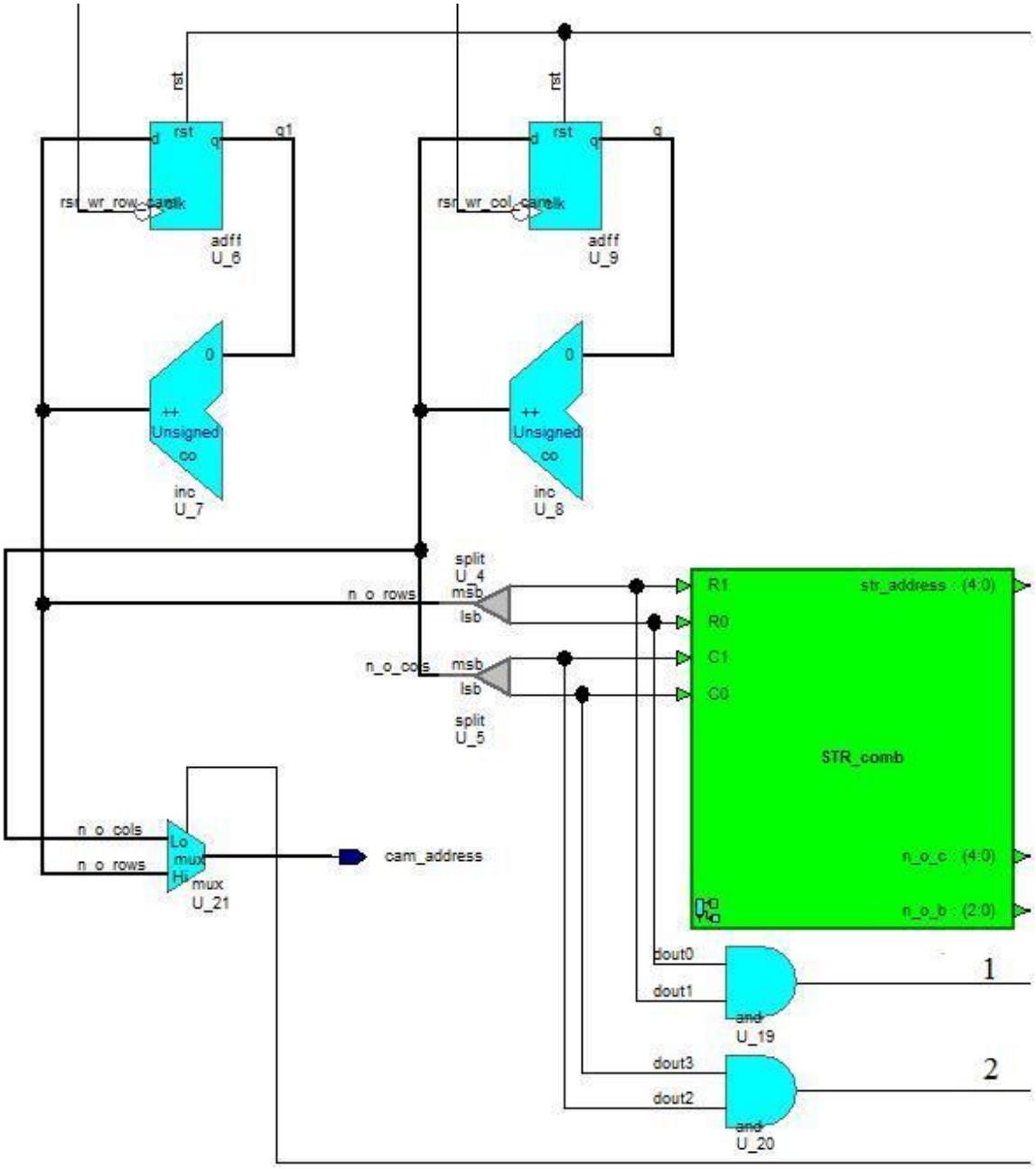


Obr. A.14: Zapojenie počítadiel a adresných dekóderov

A.5 Generátor stratégie

Súčasťou obvodu generátora stratégie sú okrem samotnej pamäte obsahujúcej stratégie opravy aj rôzne počítadlá, vďaka ktorým vie konečný stavový automat riadiaci celý priebeh opravy pamäte pristupovať k jednotlivým stratégiám alebo získavať potrebné stavové informácie ako napr. informácia o tom, že všetky zálohy boli použité, informácia o tom, že všetky stratégie boli použité a pod.

Pre prehľadnosť obvod opíšem v dvoch blokoch. Prvý blok je zobrazený na obrázku Obr. A.15. V hornej časti obrázku sú zobrazené 2 počítadlá (princíp zapojenia je už známy). Počítadlo vľavo počíta použité riadkové zálohy, počítadlo vpravo počíta použité stĺpcové zálohy. Výstupy z týchto počítadiel sú vstupmi kombinačného obvodu, ktorého správanie je opísané v kapitole 4.2.2. Signály v dolnej časti obrázku – výstupy log. členov AND indikujú vyčerpanie všetkých riadkových záloh (horný signál č. 1), resp. vyčerpanie všetkých stĺpcových záloh (dolný signál č. 2). Tieto signály sú stavovými signálmi pre konečný stavový automat riadiaci celý priebeh opravy pamäte.



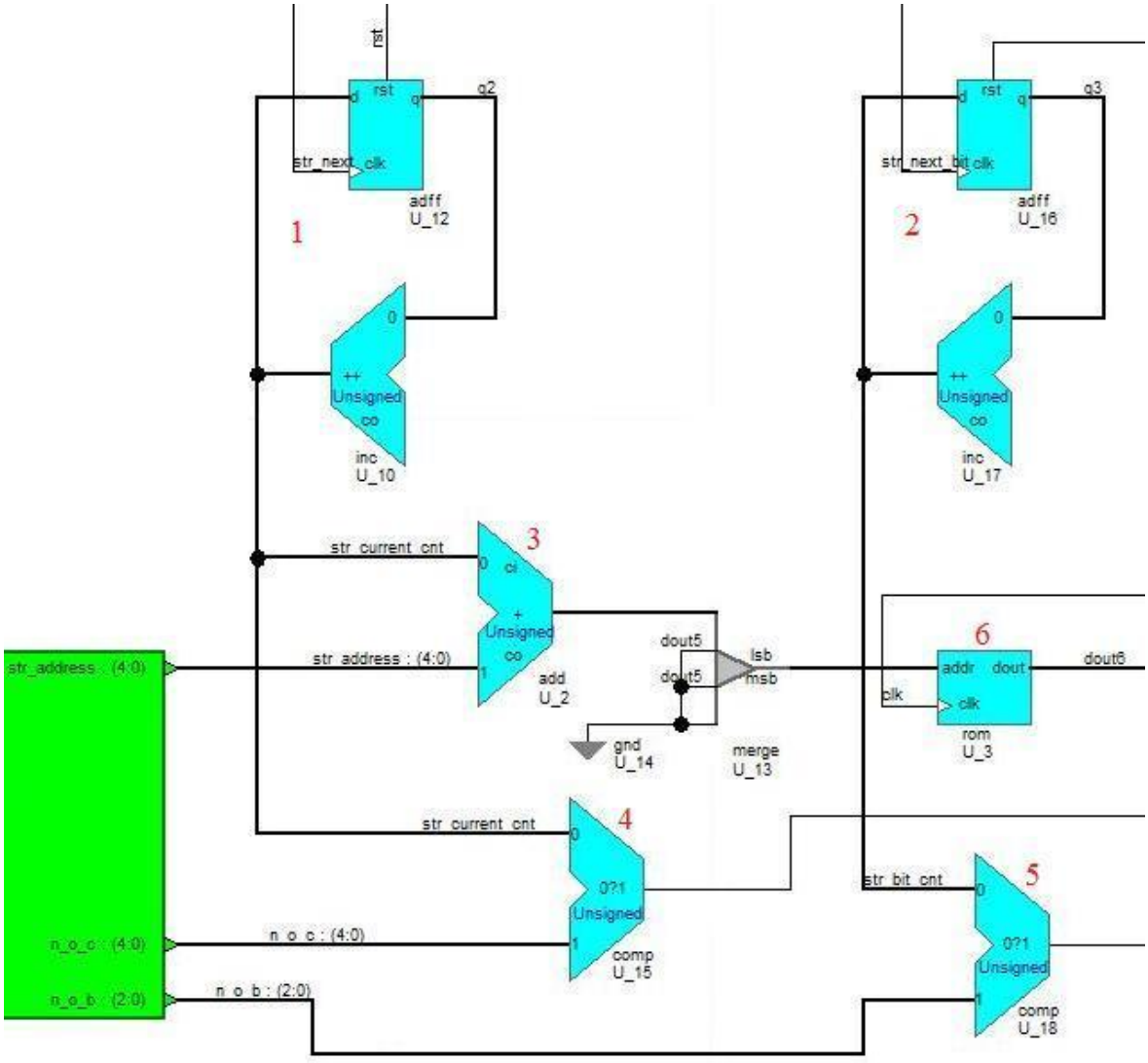
Obr. A.15: 1. blok obvodu generátora stratégie

Druhý blok je zobrazený na obrázku Obr. A.16. V ľavej hornej časti sa nachádza počítadlo adresy (číslo 1) do pamäte obsahujúcej stratégie (číslo 6). Počítadlo je inkrementované signálom, ktorého impulzom si konečný stavový automat vyžiada ďalšiu stratégiu. V pravej hornej časti sa nachádza počítadlo bitu v rámci stratégie (číslo 2, koľko záloh už bolo aplikovaných). Počítadlo je inkrementované signálom, ktorého impulzom si konečný stavový automat vyžiada typ ďalšej zálohy.

Obvod číslo 3 je sčítačka, ktorá spočíta základnú adresu do pamäte obsahujúcej stratégie (výstup kombinačného obvodu) spolu s hodnotou počítadla adresy do pamäte obsahujúcej stratégie.

Obvod číslo 4 je komparátor, ktorý porovnáva počet otestovaných stratégií s maximálnym počtom stratégií (výstup kombinačného obvodu). Aktívny výstup komparátora indikuje konečnému stavovému automatu vyčerpanie všetkých dostupných stratégií.

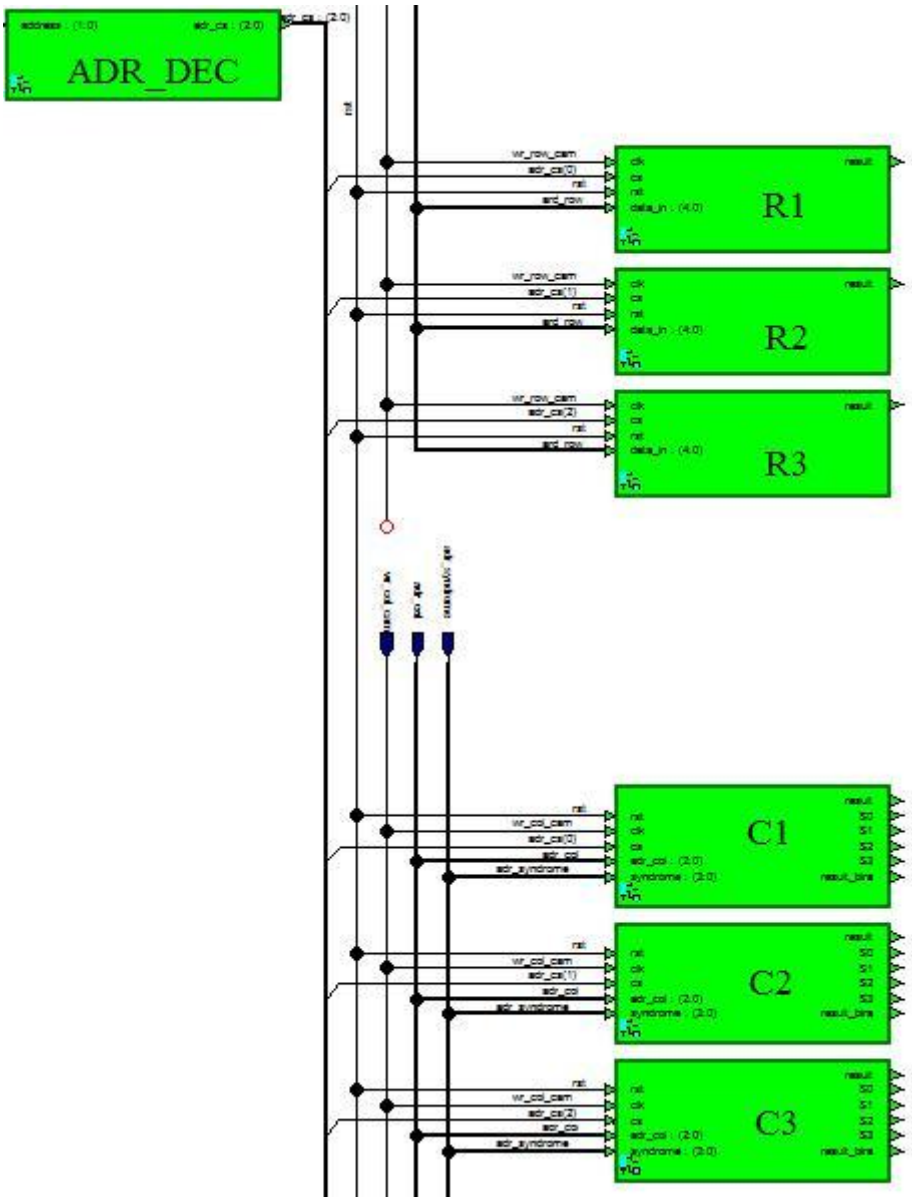
Obvod číslo 5 je komparátor, ktorý porovnáva počet aplikovaných záloh v rámci stratégie s počtom dostupných záloh (výstup kombinačného obvodu). Aktívny výstup komparátora indikuje konečnému stavovému automatu vyčerpanie všetkých dostupných záloh v rámci testovanej stratégie.



Obr. A.16: 2. blok obvodu generátora stratégie

A.6 Register vykonaných opráv

Na obrázku Obr. A.17 je zobrazené principiálne zapojenie CAM pamätí pre uchovanie informácií o opravách záložným riadkom a stĺpcom. CAM pamäte pre záznamy o oprave záložným riadkom sú označené *R1*, *R2* a *R3*, CAM pamäte pre záznamy o oprave záložným stĺpcom sú označené *C1*, *C2* a *C3*. Obvod *ADR_DEC* je adresný dekóder pre tieto CAM pamäte. Adresný dekóder postačuje len jeden z toho dôvodu, že v jednom čase sa môže vykonávať operácia zápisu len do jednej CAM pamäte v registri vykonaných opráv. Štruktúra buniek riadkovej CAM pamäte je rovnaká ako štruktúra buniek CAM pamätí adresných registrov použitých v bitmape. Štruktúra buniek stĺpcovej CAM pamäte sa líši pridaním logiky pre poskytnutie výstupu o syndróme opraveného stĺpca.



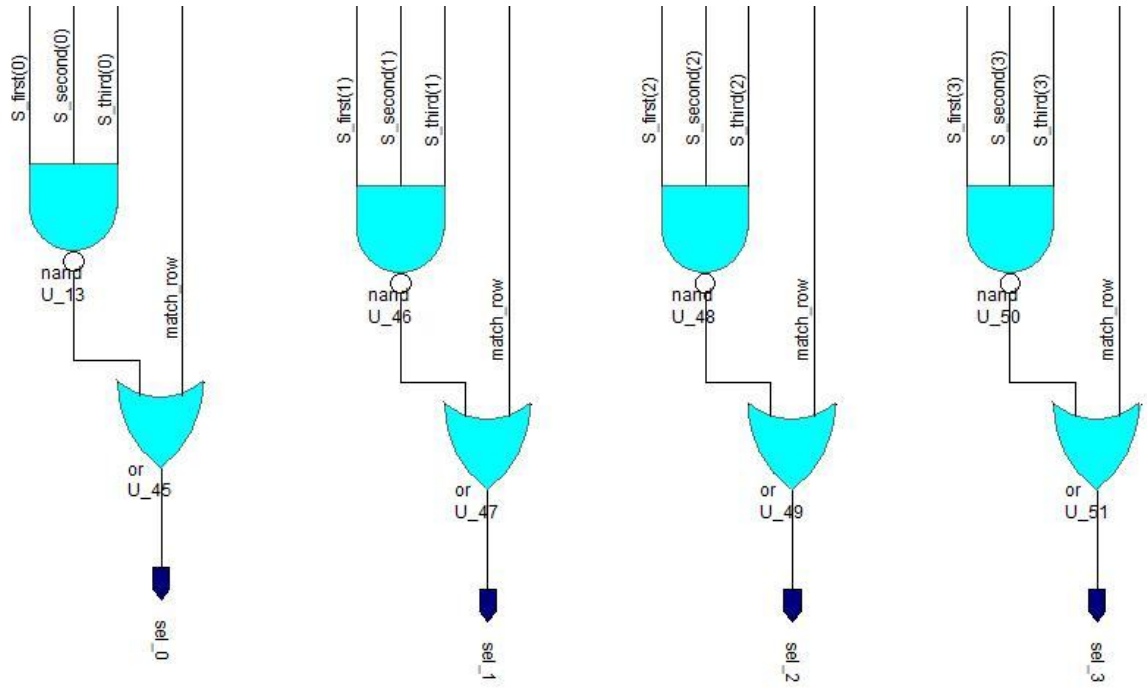
Obr. A.17: Princíp zapojenia CAM pamätí v registri vykonaných opráv

Počet buniek CAM pamätí v registri vykonaných opráv závisí od počtu dostupných záloh. V tomto prípade je to 3 a 3. Záložná pamäť je logicky rozdelená na riadkovú a stĺpcovú záložnú pamäť a je adresovaná podľa výstupu *result* z CAM pamätí. Ak používateľ dopytuje operáciu nad opravenou bunkou, operácia sa vykoná nad záložnou bunkou v záložnej pamäti a výstupom registra vykonaných opráv sú signály pre nastavenie multiplexorov slúžiacich na prešírenie správnej hodnoty na používateľský údajový výstup z pamäte a takisto údajový výstup zo záložnej pamäte (tá je implementovaná ako súčasť obvodu register vykonaných opráv).

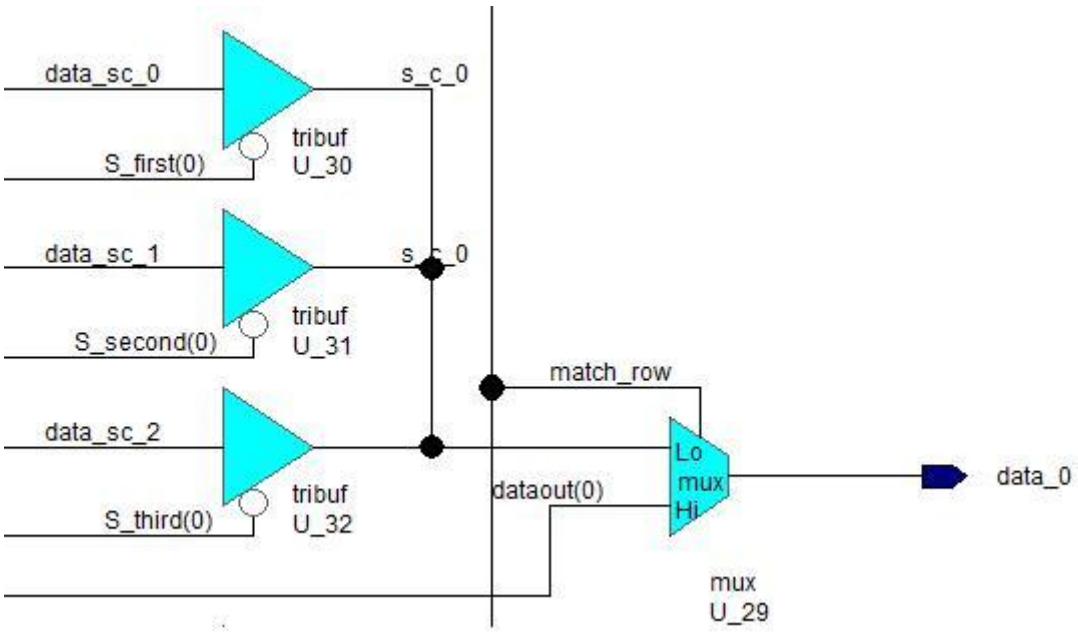
Slovo hlavnej pamäte má šírku 4 bity, preto je potrebné pripojiť údajové výstupy z hlavnej pamäte a údajové výstupy zo záložnej pamäte bit po bite na multiplexor. Multiplexor je potom ovládaný jedným zo signálov *sel_0*, *sel_1*, *sel_2* alebo *sel_3* (logika generovania signálov zobrazená na obrázku Obr. A.18), záleží od toho, či sa jedná o najviac významový bit slova (*sel_3*) alebo najmenej významový (*sel_0*). V tomto riešení platí pravidlo, že oprava riadkom je nadradená oprave stĺpcom. Preto, keď je dopyt na bunku pamäte, ktorá je opravená aj riadkom, aj stĺpcom, pracuje sa s jej hodnotou v riadkovej záložnej pamäti. Cez multiplexor sa na výstup prešíria údaje z hlavnej pamäte, pokiaľ dopytovaná bunka nebola opravovaná (vtedy signál *sel_x* = ‘0’) a údaje zo záložnej pamäte sa prešíria v prípade, že bunka bola opravovaná (vtedy signál *sel_x* = ‘1’). Ako možno vidieť zo zapojenia logických členov pre generovanie signálov *sel_x*, signály sú aktívne vždy, keď je zhoda s riadkovou adresou dopytovanej bunky v riadkovej CAM pamäti. Keď je zhoda so stĺpcovou adresou dopytovanej bunky v stĺpcovej CAM pamäti, aktívne sú signály *sel_x* len tých bitov, ktoré boli v danom stĺpci opravené.

Ktorá hodnota zo záložnej pamäte (riadkovej alebo stĺpcovej) sa prešíri na celkový údajový výstup záložnej pamäte, určuje jednoduché zapojenie log. členov zobrazené na obrázku Obr. A.19. Multiplexor vyberá z dvoch údajových vstupov. Na prvom vstupe multiplexora je pripojený výstup zo stĺpcovej záložnej pamäte, na druhom vstupe je pripojený výstup z riadkovej záložnej pamäte (*dataout(0)*). Takéto zapojenie sa v registri vykonaných opráv nachádza štyrikrát – pre každý údajový bit slova. Keď je zhoda s riadkovou adresou dopytovanej bunky v riadkovej CAM pamäti, signál selektora má vždy hodnotu ‘1’, inak má hodnotu ‘0’ (prešíri sa výstup zo stĺpcovej záložnej pamäte). Zo stĺpcovej záložnej pamäte idú 3 výstupy – *data_sc_0*, *data_sc_1* a *data_sc_2*. Tieto výstupy musia byť pripojené cez trojstavové hradlá, ktoré sú aktivované signálom syndrómu zo stĺpcovej CAM pamäte. Ak bol opravovaný najmenej významový bit v dopytovanom stĺpci, prešíri

sa na údajový výstup záložnej pamäte jeden z vyššie spomenutých signálov *data_sc_x* podľa toho, či bol opravený prvou, druhou alebo treťou stĺpcovou zálohou.



Obr. A.18: Zapojenie log. členov pre generovanie signálov selektor



Obr. A.19: Princíp prešírenia hodnoty zo záložnej pamäte

Príloha B: Používateľská príručka

B.1 Nahratie testu

Pre nahratie testu je potrebné vytvoriť si test typu march pre slovne orientovanú pamäť podľa vašich požiadaviek. Ste limitovaný zložitosťou testu (255N) a test nemôže využívať oneskorenia. Testovacie vzorky sú 4-bitové, podľa šírky slova testovanej pamäte. Následne svoj test zakódujte do formátu uvedeného v kapitole 4.1.3. Príklad zakódovania testu march C- (pri operáciách s možnosťou voľby smeru adresovania bol zvolený vzostupný smer):

100110000 = ⇳ (w0000)
110100000 = ⇑ (r0000)
111111111 = ⇑ (w1111)
110101111 = ⇑ (r1111)
111110000 = ⇑ (w0000)
110000000 = ⇓ (r0000)
111011111 = ⇓ (w1111)
110001111 = ⇓ (r1111)
111010000 = ⇓ (w0000)
100100000 = ⇳ (r0000)
000000000 = finish

Keď máte pripravený svoj test, zapíšete ho nastavovaním signálov *bist_addr*, *bist_data* a *bist_clk* nasledovným spôsobom (je dôležité, aby signál *bist_run* mal hodnotu log. 0):

- signálom *bist_addr* adresujete pamäť inštrukcií, začínate na adrese 0 a pokračujete inkrementovaním adresy o 1,
- na signál *bist_data* je potrebné postupne priviesť všetky zakódované inštrukcie,
- nábežnou hranou signálu *bist_clk* vykonáte zápis inštrukcie zo signálu *bist_data* do pamäte inštrukcií na adresu *bist_addr*.

Príklad sekvencie pre nahranie vyššie uvedeného testu march C- do pamäte inštrukcií:

```
bist_run <= '0';
bist_addr <= "00000000";
bist_data <= "100110000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= "00000000";
bist_data <= "100110000";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= "00000001";
bist_data <= "110100000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= "00000001";
bist_data <= "110100000";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= "00000010";
bist_data <= "111111111";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= "00000010";
bist_data <= "111111111";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= "00000011";
bist_data <= "110101111";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= "00000011";
bist_data <= "110101111";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= "00000100";
bist_data <= "111110000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= "00000100";
bist_data <= "111110000";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= "00000101";
bist_data <= "110000000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= "00000101";
bist_data <= "110000000";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= " 00000110";
bist_data <= " 111011111";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= " 00000110";
bist_data <= " 111011111";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= " 00000111";
bist_data <= " 110001111";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= " 00000111";
bist_data <= " 110001111";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= " 00001000";
bist_data <= " 111010000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= " 00001000";
bist_data <= " 111010000";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= " 00001001";
bist_data <= " 100100000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= " 00001001";
bist_data <= " 100100000";
bist_clk <= '0' ;
wait for 1 ns;
bist_addr <= " 00001010";
bist_data <= "000000000";
bist_clk <= '1' ;
wait for 1 ns;
bist_addr <= " 00001010";
bist_data <= "000000000";
bist_clk <= '0' ;
```

B.2 Spustenie simulácie

Otvorte projekt *diplomovka* na priloženom médiu v programe HDL Designer. Kliknite na jednotku *obvod_tb* a kliknite na *Simulate* v ponuke v ľavom paneli. V prípade potreby si upravte priebeh testu v súbore *obvod_tester* podľa svojich predstáv, prednastavený prebieh testovania je nahratie testu march C- do pamäte inštrukcií, spustenie testu a následne zápis a prečítanie údajov z vybraných adries hlavnej pamäte pre overenie úspešnosti vykonanej opravy. V projekte je aj priložený waveform *final_test_no_fault.do*, ktorý si môžete nahrať do okna *Wave* programu ModelSim. Uvidíte tam signály jednotlivých blokov podľa popisu v tomto dokumente. Základne rozhranie tvoria signály:

- *mem_addr* – adresovanie hlavnej pamäte,
- *mem_we* – ovládanie operácie zápisu (‘1’) a čítania (‘0‘) nad hlavnou pamäťou,
- *mem_data* – údajový vstup hlavnej pamäte,
- *mem_data_out* – údajový výstup hlavnej pamäte,
- *finish* – signál indikujúci ukončenie opravy pamäte,
- *unrepairable* – signál indikujúci úspešnosť (‘0‘) / neúspešnosť (‘1’) opravy,
- *bist_run* – signál pre spustenie testovania a opravy pamäte,
- *bist_addr*, *bist_clk*, *bist_data* – signály sú opísané v predošlej kapitole.

Poruchu do pamäte injektujete využitím kontextovej funkcie *Force* programu ModelSim. Bunky hlavne pamäte sú prístupné v skupine signálov s názvom *RAM*, po jednotlivých stĺpcoch, následne riadkoch v danom stĺpci, a nakoniec po jednotlivých bitoch zvoleného slova.

Príloha C: Obsah elektronického nosiča

\annot	anotácie v slovenskom a anglickom jazyku (.doc + .pdf)
\doc	tento dokument (.doc + .pdf)
\examples	príkazy force pre jednotlivé príklady uvedené v kapitole 5.1
\iit.src	príspevok prezentovaný na IIT.SRC 2014 (článok .pdf + plagát .pptx)
\project	implementácia práce formou projektu v programe HDL Designer
\synthesizable	alternatívny zdrojový kód riadiacej jednotky BIST pre syntézu
contents.txt	obsah elektronického nosiča

Použité skratky

BIRA	Built-in Repair Analysis – vstavaná analýza opravy
BISR	Built-in Self-Repair – vstavaná samočinná oprava
BIST	Built-in Self-Test – vstavané samočinné testovanie
bitmapa	malá pamäť, do ktorej sú ukladané informácie o detegovaných poruchách testovanej pamäte
MLB	Maximal-Size Local Bitmap – bitmapa
DRAM	dynamic RAM – dynamická pamäť typu RAM
FSM	Finite State Machine – konečný stavový automat
RAM	Random Access Memory – pamäť s náhodným prístupom
ROM	Read Only Memory – pamäť, z ktorej je možné len čítať
RWM	Read Write Memory – pamäť, z ktorej je možné čítať a tiež do nej zapisovať
SoC	System on Chip – systém na čipe
SRAM	static RAM – statická pamäť typu RAM
VHLD	Very High Speed Integrated Circuit Hardware Description Language – jazyk pre opis hardvéru
CAM	pamäť adresovateľná obsahom
FIFO	First In – First Out, rad obsluhovaný spôsobom prvý príde, prvý aj odíde
RSR	Repair Signature Register – register vykonaných opráv

Použitá literatúra

[1] International Technology Roadmap for Semiconductors (ITRS). 2011. *Test and Test Equipment*. http://www.itrs.net/Links/2011ITRS/2011Chapters/2011Test.pdf (2.5.2013)

[2] ADAMS, R. D. *High Performance Memory Testing. Design Principles, Fault Modeling and Self-Test*, Kluwer Academic Publisher, 2003, 246 s, ISBN 1-4020-7255-4.

[3] HAMDIOUI, S., GAYDADJIEV, G., VAN DE GOOR, A. J. *The State-of-art and Future Trends in Testing Embedded Memories*. In: *Records of the International Workshop on Memory Technology*, *Design and Testing*, 2004, s. 54-59.

[4] WUNDERLICH, H.-J. (ed.). *Models in Hardware Testing*. Springer, 2010, 257 s, ISBN 978-90-481-3281-2.

[5] MOŤOVSKÝ, E.: Metódy testovateľnosti a rekonfigurovateľnosti pamätí RAM vhodných pre vnorenie do SoC. Bratislava: FIIT STU, 2010. 65 s. Diplomový projekt.

[6] CHEN, T.-J., LI, J.-F, TSENG, T.-W. *Cost-Efficient Built-In Redundancy Analysis With Optimal Repair Rate for RAMs*. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 31, 2012, no. 6, s. 930-940.

[7] CHANG, Y.-J., HUANG, Y.-J, LI, J.-F. *A Built-In Redundancy-Analysis Scheme for RAMs with 3D Redundancy*. In: IEEE Transactions on VLSI-DAT, vol. 22, 2011, no. 6, s. 836-839.

[8] LEE, M., DENQ, L.-M., WU, C.-W. *A Memory Built-In Self-Repair Scheme Based on Configurable Spares*. In: IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, 2011, no. 6, s. 919-929.

[9] SU, C.-L., HUANG, R.-F, WU, C.-W. *A Built-In Self-Diagnosis and Repair Design With Fail Pattern Identification for Memories*. In: IEEE Transactions on VLSI Systems, vol. 19, 2011, no. 12, s. 2184-2194.

[10] TSENG, T.-W., LI, J.-F. *A Low-Cost Built-In Redundancy-Analysis Scheme for Word-Oriented RAMs With 2-D Redundancy*. In: IEEE Transactions on VLSI Systems, vol. 19, 2011, no. 11, s. 1983-1995.

[11] KAPSE, V., ARIF, M. *Optimization of Microcode Built-In Self Test By Enhanced Faults Coverage for Embedded Memory*. In: IEEE Student's Conference on Electrical, Electronics and Computer Science, 2012, s. 1-6.

[12] LOTFI, A., KABIRI, P., NAVABI, Z. *Configurable Architecture for Memory BIST*. In: 9th East-West Design & Test Symposium (EWDTS), 2011, s. 1-5.

[13] DU, X., MUKHERJEE, N., CHENG, W.-T., REDDY, S.M. *Full-Speed Field-Programmable Memory BIST Architecture*. In: Proceedings of IEEE International Test Conference, 2005, s. 1165-1173.

[14] NOOR, N. M., SAPARON, A., YUSOF, Y. *Programmable MBIST Merging FSM and Microcode Techniques Using Macro Commands*. In: IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems, 2010, s. 115-121.

[15] JEONG, W., KANG, I., JIN, K., KANG, S. *A Fast Built-in Redundancy Analysis for Memories with Optimal Repair Rate Using a Line-Based Search Tree*. In: IEEE Transactions on VLSI systems, vol. 17, 2009, no. 12, s. 1665-1678.