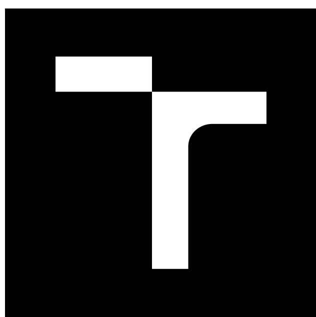
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky
a komunikačních technologií

DIPLOMOVÁ PRÁCE

Brno, 2017

Bc. Lukáš Vykydal



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

ÚSTAV RADIOELEKTRONIKY

DEPARTMENT OF RADIO ELECTRONICS

MIKROPROGRAMEM ŘÍZENÝ RAM BIST

MICROCODE-CONTROLLED RAM BIST

DIPLOMOVÁ PRÁCE

MASTER'S THESIS

AUTOR PRÁCE — **Bc. Lukáš Vykydal**

AUTHOR

VEDOUCÍ PRÁCE — **Ing. Michal Kubíček, Ph.D.**

SUPERVISOR

BRNO 2017

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ | FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

Diplomová práce

magisterský navazující studijní obor **Elektronika a sdělovací technika**

Ústav radioelektroniky

Student: Bc. Lukáš Vykydal ***ID:*** 154908

Ročník: 2 ***Akademický rok:*** 2016/17

NÁZEV TÉMATU:

Mikroprogramem řízený RAM BIST

POKYNY PRO VYPRACOVÁNÍ:

Analyzujte dostupné BIST algoritmy pro paměti typu RAM. Zaměřte se na algoritmy MARCH. Navrhněte architekturu RAM BIST řízeného mikroprogramem. Definujte optimální instrukční sadu mikrokódu pro MARCH algoritmy s ohledem na minimální velikost číslicového obvodu. Součástí návrhu bude i diagnostické rozhraní pro analýzu výrobních defektů RAM s možností provádět jednotlivé operace zápisu a čtení s libovolnou adresou a daty.

Návrh implementujte syntetizovatelným HDL kódem. Implementace verifikujte RTL simulací. Vytvořte aplikaci generující mikroprogram na základe popisu algoritmu používanou gramatikou pro MARCH algoritmy. V digitálním simulátoru proveďte automatickou simulaci vygenerovaných RAM BIST.

DOPORUČENÁ LITERATURA:

[1] SPEAR, Chris. a Gregory J. TUMBUSH. SystemVerilog for verification: a guide to learning the testbench language features. 3rd ed. New York: Springer, c2012.

[2] PRADHAN, Dhiraj K. a Ian G. HARRIS. Practical design verification. New York: Cambridge University Press, 2009. ISBN 0521859727.

Termín zadání: 6.2.2017 ***Termín odevzdání:*** 16.5.2017

Vedoucí práce: Ing. Michal Kubíček, Ph.D.
Konzultant: Ing. Miroslav Kašša

prof. Ing. Tomáš Kratochvíl, Ph.D.
předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č.40/2009 Sb.

ABSTRAKT

Cílem práce je seznámit se s defekty polovodičových pamětí a algoritmy určenými pro jejich detekci. Následně se práce zabývá návrhem a implementací BIST kontroléru pro test polovodičových pamětí s nízkými nároky na velikost výsledného digitálního bloku.

KLÍČOVÁ SLOVA

testování pamětí, March algoritmy, paměťový BIST

ABSTRACT

The goal of this work is to understand types of defects in semiconductor memories and algorithms for their testing. In the second part the work describes design and implementation of programmable BIST controller with small digital block size requirements.

KEYWORDS

memory testing, March algorithms, memory BIST

VYKYDAL, Lukáš *Mikroprogramem řízený RAM BIST*: diplomová práce. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, Ústav radioelektroniky, Rok. 58 s. Vedoucí práce byl Ing. Michal Kubíček, PhD

Vysázeno pomocí balíčku `thesis` verze v1.3; `http://latex.feec.vutbr.cz`

PROHLÁŠENÍ

Prohlašuji, že svou diplomovou práci na téma „Mikroprogramem řízený RAM BIST“ jsem vypracoval(a) samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor(ka) uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil(a) autorská práva třetích osob, zejména jsem nezasáhl(a) nedovoleným způsobem do cizích autorských práv osobnostních a/nebo majetkových a jsem si plně vědom(a) následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon), ve znění pozdějších předpisů, včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Brno ....

podpis autora

PODĚKOVÁNÍ

Rád bych poděkoval vedoucímu diplomové práce panu Ing.Michalu Kubíčkovi Ph.D. a konzultantovi panu Ing. Miroslavu Kaššovi za odborné vedení, konzultace, trpělivost a podnětné návrhy k práci.

Brno ....

podpis autora

Faculty of Electrical Engineering
and Communication
Brno University of Technology
Purkynova 118, CZ-61200 Brno
Czech Republic
http://www.six.feec.vutbr.cz

PODĚKOVÁNÍ

Výzkum popsaný v této diplomové práci byl realizován v laboratořích podpořených z projektu SIX; registrační číslo CZ.1.05/2.1.00/03.0072, operační program Výzkum a vývoj pro inovace.

Brno ....

podpis autora

EVROPSKÁ UNIE
EVROPSKÝ FOND PRO REGIONÁLNÍ ROZVOJ
INVESTICE DO VAŠÍ BUDOUCNOSTI

OP Výzkum a vývoj pro inovace

OBSAH

Úvod **11**

1 Paměti **12**

1.1 Typy pamětí . . . 12

1.2 Organizace paměťových buněk . . . 13

2 Defekty pamětí **15**

2.1 Defekty adresního dekodéru . . . 15

2.2 Defekty paměťových buněk . . . 16

2.2.1 Stack at fault . . . 17

2.2.2 Stuck open fault . . . 17

2.2.3 Transition fault . . . 17

2.2.4 Destructive read . . . 17

2.2.5 Data retention fault . . . 18

2.2.6 Write disturb fault . . . 18

2.2.7 Coupling fault . . . 18

2.2.8 Pattern sensitive faults . . . 18

2.2.9 Linked faults . . . 19

2.3 Defekty čtecí a zápisové logiky . . . 19

3 Algoritmy pro test pamětí **20**

3.1 Klasické testovací algoritmy . . . 20

3.1.1 Algoritmus 0–1 . . . 20

3.1.2 GALPAT, Walking 1/0 . . . 21

3.1.3 Sliding diagonal . . . 21

3.2 March algoritmy . . . 21

3.2.1 Volnosti march algoritmů . . . 22

3.2.2 Paměti s přístupem po slovech . . . 25

3.3 Smarch algoritmy . . . 25

4 Implementace **27**

4.1 Bloková struktura . . . 27

4.1.1 Vnější rozhraní kontroléru . . . 27

4.1.2 Řízení BIST kontroléru . . . 29

4.1.3 Řádkový a sloupcový čítač . . . 29

4.1.4 Čítač šířky datového slova . . . 29

4.1.5 Mikrokód . . . 29

4.1.6 Řízení . . . 29

4.1.7 Interface paměti 30
4.2 Mikrokód 30
4.2.1 Zápis mikrokódu 33
4.3 Čítače 33
4.3.1 Binární 33
4.3.2 LFSR 34
4.3.3 CFSR 37
4.3.4 Porovnání 40

5 Verifikace 42
5.1 Test bench 42
5.2 Interface testů 42
5.3 Testy 44

6 Generování BIST kotroléru 46
6.1 Model paměti 46
6.2 Šablony souborů 49
6.3 Tcl API 49
6.4 Užití 52

7 Závěr 54

Literatura 55

Seznam zkratek 57

SEZNAM OBRÁZKŮ

1.1 Blokové schéma typické paměti . . . 12
1.2 Paměťová buňka SRAM v technologii CMOS. Tato buňka je označována jako 6T. . . . 13
1.3 Foto integrované paměti Intel 2102 . . . 14
1.4 Blokové schéma paměti Intel 2102 [5] . . . 14
2.1 Defekt adresního dekodéru ADOF . . . 16
3.1 Přístup k datovým bitům v smarch algoritmu . . . 25
4.1 Blokové rozdělení testovacího kontroléru . . . 27
4.2 Časování signálů BIST kontroléru . . . 28
4.3 Časování signálů pro sériový přístup k paměti pomocí BIST kontroléru. 28
4.4 Zapojení MAI2MAI interface uvnitř modulu interafce paměti . . . 30
4.5 Fibonacci LFSR $n = 3$ a $p(x) = x^3 + x^2 + 1$ . . . 36
4.6 Reverzní Fibonacci LFSR $n = 3$ a $p(x) = x^3 + x^2 + 1$ . . . 37
4.7 Implementace dopředného čítače. Základem je LFSR čítač popsaný výrazem $x^3 + x^2 + 1$, doplněný o zpětnou vazbu dle výrazu 4.11. . . . 39
4.8 Implementace zpětného čítače. Základem je zpětný LFSR čítač popsaný výrazem $x^3 + x^2 + 1$, doplněný o zpětnou vazbu dle výrazu 4.14. . . . 39
4.9 Porovnání velikostí čítačů . . . 41
5.1 Blokové schéma verifikačního prostředí . . . 42

SEZNAM TABULEK

2.1 Gramatika pro popis defektů na paměti 16
3.1 Závislost požadovaného testovacího času pro různou složitost algoritmu. 20
3.2 Gramatika march testů 22
3.3 March testy 23
3.4 Pokrytí jednobitových chyb march algoritmy 24
3.5 Pokrytí vícebitových chyb march algoritmy 24
4.1 Nejjednodušší mikrokód 31
4.2 Mikrokód rozšířený pro více různých datových slov a návratovou adresu 31
4.3 Mikrokód rozšířený o adresaci „po sloupcích“ a „po řádcích“ a o podporu smarch algoritmů 32
4.4 Použitá verze mikrokódu; n udává šířku adresy mikroinstrukce. 32
4.5 Textový zápis march algoritmů 33
4.6 Odhad velikosti obousměrného binárního čítače 34
4.7 Odhad velikosti dopředného Fibonacci LFSR 37
4.8 Odhad velikosti obousměrného Fibonacci LFSR 37
4.9 Odhad velikosti dopředného CFSR čítače 39
4.10 Odhad velikosti obousměrného nulovatelného CFSR čítače 40
4.11 Porovnání odhadu velikostí čítačů 40
6.1 Popis konfiguračních sekcí a užitých klíčů. 47

ÚVOD

Význam polovodičových pamětí v integrovaných obvodech stále roste a s tím i zabraná plocha na čipu. Tento trend je navíc ještě umocněn využitím mikroprocesorových jader v integrovaném obvodu.

Při výrobě polovodičového obvodu dochází ke vzniku defektů, které je třeba co nejdříve odhalit a defektní čip vyřadit ideálně již během prvních testů. Při testování je požadavek na co nejvyšší rychlost testu. Z toho důvodu jsou části testovací logiky vyrobeny již v integrovaném obvodu. Jde ovšem o logiku, která není potřebná pro běžnou činnost, a proto je snaha o její co nejmenší velikost (tj. nejmenší cenu).

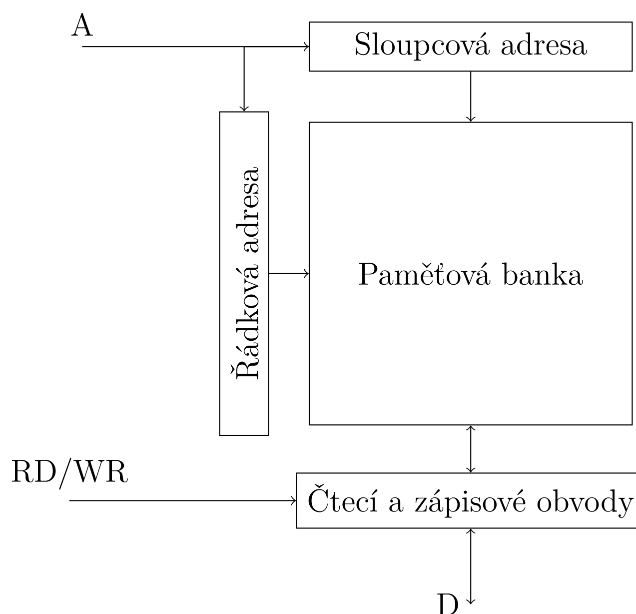
Vývoj se nevyhýbá ani algoritmům pro testování pamětí. Pro postupné zdokonalování pokrytí výrobních defektů je vhodné mít možnost změnit testovací algoritmus bez nutnosti přenavrhovat blok pro test pamětí. Řešením jsou mikrokódem programovatelné BIST kontroléry. Dostupné verze však nevyhovují z důvodu velikosti.

Cílem této práce je přiblížit problematiku testování polovodičových pamětí, následně provést souhrn existujících testovacích algoritmů a předložit koncept testovacího kontroléru pro polovodičové paměti, který bude možné použít pro self-test[1] pamětí.

[1] Test implementovaný na samotném polovodičovém čipu.

1 PAMĚTI

Polovodičová paměť je obvodová struktura určená ke krátkodobému nebo dlouhodobému uchování informace. Typická paměť obsahuje adresní dekodéry a čtecí a případně zápisové obvody. Taková paměť pak může mít strukturu podle obrázku 1.1. Toto je běžná struktura polovodičové paměti.[1]



Obr. 1.1: Blokové schéma typické paměti

1.1 Typy pamětí

Polovodičové paměti můžeme dělit podle několika hledisek. Paměti můžeme dělit podle uchování dat na trvalé (non-volatilní) a dočasné (volatilní), nebo podle upravitelnosti dat na přepisovatelné a pouze pro čtení.

Paměti určené pouze pro čtení („ROM“) jsou vždy trvalého charakteru. Tyto paměti mnohou být programovány maskou při výrobě (tzv. maskou programovatelné paměti), přepálením pojistky (tzv. OTP) nebo jde o EPROM[2].

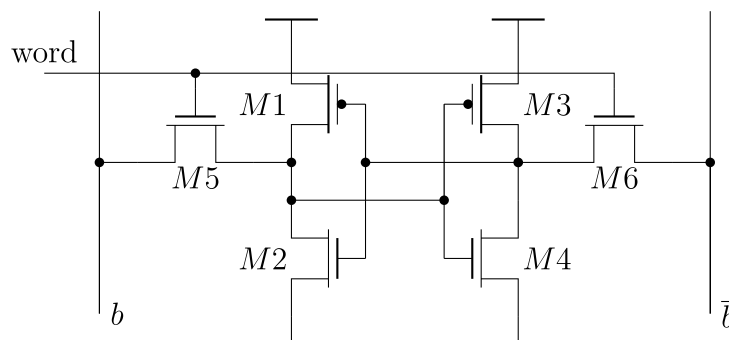
Přepisovatelné paměti trvalého charakteru jsou paměti, které se používají k uložení dat nebo konfigurace. Běžně se používají paměti EEPROM (typicky pro uložení malého množství dat) a paměti FLASH (NAND a NOR typu, dnes dostupné i pro velké kapacity).

[1] Zanedbáme-li speciální sériové paměti, které mohou reprezentovat fonty a posuvné registry.

[2] EPROM je přeprogramovatelná paměť, ale z pohledu této práce si ji dovolím zařadit mezi ROM z důvodu nemožnosti ji elektronicky přepsat.

Paměti dočasného charakteru jsou prakticky vždy přepisovatelné. Tyto paměti dále rozdělujeme na statické (SRAM) a dynamické (DRAM).

SRAM je polovodičová paměť, jejíž základní buňkou je klopný obvod (tzv. flip--flop) tvořený dvěma invertory zapojenými do kladné zpětné vazby. Příklad takové buňky v technologii CMOS lze vidět na obrázku 1.2. Takto tvořená buňka drží uloženou hodnotu, dokud je připojeno napájecí napětí a není třeba data pravidelně obnovovat.



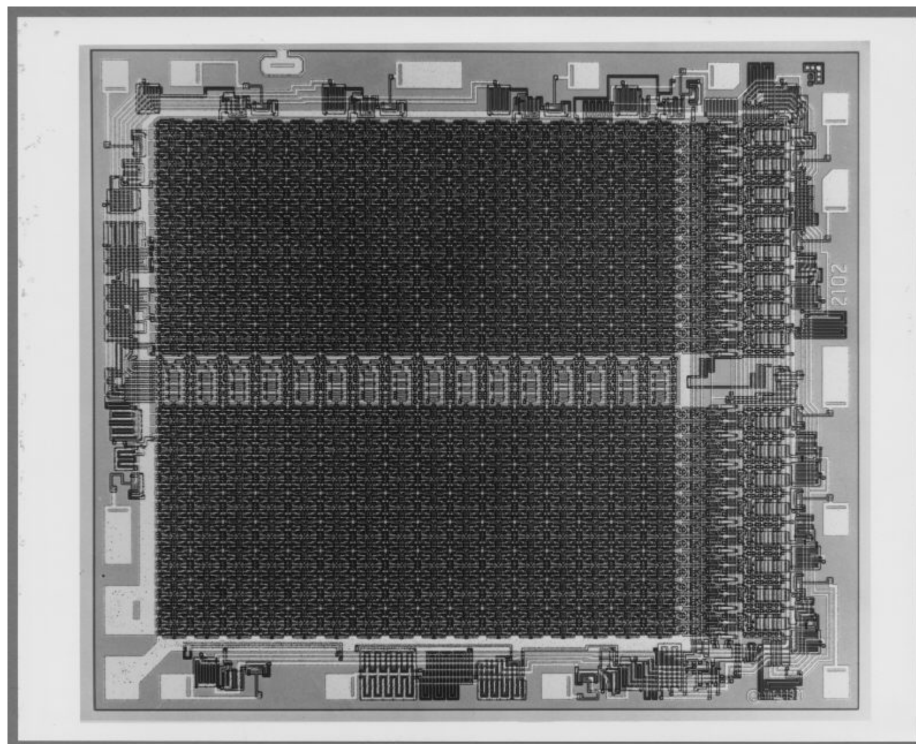
Obr. 1.2: Paměťová buňka SRAM v technologii CMOS. Tato buňka je označována jako 6T.

DRAM je polovodičová paměť, jejíž základní buňku tvoří kondenzátor vytvořený ve struktuře integrovaného obvodu. Z toho plyne nutnost data v dynamické paměti pravidelně obnovovat (tzv. refresh) z důvodu vybíjení paměťové kapacity svodem. Taktéž čtení dynamické paměti je pro uložená data destruktivní. Z toho vyplývá nutnost opětovného zápisu dat do paměti po přečtení.

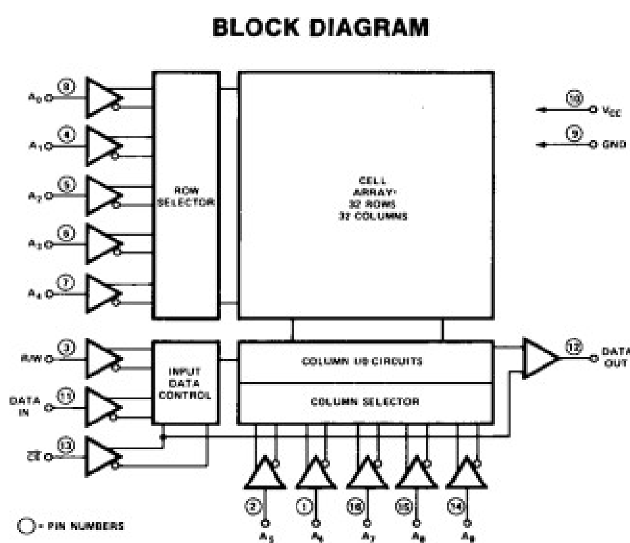
Paměti můžeme dále rozdělit podle jejich komunikačních rozhraní na synchronní a asynchronní. U synchronních pamětí probíhá komunikace v pravidelných časových okamžicích - taktech. U asynchronních je pouze udána doba, do které paměť zareaguje na žádanou operaci.

1.2 Organizace paměťových buněk

Paměť se skládá z většího množství paměťových buněk rozmístěných po ploše integrovaného obvodu. Jako příklad může posloužit SRAM paměť Intel 2102 (1024x1). Její popis je dostupný v datasheetu, například na [5]. Blokové schéma je na obrázku 1.4. Vyrobený integrovaný obvod je vidět na obrázku 1.3. Z datasheetu a fotky čipu je poznat, že jde o paměť s organizací 32 řádků a 16 sloupců o dvou bankách. Adresní sběrnice A[9:0] se dělí na řádkovou adresu A[4:0] a sloupcovou adresu A[9:5]. Bit, který vybírá banku paměti, nelze z datasheetu poznat.



Obr. 1.3: Foto integrované paměti Intel 2102[3]



Obr. 1.4: Blokové schéma paměti Intel 2102 [5]

Informace o rozdělení adresy na řádkovou adresu, sloupcovou adresu a bankovou adresu je nezbytné znát pro určení fyzické pozice paměťové buňky a logických adres sousedních buněk.

[3]Převzato z `http://www.oac.cdlib.org/ark:/13030/kt7k4022nm/?brand=oac4`

2 DEFEKTY PAMĚTÍ

Pro dělení defektů pamětí vyjdeme ze struktury na obrázku 1.1. Adresní dekodéry budou v textu chápány jako jediný blok.

Defekty pamětí můžeme rozdělit na globální a lokální. Za globální defekt považujeme chybu, která postihne celou křemíkovou desku. Může jít například o vynechaný výrobní krok, případně odchylku parametrů mimo výrobní toleranci. Za lokální chyby považujeme chyby, které postihnou pouze malou oblast křemíkové desky. Může jít například o prachovou částici nebo jinou nečistotu, která se během výroby dostala na křemíkovou desku. V následujícím textu se zabýváme lokálními defekty.

2.1 Defekty adresního dekodéru

Defekty v adresním dekodéru (a to v řádkovém i sloupcovém) myslíme takové chyby, kdy je pro operaci nad pamětí vybrána jiná paměťová buňka, než udává použitá adresa, případně není vybrána žádná. Rozlišujeme následující chyby:

1. Zvolená adresa nepřistupuje na žádnou paměťovou buňku.
2. Na paměťovou buňku nelze přistoupit žádnou adresou.
3. Jedna adresa vybere dvě paměťové buňky.
4. Dvě adresy přistupují na jednu paměťovou buňku.

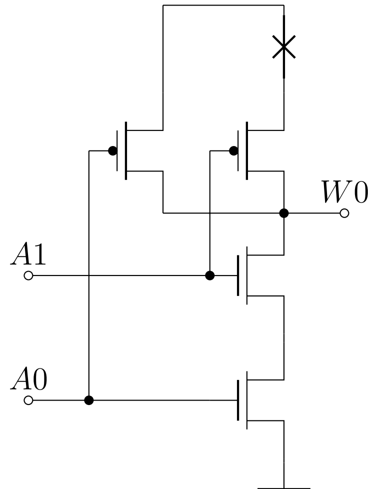
Tyto chyby se z pohledu detekce vyskytují vždy v párech, a to (1,2), (2,3), (1,4) a (3,4). Pro detekci těchto chyb postačuje přečíst všechny paměťové buňky ve stavech 1 a 0.

Dále můžeme v CMOS technologii sledovat defekt označený jako ADOF[1][2]. Pro definici ADOF uvažujeme adresní dekodér dle schématu 2.1. Pokud dojde k defektu naznačenému na obrázku 2.1 křížkem, adresní dekodér selže při klesající hraně na A1, kdy se výstup W0 nezmění a paměť přistoupí na starou adresu. Pro stimulaci chyby v tomto případě je třeba provést následující:

- Zapsat x na adresu 00.
- Zapsat $\overline{x}$ na adresu 10.
- Přečíst hodnotu na adrese 00. Hodnota $\overline{x}$ znamená chybu.

Pro stimulaci ADOF defektů postačuje test pro běžné chyby adresního dekodéru, ale adresní posloupnost musí obsahovat všechny možné přechody, při kterých dojde ke změně pouze jediného bitu adresy.

[1]angl. Address Decoder Open Fault



Obr. 2.1: Defekt adresního dekodéru ADOF

2.2 Defekty paměťových buněk

Popis defektu paměti se skládá ze dvou výrazů. První popisuje operaci, která chybu nad pamětí zcitliví. Druhá operace popisuje efekt, který má chyba na paměťovou buňku. Popis chyby zapíšeme jako $\langle S, F \rangle$, kde S popisuje krok (nebo kroky) ke zcitlivění chyby a F její projev. Použitý zápis je přebrán z článku [4].

Tab. 2.1: Gramatika pro popis defektů na paměti

Stimulace - S	
L	logická hodnota uložená v paměti
R	čtení paměti
0	zápis log. 0
1	zápis log. 1
$\uparrow$	zápis log. 1 do buňky obsahující log. 0
$\downarrow$	zápis log. 0 do buňky obsahující log. 1
$\updownarrow$	zápis opačné hodnoty, než je nyní v buňce
$\forall$	libovolná operace
L_T	chyba, která se projeví po uplynutí doby na buňce v úrovni L
Chybná hodnota - F	
0	log. 0
1	log. 1
L	původní hodnota buňky

Dále defekty paměťových buněk dělíme dle počtu zúčastněných buněk na defekty jediné buňky, vázané defekty a defekty citlivé na stav okolních buněk.

2.2.1 Stack at fault

Stack at fault (zkráceně SaF nebo SA0, SA1), jsou chyby, kdy defektní paměťová buňka vždy vrátí stejnou hodnotu bez závislosti na stimulaci. Principiálně jde o chyby $\langle\forall, 0\rangle$ a $\langle\forall, 1\rangle$. Jde o nejčastější chyby [4].

Pro detekci postačuje zapsat a přečíst všechny paměťové buňky v obou stavech.

2.2.2 Stuck open fault

Stuck open fault (zkráceně SoF), je chyba, kdy paměťovou buňku nelze adresovat. Projev této chyby záleží na implementaci čtecích zesilovačů. Pokud neobsahují latch, projeví se jako SaF. Pokud čtecí zesilovač obsahuje latch, může dojít k opětovnému vrácení poslední čtené hodnoty. Chyba se pak projeví jako $\langle\forall, L\rangle$.

Pro detekci musí testovací algoritmus během přístupu k jedné adrese přečíst hodnotu x a $\overline{x}$. Tento krok je potřeba provést pro $x = 0, 1$.

2.2.3 Transition fault

Transition fault (zkráceně TF) je chyba, kdy paměťová buňka nezvládne přechod ze stavu 0 do 1 nebo naopak. Jde o chybu $\langle\uparrow, 0\rangle$ $\langle\downarrow, 1\rangle$.

Chyba je popisem i detekcí velmi podobná SaF, jde ale o odlišné chyby. Testovací algoritmus, který pokryje TF, pokryl automaticky i všechny SaF. Pro detekci je třeba provést přechod do obou stavů na každé buňce (0->1 a 1->0). Po obou přechodech je třeba provést čtení paměťových buněk.

2.2.4 Destructive read

Destructive read (zkráceně DR) je chyba, kdy dojde ke změně dat v paměti při čtecí operaci. U této chyby rozlišujeme dvě verze.

První verze je DR, kdy dojde k poruše dat a paměť vrátí chybnou hodnotu. Systematicky ji popíšeme jako $\left\langle R, \overline{L}\right\rangle$. Test se provede stejně jako u SaF a TF.

Druhou verzí je dDR[2]. Při této chybě dojde k vrácení očekávaného výsledku při současném poškození dat uložených v paměti. Jde o chybu $\left\langle RR, \overline{L}\right\rangle$. Pro test této chyby musí dojít k dvojnásobnému čtení paměti[3].

[2] angl. deceptive destructive read
[3] angl. back-to-back read

2.2.5 Data retention fault

Data retention fault (zkráceně DRF) je chyba, kdy dojde po uplynutí určité doby ke změně hodnoty uložené v paměti. Jde o chyby $\langle 1_T, 0\rangle$ a $\langle 0_T, 1\rangle$. Jedna paměťová buňka může vykazovat obě chyby zároveň.

Pro detekci DRF je třeba testovací algoritmy rozšířit o zpoždění, které je potřeba použít pro obě úrovně testované buňky.

2.2.6 Write disturb fault

Write disturb fault (zkráceně WDF) je chyba, kdy dojde zápisem stejné hodnoty do paměťové buňky k inverzi dat. Jde o chyby $\langle 00, 1\rangle$ a $\langle 11, 0\rangle$.

Pro detekci chyb je třeba provést vícenásobný zápis dat do paměti pro oba stavy paměťové buňky.

2.2.7 Coupling fault

Coupling fault (zkráceně CF) je chyba, kdy operace nad jednou buňkou (tzv. agresor), kterou označíme jako i, poškodí data v jiné buňce (tzv. oběť[4]), kterou označíme jako j. Rozlišujeme tři typy:

- Inversion coupling (CFin). Jde o chyby, kdy zápis dat do paměťové buňky i vyústí ve změnu dat v paměťové buňce j. Jde o chyby $\langle \uparrow_i, \downarrow_j\rangle$ a $\langle \downarrow_i, \uparrow_j\rangle$. Obě chyby můžou existovat zároveň nad jedním párem i, j buněk.
- Idempotent coupling (CFid). Jde o chyby, kdy po zápisu dat do paměťové buňky i dojde ke změně paměťové buňky j do hodnoty 0 nebo 1. Jde o chyby $\langle \uparrow_i, 0_j\rangle$, $\langle \downarrow_i, 0_j\rangle$, $\langle \uparrow_i, 1_j\rangle$ a $\langle \downarrow_i, 1_j\rangle$.
- State coupling (CFst). Jde o chybu, kdy buňka i ve stavu 0 nebo 1 vynutí konstantní hodnotu v buňce j. Jde o chyby $\langle 0_i, 0_j\rangle$, $\langle 0_i, 1_j\rangle$, $\langle 1_i, 0_j\rangle$ a $\langle 1_i, 1_j\rangle$.

Pro analýzu pokrytí CF chyb je rozdělíme na dvě části podle poloh buněk i, j, tj. kdy agresor je na nižší adrese a kdy je agresor na vyšší adrese. Následně sledujeme, v jakých stavech se nachází buňky i, j při čtecích operacích.

2.2.8 Pattern sensitive faults

Pattern sensitive faults je chyba paměťové buňky, která se projeví pouze při vhodných hodnotách ostatních buněk změnou stavu testované paměťové buňky. Množinu ostatních paměťových buněk často omezíme pouze na sousední paměťové buňky. Proto je třeba pro testování znát fyzické rozdělení paměti na čipu.

[4] angl. victim

Pro test každé buňky by bylo potřeba projít všech 2^8 kombinací okolních buněk, což představuje vysokou časovou zátěž. Pro zjednodušení jsou zavedena omezená okolí buňky - čtyřokolí a dvouokolí. Typicky je pro test použito dvouokolí, označované jako šachovnice[5]. To umožní pokrýt omezenou množinu NPSF[6]

2.2.9 Linked faults

Za linked faults (vázané chyby) označíme takovou n-tici chyb (typu CF nebo NPSF), která ovlivňuje stejnou cílovou buňku. V takovém případě dojde k maskování chyb paměťové buňky a paměť se může zdát bezchybná i přesto, že obsahuje chyby. Většina testovacích algoritmů detekuje pouze některé vázané chyby.

2.3 Defekty čtecí a zápisové logiky

Statické defekty čtecí a zápisové logiky paměti považujeme za pokryté pomocí testů paměťových buněk, kde každou paměťovou buňku čteme v hodnotě 0 i 1.

Na čtecí logice můžeme pozorovat problém doby zotavení (angl. recovery time), kdy po velkém počtu čtení stejné hodnoty dojde při prvním čtení buňky obsahující opačnou hodnotu k přečtení nesprávné hodnoty.

[5]angl. checkerboard pattern

[6]zkratka z angl. Neighbourhood pattern sensitive fault

3 ALGORITMY PRO TEST PAMĚTÍ

Pro plné otestování paměti by bylo potřeba přečíst každý bit v obou stavech pro všechny kombinace všech ostatních buněk. Náročnost takového testování by byla $\mathcal{O}(n \cdot 2^n)$. Tato složitost v praxi znemožňuje kompletní test paměti a každý testovací algoritmus představuje podstatné zkrácení doby. Testy se snažíme zjednodušit tak, aby zanedbaly pouze nepravděpodobné chyby v paměti.

Pro představu časové náročnosti nám pomůže tabulka 3.1. Zde je jasně vidět důvod snahy použít algoritmy se složitostí $\mathcal{O}(c \cdot n)$.

Tab. 3.1: Závislost požadovaného testovacího času pro různou složitost algoritmu. Předpokládáme testovací kontrolér s jedním cyklem na operaci pracujícím na frekvenci $10MHz$.

	t			
Velikost paměti [B]	$\mathcal{O}(n)$	$\mathcal{O}(n \cdot log(n))$	$\mathcal{O}(n^2)$	$\mathcal{O}(2^n)$
16 k	1.6 ms	6.9 ms	26.8 s	10^{11340} s
512 k	52.4 ms	300.0 ms	7.6 h	
2 M	210.0 ms	1.3 s	5 dní	
100 M	10.5 s	84.1 s	35 let	
2 G	3.6 min	33.4 min	146 století	

Je také důležité zmínit, že se v textu budeme zabývat výlučně výskytem jediné chyby v paměti. V případě vícenásobné chyby může z pohledu testovacího algoritmu docházet k maskování chyb, kdy dvě různé chyby ovlivňující jedinou buňku vedou ke zdánlivě správným výsledkům.

3.1 Klasické testovací algoritmy

3.1.1 Algoritmus 0–1

Jde o nejzákladnější testovací algoritmus pro polovodičové paměti. Podstatou algoritmu je přečíst všechny paměťové buňky v hodnotě 0 a 1. Jde o test popsaný algoritmem:

1. Zapiš na všechny paměťové pozice hodnotu 0.
2. Přečti všechny paměťové pozice a testuj na hodnotu 0.
3. Opakuj body 1 a 2 pro hodnotu 1.

Tento algoritmus plně pokryje defekty SA1 a SA0, pokud je adresní dekodér bez defektu. Úpravou tohoto algoritmu vznikne algoritmus MATS+, který je nejjednodušším algoritmem ze skupiny march algoritmů.

3.1.2 GALPAT, Walking 1/0

Jde o algoritmy se složitostí $\mathcal{O}(n^2)$, které jsou zaměřeny na pokrytí CF. Algoritmus testuje, zda zápis na libovolnou adresu paměti nezmění současně data i na jiné adrese.

GALPAT algoritmus je popsán následujícími kroky:

1. Na všechny adresy je zapsána hodnota x.
2. Na aktuální pozici se zapíše $\overline{x}$.
3. Provede se čtení všech paměťových pozic a porovná se s očekávanou hodnotou. Po každé hodnotě se navíc čte i aktuální pozice.
4. Na aktuální pozici se zapíše x a pokračuje se bodem 2 pro další adresu.
5. Po projití celé paměti se test zopakuje s $x = \overline{x}$.

Algoritmus Walking 1/0[1] ve 3. bodě opakovaně nečte aktuální pozici a její čtení provede až na konci smyčky.

Tyto algoritmy plně pokryjí všechny SA0, SA1, TF, CF a chyby adresního dekodéru.

3.1.3 Sliding diagonal

Princip algoritmu je stejný jako v případě GALPAT. Aktuální pozice však není jediná adresa, ale množina adres tvořících diagonálu přes geometrické rozložení paměťového pole. Tím dojde ke snížení složitosti na $\mathcal{O}(n^{2/3})$ za cenu toho, že některé CF nejsou detekovány.

3.2 March algoritmy

March algoritmy jsou množinou algoritmů pro testování pamětí s náročností $\mathcal{O}(c{\cdot}n)$. March algoritmus sestává ze sekvence čtení a zápisů do paměti, které jsou seskupené do march elementů, které aplikujeme na každou paměťovou pozici paměťového pole ve vzestupném nebo sestupném pořadí adres. March algoritmy předpokládají paměť s bitovým přístupem.

Pro zápis march algoritmů využijeme gramatiku podle tabulky 3.2.

March algoritmy se označují jménem, případně délkou algoritmu. Seznam základních march algoritmů je v tabulce 3.3. Pokrytí jednobitových chyb uvedenými algoritmy je vidět v tabulce 3.4. Pokrytí WDF, DRF a dDR chyb lze jednoduše doplnit i do ostatních testů zdvojením některých čtecích a zápisových operací a doplněním zpoždění do testu.

[1] Označován také jako WALPAT (WALking PATtern).

Tab. 3.2: Gramatika march testů

Zápis	Význam
$w0/w1$	Zápis symbolu 0/1 na aktuální paměťovou pozici
$r0/r1$	Čtení aktuální paměťové pozice a porovnání s očekávaným symbolem
$\Uparrow$ (...)	Skupina čtení/zápisů prováděná na celé paměti adresované postupně ve vzestupném pořadí
$\Downarrow$ (...)	Skupina čtení/zápisů prováděná na celé paměti adresované postupně v sestupném pořadí
$\Updownarrow$ (...)	Skupina čtení/zápisů prováděná na celé paměti adresované postupně v libovolném pořadí
Del	Pauza v testovacím algoritmu pro stimulaci DRF[2]

3.2.1 Volnosti march algoritmů

Při implementaci march algoritmů můžeme provést jistá zjednodušení nebo optimalizace při zachování vlastností implementovaného algoritmu. Tyto oblasti označíme jako volnosti march algoritmů [6]. V této práci užijeme následující dva z šesti pozorovaných:

Jako první stupeň volnosti představuje možnost použít jiný než prostý binární čítač pro adresaci paměťových buněk. Podmínkou pro implemetovaný čítač je projití všech 2^n stavů. V sekvenci se také musí vyskytnout každý stav právě jednou. U vybraného čítače musí existovat možnost čítat v opačném pořadí pro implementaci „čítání směrem dolů“.

Druhý stupeň volnosti je v testovacích datech. Pro test je třeba využít dvojici navzájem inverzních dat, tj. 0 a 1. Tato testovací slova nemusí být stejná pro všechny paměťové pozice a lze je generovat například z adresy. Nutnou podmínkou je, aby datová slova odpovídající 0 a 1 byla pro danou adresu konstantní během celého march algoritmu.

[2] Patří k operacím, o které se standardní march testy rozšiřují.

Tab. 3.3: March testy

Název testu	Kód testu	Zdroj
MATS ($4 \cdot n$)	$\Updownarrow (w0) \Updownarrow (r0, w1) \Updownarrow (r1)$	[7]
MATS+ ($5 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0)$	[4][7][8]
MATS++ ($6 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0, r0)$	[7]
March X ($6 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1) \Downarrow (r1, w0) \Updownarrow (r0)$	[7]
March C ($11 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Uparrow (r0) \Downarrow (r0, w1)$ $\Downarrow (r1, w0) \Updownarrow (r0)$	[7]
March C- ($10 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1) \Uparrow (r1, w0) \Downarrow (r0, w1)$ $\Downarrow (r1, w0) \Updownarrow (r0)$	[4][7][8]
March A ($15 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, w0, w1) \Uparrow (r1, w0, w1)$ $\Downarrow (r1, w0, w1, w0) \Downarrow (r0, w1, w0)$	[7]
March Y ($8 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, r1) \Downarrow (r1, w0, r0) \Updownarrow (r0)$	[7]
March B ($16 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, r1, w0, w1) \Uparrow (r1, w0, w1)$ $\Downarrow (r1, w0, w1, w0) \Downarrow (r0, w1, w0)$	[4][7][8]
March G ($23 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, r1, w0, r0, w1) \Uparrow (r1, w0, w1)$ $\Downarrow (r1, w0, w1, w0) \Downarrow (r0, w1, w0) Del$ $\Updownarrow (r0, w1, r1) Del \Updownarrow (r1, w0, r0)$	[4]
March SS ($22 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, r0, w0, r0, w1)$ $\Uparrow (r1, r1, w1, r1, w0) \Downarrow (r0, r0, w0, r0, w1)$ $\Downarrow (r1, r1, w1, r1, w0) \Updownarrow (r0)$	[8]
March U ($13 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, r1, w0) \Uparrow (r0, w1)$ $\Downarrow (r1, w0, r0, w1) \Downarrow (r1, w0)$	[8]
March LR ($14 \cdot n$)	$\Updownarrow (w0) \Downarrow (r0, w1) \Uparrow (r1, w0, r0, w1) \Uparrow (r1, w0)$ $\Uparrow (r0, w1, r1, w0) \Uparrow (r0)$	[8]
March SR ($14 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, r1, w0) \Uparrow (r0, r0) \Uparrow (w1)$ $\Downarrow (r1, w0, r0, w1) \Downarrow (r1, r1)$	[8]
PMOVI ($13 \cdot n$)	$\Updownarrow (w0) \Uparrow (r0, w1, r1) \Uparrow (r1, w0, r0)$ $\Downarrow (r0, w1, r1) \Downarrow (r1, w0, r0)$	[8]

Tab. 3.4: Pokrytí jednobitových chyb march algoritmy

March test	SaF	SoF	TF	DR	dDR	DRF	WDF
MATS ($4 \cdot n$)	2/2	0/1	0/2	2/2	0/2	0/2	0/2
MATS+ ($5 \cdot n$)	2/2	0/1	1/2	2/2	0/2	0/2	0/2
MATS++ ($6 \cdot n$)	2/2	0/1	2/2	2/2	0/2	0/2	0/2
March X ($6 \cdot n$)	2/2	0/1	2/2	2/2	0/2	0/2	0/2
March C ($11 \cdot n$)	2/2	0/1	2/2	2/2	0/2	0/2	0/2
March C- ($10 \cdot n$)	2/2	0/1	2/2	2/2	0/2	0/2	0/2
March A ($15 \cdot n$)	2/2	0/1	2/2	2/2	0/2	0/2	0/2
March Y ($8 \cdot n$)	2/2	1/1	2/2	2/2	0/2	0/2	0/2
March B ($16 \cdot n$)	2/2	0/1	2/2	2/2	0/2	0/2	0/2
March G ($23 \cdot n$)	2/2	1/1	2/2	2/2	0/2	2/2	0/2
March SS ($22 \cdot n$)	2/2	0/1	2/2	2/2	2/2	0/2	2/2
March U ($13 \cdot n$)	2/2	1/1	2/2	2/2	0/2	0/2	0/2
March LR ($14 \cdot n$)	2/2	1/1	2/2	2/2	0/2	0/2	0/2
March SR ($14 \cdot n$)	2/2	1/1	2/2	2/2	2/2	0/2	0/2
PMOVI ($13 \cdot n$)	2/2	1/1	2/2	2/2	2/2	0/2	0/2

Tab. 3.5: Pokrytí vícebitových chyb march algoritmy

March test	CFst	CFid	CFin
MATS+ ($5 \cdot n$)	4/8	3/8	0/4
MATS++ ($6 \cdot n$)	4/8	3/8	0/4
March X ($6 \cdot n$)	5/8	4/8	0/4
March C ($11 \cdot n$)	8/8	8/8	4/4
March C- ($10 \cdot n$)	8/8	8/8	4/4
March A ($15 \cdot n$)	6/8	8/8	4/4
March Y ($8 \cdot n$)	5/8	4/8	0/4
March B ($16 \cdot n$)	6/8	8/8	4/4
March G ($23 \cdot n$)	6/8	8/8	4/4
March SS ($22 \cdot n$)	8/8	8/8	4/4
March U ($13 \cdot n$)	8/8	8/8	4/4
March LR ($14 \cdot n$)	8/8	8/8	4/4
March SR ($14 \cdot n$)	8/8	8/8	4/4
PMOVI ($13 \cdot n$)	8/8	7/8	3/4

3.2.2 Paměti s přístupem po slovech

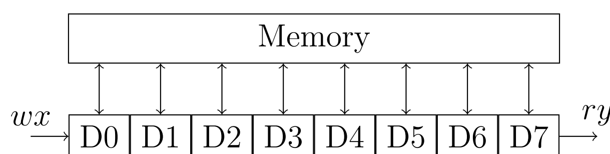
Při testu pamětí, které mají datové slovo širší než jeden bit, dochází k problému, protože nelze k paměti přistupovat po jednotlivých bitech. Z toho důvodu by nešlo stimulovat CF mezi jednotlivými bity paměťového slova a tím by došlo ke snížení pokrytí chyb v testu. Pro stimulaci CF mezi jednotlivými datovými linkami použijeme různá datová slova, která budeme během testu měnit. Pro generaci testu se dá použít pevná množina datových slov, která testují různé kombinace a jejich inverze (např.: 000, 111, 101, 010). Tento postup používá například BIST kontrolér firmy Virage Logic.

Pokud se zaměříme pouze na pokrytí datových linek a čtecích zesilovačů, bude možno využít pseudonáhodné generování testovacích dat. Toto si můžeme dovolit díky volnosti v march algoritmech. Testovací data generujeme logickou funkcí, která má za vstup aktuální adresu. Tato změna je velice nenáročná na velikost BIST kontroléru.

Poslední způsob pro testování pamětí s přístupem po slovech je použít smarch algoritmy, které budou předmětem další kapitoly.

3.3 Smarch algoritmy

Smarch neboli „serial march“ algoritmy jsou odvozeny z původních march algoritmů pro testování paměti, která přistupuje k datům na úrovni slov. Smarch algoritmy využívají sériový přístup k paměti, kdy jsou jednotlivé bity při každé operaci na paměti rotovány přes paměťové slovo [9]. Princip takového přístupu je znázorněn na obrázku 3.1.



Obr. 3.1: Přístup k datovým bitům v smarch algoritmu

Tato změna také znamená, že operace zápis a čtení je potřeba provádět vždy současně. V popisu smarch algoritmu se budou vyskytovat pouze přístupy $wxry$, kde x jsou data určená k zápisu do paměti a y data právě čtená z paměti. Výstup y je nutné v prvních march elementech ignorovat. O tuto možnost je třeba rozšířit gramatiku march testů – tuto operaci formálně zapíšeme jako $(wxry)^n$.

Příkladem smarch algoritmů může být algoritmus SMarchCHKBci od firmy Mentor Graphics Corporation [10]. Tento algoritmus je popsán výrazem 3.1[3]. Tento algoritmus plně pokryje chyby SaF, SoF, TF, DR, dDR, DRF, WDF, CFst, CFid, CFin a defekty adresního dekodéru.

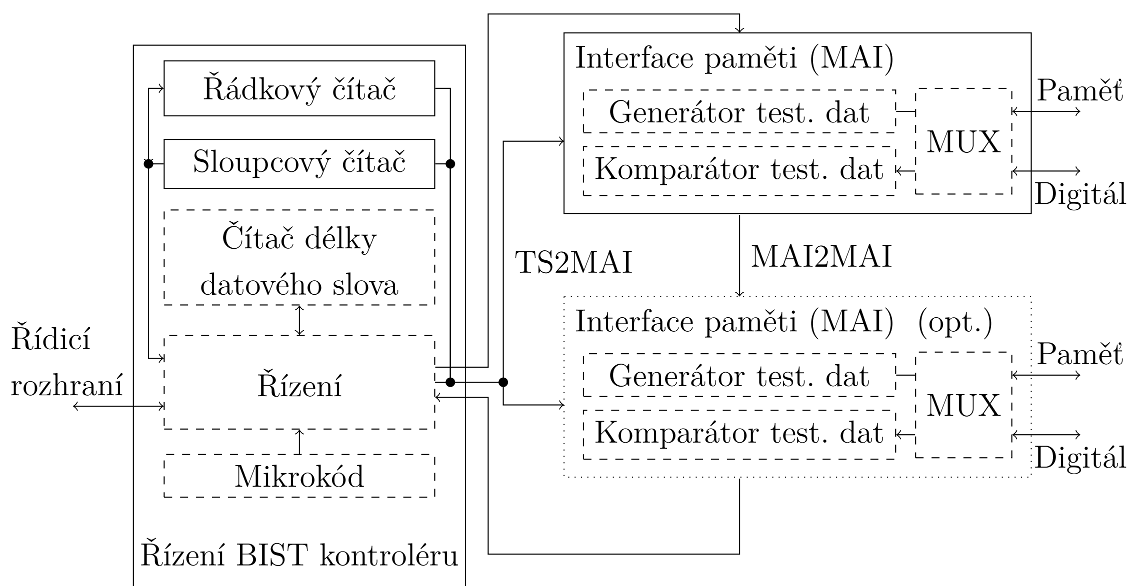
$$\begin{aligned}
&\Updownarrow ((w1rx)^n, r1, w1) \Updownarrow ((w0r1)^n, r0, r0) \Uparrow ((w0rx)^n, r0)\\
&\Uparrow ((w0r0)^n) \Uparrow ((w1r0)^n) \Uparrow ((w1r1)^n) \Uparrow ((w0r1)^n)\\
&\Updownarrow ((w0rx)^n, r0, r0) \Uparrow (w0, r0) \Uparrow ((w1r0)^n, r1, w1)\\
&\Uparrow ((w0r1)^n, r0, w0) \Downarrow ((w1r0)^n, r1, r1)\\
&\Downarrow ((w0r1)^n, r0, r0) \Downarrow ((w0r0)^n) \Updownarrow ((w1rx)^n, r1, r1)\\
&\Uparrow (w1, r1) \Uparrow ((w1r1)^n) \Updownarrow ((w0r1)^n, w0, r0) \Uparrow (w0, r0)
\end{aligned} \tag{3.1}$$

[3] Algoritmus není přepsán úplně správně. V originálním algoritmu jsou bloky bez udání směru čítání provedeny pouze pro jedinou adresu.

4 IMPLEMENTACE

4.1 Bloková struktura

Testovací march kontrolér můžeme systematicky rozdělit na dvě části, přičemž jedna se stará o interpretaci mikrokódu a řízení kontroléru a druhá zajišťuje přístup k paměťovému poli. Toto rozdělení umožňuje přepoužít část logiky testovacího kontroléru a tím snížit nároky na implementaci a v konečném důsledku uspořit plochu křemíkového čipu. Bloková struktura je na obrázku 4.1.



Obr. 4.1: Blokové rozdělení testovacího kontroléru

4.1.1 Vnější rozhraní kontroléru

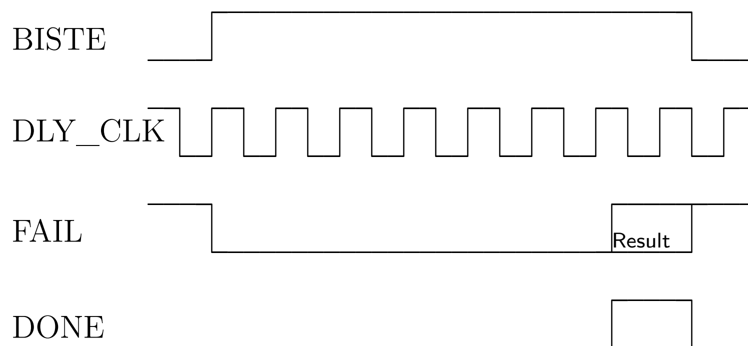
Rozhraní pro ovládání BIST kontroléru sestává z následujících signálů:

- BISTE – vstup – povolení testovacího kontroléru a spuštění testu.
- DLY_CLK – vstup – hodinový signál udávající dobu čekání pro delay operaci.

Pro zjištění výsledku testu se použijí následujíci dva signály:

- FAIL – výstup – v případě, že `FAIL = 1`, kontrolér našel v testované paměti chybu.
- DONE – výstup – kontrolér prošel celý mikroprogram a zastavil se.

Časování těchto signálů je vidět na obrázku 4.2.

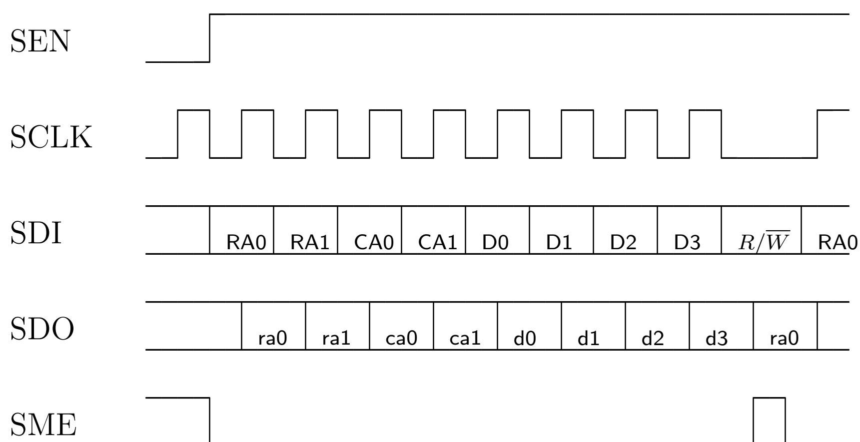


Obr. 4.2: Časování signálů BIST kontroléru

V případě neúspěchu testu je také nezbytné mít k dispozici diagnostické rozhraní, pomocí kterého můžeme detailněji analyzovat chybu. Jde o velmi jednoduché sériové rozhraní přirovnatelné k SPI. Signály jsou následující:

- SEN – vstup – povolení sériového přístupu
- SCLK – vstup – hodiny sériového rozhraní
- SDI – vstup – sériový vstup dat
- SDO – výstup – sériový výstup dat
- SME – vstup – provedení vybrané operace nad pamětí

SDI je vzorkované na nástupné hraně signálu SCLK. Data na SDO jsou stabilní při sestupné hraně hodin. Časování je na obrázku 4.3. Malými písmeny jsou označeny hodnoty vyčítané z BIST kontroléru. Příklad je uveden pro čtyřbitovou adresní i datovou sběrnici. Přejít do sériového řízení bude možno ihned po skončení testu.



Obr. 4.3: Časování signálů pro sériový přístup k paměti pomocí BIST kontroléru.

4.1.2 Řízení BIST kontroléru

Blok je implementovaný v souboru `mbist_ts_core`. Jde o část BIST, která sdružuje bloky, které jsou znovupoužitelné pro více pamětí.

4.1.3 Řádkový a sloupcový čítač

Jde o čítače paměťové adresy (řádkové a sloupcové) pro march mikroprogram. Tyto dva tvoří dohromady adresní čítač. Jsou takto rozděleny, aby mohl algoritmus přistupovat k paměti po sloupcích nebo po řádcích. Pro tyto čítače platí volnost march algoritmu. Jejich implementací se podrobněji zabývá kapitola 4.3.

Čítač je implementovaný v modulu `mbist_adr_cnt`.

4.1.4 Čítač šířky datového slova

Jde o čítač pro smarch režim. Tento čítač odpočítává počet operací čtení/zápisu v jedné mikroinstrukci. Jedná se o obyčejný binární čítač se zkráceným cyklem.

Čítač je implementovaný v modulu `mbist_ncnt`.

4.1.5 Mikrokód

Paměť s mikrokódem je implementována jako nejmenší look-uptabulka o délce 2^n. Popisu mikrokódu se věnuje kapitola 4.2.

4.1.6 Řízení

Jde o logiku implementovanou v modulu řízení BIST kontroléru. Jedná se o:

- Detekci náběžné hrany na signálu SCLK
- Generování výstupních signálů FAIL a DONE
- Generování šachovnicového vzoru

Pro komunikaci s jednotlivými interface pamětí využívá blok řízení rozhraní `TS2MAI` a `MAI2MAI`.

`TS2MAI` je použito k signalizování žádaných operací a adres, na kterých má být provedena. Skládá se z následujících signálů:

- `col_addr` – aktuální řádková adresa
- `row_addr` – aktuální sloupcová adresa
- `op` – aktuální operace prováděná nad pamětí
- `op_dly` – operace prováděná v minulém kroku
- `data` – datový vektor k použití
- `sdi` – signál `SDI`
- `sclk_edge` – hrana na SCLK signálu pro sériový přístup

- `sme` – signál `SME`
- `sen` – signál `SEN`
- `biste` – signál `BISTE`

`MAI2MAI` se využívá pro zapojení jednotlivých interface pamětí do jednoho řetízku. Tohoto uspořádání se využívá pro sériový přístup k paměti a testy pracující v smarch režimu. Skládá se z následujících signálů:

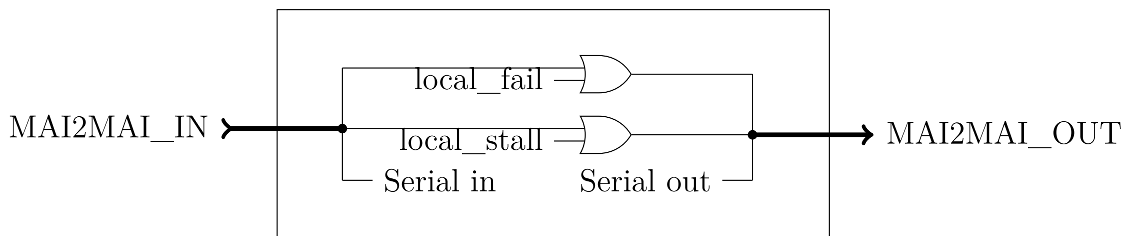
- `sd` – sériová data mezi interface pamětí
- `fail_c` – byla detekována chyba na paměti
- `stall_c` – požadavek na pozdržení testovací sekvence

4.1.7 Interface paměti

Interface paměti zajišťuje generování testovacích dat a jejich verifikaci v normálním módu. Pokud BIST kontrolér pracuje v módu smarch nebo sériového přístupu k paměti, chovají se moduly pouze jako posuvný registr.

Interface paměti také musí řešit přístup na neexistující adresu paměti. Tento problém vzniká při použití jediného bloku řízení BIST kontroléru pro více pamětí.

Každý interface paměti má vstup a výstup sběrnice `MAI2MAI`. Tato sběrnice musí být zapojena podle obrázku 4.4.



Obr. 4.4: Zapojení MAI2MAI interface uvnitř modulu interafce paměti

4.2 Mikrokód

Nejjednodušší implementace mikrokódu vychází přímo z march zápisu převedeného do bitové podoby 4.1. Konec programu je v tomto případě detekován jako přetečení adresy mikrokódu. Kratší mikroprogramy musí být tedy doplněny výplňovými instrukcemi do délky 2^n.

Tab. 4.1: Nejjednodušší mikrokód

Číslo bitu	Význam
0	Adresace vzestupně/sestupně
1	Operace čtení/zápis
2	Slovo 0/1
3	Konec smyčky (konec march elementu)

Tento mikrokód však není vhodný pro implementaci a je omezující pro march testy. Omezení plyne z možnosti využít pouze jediný pár slov pro test paměti a nemožnosti využití šachovnice pro test NPSF. Dále se dá ušetřit na logice řadiče, pokud se rozhodneme na návrat ve smyčce uvést přímo adresu a nevyužít registr s návratovou adresou. Příklad takového mikrokódu je v tabulce 4.2. V této tabulce ve sloupci číslo bitu n označuje počet bitů pro identifikaci datového slova a m udává počet bitů pro šířku adresy mikrokódu.

Tab. 4.2: Mikrokód rozšířený pro více různých datových slov a návratovou adresu

Číslo bitu	Význam
0	Adresace vzestupně/sestupně
1	Operace čtení/zápis
2	Generování šachovnice
3…n+3	Identifikace použitého slova
n+4	Konec smyčky (konec march elementu)
n+5…n+5+m	Návratová adresa

Při rozdělení adresních čítačů je vhodné mít možnost je využít nezávisle na sobě pro adresaci „po řádcích“ nebo „po sloupcích“. Také je třeba implementovat podporu pro smarch algoritmy. Návrh mikrokódu je v tabulce 4.3. Tento mikrokód neomezí implementaci march a smarch testovacích algoritmů. Široké mikroinstrukce umožní použití jednoduché řídicí logiky v MBIST kontroléru. Samotnou implementaci mikrokódu look-up tabulkou zjednoduší syntéza na minimální rozměr.[1]

Pro implementaci mikrokódu byl nakonec zvolen mikrosequencer. Mikrokód tak byl rozšířen o adresu, kam se má skočit, pokud podmínka neplatí. Implementovanou verzi mikrokódu popisuje tabulka 4.4. Identifikace použitých dat pro operaci march algoritmu musí být alespoň 3-bitová, protože se stejný vektor používá i pro data pro smarch operaci. V takovém případě nultý bit udává data zapisovaná do paměti a první bit udává očekávaná čtená data. Druhý bit umožňuje ignorovat výsledek čtecí operace.

[1]Očekáváme povolenou optimalizaci přes hranice buněk.

Tab. 4.3: Mikrokód rozšířený o adresaci „po sloupcích“ a „po řádcích“ a o podporu smarch algoritmů

Číslo bitu	Význam
0	Povolení čítání řádkové adresy
1	Nulování řádkové adresy
2	Čítání řádkové adresy nahoru/dolu
3	Povolení čítání sloupcové adresy
4	Nulování sloupcové adresy
5	Čítání sloupcové adresy nahoru/dolu
6	Generování šachovnice
7	Operace zápis/čtení (v případě `SMARCH = 1` data k zápisu)
8...n+8	Testovací datový vektor (v případě `SMARCH = 1` očekávaná data)
n+9	Operaci provést v smarch režimu
n+10...n+10+m	Návratová adresa
n+11+m...n+11+m	Podmínka ukončení smyčky

Tab. 4.4: Použitá verze mikrokódu; n udává šířku adresy mikroinstrukce.

Číslo bitu	Typ	Význam
0	std_logic	Směr čítání sloupcové adresy
1	std_logic	Nulování sloupcové adresy
2	std_logic	Povolení čítání sloupcové adresy
3	std_logic	Směr čítání řádkové adresy
4	std_logic	Nulování řádkové adresy
5	std_logic	Povolení čítání řádkové adresy
6	std_logic	Generování šachovnice podle sloupce
7	std_logic	Generování šachovnice podle řádku
8,9	t_OPERATION	Operace nad pamětí
10,11,12	t_DATA_IDX	Data použitá při operaci
13,14	t_CONDITION	Podmínka pro skok
15,15+n	t_MC_ADDR	Odkaz na další adresu v případě, že podmínka neplatí
15+n+1, 15+2n	t_MC_ADDR	Odkaz na další adresu v případě, že podmínka platí

4.2.1 Zápis mikrokódu

Pro zjednodušení zápisu mikrokódu bude využit jazyk umožňující popsat march a smarch algoritmy. Při jeho tvorbě vyjdeme z gramatiky march testu, těžko zapsatelné symboly nahradíme jednoduššími a rozšíříme o další možnosti:

- Generování šachovnice pro pokrytí skupiny NPSF. Generování bude probíhat podle exkluzivního součtu nejméně významných bitů řádkové a sloupcové adresy. Dle jeho výsledku bude proveden zápis vybraného testovacího slova nebo jeho inverze.
- Možnost adresovat paměť po řádcích nebo po sloupcích. K tomuto využijeme fakt, že adresní dekodér je rozdělen na dvě části, které dokáží čítat nezávisle na sobě.

Význam symbolů pro zápis algoritmu je uveden v tabulce 4.5.

Tab. 4.5: Textový zápis march algoritmů

<table>
<tr><th>Textový popis</th><th>Význam</th></tr>
<tr><td><code>.(...)</code></td><td>Provedení operace pouze pro jednu adresu</td></tr>
<tr><td><code>V(...)</code></td><td>$\Downarrow (\ldots)$</td></tr>
<tr><td><code>^(...)</code></td><td>$\Uparrow (\ldots)$</td></tr>
<tr><td><code>^_(...)</code></td><td>$\Uparrow (\ldots)$ po řádcích</td></tr>
<tr><td><code>^|(...)</code></td><td>$\Uparrow (\ldots)$ po sloupcích</td></tr>
<tr><td><code>rx</code></td><td>rx</td></tr>
<tr><td><code>wx</code></td><td>wx</td></tr>
<tr><td><code>rcx</code></td><td>rx šachovnice</td></tr>
<tr><td><code>wcx</code></td><td>wx šachovnice</td></tr>
<tr><td><code>(wxry)^n</code></td><td>Provedení n operací v režimu smarch</td></tr>
<tr><td><code>(wcxrcy)^n</code></td><td>Provedení n operací v režimu smarch při současném generování šachovnice v paměti</td></tr>
<tr><td><code>DELAY</code></td><td>Zpoždění vložené do testovacího algoritmu</td></tr>
</table>

4.3 Čítače

4.3.1 Binární

Běžně se při implementaci march testů využívá obousměrný binární čítač. Takový čítač se skládá z úplných sčítaček (XOR3 a majoritní hradlo) a logiky, která volí vstupní datový vektor pro přičítání $+1$ nebo -1. Nulování musí být prováděno do stavu samých nul nebo jedniček podle nastavení směru čítání.

Požadavky na implementaci obousměrného čítače jsou shrnuty v tabulce 4.6. Velikosti hradel jsou uvedeny poměrově proti hradu NAND2; hradla jsou z technologie amis350uboscd.

Tab. 4.6: Odhad velikosti obousměrného binárního čítače

Hradlo	Velikost	Počet
DFF	5	n
MUX4	6.333	n
ADFULL	5	$n-1$
HALFADD	3	1
Σ		$16.333 \cdot n - 2$

4.3.2 LFSR

LFSR[2] je n-bitový lineární čítač generující pseudonáhodnou posloupnost slov o maximální možné délce $2^n - 1$. Čítač LFSR se skládá z posuvného registru délky n a zpětné vazby, ve které je využita funkce XOR. Z toho vyplývá jeho nenáročnost na velikost implementace.

Jelikož se jedná o lineární čítač, je možno použít k jeho řešení lineární matematické postupy. Následující popis bude probíhat v modulo-2 aritmetice. Sčítání si definujeme jako logickou funkci XOR, násobení jako AND.

Čítač lze popsat maticovou rovnicí 4.1, kde matice **A** plně popisuje zkoumaný čítač včetně typu implementace.

$$\mathbf{X}(n) = \mathbf{A} \cdot \mathbf{X}(n-1) \tag{4.1}$$

Typičtěji se však pro popis LFSR čítače používá charakteristický polynom matice **A**. Ten lze určit dle vzorce 4.2.

$$|x \cdot \mathbf{I} - \mathbf{A}| = p_A(x) \tag{4.2}$$

Pro popis čítače s délkou $2^n - 1$ musí být charakteristický polynom primitivní.

Z rovnice 4.1 vyplývá důvod, proč LFSR čítač maximální délky pokryje pouze $2^n - 1$ stavů. Každou hodnotu čítače je třeba vyjádřit pomocí součtu vybraných hodnot předchozího stavu. Pokud jsou v čítači samé nuly, nemůže v žádném místě čítače vzniknout hodnota různá od nuly a čítač setrvá v nulovém stavu i nadále.

[2] Linear Feedback Shift Register

Reverzní LFSR čítač

Reverzním čítačem je myšlen čítač, který prochází své stavy v opačném pořadí, než čítač dopředný. Postup vedoucí k vytvoření takového čítače je popsán v následujících řádcích. Pro čítač dle rovnice 4.1 lze vyjádřit předchozí stav vynásobením pomocí $\mathbf{A}^{-1}$ zleva jako 4.3.

$$\mathbf{A}^{-1} \cdot \mathbf{X}(n) = \mathbf{X}(n-1) \tag{4.3}$$

Pro praktické použití bude lepší najít charakteristickou rovnici pro $\mathbf{A}^{-1}$ ze znalosti $p_A(x)$. Začneme úpravou rovnice 4.2 vynásobením pomocí $|\mathbf{A}^{-1}|$ a následnou úpravou pomocí rovnosti $|\mathbf{A}| \cdot |\mathbf{B}| = |\mathbf{A} \cdot \mathbf{B}|$.

$$\left|x \cdot \mathbf{A}^{-1} - \mathbf{I}\right| = \left|\mathbf{A}^{-1}\right| \cdot p_A(x) \tag{4.4}$$

Následně provedeme substituci $\mu = x^{-1}$.

$$\left|\mu^{-1} \cdot \mathbf{A}^{-1} - \mathbf{I}\right| = \left|\mathbf{A}^{-1}\right| \cdot p_A(\mu^{-1}) \tag{4.5}$$

Dále rovnici rozšíříme výrazem $-\mu^n$, který na levé straně zapíšeme jako $|-\mu \cdot \mathbf{I}|$. Tím rovnici upravíme na 4.6, která je na levé straně formálně stejná jako 4.2 pro matici $\mathbf{A}^{-1}$.

$$\left|\mu\mathbf{I} - \mathbf{A}^{-1}\right| = -\mu^n \cdot \left|\mathbf{A}^{-1}\right| \cdot p_A(\mu^{-1}) = p_{A^{-1}}(\mu) \tag{4.6}$$

Zde jsme dokázali, že znalost 4.2 nám stačí k realizaci reverzního LFSR čítače dle rovnice 4.6[3].

Porovnáním obou polynomů zjistíme, že došlo pouze k obrácení pořadí koeficientů. Tyto poznatky se shodují s [1].

Implementace

Rozlišujeme dva způsoby, jak implementovat LFSR čítač - Fibonacci LFSR a Galois LFSR. Pro implementaci byla vybrána verze Fibonacci LFSR. Tato verze má z DFF(D flip-flop) poskládaný nepřerušený posuvný registr, což je výhodné pro implementaci sériového přístupu k paměti.

Fibonacci LFSR

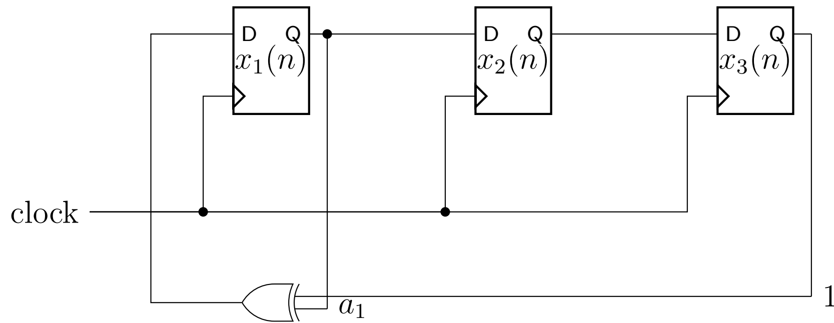
Fibonacci LFSR čítač, jinak také nazývaný standardní, „many-to-one“, „external XOR gate“, využívá posuvný registr délky n, ze kterého jsou vzaty jednotlivé odbočky do zpětnovazební funkce definované charakteristickým polynomem. Fibonacci

[3] Předpokládáme splněnou podmínku $\left|A^{-1}\right| \neq 0$.

LFSR je popsán maticí **A** ve tvaru 4.7. Z této matice lze vypočítat polynom 4.8. Ukázkovou realizaci pro $n = 3$ a $p(x) = x^3 + x^2 + 1$ lze vidět na obrázku 4.5.

$$\mathbf{A} = \begin{vmatrix} a_1 & a_2 & a_3 & \dots & a_{n-1} & 1 \\ 1 & 0 & 0 & \dots & 0 & 0 \\ 0 & 1 & 0 & \dots & 0 & 0 \\ 0 & 0 & 1 & \dots & 0 & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots & \vdots \\ 0 & 0 & 0 & \dots & 1 & 0 \end{vmatrix} \tag{4.7}$$

$$p_A(x) = x^n + a_1 \cdot x^{n-1} + a_2 \cdot x^{n-2} + \ldots + a_{n-1} \cdot x + 1 \tag{4.8}$$



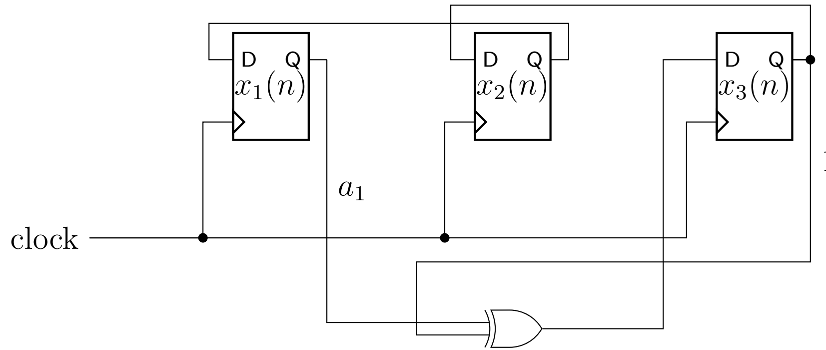
Obr. 4.5: Fibonacci LFSR $n = 3$ a $p(x) = x^3 + x^2 + 1$

Podle předchozích poznatků z kapitoly 4.3.2 můžeme odvodit matici A pro reverzní čítač v implementaci Fibonnaci LFSR. Výpočtem inverzní matice k matici A dostaneme matici 4.9, jejíž charakteristický polynom je popsán výrazem 4.10. Realizace inverzního čítače je na obrázku 4.6.

$$\mathbf{A}^{-1} = \begin{vmatrix} 0 & 1 & 0 & \dots & 0 \\ 0 & 0 & 1 & \dots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & a_1 & a_2 & \dots & a_{n-1} \end{vmatrix} \tag{4.9}$$

$$p_{A^{-1}}(x) = x^n + a_{n-1} \cdot x^{n-1} + a_{n-2} \cdot x^{n-2} + \ldots + a_1 \cdot x + 1 \tag{4.10}$$

Souhrn požadavků na realizaci je vidět v tabulce 4.7, kde k je počet nenulových koeficientů charakteristického polynomu. Pro tuto realizaci ovšem není podmínkou použití hradel XOR2 a lze využít i složitější hradla a ušetřit místo.



Obr. 4.6: Reverzní Fibonacci LFSR $n = 3$ a $p(x) = x^3 + x^2 + 1$

Tab. 4.7: Odhad velikosti dopředného Fibonacci LFSR

Hradlo	Velikost	Počet
DFF	5	n
MUX2	2.333	n
XOR2	2	$k - 1$
AND2	1.333	$n - 1$
OR2	1.333	1
Σ		$8.666 \cdot n + 2 \cdot k - 2$

Tab. 4.8: Odhad velikosti obousměrného Fibonacci LFSR

Hradlo	Velikost	Počet
DFF	5	n
MUX3[4]	3.5	n
XOR2	2	$k - 1$
AND2	1.333	$n - 1$
OR2	1.333	1
MUX2	2.333	k
Σ		$9.8333 \cdot n + 4.333 \cdot k - 2$

4.3.3 CFSR

CFSR[5] je čítač vzniklý rozšířením LFSR čítače o stav `(others => '0')` a tím dosažení všech 2^n stavů, kde n je šířka čítače. Jako základ čítače CFSR využijeme Fibonacci LFSR čítač 4.1 a zavedeme další zpětnou vazbu. Výsledek se chová dle

[5] Complete Feedback Shift Register

rovnice 4.11.

$$\mathbf{X}(n+1) = \mathbf{A} \cdot \mathbf{X}(n) + \begin{bmatrix} \overline{x_0(n)} \cdot \overline{x_1(n)} \cdot \overline{x_2(n)} \cdot \ldots \overline{x_{n-1}(n)} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.11}$$

$$\mathbf{A}^{-1} \cdot \mathbf{X}(n+1) - \mathbf{A}^{-1} \cdot \begin{bmatrix} \overline{x_1(n)} \cdot \overline{x_2(n)} \cdot \overline{x_3(n)} \cdot \ldots \overline{x_{n-1}(n)} \\ 0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \mathbf{X}(n) \tag{4.12}$$

První výraz na levé straně popisuje reverzní LFSR čítač. Druhý výraz popisuje nelineární část zpětovazební funkce pro CFSR čítač. Protože matice **A** je známého tvaru Fibonacci LFSR čítače, můžeme druhý člen přepsat pro stav o jeden krok posunutý k vyšším hodnotám podle výrazu 4.13.

$$\begin{bmatrix} \overline{x_1(n)} \cdot \overline{x_2(n)} \cdot \ldots \cdot \overline{x_{n-1}(n)} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \Rightarrow \begin{bmatrix} \overline{x_2(n+1)} \cdot \overline{x_3(n+1)} \cdot \ldots \cdot \overline{x_n(n+1)} \\ 0 \\ \vdots \\ 0 \end{bmatrix} \tag{4.13}$$

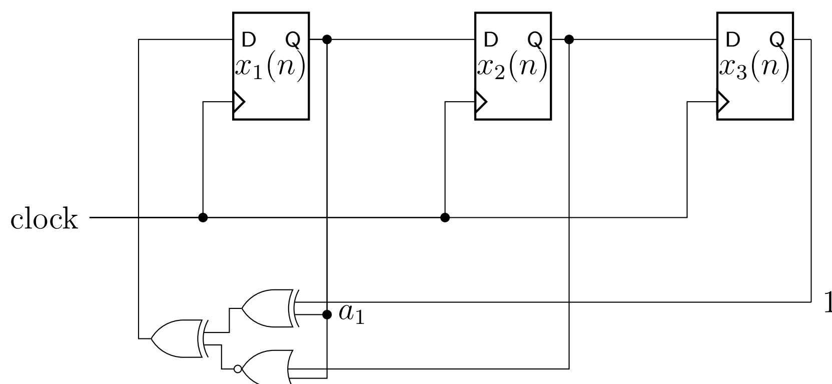
Matice $\mathbf{A}^{-1}$ přičítá vstup x_0 pouze k bitu x_n. Následkem toho rovnici 4.12 přepíšeme na rovnici 4.14. Tuto rovnici použijeme pro implementaci reverzního čítače.

$$\mathbf{A}^{-1} \cdot \mathbf{X}(n+1) + \cdot \begin{bmatrix} 0 \\ 0 \\ \vdots \\ \overline{x_2(n+1)} \cdot \overline{x_3(n+1)} \cdot \overline{x_4(n+1)} \cdot \ldots \overline{x_n(n+1)} \end{bmatrix} = \mathbf{X}(n) \tag{4.14}$$

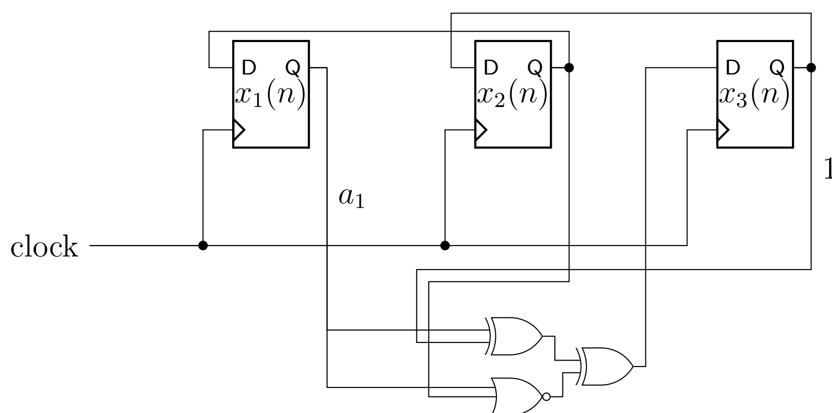
Reálná implementace CFSR čítače je na obrázcích 4.7 a 4.8.

Požadavky na implementaci CFSR čítačů jsou shrnuty v tabulkách 4.9 a 4.10. Pro porovnání s dekadickým čítačem nás budou zajímat dva extrémy, tj. situace, kdy zpětovazební polynom má pouze tři členy ($x^n + x^k + 1$), a situace, kdy by byly využity všechny koeficienty[6].

[6]Tento stav je v praxi neočekávaný.



Obr. 4.7: Implementace dopředného čítače. Základem je LFSR čítač popsaný výrazem $x^3 + x^2 + 1$, doplněný o zpětnou vazbu dle výrazu 4.11.



Obr. 4.8: Implementace zpětného čítače. Základem je zpětný LFSR čítač popsaný výrazem $x^3 + x^2 + 1$, doplněný o zpětnou vazbu dle výrazu 4.14.

Tab. 4.9: Odhad velikosti dopředného CFSR čítače

Hradlo	Velikost	Počet
DFF	5	n
MUX2	2.333	n
XOR2	2	$k - 1$
AND2	1.333	$n - 1$
OR2	1.333	1
OR2	1.333	$n - 1$
INV1	0.666	1
Σ		$9.999 \cdot n + 2 \cdot k - 2.666$

Tab. 4.10: Odhad velikosti obousměrného nulovatelného CFSR čítače

Hradlo	Velikost	Počet
DFF	5	n
MUX3[7]	3.5	n
XOR2	2	$k-1$
AND2	1.333	$n-1$
OR2	1.333	1
MUX2	2.333	k
OR2	1.333	$n-1$
INV1	0.666	1
MUX2	2.333	1
Σ		$11.1333 \cdot n + 4.333 \cdot k - 0.333$

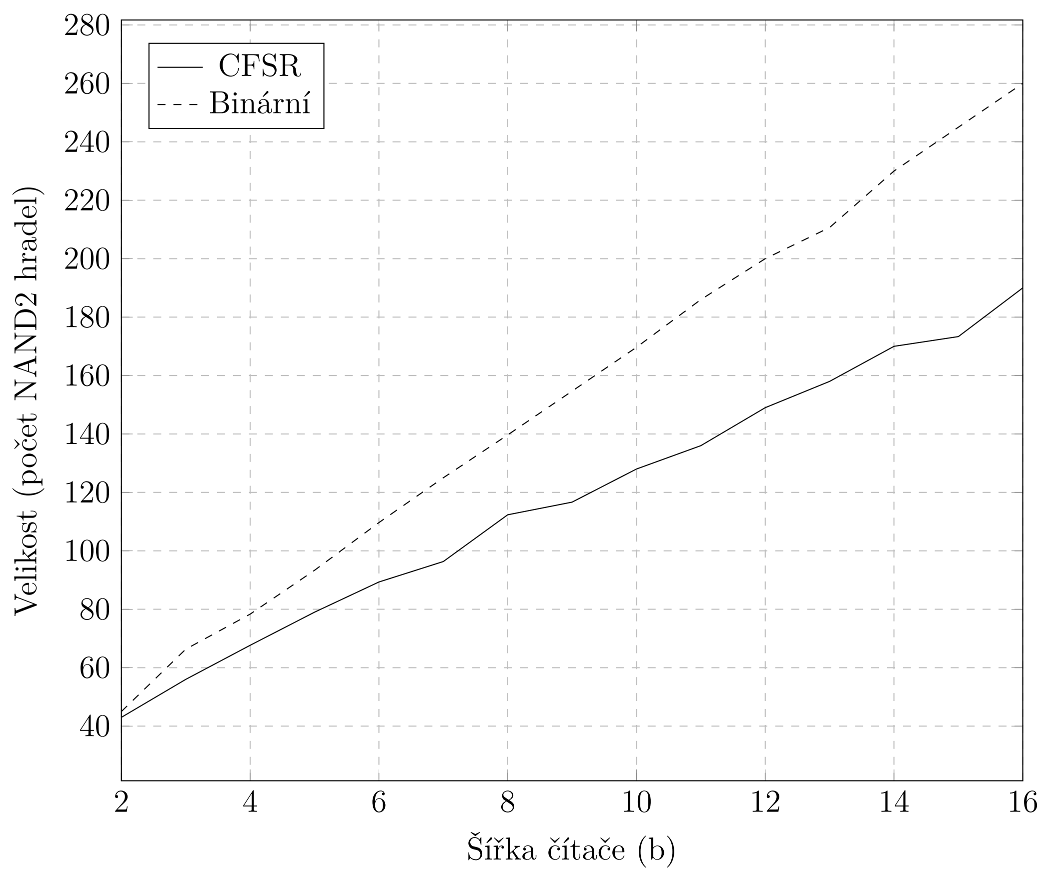
4.3.4 Porovnání

Jako hlavní požadavek na čítač adresy byla zvolena velikost jeho implementace na čipu. Pro porovnání využijeme odhady velikosti z předchozích kapitol. Toto porovnání je vidět v tabulce 4.11. Z tohoto porovnání vychází, že se vyplatí použít CFSR čítač pro šířky větší než 2 bity. Pro ověření této domněnky byly syntetizovány dekadické a CFSR čítače šířky 2-16 bitů. Výsledek porovnání je vidět na obrázku 4.9.

Tab. 4.11: Porovnání odhadu velikostí čítačů

Typ čítače	Odhad velikosti	n pro které vychází plocha menší než binární obousměrný čítač
Binární obousměrný	$16.333 \cdot n - 2$	-
CFSR obousměrný - nejlepší případ	$11.133 \cdot n + 4$	2
CFSR obousměrný - nejhorší případ	$15.433 \cdot n - 0.333$	1

Požadavky na sekvenční část implementace (počet DFF registrů) byl shodný mezi typy, protože čítače mají stejnou šířku. Kombinační část obvodu CFSR byla oproti binárnímu čítači poloviční.

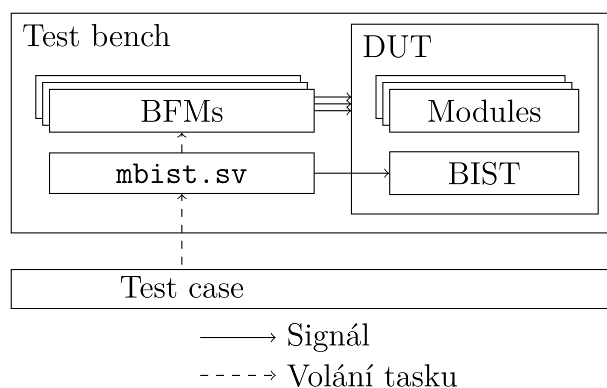


Obr. 4.9: Porovnání velikostí čítačů

5 VERIFIKACE

5.1 Test bench

Implementace testovacího prostředí odpovídá obrázku 5.1. Test bench je modul obsahující jak testovaný obvod (dále označovaný jako DUT), tak moduly sloužící k jeho stimulaci. Samotné testy využívají pouze tasky (případně funkce) jednotlivých BFM[1] pro stimulaci DUT. Toto rozložení otestuje i integraci BIST kontroléru do DUTu. Provede se tedy ověření toho že lze nastavit DUT do takového stavu, aby uživatel mohl přistupovat k BIST kontroléru.



Obr. 5.1: Blokové schéma verifikačního prostředí

Stimulace BIST kontroléru se provádí pomocí BFM v souboru `mbist.sv`. Tento BFM v sobě zahrnuje veškerou komunikaci mezi BIST kontrolérem a testcase. Přesný popis funkcí a jejich chování bude obsažen v následující kapitole.

5.2 Interface testů

Pro stimulaci testovaného obvodu používají testy rozhraní definované v souboru `mbist.sv`. Toto rozhraní slouží k přizpůsobení přístupu k BIST kontroléru v závislosti na požadavcích DUTu. Jde o SystemVerilogový model obsahující interface:

Nastavení testů

function time f_get_test_to()

Vrátí maximální akceptovatelnou dobu trvání testu paměti.

[1] angl. Bus Function Model

function logic[] f_get_address(input logic[] logicAddress)

logicAddress Logická adresa v rámci paměti

Vrátí fyzickou adresu paměťové buňky pro zadanou logickou adresu v rámci paměti. Při výpočtu adresy dbá na rozsahy a pozice řádkových a sloupcových adresních čítačů.

Rozhraní BIST kontroléru

task tsk_enable(input logic _biste, input logic _sen)

_biste Hodnota pro vystavení na `BISTE` pinu

_sen Hodnota pro vystavení na `SEN` pinu

Tato funkce má za úkol nastavení správného test módu integrovaného obvodu, ve kterém se nachází testovaný BIST kontrolér. Následně provede zápis hodnot na pin odpovídající pinům `BISTE`, `SEN` na BIST kontroléru.

task tsk_read_done(input logic exp_done)

exp_done Očekávaná hodnota `DONE` signálu

Tato funkce přečte a porovná hodnotu `DONE` signálu s očekávanou hodnotu. V případě neshody musí zajistit nahlášení chyby a eventuální zastavení testu.

task tsk_wait_done(input t_EDGE edg, input time timeout)

edg Na kterou hranu na signálu `DONE` má task čekat. Možnosti: `{e_POSEDGE, e_NEGEDGE, e_ANYEDGE}`

timeout Maximální doba čekání na hranu

Task počká na **edg** hranu na signálu `DONE`. Pokud ke hraně nedojde do **timeout** času ohlásí task chybu a případně dojde k zastavení testu.

task tsk_read_fail(input logic exp_fail)

exp_fail Očekávaná hodnota `FAIL` signálu

Tato funkce přečte a porovná hodnotu `FAIL` signálu s očekávanou hodnotu. V případě neshody musí zajistit nahlášení chyby a eventuální zastavení testu.

task tsk_wait_fail(input t_EDGE edg, input time timeout)

edg Na kterou hranu na signálu `FAIL` má task čekat. Možnosti: `{e_POSEDGE, e_NEGEDGE, e_ANYEDGE}`

timeout Maximální doba čekání na hranu

Task počká na **edg** hranu na signálu `FAIL`. Pokud ke hraně nedojde do **timeout** času ohlásí, task chybu a případně dojde k zastavení testu.

Sériová diagnostika

task tsk_shift(input logic[] address, input logic[] data, input logic[1:0] operation, input logic rnd_clock)
address Fyzická adresa, na kterou se má přistoupit
data Data pro paměť
operation Operace pro paměť. Možnosti: `{e_OP_RD, e_OP_WR, e_NOP}`
rnd_clock Povolení přidání jitteru do hodinového signálu

Provede sériový přístup k paměti na adresu **address** a dle vybrané operace buď zapíše data do paměti, nebo porovná data přečtená z paměti s očekávanou hodnotou.

Při generování sériového průběhu ponechá signál `SDI` v žádané hodnotě pouze setup/hold time okolo náběžné hrany `SCLK`. Data ze sériového rozhraní čte při sestupné hraně `SCLK`.

Simulace defektu

task tsk_corrupt_memory(input logic[] address, input logic corrupt)
address Na které adrese se má simulovat chyba
corrupt Povolení/zakázání simulace chyby na dané adrese

Pokud **corrupt** = 1, task provede nastavení paměťového modelu tak, aby po přečtení dat z adresy **address** paměť vracela vždy negovanou hodnotu.

Pokud **corrupt** = 0, task nastaví paměťový model tak, aby se na adrese **address** choval jako bezchybná paměť.

5.3 Testy

Výstupy BIST kontroléru jsou permanentně testovány na nežádoucí krátké zákmity (tzv. glitche). Glitch se nesmí objevit na žádném výstupním portu s výjimkou paměťové sběrnice. Jako další je implementován test na nerozhodnou úroveň `X`. V případě jejich výskytu dojde k zastavení testu a nahlášení neúspěchu.

Konstanta `CLKPER` udává hodinovou periodu.

Testy BIST kontroléru

1. Nastavit `BISTE = '1`, počkat `CLKPER` a otestovat, že `FAIL = '0`.
2. Počkat na nástupnou hranu na signálu `DONE`. Za celou dobu se nesmí objevit nástupná hrana na signálu `FAIL`.
3. Nastavit `BISTE = '0`, počkat `CLKPER` a otestovat, že `FAIL = '1, DONE = '0`.
4. Zopakovat body 1–3. (Otestuje, zda jde test paměti opět spustit.)

5. Zopakovat body 1–3. Během testu simulovat chybu v paměti. V bodě 2 se musí objevit nástupná hrana na signálu `FAIL`.
6. Zopakovat body 1–3. Během bodu 2 nastavit `BISTE = '0` a okamžitě přejít na bod 3. (Otestuje přerušení testu v půlce.)
7. Zopakovat body 1–3. Změřit délku celého testu a porovnat s trváním prvního a druhého proběhnutí testu. Délka testu musí být podobná (±`CLKPER`)[2].

Test sériového přístupu

1. Nastavit `BISTE = '0, SEN = '1`.
2. Provede se zapsání adresy a dat do BIST kontroléru pomocí sériové sběrnice. Vstupní data se ponechají validní pouze v setup/hold time okolo nástupné hrany hodin. Na sériové sběrnici se vystaví operace zápisu.
3. Provede se zapsání negace adresy a dat do BIST kontroléru. Zkontroluje se, že na výstupu obdržíme původní adresu a data. Na sériové sběrnici se nevystaví žádná operace.
4. Simulujeme negaci dat v paměti na testované adrese.
5. Provedeme zapsání adresy a dat do BIST kontroléru pomocí sériové sběrnice. Provedeme operaci čtení.
6. Přečteme data z kontroléru a porovnáme, že přečtená data jsou negací očekávaných. Zrušíme negaci dat v paměti na testované adrese.
7. Opakujeme 2–6 pro walking one / walking zero v adrese a datech.

[2] Lze prodloužit v případě, že délku ovlivňují operace, které nemusejí mít opakovatelnou délku.

6 GENEROVÁNÍ BIST KOTROLÉRU

Pro generování BIST kontroléru byla napsaná aplikace `mbistgen`. Tato aplikace se skládá ze dvou částí.

V první části je implementováno načítaní konfiguračních souborů (march algoritmu, souboru s primitivními polynomy a modelu paměti). Jedná se o část psanou v jazyce C s využitím generátoru parserů Flex a Bison[1]. Zde je také implementována kompilace mikrokódu.

V druhé části jde o generování výstupních souborů na základě šablon a generování projektu BIST kontroléru. Tato část využívá jazyka Tcl a velmi jednoduché šablonovací knihovny.

Pro generování CFSR čítačů je potřeba znát alespoň jeden primitivní polynom pro danou šířku čítače. Aplikace využívá seznam primitivních polynomů dostupný na `https://www.commsys.isy.liu.se/~mikol92/polynomials/binary/primitive/one/BinaryPrimPolyList-0002-0100.txt`.

Aplikace byla vyvíjena na distribuci Debian Jessie. Mělo by ji být možné zkompilovat i na jiných POSIX kompatibilních systémech, během práce to však testováno nebylo.

6.1 Model paměti

Pro automatizované generování BIST kontroléru je třeba znát základní informace o testovaných pamětech, primárně jejich interface a logické uspořádání buněk. Z důvodu kompatibility s existujícími nástroji jsem využil existující formát využívaný firmou Mentor Graphics Corporation. Jde o uloženou stromovou strukturu s datovými položkami. Syntaktický formát je následující:

```
/* comment */
section_type {
  property1: 4'b0101;
  property2: 8'hAB;
  property3: -0998;
  property4: yes;
  property5: TEST[3:0];
  subsection (section_name){
    property: value;
  }
}
```

[1] GNU implementace lex a yacc

Hlavní sekcí konfiguračního souboru je `MemoryTemplate`. Popis jednotlivých klíčů je v tabulce 6.1.

Tab. 6.1: Popis konfiguračních sekcí a užitých klíčů.

Sekce MemoryTemplate	
CellName	Název buňky RAM paměti
MemoryType	Typ paměti, musí být `"SRAM"`
NumberofBits	Šířka datového slova paměti
AddressCounter	Sekce popisující rozložení adresy na řádkovou a sloupcovou
Port	Sekce popisující jeden port na paměti
Sekce Port (*Název portu*)	
Direction	Směr portu na paměťovém IP. Možnosti `INPUT` nebo `OUTPUT`.
Function	Využití portu. Možnosti `Data, Address, WriteEnable, Select, Clock, LogicLow, LogicHigh`
Polarity	Aktivní úroveň pinu. Možnosti `ActiveHigh, ActiveLow`
Sekce AddressCounter	
Function(*Name*)	Jeden element popisující adresní čítač. Generující aplikace tento element vyžaduje 3x - s názvy: *Address, RowAddress, ColumnAddress*
Sekce Function (*Name*)	
CountRange[*hi*:*lo*]	Čítač *Name* má rozsah adres od *lo* k *hi*
LogicAddressMap	Udává složení fyzické adresy ze sloupcové/řádkové

Sekce LogicAddressMaP přiřazuje jednotlivé čítače do logické adresy. Jednotlivé položky mají formát: `(Název čítače) [hi:lo]: Address [hi:lo];`.

Příkladem modelu paměti může být výpis 6.1. Z těchto dat je možno vygenerovat vhodný interface k paměti. Položka `NumberofBits` a rozsahy čítačů se použijí pro konfiguraci test kontroléru. Neznáme konfigurační sekce a klíče jsou ignorovány.

Výpis 6.1: Ukázková konfigurace paměti SRAM 512x22

```
MemoryTemplate ( spram512x22cm4Mhz10 ) {
   CellName:        spram512x22cm4Mhz10;
   MemoryType:      SRAM;
   NumberofWords:   512;
   NumberofBits :   22;
   AddressCounter {
     Function (Address) {
       LogicalAddressMaP {
         ColumnAddress [1:0] : Address [1:0] ;
         RowAddress [6:0] : Address [8:2] ;
       }
     }
     Function (Rowaddress) {
       CountRange [0:127];
     }
     Function (Columnaddress) {
       CountRange [0:3];
     }
   }
   Port ( Q[21:0] )
   {
      Direction: OUTPUT;
      Function: data;
   }
   Port ( ADR[8:0] )
   {
     Direction: INPUT;
     Function: Address;
   }
   Port ( D[21:0] )
   {
     Direction: INPUT;
     Function: data;
   }
   Port ( WE )
   {
     Direction: INPUT;
     Function: WriteEnable;
     Polarity: ActiveHigh;
   }
   Port  ( ME )
   {
     Direction: INPUT;
     Function: Select;
     Polarity: ActiveHigh;
   }
   Port ( CLK )
   {
     Direction: INPUT;
     Function: Clock;
     Polarity: ActiveHigh;
   }
}
```

6.2 Šablony souborů

Pro generování výstupních zdrojových kódů byl využít velmi jednoduchý šablonovací systém TemplaTcl[2]. Tento šablonovací systém využívá dvou druhů „tagů“ pro řízení generování výstupního souboru. Tag `<%= expression %>` slouží k vložení hodnoty proměnné nebo výsledku výrazu do výstupu. Tag `<% code %>` slouží pro uvození Tcl kódu k řízení generování výstupu.

Jednoduchý příklad využití je vidět na souboru `templates/mbist_fof.tcl.ttcl` ve výpisu 6.2. Je zde vidět generování seznamu souborů pro kompilaci. Na 5.–7. řádku je smyčka pro všechny prvku předané v seznamu $Memories. Řádek 6 je tedy ve výsledku zopakován pro každý element v poli $Memories, pokaždé s odpovídajícím názvem.

Výpis 6.2: Soubor templates/mbist_fof.tcl.ttcl

```
  VHDL_COMP $env(BM_RTL_DIR)/mbist_pkg.vhd
  VHDL_COMP $env(BM_RTL_DIR)/mbist_cnt.cfsr.rtl.vhd
  VHDL_COMP $env(BM_RTL_DIR)/mbist_ncnt.dec.rtl.vhd
  VHDL_COMP $env(BM_RTL_DIR)/mbist_ts_core.rtl.vhd<%
foreach {memory} $Memories { %>
  VHDL_COMP $env(BM_RTL_DIR)/mbist_mai_<%= [dictGetCI $memory
      ↪ ComponentName] %>.rtl.vhd<%
} %>
  VHDL_COMP $env(BM_RTL_DIR)/mbist_top.str.vhd
```

6.3 Tcl API

V Tcl části jsou nachystané high level metody pro práci s generátorem BIST kontroléru. Všechny Tcl funkce jsou definovány v souboru `bootstrap.tcl`, který se načte po startu. Jedná se o funkce:

createTS targetDir sourceFile memories

Zkompiluje mikrokód ze souboru $sourceFile a vygeneruje blok řízení test kontroléru. Šířky čítačů a počet kroků pro smarch operaci jsou určeny z pamětí v seznamu $memories.

Tato funkce generuje soubory `mbist_ts_core.rtl.vhd` a `mbist_pkg.vhd`, které uloží do adresáře $targetDir. Ke generování použije šablonu `templates/mbist_ts_core.rtl.vhd.ttcl`.

[2] K dispozici na: `http://wiki.tcl.tk/18175`.

createSS targetDir memoryFile

Vygeneruje interface paměti pro model paměti v souboru $memoryFile. Současně vrátí slovník obsahující informace o paměti pro generování TOP modulu.

Výsledek uloží do souboru `mbist_mai_(cellName).rtl.vhd` v adresáři $targetDir. Ke generování používá šablonu `templates/mbist_mai_ssram.rtl.vhd.ttcl`.

createTop targetDir memories memoryNames memoryWords

Vygeneruje top modul pro BIST kontrolér. Do TOP modulu vloží instanci test sequenceru a interface pro všechny paměti uvedené v seznamu $memories. Ke vstupům a výstupům přidá předponu podle seznamu v $memoryNames. Jako testovací vektory používá slova ze seznamu $memoryWords.

Výsledek uloží do souboru `mbist_top.str.vhd`. Ke generování použije šablonu `templates/mbist_top.str.vhd.ttcl`.

createBfm targetDir memories testSequencer

Vygeneruje BFM pro BIST kontrolér. Pro jeho nastavení využije parametry pamětí v seznamu $memories a parametry řízení BIST kontroléru ze slovníku $testSequencer. Výsledek uloží do souboru `mbist.sv`. Ke generování použije šablonu `templates/mbist_bfm.sv.ttcl`.

createFof targetDir memories

Tato ukazuje možnost rozšíření generování o další soubory. V ukázkové implementaci generuje skript sloužící jako seznam jednotlivých HDL souborů ke kompilaci.

Výsledek uloží do souboru `mbist.fof.tcl` v adresáři $targetDir. Ke generování použije šablonu `templates/mbist.fof.tcl.ttcl`.

Příklad jejich využití je na výpise 6.3.

Výpis 6.3: Ukázkový projekt pro mbistgen – paměť SRAM 512x22

```
#Define target directories for specified output types
set targetDir /cesta/k/adresari/s/rtl
set modelDir  /cesta/k/adresari/s/model
set configDir /cesta/k/adresari/s/config

#Prepare lists for data
set memories {}
set words {}
set memNames {}

#Load data about memory
lappend memories [createSS $targetDir $modelDir/
    ↪ spram512x22cm4Mhz10.lvlib]
lappend words [list x\"000000\" x\"3FFFFF\" x\"155555\" x\"2
    ↪ AAAAA\" x\"0CCCCC\" x\"333333\" x\"0F0F0F\" x\"30F0F0\"]
lappend memNames Name

#Create test sequencer
set ts [createTS $targetDir march/SMarchCHKBci.m $memories]

#Create BIST top module
createTop $targetDir $memories $memNames $words

#Create BFM
createBfm $modelDir $memories $ts

#Create files with list of generated files
createFof $configDir $memories
```

6.4 Užití

Aplikace určená pro generování BIST kontroléru má dva základní módy použití.

První je možnost využít napsaný „projekt“ ve formě Tcl skriptu podobného tomu ve výpisu 6.3. Tento způsob má větší flexibilitu a uživatel může nakonfigurovat jednotlivé kroky generování sám. Může také využít aplikaci pro generování BIST kontroléru pro více pamětí. Nachystaný projekt se poté předá jako parametr aplikaci `mbistgen` následovně: `./mbistgen -d projekt.tcl`.

Druhá možnost je využít zjednodušený mód pro generování BIST kontroléru pouze pro jedinou paměť. V tomto případě se všechny potřebné informace předají pomocí parametrů aplikaci `mbistgen`. Formát příkazové řadky je: `./mbistgen -s --memory[memoryModel.lvlib] --target [targetDirectory] --name [connectionPrefix]`, kde `memoryModel.lvlib` představuje soubor s modelem paměti, `connectionPrefix` předponu názvu pro signály pro danou paměť a `targetDirectory`, který ukazuje na adresář s cílovým HDL designem. Aplikace v takovém případě využije předpřipravený projekt v souboru `simple.tcl`. V tomto projektu jsou již nachystané cesty do podadresářů pro jednotlivé generované části. Ten projekt tedy očekává, že v adresáři targetDirectory bude již nachystaná následující adresářová struktura:

```
[targetDir]
├── config
└── hdl
    ├── rtl
    ├── model
    └── testcase
```

V obou případech je interface TOP modulu `MBIST_TOP` následující:

```
entity MBIST_TOP is
  port (
    --Global signals
    CLK    : in std_logic;          -- Global clock signal
    RST_B  : in std_logic;          -- Global reset signal

    --BIST interface
    BISTE  : in std_logic;          -- Enable of bist controller
    DLY_CLK: in std_logic;          -- Clock for delay operation
    FAIL   : out std_logic;         -- Test found error in
       ↪ memory
    DONE   : out std_logic;         -- Test sequence hit end

    -- Section for memory $cennectionPrefix
```

```
    $connectionPrefix_MEM_...          -- Connections for memory
        ↪ cell
    $connectionPrefix_DIG_...          -- Connections for digital
        ↪ logic

    -- Serial diagnostic interface
    SDI    : in std_logic;
    SCLK   : in std_logic;
    SME    : in std_logic;
    SEN    : in std_logic;
    SDO    : out std_logic
  );
end entity MBIST_TOP;
```

7 ZÁVĚR

Cílem diplomové práce bylo využít poznatky o testování polovodičových pamětí nabyté během tvorby semestrální práce k implementaci BIST kontroléru pro test pamětí.

Navržená struktura BIST kontroléru byla revidována a upravena pro možnost testování více polovodičových pamětí pomocí jediného BIST kontroléru. Bylo rozhodnuto, že mikrokód bude implementován pomocí mikrosequenceru.

Pro adresní čítače bylo rozhodnuto využít CFSR čítače z důvodu jejich menších velikostí. Při srovnávání vyšlo najevo, že velikost plochy zabrané kombinační logikou je u nich v implementovaném kontroléru přibližně poloviční.

Pro zjednodušení využití byla naprogramována aplikace pro generování BIST kontroléru na základě popisu algoritmu a modelu paměti. Generování ovšem nedokáže zohlednit složitější fyzické rozmístění pamětí, než pouhé rozdělení na řádky a sloupce. Tyto informace nebyly součástí dostupných modelů pamětí. Tento problém je řešitelný s pomocí manuální úpravy interface paměti.

Při srovnání celé implementace BIST kontroléru s řešením od firmy Mentor Graphics Corporation lze pozorovat, že zde navrhovaně řešení je o 40% menší[1].

[1]Paměť 512x22b, algorimtus SmarchCHKBci, technologie AMIS 350μm. Plocha memoryBIST 2310, zde prezentované řešení 1231.

LITERATURA

[1] Dhingra Sachin. *Comparison of LFSR and CA for BIST*. Auburn University, USA Dostupné z URL: <http://www.eng.auburn.edu/~agrawvd/COURSE/E7250_05/REPORTS_TERM/Dhingra_LFSR.pdf>

[2] J. Otterstedt, D. Niggemeyer and T. W. Williams, *Detection of CMOS address decoder open faults with March and pseudo random memory tests* Test Conference, 1998. Proceedings., International, Washington, DC, 1998, pp. 53-62.

[3] Niel H. E. Weste, Davbid Money Harris. *CMOS VLSI Design, A circuits and system prespective Forth edition* ISBN-10 0-321-54774-8

[4] A. J. Van De Goor, *Using march tests to test SRAMs*, in IEEE Design & Test of Computers, vol. 10, no. 1, pp. 8-14, March 1993. Dostupné z URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=199799&isnumber=5192>

[5] Intel *1024 bit fully decoded static mos random access memory*. Dostupné z URL: <https://drive.google.com/file/d/0B9rh9tVI0J5mMmZlYWRlMDQtNDYzYS00OWJkLTg4YzYtZDYzMzc5Y2ZlYmVk/view>

[6] D. Niggemeyer, M. Redeker and J. Otterstedt, *Integration of non-classical faults in standard March tests*, Proceedings. International Workshop on Memory Technology, Design and Testing (Cat. No.98TB100236), San Jose, CA, 1998, pp. 91-96. Dostupné z URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=705953&isnumber=15293>

[7] Arvind Raghuraman, *Walking, marching and galloping patterns for memory tests*, Dostupné z URL: <http://www.eng.auburn.edu/~agrawvd/COURSE/E7250_05/REPORTS_TERM/Raghuraman_Mem.doc>

[8] S. Hamdioui, A. J. van de Goor and M. Rodgers, *March SS: A Test for All Static Simple RAM Faults*, Proceedings of the 2002 IEEE International Workshop on Memory Technology, Design and Testing (MTDT2002), 2002, pp. 95-100. Dostupné z URL: <http://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=1029769>

[9] B. Nadeau-Dostie, A. Silburt and V. K. Agarwal, *A serial interfacing technique for built-in and external testing of embedded memories*, 1989 Proceedings of the IEEE Custom Integrated Circuits Conference, San Diego, CA, USA, 1989,

pp. 22.2/1-22.2/5. Dostupné z URL: <`http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=5726275&isnumber=2057`>

[10] Mentor Graphics Corporation *Tessent MemoryBIST User's and Reference Manual v2016.4, December 2016*

SEZNAM ZKRATEK

BIST	Built-In Self Test - v této práci myšlen test implementovaný v samotném integrovaném obvodu.
SRAM	Statická paměť typu RAM. Paměťové buňky jsou klopné obvody.
DRAM	Dynamická paměť typu RAM. Paměťové buňky jsou kondenzátory.
CMOS	Complementary MOS - je výrobní technologie využívající PMOS a NMOS tranzistory na jednom čipu.
ADOF	Address Decoder Open Fault - defekt adresního dekodéru. Chyba se projeví při hraně na defektních bitech.
SaF	Stuck at Fault - chyba, kdy paměťová buňka vrací stále jednu hodnotu.
TF	Transition Fault - chyba, kdy paměťová buňka nezvládne zápis bitu.
DR	Destructive Read - chyba, kdy čtením dojde k poškození dat v paměti.
dDR	Deceptive Destructive Read - chyba, kdy čtením dojde k poškození dat. Při čtení paměť vrátí správnou hodnotu.
DRF	Data Retention Fault - chyba, kdy dojde ke změně dat po určitém čase.
WDF	Write Disturb Fault - chyba, kdy zápis stejné hodnoty do pamětí vyústí v chybnou hodnotu.
CF	Coupling Fault - chyba, kdy operace nad jednou buňkou poškodí data v jiné buňce.
NPSF	Neighbourhood Pattern Sensitive Faults - chyby, projevující se při „vhodných“ datech v okolních buňkách.
SA0/1	Stuck at 0/1 - chyba SaF na hodnotě 0/1.
AF	Address decoder Fault - chyba adresního dekodéru (libovolná).
LF	Linked Faults - množina CF, která má stejnou cílovou buňku.
BFM	Bus Function Model - model budící komunikační sběrnici. Zajišťuje komunikaci mezi testcase a testovaným integrovaným obvodem.
LUT	Look-up Table - Jde o ROM paměť implementovanou pomocí logiýhch hradel

SPI Serial Periherial Interface - sériová sběrnice, běžně používaná pro komunikaci mezi integrovanými obvody.