VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

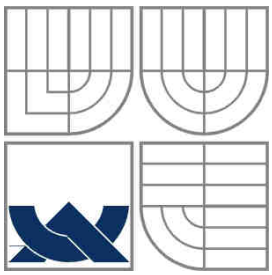
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SAMOČINNÝ TEST ALU ZA PROVOZU

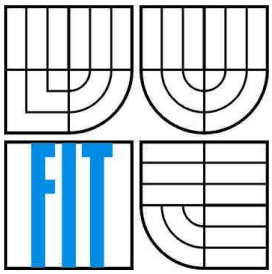
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE Bc. JAROSLAV BEDNÁŘ
AUTHOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

SAMOČINNÝ TEST ALU ZA PROVOZU

ALU BUILT-IN SELF TEST

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. Jaroslav Bednář

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. Vladimír Drábek, CSc.

BRNO 2010

Abstrakt

Práce se zabývá různými poruchami, které mohou nastat při výrobě a provozu ALU. Nastiňuje vícero dělení poruch a jejich modely. Jsou rozebrány přístupy k zajištění bezpečnosti systému zejména po hardwarové stránce. Následuje shrnutí softwarových metod pro testování ALU. Poslední kapitola rozebírá návrh knihovny pro testy mikrokontroléru MSP430.

Klíčová slova

ALU, aritmeticko-logická jednotka, porucha, chyba, test, modely poruch, mikrokontrolér.

Abstract

This work deals with faults, errors and failures, which can occur during manufacturing and long term operation. Work describes the various failures and fault models. There are some approaches to get fault tolerant systems, mainly in hardware. The thesis continues with a summary of methods for ALU software testing. The last chapter is about tests generation for microcontroller MSP430.

Keywords

ALU, arithmetic logic unit, fault, error, defect, test, fault models, microcontroller.

Citace

Bednář Jaroslav: Samočinný test ALU. Brno, 2010, diplomová práce, FIT VUT v Brně.

Samočinný test ALU za provozu

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením doc. Ing. Vladimíra Drábka, CSc.
Další informace mi poskytli Ing. Martin Kravka a Ing. Radomír Svoboda.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

Jaroslav Bednář
31.5.2010

Poděkování

Velmi děkuji za odbornou pomoc a velice užitečné připomínky doc. Ing. Vladimíru Drábkovi, CSc. a Ing. Martinu Kravkovi.

Obsah

1 Úvod

Aritmeticko-logická jednotka je základní součástí procesorů. Zajišťuje provádění, jak z názvu vyplývá, aritmetických a logických operací. Byla navržena matematikem Johnem von Neumannem jíž v roce 1945 ve zprávě o novém počítači EDVAC.

V dnešní době se s různými systémy obsahující procesor, a tedy i ALU, potkáváme velmi často. Procesor obsahují jak počítače pro všeobecné použití, tak i různé vestavěné systémy. Vestavěných systémů pro rozmanité úlohy je velmi mnoho a setkáváme se s nimi mnohem častěji než s procesory pro všeobecné použití. V normálním každodenním životě ani nevnímáme, že pracujeme se zařízením, které obsahuje mikrokontrolér a potažmo tedy procesor. Mnoho zařízení obsahující vestavěné systémy pracuje velmi dlouho bez přestávky v různě náročných podmínkách. U těchto zařízení je důležité, aby fungovala bez chyb, případně aby chyby byly co s největší pravděpodobností detekovány a mohla být učiněna protiopatření. Z teoretického hlediska, pokud by byly dokonalé součástky a dokonalé návrhy zařízení, nebylo by zapotřebí žádného testování a protiopatření. Realita je od teorie natolik vzdálená, že ani bodu dokonalého návrhu zařízení v dohledné době nedosáhneme. Je tedy nutné, a pravděpodobně tomu tak ještě dlouho bude, provádět testování. Situaci napomáhá navrhování pomocí modulů, kdy jednotlivé moduly jsou odladěny zvlášť a teprve potom se přidávají do návrhu systému. Složitější systém je skládán z jednotlivých modulů, doplněný o potřebné obvody. Jednodušší obvody modulů snižují náročnost návrhu i pravděpodobnost chyby v návrhu. Z pohledu součástek a stále větší integrace obvodů jsou na vstupní materiály kladeny daleko větší nároky. Nečistoty obsažené v materiálech při nižší integraci nemusí způsobit problémy, ale jejich vliv na chyby stoupá.

Nejdříve je ve druhé kapitole rozebrána úloha diagnostiky a základní obvyklé názvosloví, které se používá. Dále je v kapitole naznačeno možné dělení poruch, modely poruch a dělení modelů podle úrovně abstrakce. V závěru kapitoly je malé shrnutí o spolehlivosti, bezpečnosti a různých způsobech, jak bezpečnosti docílit.

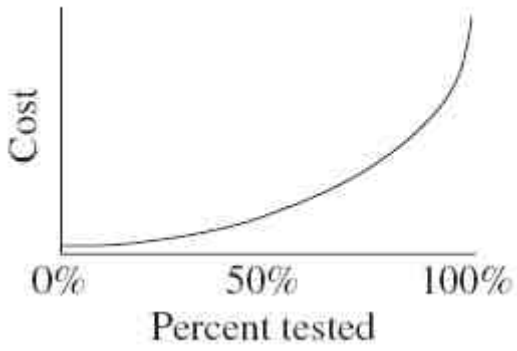
V následné kapitole je souhrn softwarového testování ALU, o kterém jsem se dozvěděl.

V poslední kapitole je rozveden návrh knihovny pro zvolený mikrokontrolér MSP430.

2 Diagnostika

Cílem diagnostiky je nalezení efektivních postupů, které dokáží ověřit správnou funkci zařízení. V případě nalezení jakékoliv nekorektní funkce zařízení je důležité, aby byly vráceny co nejpodrobnější informace o chybě a bylo možno ji opravit, nebo přijmout příslušná protiopatření.

Pro diagnostickou testovací sekvenci je zažit termín test. Jinými slovy, jak je uvedeno v [7], je test množina dvojic vstupních a jim přiřazených výstupních vektorů, které jsou určeny pro nalezení chyby. Jedna taková dvojice vektorů se nazývá krok testu, počet kroků udává délku testu. Důležitým parametrem testu je diagnostické pokrytí, udává počet poruch detekovaných (tj. pokrytých) testem. Pokrytí se udává buď absolutně (výčtem detekovaných či nedetekovaných poruch) nebo relativně vzhledem k celkovému počtu možných poruch. Test, který pokrývá 100% chyb, se nazývá úplný. Každý takovýto test lze stále rozšiřovat, aniž by se měnilo pokrytí tohoto testu. Z toho vyplývá skutečnost, že úplných testů je nekonečné množství. Při vypouštění kroků z testu nastává riziko porušení úplnosti testu, a proto je v tomto případě nutná obezřetnost a opatrnost. Test ze kterého nelze vypustit žádný krok, aniž by došlo k porušení úplnosti testu, se nazývá neredundantní. Pro jeden obvod může existovat více neredundantních testů. Nejkratší úplný test se nazývá minimální. Minimálních testů se může pro jeden obvod vyskytovat také více. Další možností testování je triviální test. Jedná se o vyzkoušení všech možných funkcí (vstupních vektorů), které má testovaný obvod realizovat. Je to velmi náročná varianta na délku testu, pro n-vstupový kombinační logický obvod má triviální test délku 2^n kroků a skládá se ze všech možných n-bitových vektorů. Triviální test je použitelný pouze pro obvody s málo vstupy. Pro reálné obvody je obvyklé pokrytí poruch mezi 70 - 75%. Je to dáno i finančními nároky na testování. V literatuře [13] je téma financí rozebráno více; zde uvádím některá důležitá fakta. U VLSI obvodů se v jisté chvíli z důvodu rentability nevyplatí zvyšovat pokrytí poruch, jak ukazuje obrázek 2.1. Čím je pokrytí poruch větší, tím je potřeba na jeho rozšiřování vyšších nákladů. Vyplývá to ze vzorce $DL = 1 - Y^{(1-T)}$; kde DL (defekt level) je úroveň defektů a je definována jako DL = počet prodaných špatných kusů / celkový počet prodaných kusů, Y (yield) znamená výtěžnost procesu výroby, tedy kolik procent součástek je po výrobě funkčních. T je procento otestování dané součástky. Za zmínku stojí poznámka, že v moderních SoC až 90% plochy čipu zabírají paměti a proto je vhodné tuto oblast testování mít dobře pokrytu. S aspektem ceny výroby a testů souvisí pravidlo desíti. Říká nám, že pokud objevení chyby v součástce nás bude stát nějakou hodnotu, tak po osazení vadné součástky do desky nás nalezení této vady bude stát 10krát tolik. Při osazení desky do systému pak 100krát více. Z daného pohledu je velmi vhodné objevení vadné součástky co nejdříve.



Obrázek 2.1 Náklady na testování

V literatuře [8] je uvedeno metodické hledisko pro testovací strategie. První strategií je funkční testování, provádí se na základě porovnávání vstupních a výstupní vektorů. Testuje se vlastní funkcionalita. Vnitřní struktura obvodů nemusí být známa. Druhou strategií je strukturní test. Pomocí kritické cesty se strategie snaží nalézt fyzické defekty.

2.1 Selhání, chyba, porucha

V anglicky psané literatuře se vyskytuje poměrně nesoulad mezi termíny poruch na jednotlivých úrovních abstrakce [1][2]. V této práci budu tyto tři termíny definovat takto:

- Porucha je jakási fyzikální změna vlastností některé součástky logického obvodu [3].
- Chyba je nesprávná logická hodnota zaviněná poruchou.
- Selhání je nesprávná hodnota signálu vycházející ze systému.

2.2 Dělení poruch

Poruchy mohou nastat v hardwaru, v datech a dále mohou nastat algoritmické chyby.

Poruchy v hardwaru lze dělit podle různých hledisek [3]:

Podle podmínek vzniku poruchy:

- Porucha z vnějších příčin – je způsobena nedodržením stanovených provozních podmínek a předpisů pro zatěžování, obsluhu a údržbu. Např. vliv prostředí, lidský faktor, náhodné události (povodeň, požár, zemětřesení, pád letadla).
- Porucha z vnitřních příčin – je způsobena vlastní nedokonalostí výrobku při zachování stanovených provozních podmínek a předpisů pro zatěžování, např. vývojová chyba, výrobní vada, vada materiálu.

Podle časového průběhu změn parametrů:

- Náhlá porucha – její vznik nemůže být předvídán předchozí prohlídkou nebo zkouškou, protože změna parametrů nastane skokem.
- Postupná porucha – parametry se mění postupně a poruchu lze tedy zjisti nebo předvídat na základě prohlídky nebo zkoušky.
- Nestálá porucha – porucha zařízení, která trvá po omezenou dobu, u zařízení se obnoví bezporuchový stav bez vnějšího zásahu. Příčinami mohou být vlhkost, kontaminace, elektrostatické náboje nebo náhodný šum.
- Stálá porucha – porucha která trvá od okamžiku vzniku po neomezenou dobu.

Podle stupně poruchy:

- Úplná porucha – odchylka parametrů je taková, že je znemožněna funkce výrobku.
- Částečná porucha – odchylka parametrů není tak veliká, aby se znemožnila funkce zařízení.

Autor v knize [2] uvádí ještě jedno hledisko dělení poruch, a to **podle možnosti jejich detekovatelnosti:**

- Nestálá porucha – poruchy, které zapříčiňují chyby v rychlosti. Chyby se projevují až ve vyšších frekvencích nebo produkují teplo. Je nutné mít dva nebo více testových vzorů pro odhalení tohoto druhu poruch. Test je nutné provádět v podobných rychlostech, ve kterých se chyby vyskytují, například jsou to vysokoodporové můstky a tunelový efekt.
- Stálé poruchy – poruchy se projevují ve všech frekvencích. Na odhalení takové chyby je potřeba jen jeden testový vzor a test může probíhat za malé rychlosti.

Poruchy v datech mohou být:

- Stálé – vytvořené při pořizování dat.
- Náhodné – vyvolané vnějšími vlivy (alfa částice – z vesmíru, z pouzder úložných zařízení, elektromagnetické rušení).

Algoritmické chyby – špatná formální specifikace, verifikace.

2.3 Úroveň abstrakce a modely poruch

Pro zjednodušení složitosti generace testů potřebujeme aktuální poruchy na čipu modelovat pomocí modelů poruch na vyšší úrovni abstrakce. Úrovně abstrakce obvyklé u popisu elektronických systémů jsou [1]: behaviorální, funkční, strukturní, klopných obvodů a obvodů samotných. Behaviorální a funkční modely můžeme zařadit do kategorie vysoko abstraktních modelů. Dovolují zvýšení rychlosti

pro vyhodnocení pokrytí chyb. Vysoko abstraktní modely mohou být explicitní nebo implicitní. Explicitní model identifikuje každou chybu zvlášť. Pro každou chybu je generován test. U implicitního modelu se předpokládají třídy chyb, které mají podobné vlastnosti a chyby v jedné kategorii je možné detekovat podobnými postupy. Výhodou implicitního modelu není nutnost explicitně vyjmenovávat jednotlivé poruchy uvnitř jedné kategorie. Kromě dříve zmíněných vysoko abstraktních chybových modelů patří do této kategorie ještě model pro tuto práci nejvhodnější, a tím je funkční chybový model pro mikroprocesory.

2.3.1 Behaviorální chybový model

Je to nejabstraktnější model. Popis chování digitálního systému se zadává pomocí programovacího jazyka, který dokáže popsat hardware. Těmi jsou například VHDL nebo Verilog. Zachycuje tok dat a tok programu. Chyby na této úrovni znamenají rozdílné chování oproti popisu (programu). Co vše za chyby do této kategorie patří, záleží na nižších vrstvách abstrakce. Patří sem modely typu:

- **„for (P) {T}"**, kde podmínka cyklu P může selhat takovým způsobem, že tělo cyklu T je vykonáno vždy nebo nikdy.
- **„switch (Id)"**, kde podmínka přepínače může selhat dvěma způsoby:
  - Přepínač může vybrat takový případ, který odpovídá hornímu nebo spodnímu extrému hodnoty proměnné v Id.
  - Chybně není vybrána žádná varianta.
- **„if (V) then {T_1} else {T_2}"**, kde celá podmínka může selhat následujícími případy:
  - Skupina výrazů {T_1} není nikdy vykonána a skupina příkazů {T_2} je vykonána vždy.
  - Skupina výrazů {T_1} je vykonána vždy a skupina příkazů {T_2} není vykonána nikdy.
  - Skupina výrazů {T_1} je vykonána tehdy, pokud výsledek logického výrazu V je nepravdivý a skupina výrazů {T_2} je vykonána tehdy, pokud výsledek logického výrazu V je pravdivý.
- **Příkaz přiřazení „X:=Y"**, při vykonávání příkazu může dojít k těmto chybám:
  - Hodnota X zůstane beze změny.
  - Do X se zapíší pouze logické 0 nebo 1.
  - V X bude uložená hodnota logických 0 nebo 1 podle nějaké pravděpodobnostní funkce.

2.3.2 Funkční chybové modely

Popis systému je na úrovni přenosu mezi registry a funkčními bloky (RTL), jako jsou sčítačky, ALU, paměti, multiplexory a sběrnice. Model chyb může být na úrovni instrukcí nebo RTL.

V knize [2] je uveden pro model RTL všeobecný vzorec:

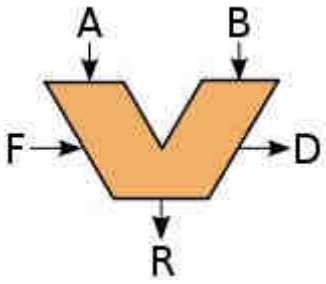
K: (T,C) $R_d \leftarrow f(R_{S1}, R_{S2},\ldots, R_{Sn}), \rightarrow N$

Kde K je označení RTL výrazu, T je časování a C je logická podmínka pro vykonání příkazu, R_d je cílový registr, R_{Si} je i-tý zdrojový registr, f je operace nad zdrojovými registry, $\leftarrow$ reprezentuje přenos dat a $\rightarrow$ N reprezentuje skok na výraz N.

Na základě výše uvedeného výrazu můžeme definovat 9 kategorií funkčních chyb. Kategorie jsou následující:

- Chyby v označení vyjádřené pomocí výrazu (K/K'), což znamená, že označení K je změněno na K' kvůli chybám na nižší úrovni (např. stálá 1 nebo 0, zkraty nebo při citlivosti na vzorky).
- Chyby časování (T/T').
- Chyby v logické podmínce (C/C').
- Chyby v dekódování funkce (R_i/R_i').
- Kontrolní chyby ($\rightarrow N/\rightarrow N$').
- Chyby při správě dat ((R_i)/(R_i)'). Výraz znamená změnu obsahu registru R z obsahu (R) na obsah (R)' vlivem chyb na nižších úrovních.
- Chyby při přesunu dat ($\leftarrow/\leftarrow$'). Výraz znamená chybu na cestě přesunu ze zdroje do cíle.
- Chyby při zpracování dat (vykonávání funkcí) ((f)/(f)'). Znamená chybu při zpracování operace. Operace f je provedena, ale výsledek operace je nesprávný.

Tato skupina poruch je pouhý souhrn, protože vnitřní chování každého obvodu lze popsat sekvencí RTL výrazů.



Obrázek 2.2 Nákres ALU jako funkční blok (RTL)

2.3.3 Funkční chybový model pro mikroprocesor

Vyskytující se chyby v procesorech se mohou rozdělit do těchto kategorií:

- Chyby adresování ovlivňující dekódování registru.

  Pokud je do multiplexoru s poruchou poslána adresa, mohou nastat následující případy:

  - Není zadán žádný zdroj.
  - Je zadán špatný zdroj.
  - Je zadán více než jeden zdroj a výstup multiplexeru je buď uzlový součin nebo součet zdrojů. Uzlová funkce záleží na použité technologii.

  Pokud je do demultiplexory s poruchou poslána adresa cíle, mohou nastat následující případy:

  - Žádný cíl není vybrán.
  - Místo správného cíle je vybrán jiný cíl či cíle.
  - Se správným cílem je vybrán jeden čí více cílů zároveň.

- Chyby adresování ovlivňující dekódování a pořadí instrukcí.

  Na instrukci lze nahlížet jako na sekvenci mikroinstrukcí, kde každá mikroinstrukce je množina mikropovelů, které jsou prováděny paralelně. Mikropovely reprezentují základní operace pro přenos dat a manipulaci s daty.

  Adresové chyby ovlivňující vykonávání instrukcí mohou způsobit jeden nebo více z následujících chybových efektů:

  - Jeden nebo více mikropovelů není mikroinstrukcí provedeno.
  - Mikropovely jsou chybně provedeny mikroinstrukcí.
  - Je provedena rozdílná sada mikroinstrukcí, ať již i se správnou sadou mikroinstrukcí či bez ní.

  Tento chybový model je všeobecný a připouští i možnost vykonání pouze části instrukce nebo vykonání zcela nové instrukce, která není ani v instrukčním setu.

- Chyby v ukládání a čtení dat.

  Zařízením na ukládání dat je většinou paměť. Proto zde může být použit chybový model pro paměti. V buňkách pamětí mohou nastat tyto situace:

  - Jedna nebo více buněk má stále zapsáno 0 nebo 1.
  - Jedna nebo více buněk nemůže vykonat přechod z 0 → 1 nebo 1 → 0.
  - V paměti se nalézají pár nebo páry buněk, které se ovlivňují při změně. Pokud je provedena změna v jedné paměťové buňce, ve druhé paměťové buňce v páru může dojít ke stejné či opačné změně.

- Chyby v přenosu dat.

  V kategorii se nalézají chyby, které vznikají při přenosu dat ve sběrnicích mezi registry a funkčními bloky mikroprocesoru. Pro chybovou sběrnici mohou nastat tyto případy:

  - Jedna nebo více linek může být trvale ve stavu 0 nebo 1.

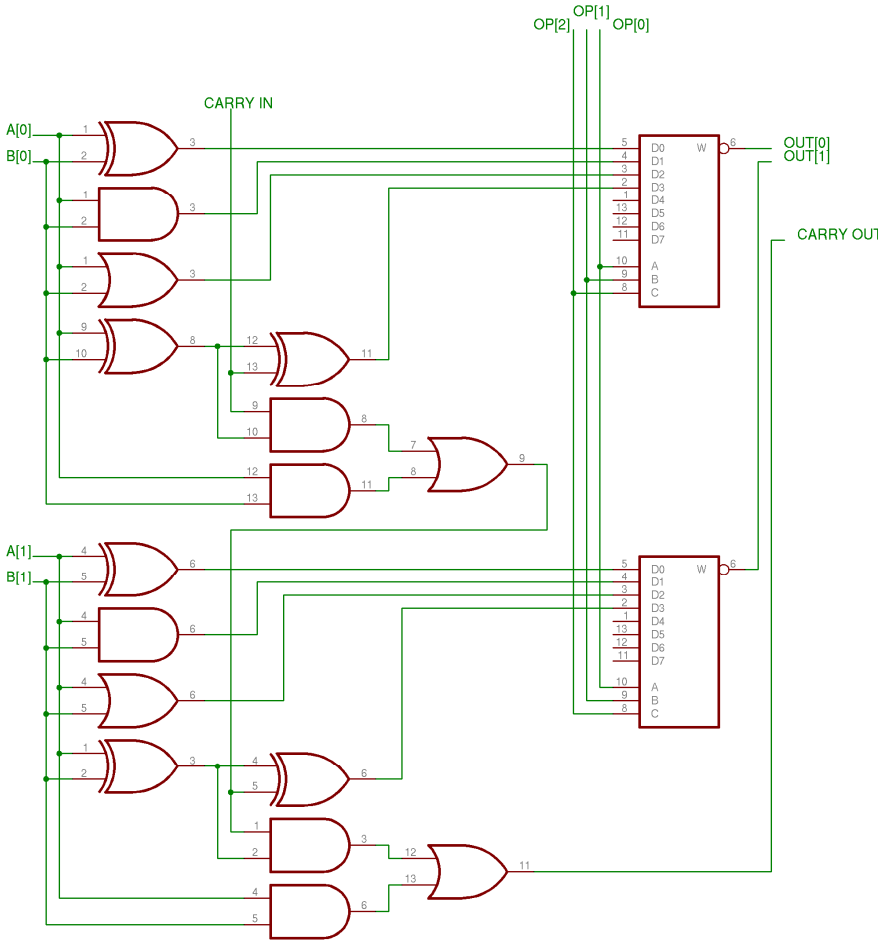
- Jedna nebo více linek může vytvořit uzlový součin či součet vlivem zkratů nebo rušení.

- Chyby při práci s daty.

  Pro tyto chyby není všeobecný model kvůli velmi širokému rozsahu použitých návrhů mikroprocesorů. Testovací množina se odvodí od funkčních jednotek.

2.3.4 Strukturální chybové modely

Úroveň logických členů. Popis systému se skládá z jednotlivých logických členů jako jsou AND, OR, XOR atd. a spojení mezi nimi. Příklad grafického zobrazení 2-bitové ALU na úrovni logických členů je na obrázku 2.2. Je to nejčastěji používaná úroveň abstrakce. Nejvíce známý model poruch trvalá jednička nebo trvalá nula. Toto je nejrozšířenější model používaný v průmyslu. Je populární pro jeho použitelnost pro různé polovodičové technologie. Detekcí jednotlivých chyb trvalá 1 nebo 0 je zjištěna většina fyzických poruch (často je takto detekováno až 80 – 85% chyb). Dnešním trendem je tento model poruch rozšiřovat i o jiné modely, které dokáží detekovat chyby tímto modelem nedetekovatelné.

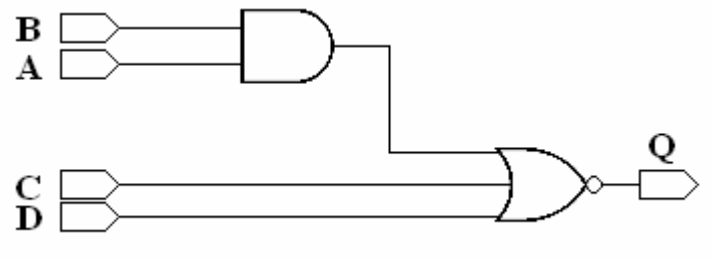


Obrázek 2.3 Jednoduchá 2-bitová ALU s funkcemi XOR, AND, OR a sčítáním seskládaná z logických členů

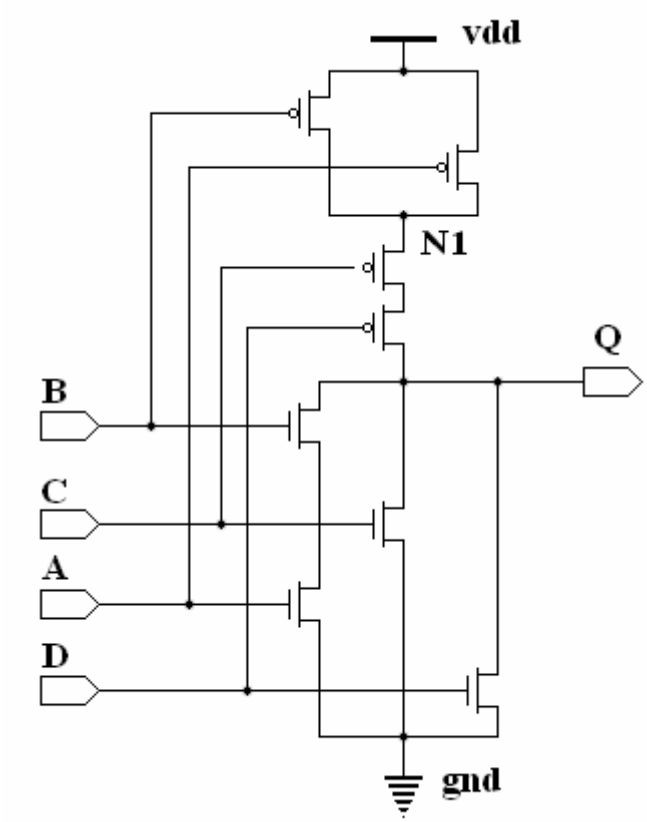
Z literatury [2] je známo, že klasické metody pro vytvoření testů nemohou zvládnout popsat aktuální chybné chování integrovaného obvodu, který je implementován pomocí technologie CMOS. Testy pouze vygenerují množinu testovacích vektorů pomocí chybových modelů na úrovni logických členů. Reálná struktura obvodu, charakteristiky poruchovosti obvodu a aktuální chování obvodu je tímto přístupem zcela ignorováno.

Pro pokrytí těchto nevýhod byly navrženy metody vytváření testů pro reálné struktury obvodu. V tomto přístupu je složitý obvod testován jako jeden celek, to je výpočetně velmi náročné a v praxi nepraktické. Proto je navrhnut i alternativní přístup, kdy jsou analyzovány všechny logické členy ze standardní knihovny buněk na úrovni tranzistorů. Testují se na všechny možné poruchy, což může být výpočetně náročné, ale pro každou buňku standardní knihovny je tento proces nutný vykonat pouze jednou. Tyto informace jsou pak dále využívány pro modely na vyšších úrovních abstrakce. V knize [2] je brán v potaz pouze jeden druh poruchy v CMOS obvodech a tím je zkrat mezi dvěma vodiči. Analýza může být rozšířena i na jiné druhy poruch, např. rozpojení.

Uvádím zde příklad výsledků jedné takové analýzy pro čtyřvstupový obvod s logickou funkcí NOT((A AND B) OR C OR D). Analýza a testování bylo provedeno na 0,8μm CMOS technologii. Zapojení logických členů je na obrázku 2.3. Na obrázku 2.4 je schéma stejného obvodu.



Obrázek 2.4 Logické členy pro logickou funkci NOT((A AND B) OR C OR D)

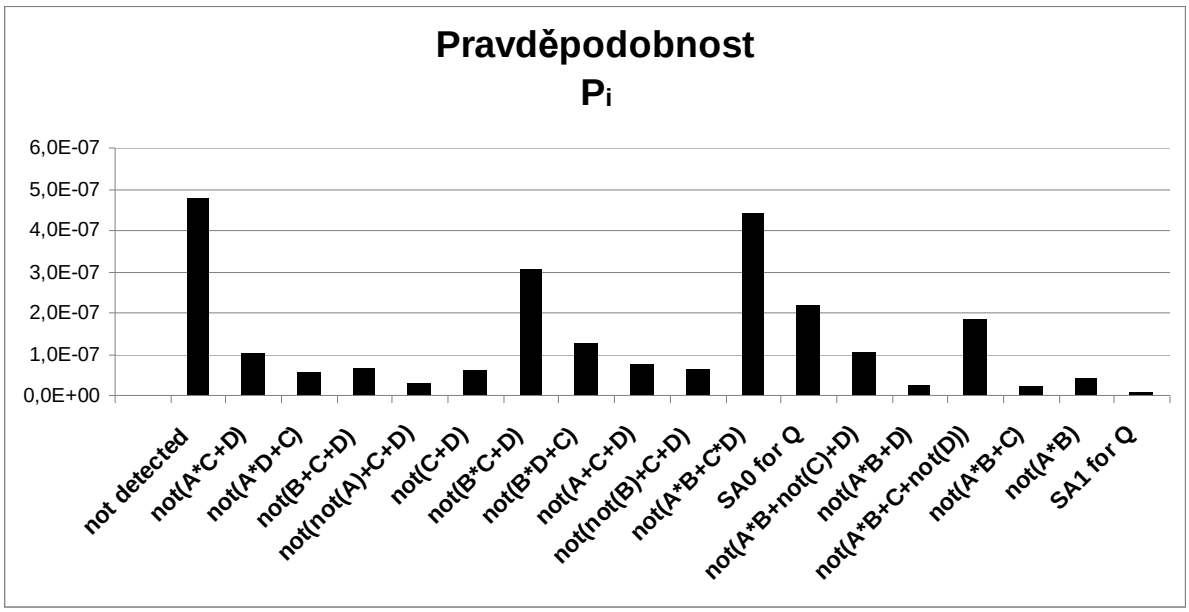


Obrázek 2.5 Schéma obvodu pro logickou funkci NOT((A AND B) OR C OR D)

Tabulka udává výsledky a pravděpodobnosti zkratů na různých částech obvodu. Při této technologii se zkraty chovají jako uzlový logický součin. V poli „porucha“ jsou uvedeny body, mezi kterými je zkrat. Jednotlivá pole vstupních vektorů znamenají, zda je porucha daným vstupním vektorem detekována (v daném poli se nachází 1), nebo není (pole je prázdné). Pod tabulkou na obrázku 2.5 se nachází graf pravděpodobností výskytů jednotlivých chybových funkcí v testovacím obvodě. Zajímavým faktem je, že poruchy stálá 1 (SA1) a stálá 0 (SA0) nejsou převažující.

i	Porucha d_i	Chybová funkce f^{di}	Pravděpodobnost P_i	Vstupní vector t_j <ABCD>															
				0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
1	A/B	**nedetekováno**	$3{,}2938*10^{-7}$	**Žádný chybový vektor**															
2	A/C	not(A*C+D)	$1{,}0199*10^{-7}$			1				1						1			
3	A/D	not(A*D+C)	$5{,}8123*10^{-8}$		1				1							1			
4	A/ Nl	not(B+C+D)	$5{,}0892*10^{-8}$					1											
5	A/Q	not(not(A)+C+D)	$3{,}1540*10^{-8}$	1				1								1			
6	A/gnd	not(C+D)	$3{,}4523*10^{-8}$													1			
7	A/vdd	not(B+C+D)	$1{,}8743*10^{-8}$					1											
8	B/C	not(B*C+D)	$3{,}0736*10^{-7}$			1								1		1			
9	B/D	not(B*D+C)	$1{,}2855*10^{-7}$		1								1			1			
10	B/Nl	not(A+C+D)	$5{,}5331*10^{-8}$									1							
11	B/Q	not(not(B)+C+D)	$6{,}5150*10^{-8}$	1								1				1			
12	B/gnd	not(C+D)	$2{,}7366*10^{-8}$													1			
13	B/vdd	not(A+C+D)	$2{,}1387*10^{-8}$									1							
14	C/D	not(A*B+C*D)	$4{,}4240*10^{-7}$		1	1			1	1			1	1					
15	C/Nl	SA0 for Q	$4{,}0733*10^{-8}$	1				1				1							
16	C/Q	not(A*B+not(C)+D)	$1{,}0515*10^{-7}$	1		1		1		1		1		1					
17	C/gnd	not(A*B+D)	$2{,}5810*10^{-8}$			1				1				1					
18	C/vdd	SA0 for Q	$1{,}4943*10^{-8}$	1				1				1							
19	D/Nl	SA0 for Q	$2{,}1978*10^{-8}$	1				1				1							
20	D/Q	not(A*B+C+not(D))	$1{,}8618*10^{-7}$	1	1			1	1			1	1						
21	D/gnd	not(A*B+C)	$2{,}1360*10^{-8}$		1				1				1						
22	D/vdd	SA0 for Q	$1{,}0609*10^{-8}$	1				1				1							
23	N/Q	not(A*B)	$4{,}2614*10^{-8}$		1	1	1		1	1	1		1	1	1				
24	N/gnd	SA0 for Q	$7{,}2557*10^{-9}$	1				1				1							
25	Nl/vdd	**Nedetekováno**	$1{,}4613*10^{-7}$	**Žádný chybový vektor**															
26	Q/gnd	SA0 for Q	$1{,}2459*10^{-7}$	1				1				1							
27	Q/vdd	SA1 for Q	$7{,}0250*10^{-9}$		1	1	1		1	1	1		1	1	1	1	1	1	1
28	gnd/vdd	**Nedetekováno**	$3{,}7640*10^{-9}$	**Žádný chybový vektor**															

Tabulka 2.1 Výsledky analýzy pro testovaný obvod



Obrázek 2.6 Pravděpodobnost výsledných chybných logických funkcí pro testovací obvod

2.3.5 Chybové modely na úrovni klopných obvodů

Tyto modely popisují obvod na úrovni tranzistorů. Nejvíce používaným modelem v této kategorii je trvale otevřen, trvale zapnut. Poruchu trvale otevřen nazýváme poruchu u tranzistoru, který je stále otevřen a nevede proud. Poruchu trvale zapnut nazýváme poruchu u tranzistoru, který je stále sepnut a stále vede proud. Tyto modely jsou přizpůsobeny technologii CMOS.

2.3.6 Chybové modely pro popis obvodů

Obvodové chybové modely popisují obvod na úrovni rozložení jednotlivých prvků. Pro tyto modely je nutná znalost rozložení na čipu. Modely využívají znalostí o šířce vodičů, o mezerách mezi vodiči a komponenty a celkových rozměrů zařízení. Zmíněné údaje modely používají pro zkoumání poruch na této úrovni abstrakce. Takovými testy lze detekovat výrobní vady. Zástupcem v této kategorii je model poruch přemostění (bridging fault model). Jedná se o zkraty mezi vodiči. Výsledná hodnota takové poruchy je závislá na aktuálním obvodu. Jsou dvě možnosti výsledku:

- Uzlový součin (wired-AND), zkratovaný obvod se chová jako hradlo pro logický součin.
- Uzlový součet (wired-OR), kde se zkratovaný obvod chová jako logický součet.

Dalším chybovým modelem v této kategorii je model přerušení (open fault model). Výsledek je také závislý na implementaci obvodu. Následky zkratů nám mimo zmíněných modelů mohou způsobit sekvenční chování obvodu (obvod začne pracovat jako paměť). Toto chování je těžko detekovatelné. V praxi je většina zkratů zachycena testy na stálou jedničku nebo nulu.

2.3.7 Model poruch zpoždění

Mimo úrovně abstrakce stojí model poruch zpoždění. Poruchy v tomto modelu mohou mít pouze dočasný charakter. Signály sice mají správnou hodnotu, ale mají menší či větší (tento případ je pravděpodobnější) zpoždění. Tato zpoždění se mohou šířit dále obvodem a na výstupu může být pomocí latch registru špatná hodnota. S větší integrací obvodů (VLSI) a zvyšováním výkonu obvodů model poruch zpoždění vzrůstá na důležitosti.

2.4 Vznik poruch

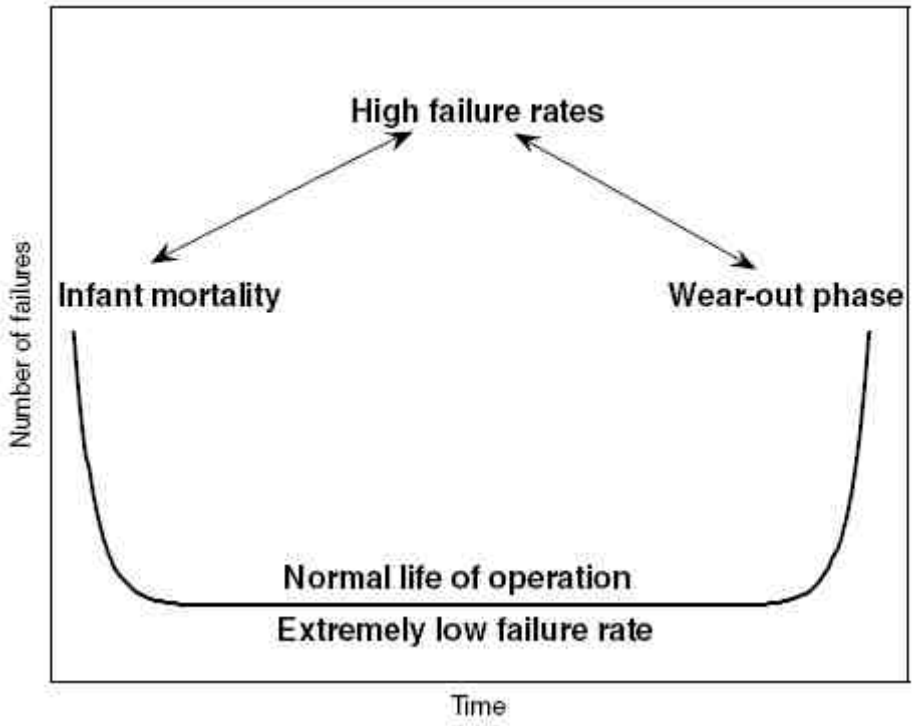
Poruchy lze dělit i podle místa vzniku:

- Poruchy polovodičových součástek (pouzdra integrovaných obvodů, tranzistory, diody).
- Poruchy pájených spojů (pájení součástek, pájení vodičů).
- Poruchy plošných spojů (mikroskopické trhliny, přeleptané spoje, neprokovené díry).
- Poruchy konektorů (nedokonalé nebo poškozené kontakty).
- Poruchy vnějších zařízení.

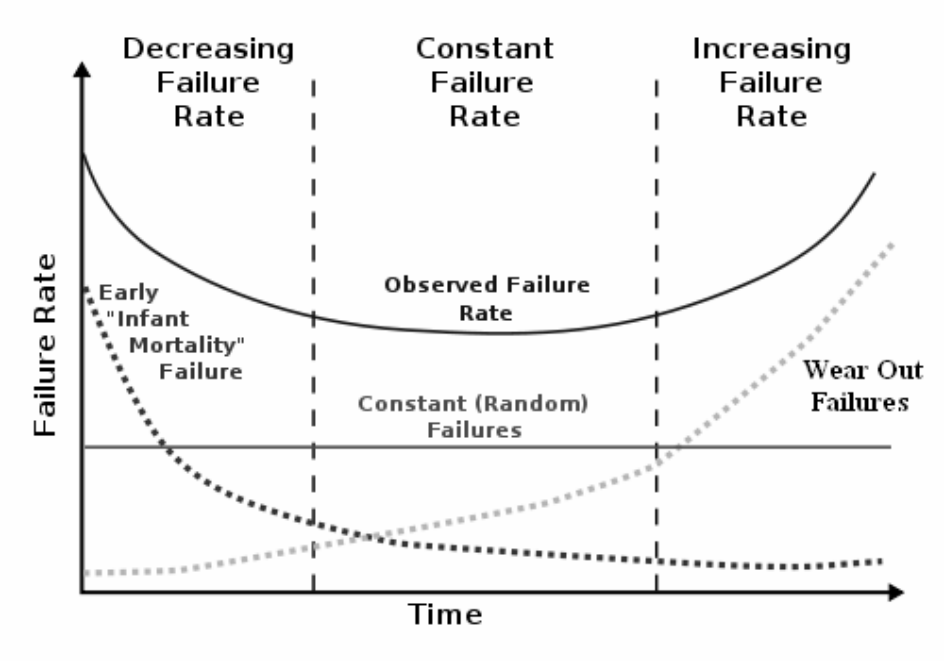
Podle [1] existují 3 hlavní kategorie vzniku poruch:

1. Elektrické namáhání – vzniká díky nedostatečnému návrhu a obvod je přetěžován.
2. Vlastní mechanizmy poruchy – toto jsou vlastní poruchy polovodičů. Patří mezi chyby a dislokace v krystalech, výrobní vady. Většinou jsou způsobeny při výrobě křemíkových desek (waferů).
3. Nevlastní mechanizmy poruchy – tyto poruchy vznikají při pouzdření polovodičů a propojování. Mezi poruchy patří například koroze a mikrotrhliny.

Dále je možno dělit podle hlediska času vzniku poruch. Tento pohled nejlépe znázorňuje vanovitá křivka intenzity poruch podle času znázorněná na obrázku 2.7 a 2.8.



Obrázek 2.7 Intenzita poruch (λ) během provozu součástky



Obrázek 2.8 Intenzita poruch (λ) a jejich skladba během provozu součástky

Oba obrázky mají totožný význam. Na obrázku 2.7 je pouze výsledná vanovitá křivka. Na obrázku 2.8 se nacházejí i dílčí křivky, ze kterých se výsledná křivka skládá. Náhodné poruchy mají konstantní hodnotu výskytu po celý životní cyklus součástek. Křivka s časnými poruchami má

největší hodnotu na začátku životního cyklu a postupem času klesá. Od jistého okamžiku nemá na celkovou poruchovost velký vliv. Třetí křivka znázorňuje poruchy opotřebením. Její vliv v čase stoupá, až dosáhne bodu, kdy je nejvlivnější.

Z výsledné křivky vychází tři fáze fungování elektronických součástek. V první fázi včasných poruch se projevují výrobní vady. Fázi včasných poruch řeší výrobce součástek pomocí tzv. zahoření. Součástky během tohoto procesu jsou po jistou dobu vystaveny poměrně vysoké teplotě, která přesahuje normální provozuschopnou teplotu. To urychluje stárnutí součástek a výskyt časných poruch. Po zahoření se provádí testování součástek a vadné jsou vyřazeny. Výzkumy v minulosti byla vyšší teplota určena jako nejrychlejší akcelerátor stárnutí oproti jiným fyzikálním jevům, např. vlhkosti. U větších zakázek si náročnost zahořování může zákazník definovat sám a tím může zvýšit kvalitu dodaných součástek. Ve druhé fázi normálního fungování dochází k velmi malé poruchovosti a intenzita poruch je téměř konstantní. Ve třetí fázi se projevuje opotřebení součástek a poruchovost stoupá. U dlouhodobého provozu zařízení je zapotřebí třetí fázi co nejlépe ošetřit, aby zařízení dokázalo i přes případné poruchy fungovat dále a nedošlo k selhání. Způsoby, které selhání jednotky oddálí, jsou uvedeny posléze.

2.5 Definice termínů souvisejících s poruchami

- Poruchovost (λ) – počet poruch za jednotku času. Toto číslo odpovídá křivce na obrázku 2.6.
- Počet poruch během času – FIT (Failure In Time). FIT je množství poruch během 10^9 provozních hodin, nebo počet selhajících zařízení po 10^9 hodin provozu. Je to nepřímá úměra mezi počtem testovaných zařízení a délkou trvání testu. Pro test tedy může být použito 1000 kusů zařízení po dobu 1 miliónu hodin nebo 1 milión zařízení po dobu 1000 provozních hodin nebo jiná kombinace.
- Střední doba mezi poruchami – MTBF (Mean Time Between Failures). MTBF je převrácená hodnota proměnné FIT a je to běžná veličina pro měření spolehlivosti systému. MTBF je časová hodnota, čím vyšší hodnota je, tím více je zařízení spolehlivé.
- Spolehlivost [R(t)]. R(t) je definována jako předpověď spolehlivě fungujících zařízení pro pevně určené časové období. Odvíjí se z FIT a je vyjádřena pomocí procent.

2.6 Bezpečnost a spolehlivost

V [3] nalezneme tyto definice pro výrazy spolehlivost, bezpečnost a integrita:

Spolehlivost znamená, že systém funguje tak, jak se od něho očekává, a to po celou dobu jeho činnosti. Spolehlivost je také míra toho, jak systém splňuje předpoklady v průběhu jeho činnosti, za

specifikovaných stavů prostředí. Hardwarová spolehlivost určuje schopnost systému odolávat selhání. Softwarová spolehlivost určuje schopnost systému vykonávat určité provozní požadavky.

Bezpečnost zahrnuje zbavení se nebezpečí, nebo více specificky, zbavení se nepřijatelných událostí, zákeřného a náhodného zneužití. Bezpečnost je také schopnost systému odolávat průniku zvenčí a zneužití zevnitř.

Integrita znamená, že určité žádoucí podmínky jsou udržovány během celého provozu. Například integrita systému souvisí s rozsahem hardwarových a softwarových změn, které byly na systému nevhodně provedeny. Integrita dat popisuje změny dat, neboli jaká data jsou a jaká by měla být. Integrita osob popisuje individuální chování v požadovaném návyku.

Důsledky poruch ve vestavěných systémech mohou být velmi různorodé a ve velké míře záleží na aplikaci, kde daný vestavěný systém je provozován. Ve spoustě případů se termíny integrita, spolehlivost a bezpečnost překrývají. Jak mnoho se překrývají, záleží na konkrétní aplikaci.

Vestavěné systémy řídí velmi různorodá zařízení. Z hlediska bezpečnosti lze rozdělit aplikace, ve kterých jsou nasazeny vestavěné systémy, na 2 základní skupiny. Vestavěné systémy, které svojí nebezpečností nebo nespolehlivostí mohou ohrozit lidské zdraví, či nikoliv. V aplikacích, kde hrozí nebezpečí zranění či dokonce smrt, jsou bezpečné vestavěné systémy velmi důležité. Mezi základní oblasti pro nasazení bezpečných systémů patří např. medicína, letectví, automobilová technika, kontrola jaderných zařízení, vesmírná a vojenská technika. Existují ale i oblasti, kde přímo o životy nejde, ale porucha může napáchat velmi velké materiální škody. Patří sem hlavně technika pracující s financemi v jakékoliv formě.

2.7 Docílení bezpečnosti

Abychom byli schopni docílit jakési úrovně odolnosti proti poruchám, je potřeba výsledky výpočtů průběžně sledovat a kontrolovat. Kontrola i sledování potřebují prostředky navíc, tedy jakousi formu redundance. Ta může být [10]:

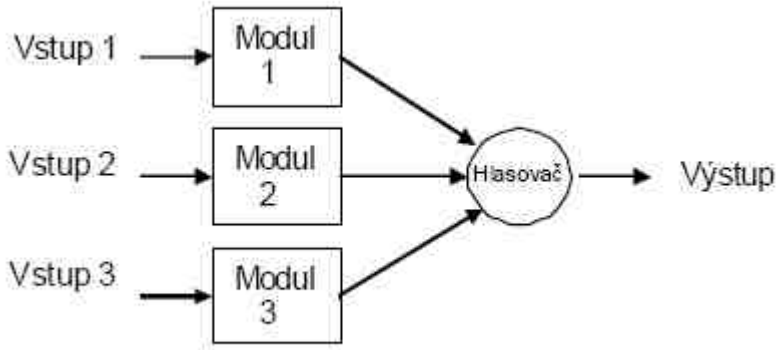
- Obvodová – detekce nebo tolerance poruch.
- Informační – detekční a informační kódy.
- Programová – znásobení programu, v nutných případech i dat. Umožňuje ošetření chybových stavů (nepovolené stavy, uváznutí) a eliminaci návrhových a algoritmických chyb. Všeobecně u softwaru lze redundanci rozdělit na statickou tzn. stále aktivní a na redundanci dynamickou tj. programové prvky se aktivují při vzniku chyby. Příkladem jsou rekonfigurační rutiny a alternativní algoritmy.

- Časová – opakování operace na stejném hardwaru. Nevýhodou je, že pokud se porucha projevuje během průběhu času stejně, není detekována.

2.7.1 Obvodová redundance

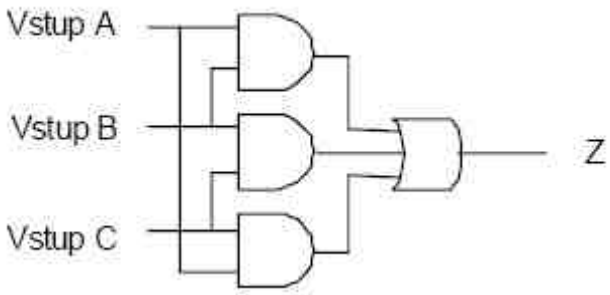
Obvodovou redundanci můžeme rozdělit na 3 kategorie:

1. **Redundance statická** – jedná se o vymaskování poruchy, porucha se neopravuje. Jedná se o N modulární systém – zkratka NMR, kde N modulů by mělo podat stejný výsledek. Hlasovací člen pak na základě dominantního výsledku rozhodne, který výsledek je správný. Ze zmíněné vlastnosti vyplývá, že N musí být liché číslo. Zajisté také není překvapení, že N modulární systém pracuje správně jen tehdy, pokud pracuje správně více jak polovina modulů. Hlavním zástupcem v této kategorii je 3 modulový systém. Více modulové systémy se téměř nevyskytují z důvodů vysoké ceny, zabraného místa a příkonu. Tří modulový systém se označuje anglickou zkratkou TMR (Triple modular redundancy) viz. obrázek 2.7.



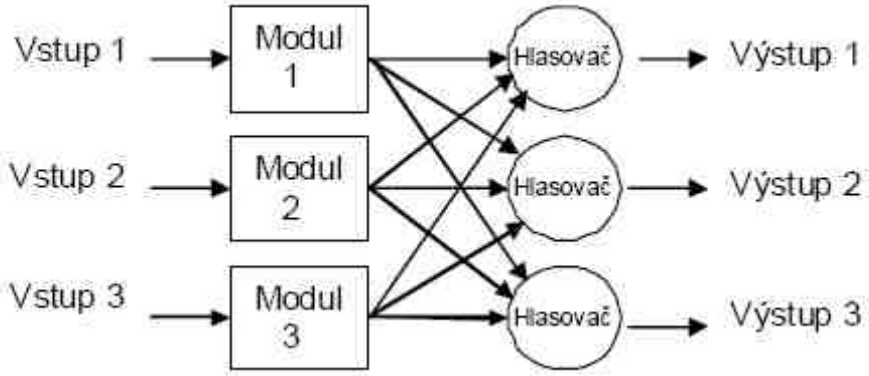
Obrázek 2.9 TMR systém

Jedná se o systém složený ze tří modulů a jednoho hlasovacího členu. Jednotlivé moduly by měly reagovat na stejné vstupy stejnými výstupy. Hlasovací člen je u takto postaveného systému slabé místo, výhodou je jeho hardwarová jednoduchost.



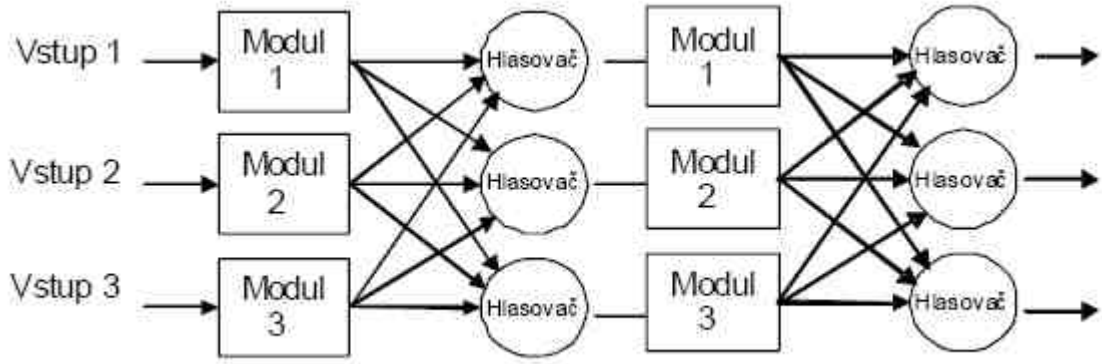
Obrázek 2.10 Jednoduchý 1 bitový hlasovací člen

Aby se vyřešila náchylnost obvodu na poruchu hlasovacího členu, je možné hlasovací člen znásobit. Vznikne tak několik na sobě nezávislých výstupů.



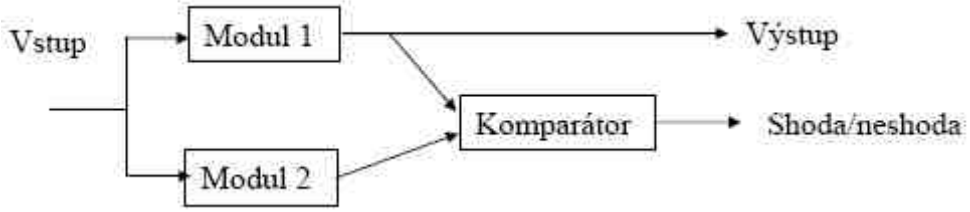
Obrázek 2.11 TMR s vícenásobným hlasovacím členem

Dále je také možno skládat jednotlivé TMR systémy do kaskády a vznikne tak vícestupňový systém TMR.



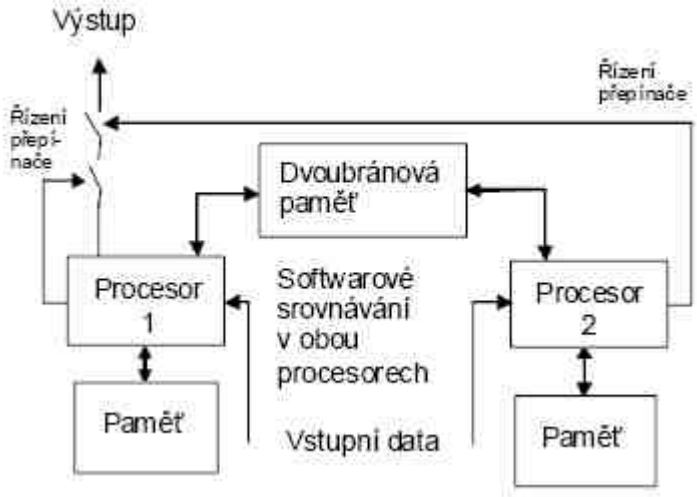
Obrázek 2.12 Vícestupňový TMR

2. **Redundance dynamická** – systémy jsou postavené na principu detekce chyby a následné lokalizaci zdroje chyby. Po lokalizaci chyby je možno provést překonfigurování systému. Základním principem dynamické redundance je hardwarové zdvojení se zálohováním.



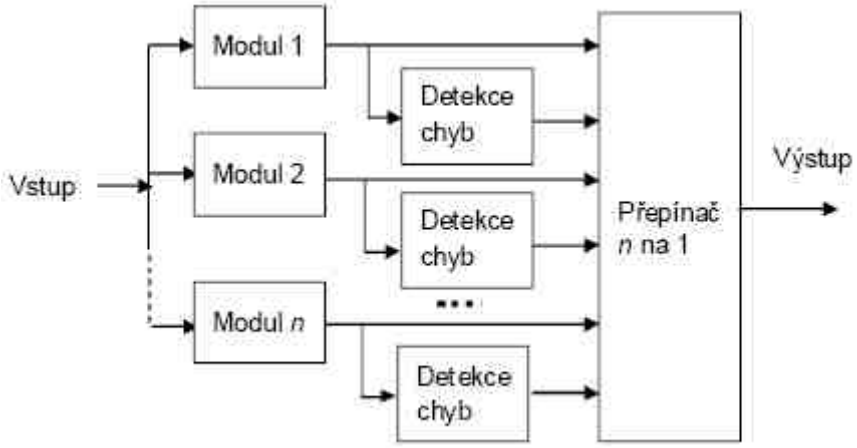
Obrázek 2.13 Zdvojený systém

Objevuje se tu podobný problém jako u TMR, komparátor nesmí selhat. Jeho selhání se dá řešit společnou pamětí modulů. Příklad takového systému je na obrázku 2.11. Při selhání společné paměti oba procesory detekují. Procesor detekující neshodu odpojí přepínač. Srovnání může probíhat pomocí softwaru nebo hardwaru. Dalším problémem je problém nedetekování chyby na společném vstupu.



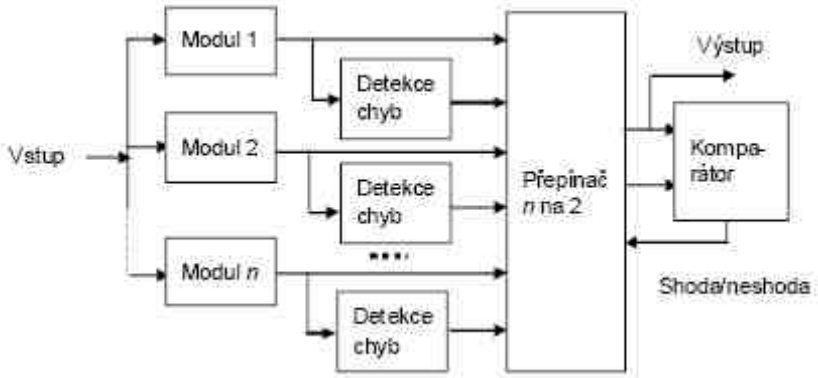
Obrázek 2.14 Procesory se společnou pamětí

Další možností je systém se zálohováním rezervami, kdy jeden modul je funkční a ostatní moduly jsou záloha. V systému se nalézají prostředky pro detekci a lokalizaci poruchy. Systém je pak pomocí přepínače n na 1 rekonfigurován a modul s poruchou je vyřazen. Při správné funkci více modulů je aktuální modul vybrán na základě určených pravidel. Slabým článkem tohoto systému je potřebný čas k rekonfiguraci, po tuto dobu je celý systém vyřazen z činnosti. Ukázka takového systému je na obrázku 2.13.



Obrázek 2.15 Systém zálohovaný rezervami

Jen velmi málo odlišný systém je systém se zálohovanou dvojicí. Zde pracují 2 moduly zároveň a výsledky jsou následně porovnávány na shodu. U tohoto systému lze obejít lokalizaci chyby tím, že při poruše jednoho modulu nahradíme celou dvojicí pracujících modulů. Samozřejmě pokud chceme nahradit pouze porouchaný modul, je nutné poruchu lokalizovat. Ukázka takového systému je na obrázku 2.14.



Obrázek 2.16 Systém se zálohovanou dvojicí

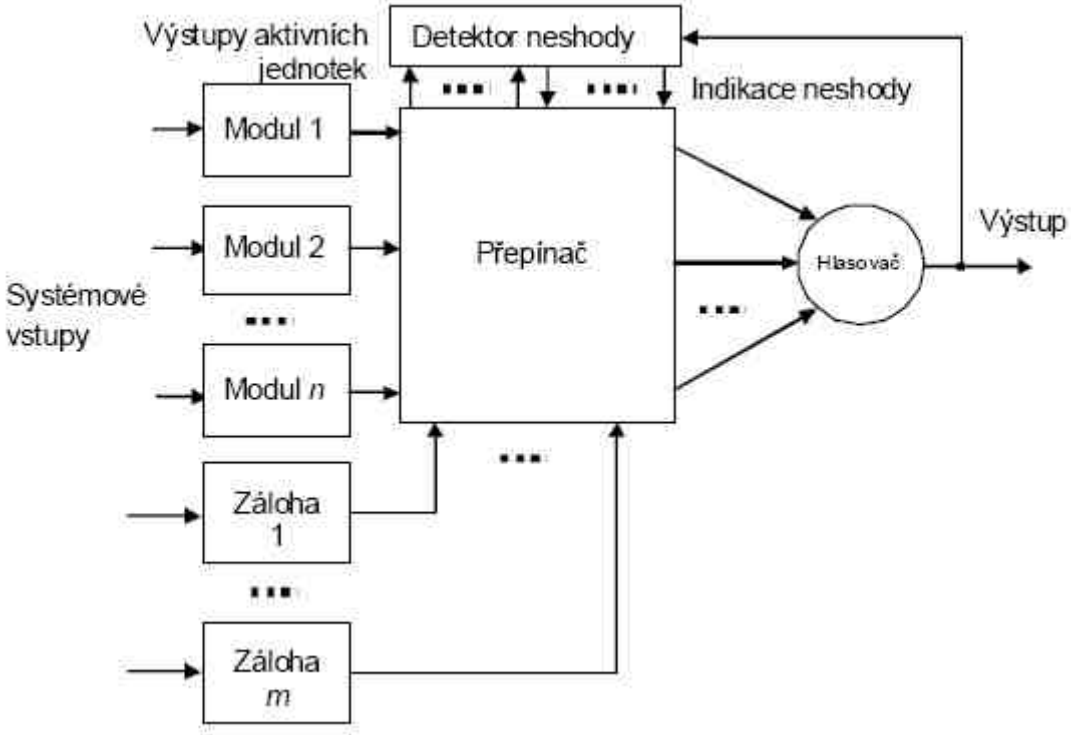
U záloh rozeznáváme režim fungování zálohy. Pokud je rezerva stále zapojena a funguje synchronně s hlavním modulem, tak tuto zálohu nazýváme zálohou horkou. Nevýhodami jsou stejné opotřebení záloh jako hlavního modulu, větší spotřeba proudu a s tím spojené větší vyzařování tepla do okolí. Zálohy, které jsou během bezchybného fungování hlavního modulu vypnuté, nazýváme studená záloha. Nevýhodou je nutná větší doba k rekonfiguraci celého systému. To je v některých případech nepřijatelné, stejně jako je nevhodné použití horkých záloh. Kompromisem mezi oběma přístupy je

použití odlehčených záloh. Pro tyto zálohy je sníženo napájecí napětí, některé jednotky nebo hodinové signály modulu mohou být vypnuty.

Systémy se zálohami mají výhodu v možnosti pokrytí tolika poruch, až kolik mají zálohových modulů.

Dalším přístupem u dynamické redundance je přidání obvodů fungujících jako watchdog timer – časový hlídač. Pokud časový hlídač dosáhne své maximální hodnoty, je hlášena porucha. Nejčastěji je to čítač, který musí být pravidelně nulován, aby nepřetekl. Při přetečení je hlášena porucha systému, ve většině případů je vyvolán reset daného modulu. Každý modul systému může mít svůj vlastní časový hlídač. Této časovač dokáže pokrýt různé softwarové i hardwarové chyby.

3. **Hybridní obvodová redundance** – kombinace aktivní a pasivní redundancí. Potírá nevýhody jednotlivých systémů (vysoká míra redundance u statické redundance, u dynamické je to hlavně drahá rekonfigurace systému). Jedná se o systém NMR + mR, tedy o N modulární systém s m zálohami. Po prvních m poruchách lze provést rekonfiguraci, následně se provádí maskování. Příklad systému je na obrázku 2.15, detektor neshody slouží na lokalizaci poruchy.



Obrázek 2.17 Systém NMR + mR

2.7.2 Informační redundance

Je velké množství různých detekčních a korekčních kódů. Lze jmenovat Hammingovy kódy, cyklické kódy, konvoluční kódy, BCH kódy, Reed-Solomonovy kódy a aritmetické kódy. Pro jednotku ALU, která provádí s daty operace, připadají v úvahu pouze kódy aritmetické. Hlavními druhy aritmetických kódů jsou kódy násobkové a zbytkové. Většinou se pro aritmetické kódy používají upravené sčítačky, což není v tomto projektu možné. Použití těchto kódů je v této práci nepraktické a použitelné pouze ve velice omezené formě, kvůli nutnosti otestovat celou instrukční sadu a neposkytuje žádné výhody.

- U násobkových kódů je obrazem čísla N číslo AN. A je vhodné celé číslo, tj. $A \neq 2^n$. A se nazývá báze. Pro operaci sčítání s čísly N_1 a N_2 v násobkovém kódu platí:

  $$A(N_1 + N_2) = AN_1 + AN_2$$

- U zbytkových tříd se číslo N vyjádří jako N = IM + r, M se nazývá báze a r je zbytek. Číslo N se pak vyjádří jako N = r mod m (číslo 11 se vyjádří: 11= 2 mod 3). S těmito kódy lze sčítat.

2.8 Důsledky poruch

Důsledky poruch lze rozdělit do několika kategorií. Rozsah poruchy může být pouze lokální a chyba neovlivňuje okolní přístroje. Nebo daná škoda může ovlivnit okolí, může se stát, že chyba lavinovitě způsobí další závažné poruchy v okolních systémech.

Pokud se podíváme na ALU na úrovni hradel, může porucha způsobit například špatné vymaskování registru a ten může ovlivnit čtení ze špatné paměti, vývod z pouzdra je v rozdílném stavu než by měl být, skok programu se provede špatně či vůbec. Poruchy se nabalují, pokud jsou na vývodech z procesoru se špatnou hodnotou nějaké členy, které přímo ovlivňují okolí a nelze ověřit, zda je hodnota korektní. Takovým případem mohou být různé aktivní členy jako relé, nebo jim podobné součástky založené na polovodičích, zapojené na vývod z procesoru. Pokud je hodnota nekorektní, relé může pracovat jinak, než se předpokládá.

3 Testování ALU

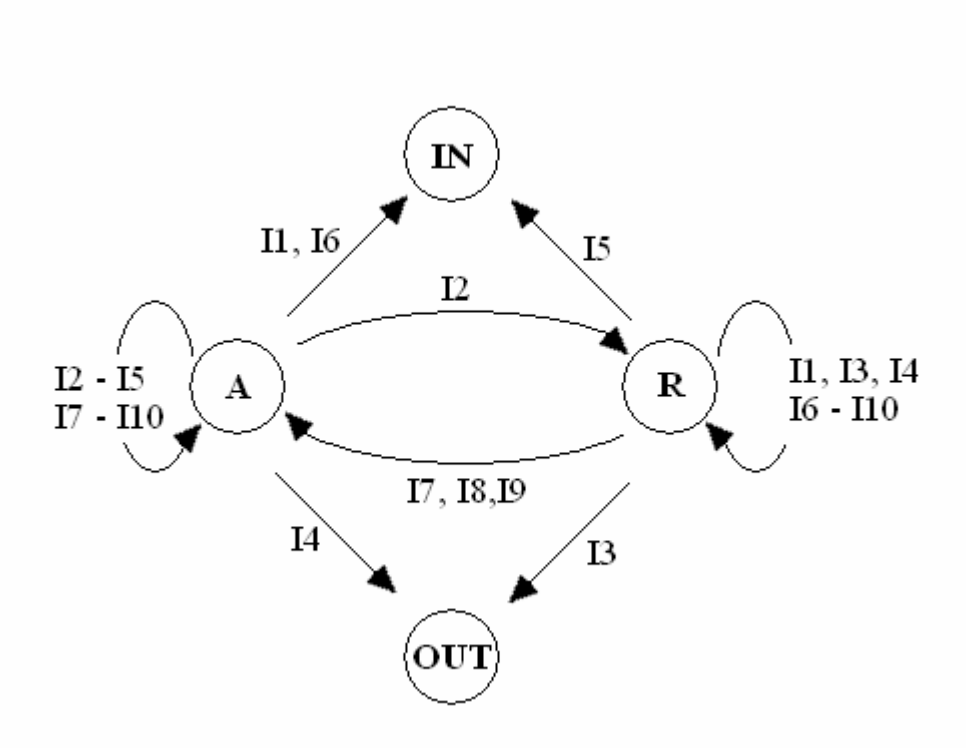
Testování mikroprocesoru je obtížný problém. Mikroprocesory jsou velmi komplexní a také nám velice často nejsou známy údaje o implementačních detailech, které byly použity. Proto je sestavování testů pro mikroprocesory na úrovni klopných obvodů a logických členů nebo na úrovní stavových diagramů nereálné. Problémem je velká různorodost mikroprocesorů, jejich uspořádání, rozsah instrukční sady, adresové módy, způsob ukládání dat, data zpracovávající jednotky a apod. Z těchto důvodů jsou testy převážně prováděny pomocí instrukční sady mikroprocesoru (ta je uvedena v každé příručce daného mikroprocesoru) na úrovni funkčního chybového modelu.

3.1 Testy založené na S-grafech

V literatuře [9] byl navržen všeobecný model procesoru pro vytvoření testů. Jmenuje se S-graf (S-graph). Mikroprocesor může být reprezentován jako vzájemný vztah registrů a rozhraním pro vnější svět. Vztahy určuje konkrétní instrukční sada daného mikroprocesoru. Každý registr mikroprocesoru je prezentován uzlem v grafu. Do grafu jsou přidány dva uzly navíc, které reprezentují okolní svět. Uzel reprezentující vstup dat se označuje jako IN, výstup označuje uzel OUT. Mezi registry R_i a R_j vede hrana I_j tehdy, pokud během provádění instrukce I_j dojde k přesunu dat (ať změněných nebo ne) z registru R_i do R_j. Podobné pravidlo platí i pro vstup a výstup dat ze systému. Hrana mezi registrem R_i a uzlem IN nebo OUT je označena I_j, pokud během instrukce I_j je proveden přesun dat z uzlu IN do registru R_i, nebo pokud jsou data z registru R_i přesunuta do uzlu OUT.

Příklad jednoduché instrukční sady neexistujícího mikroprocesoru a S-graf patřící k této sadě:

I_1: MVI A,D	A = IN	I_6: MOV A,M	A = IN
I_2: MOV R, A	R = A	I_7: ADD R	A = A + R
I_3: MOV M,R	OUT = R	I_8: ORA R	A = A ∨ R
I_4: MOV M, A	OUT = A	I_9: ANA R	A = A ∧ R
I_5: MOV R, M	R = IN	I_{10}: CMA A,D	A = ¬A



Obrázek 3.1 S-graf pro jednoduchou instrukční sadu

Na základě S-grafu a funkčního chybového modelu pro procesor popisovaného dříve byly nalezeny 3 kategorie testů [9], [2].

1. **Testy na správnou funkci dekódování registru**

   Tyto testy se provádí zápisem a čtením registrů. I přes to, že je více způsobů, jak do registrů zapisovat a číst z nich, zvolíme nejkratší sekvenci. Jedním z důvodů je potřeba ze začátku testu otestovat základní principy. Dalším důvodem je účast registrů v následujících delších sekvencích testů. Tento přístup se v [3] nazývá nabalování, kdy jsou v procesoru testovány nejjednodušší komponenty procesoru pomocí předem daných instrukcí. Po ověření správnosti výpočtů v daném bloku jsou přidávány instrukce pro další testovaný blok.

2. **Testy na správnou funkci dekódování a pořadí instrukcí.**

   Tyto testy jsou zaměřeny na chyby, které ovlivní vykonání instrukce I a způsobí chybu či spíše selhání v konečném výsledku instrukce. Tento stav nastane, pokud nějaké mikroinstrukce instrukce I nejsou provedeny nebo naopak jsou chybně provedeny některé mikroinstrukce navíc. Je všeobecně lehké detekovat chybějící mikroinstrukce. Detekovat přebývající mikroinstrukce je složitější; na detekci byla vytvořena metoda kódových slov.

3. **Testy na správnou funkci ukládání, čtení a přenosu dat.**

Funkce pro ukládání a čtení dat a funkce pro jejich přenos jsou testovány najednou. Důvodem je schopnost testovacích funkcí objevit poruchu stálá 1 nebo stálá 0 jak na přenosové cestě z A→ B, tak tyto stejné chyby objevit v registrech odpovídajících uzlům A a B. Test přenosové cesty a test registru využívají různá data. Během testu se na každém bitu transportní cesty vystřídá 0 a 1. Jakákoli dvojice bitů je nastavena zároveň na rozdílnou hodnotu 0 a 1.

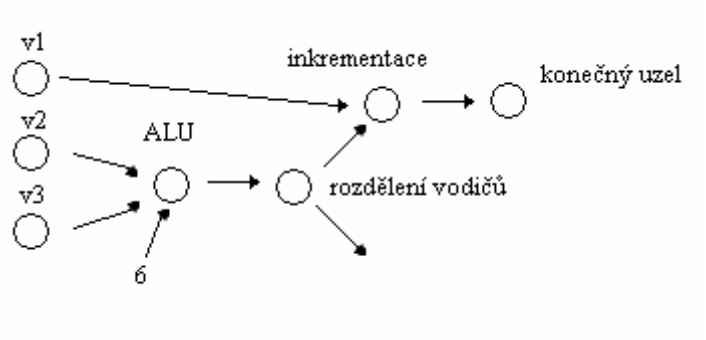
Příklad testu pro 8 bitovou sběrnici je v tabulce 3.1. Řádky jsou jednotlivé vzorky testu. Číslo sloupce označuje pozici bitu na sběrnici. Tento test objeví na sběrnici všechny chyby typu trvalá 1, trvalá 0 a všechny zkraty v jakékoli dvojici bitů.

	7	**6**	**5**	**4**	**3**	**2**	**1**	**0**
1	1	1	1	1	1	1	1	1
2	1	1	1	1	0	0	0	0
3	1	1	0	0	1	1	0	0
4	1	0	1	0	1	0	1	0
5	0	0	0	0	0	0	0	0
6	0	0	0	0	1	1	1	1
7	0	0	1	1	0	0	1	1
8	0	1	0	1	0	1	0	1

Tabulka 3.1 Příklad testu sběrnice

Žádný konkrétní chybový model pro testování funkcí pro práci s daty v [2] ani v [9] není uveden. Předpokládá se, že zatímco testovací rutiny jsou vyvíjeny na úrovni funkčního modelu, data pro test jsou navržena na úrovní hradel.

Tak jako S-graf fungují na podobném principu i decision diagrams – rozhodovací diagramy[2], které vycházejí z datové cesty. V knize[1] lze nalézt termín strukturní datová cesta. Jde o znázornění cesty na úrovni dat, jak se postupně přesouvají přes jednotlivé moduly a příslušné operace pomocí grafu. Uzly, které nejsou listové znamenají nějakou operaci, či rozdělení vodičů. Listové znázorňují vstup či výstup ze systému.



Obrázek 3.1 Příklad strukturní datové cesty

U takto podobných algoritmů můžeme návrh testu rozdělit na dvě části. První částí je nalezení správné datové cesty. Druhou částí je po jednotlivých datových cestách spouštět testy a ověřovat je.

Pro zjištění nefunkčnosti nějaké součástky je zapotřebí průběžné testování komponenty. Časové prodlevy v testech nejvíce závisí na aplikaci, v které daný mikrokontrolér pracuje. Samozřejmostí je, že čím větší nároky na správnost provozu jsou, tím častěji je nutné testy provádět. Nesmíme zapomenout ale na negativní stránku provádění testů, při testu se zařízení věnuje pouze testu a tím se může porušit požadovaná maximální doba odezvy systému. Pro velmi důležitá zařízení je vhodné mít více testů. Kratší, které pokryjí menší oblast chyb a trvají pouze velmi krátkou dobu a pak testy komplexnější, které se provádějí při menším vytížení.

Pokud bychom znali přesný návrh ALU, lze použít rozdílné metody a přístupy. Jako například metoda FMEA (failure modes and effects analysis). Pokud bychom byli přímo u návrhu ALU, je vhodné do tohoto systému vsadit vestavěný test, tzv. built-in self-test.

4 Realizace knihovny

Pro realizaci knihovny, která ověří funkčnost ALU jsem si vybral mikroprocesor MSP430F168IPM od firmy Texas Instruments. Je to 16 bitový mikrokontrolér pro všeobecné použití a je postaven jako architektura RISC, kde jeden hodinový cyklus trvá 125 ns. CPU komunikuje s okolím hlavně pomocí dvou sběrnic, jednou je sběrnice datová a druhou je sběrnice adresová. Spolehlivost výrobce definuje dobou MTBF $29*10^8$ a hodnotou FIT=3. Jeden z hlavních důvodů pro výběr MSP430F168 je umístění daného mikrokontroléru na FITkitu a také už zmiňovaná architektura RISC, kdy je instrukční sada jednoduší.

Procesor mikrokontroléru obsahuje instrukční sadu o 51 instrukcích. Z čehož je 27 instrukcí prováděno přímo a 24 instrukcí je emulováno. Emulace instrukcí nám přináší zjednodušené psaní a čtení kódu. Emulované instrukce jsou assemblerem nahrazeny přímo prováděnými instrukcemi. Přímo prováděné instrukce mají tři formáty:

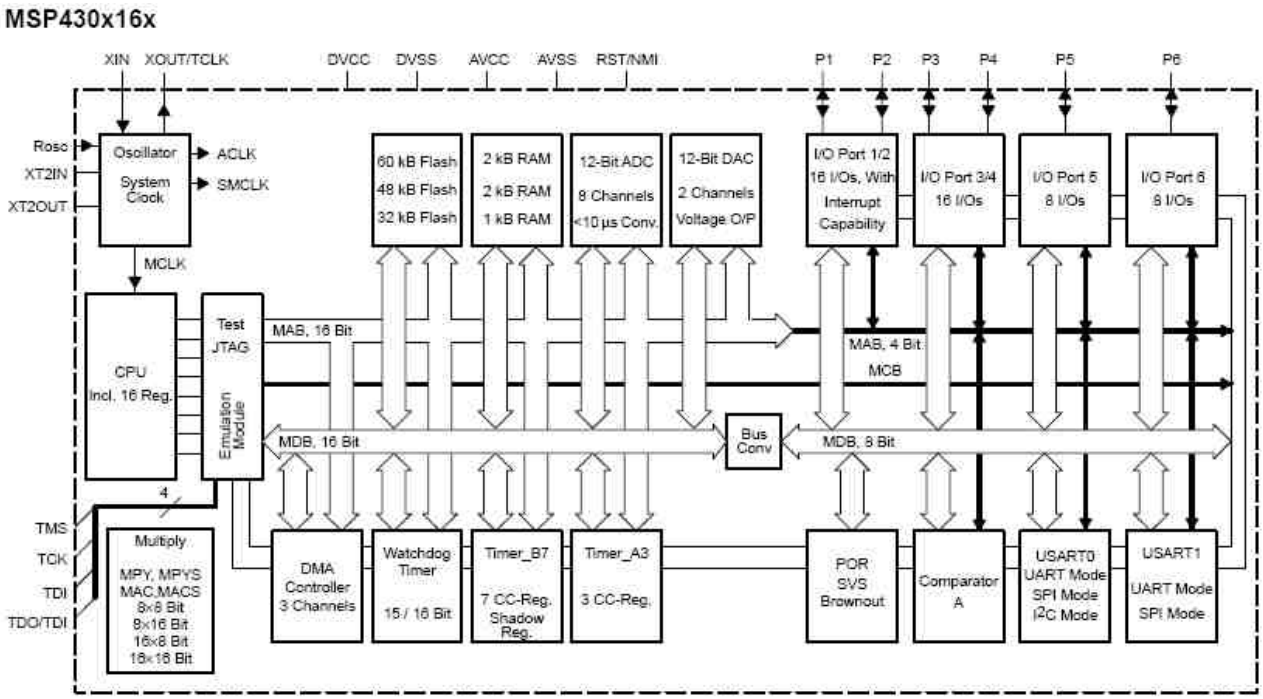
- S dvěma operandy
- S jedním operandem
- Skoky

Instrukce s operandy mohou být jak pro byte tak pro slovo. Procesor má 16 registrů, z toho 12 je všeobecně použitelných. Registr R0 je programový čítač, registr R1 je ukazatel na zásobník, R2 je statusové slovo a R3 slouží jako generátor konstant. Procesor dokáže pracovat v 7 adresových módech, všechny adresové módy jsou použitelné se všemi instrukcemi. Všech sedm adresových módů je použitelné pro zdrojové operandy a čtyři pro cílové operandy. Módy jsou následující (na konci řádku je syntaxe zápisu):

- Registrový mód – operand se nachází v registru Rn
- Indexový mód – operand se nachází v paměti na adrese vypočítané jako součet obsahu registru a konstanty (Rn + X). Konstanta je uložena v následujícím slově. X(Rn)
- Symbolický mód – operand je uložený na adrese ADDR. Je použit indexový mód s pomocí PC registru (PC + X) ukazuje na ADDR. X je uloženo za instrukcí. V reálu je použit indexový mód X(PC) ADDR
- Absolutní mód – operand je uložený na adrese ADDR. V reálu je použit indexový mód X(R2). R2 jako generátor konstant vrátí 0. &ADDR
- Nepřímý registrový mód – registr je použit jako ukazatel na operand. @Rn
- Nepřímý mód s autoinkrementem – registr je použit jako ukazatel na operand a následně je zvýšen o 1 u operace s bytem nebo o 2 pokud se pracuje se slovy. @Rn+

- Přímý operand – Konstanta je uložená ve slově za instrukcí, v reálu se použije nepřímý mód s autoinkrementem @PC+. #N

Blokové schéma mikrokontroléru:



Obrázek 4.1 Blokové schéma MSP430F168

4.1 Instrukční set mikrokontroléru MSP430

Nejdříve uvedu vysvětlivky k instrukční sadě:

- † takto označené instrukce jsou emulovány.
- (.B) u instrukcí znamená možnost přidání .B za instrukci a tím docílíme práci pouze s bytem místo slovem.
- V je overflow bit, pokud je nastaven na 1, došlo k přetečení znaménkového čísla při výpočtu:
  - U operací ADD(.B), ADDC(.B) se jedná o následující případy:
    pozitivní číslo + pozitivní číslo = negativní číslo
    negativní číslo + negativní číslo = pozitivní číslo.
  - U operací SUB(.B),SUBC(.B),CMP(.B) se jedná o následující případy:
    pozitivní číslo - negativní číslo = negativní číslo
    negativní číslo - negativní číslo = pozitivní číslo.

- N je negative bit, pokud je bit nastaven na 1, je výsledek operace negativní. Při kladném výsledku je nastaven na 0.
- Z je zero bit, pokud je nastaven na 1, je výsledek operace 0. V opačném případě je bit nastaven na 1.
- C je carry bit, je nastaven na 1 – pokud došlo k přenosu. Při nastavení na 0 k přenosu nedošlo.
- * statusový bit je operací ovlivněn.
- – statusový bit není operací ovlivněn.
- 0 statusový bit je vynulován.
- 1 statusový bit je nastaven na 1.
- Src je zdrojový operand, dst je cílový operand. Label je návěští.
- U emulovaných instrukcí uvádím instrukce, které se v reálu provedou. Jsou však uvedeny pouze varianty pro celá slova.

Instrukční sada:

Instrukce			Popis	V	N	Z	C
ADC(.B)†	dst	Přidá statusový bit C do cíle. Instrukce je emulována pomocí ADDC #0,dst.					
			dst + C → dst	*	*	*	*
ADD(.B)	src,dst	Sečte zdroj s cílem do cíle.	src + dst → dst	*	*	*	*
ADDC(.B)	src,dst	Sečte zdroj, bit C a cíl do cíle.	src + dst + C → dst	*	*	*	*
AND(.B)	src,dst	Log. funkce AND pro zdroj a cíl.	src .and. dst → dst	0	*	*	*
BIC(.B)	src,dst	Vynuluje bity v cíli.	not src .and. dst → dst	–	–	–	–
BIS(.B)	src,dst	Nastaví v cíli bity na 1.	src or dst → dst	–	–	–	–
BIT(.B)	src,dst	Test bitů v cíli.	src .and. dst	0	*	*	*
BR†	dst	Skok na cíl. Instrukce je emulována pomocí instrukce MOV dst, PC.					
			dst →PC	–	–	–	–
CALL	dst	Skok do podprogramu.	PC+2 → stack, dst → PC	–	–	–	–
CLR(.B)†	dst	Nastaví v cíli 0. Instrukce je emulována pomocí instrukce MOV #0,dst.					
			0 → dst	–	–	–	–
CLRC†		Nastaví C na 0. Instrukce je emulována pomocí instrukce BIC #1,SR.					
			0 → C	–	–	–	0
CLRN†		Nastaví N na 0. Instrukce je emulována pomocí instrukce BIC #4,SR.					
			0 → N	–	0	–	–
CLRZ†		Nastaví Z na 0. Instrukce je emulována pomocí instrukce BIC #2,SR..					
			0 → Z	–	–	0	–
CMP(.B)	src,dst	Porovná zdroj a cíl.	dst − src	*	*	*	*

DADC(.B)†	dst	Sečte dekadicky C k cíli. Instrukce je emulována pomocí instrukce DADD #0, dst.	dst + C → dst (decimally)	* * * *
DADD(.B)	src,dst	sečte dekadicky zdroj a C to cíle.	src + dst + C → dst (decimally)	* * * *
DEC(.B)†	dst	Cíl zmenší o 1. Instrukce je emulována pomocí instrukce SUB #1,dst.	dst − 1 → dst	* * * *
DECD(.B)†	dst	Cíl zmenší o 2. Instrukce je emulována pomocí instrukce SUB #1,dst.	dst − 2 → dst	* * * *
DINT†		Vypne přerušení. Instrukce je emulována pomocí instrukce BIC #8,SR.	0 → GIE	− − − −
EINT†		Zapne přerušení. Instrukce je emulována pomocí instrukce BIS #8,SR.	1 → GIE	− − − −
INC(.B)†	dst	Cíl zvětší o 1. Instrukce je emulována pomocí instrukce ADD #1,dst.	dst +1 → dst	* * * *
INCD(.B)†	dst	Cíl zvětší o 2. Instrukce je emulována pomocí instrukce ADD #2,dst.	dst+2 →dst	* * * *
INV(.B)†	dst	Negace cíle. Instrukce je emulována pomocí instrukce XOR #0FFFFh,dst.	.not.dst → dst	* * * *
JC/JHS	label	Skok pokud se C rovná 1/ Skok při vyšším cíli nebo stejné hodnotě.		− − − −
JEQ/JZ	label	Skok pokud se operandy rovnají / Skok pokud se Z rovná 1.		− − − −
JGE	label	Skok při vyšší nebo stejné hodnotě cíle.		− − − −
JL	label	Skok při nižší hodnotě cíle.		− − − −
JMP	label	Skok na PC + 2 x offset → PC.		− − − −
JN	label	Skok pokud N je 1.		− − − −
JNC/JLO	label	Skok pokud je C 0/Skok pokud je cíl menší.		− − − −
JNE/JNZ	label	Skok pokud se operandy nerovnají/ Skok pokud je Z 0.		− − − −
MOV(.B)	src,dst	Přesun zdroje do cíle.	src → dst	− − − −
NOP†		Žádná operace. Instrukce je emulována pomocí instrukce MOV #0,R3.		− − − −
POP(.B)†	dst	Vyjme položku ze zásobníku do cíle. Instrukce je emulována pomocí instrukce MOV @SP, dst.	@SP → dst, SP+2 → SP	− − − −
PUSH(.B)	src	Dá zdroj na zásobník.	SP − 2 → SP, src → @SP	− − − −
RET†		Návrat z podprogramu. Instrukce je emulována pomocí instrukce MOV @SP+, PC.	@SP → PC, SP + 2 → SP	− − − −
RETI		Návrat z přerušení.		* * * *
RLA(.B)†	dst	Rotace vlevo aritmeticky. Instrukce je emulována pomocí instrukce ADD dst, dst.		* * * *

RLC(.B)†	dst	Rotace vlevo přes C. Instrukce je emulována pomocí instrukce ADDC dst, dst.		* * * *
RRA(.B)	dst	Rotace vpravo aritmeticky.		0 * * *
RRC(.B)	dst	Rotace vpravo přes C.		* * * *
SBC(.B)†	dst	Odečte negaci (C) od cíle. Instrukce je emulována pomocí instrukce SUBC #0,dst.	dst + 0FFFFh + C → dst	* * * *
SETC†		Nastaví C na 1. Instrukce je emulována pomocí instrukce BIS #1,SR.	C 1 → C	− − − 1
SETN†		Nastaví N na 1. Instrukce je emulována pomocí instrukce BIS #4,SR.	1 → N	− 1 − −
SETZ†		Nastaví Z na 1. Instrukce je emulována pomocí instrukce BIS #2,SR.	1 → Z	− − 1 −
SUB(.B)	src,dst	Odečte zdroj od cíle.	dst + .not.src + 1 → dst	* * * *
SUBC(.B)	src,dst	Odečte zdroj a negaci (C) od cíle.	dst + .not.src + C → dst	* * * *
SWPB	dst	Přehodí byty.		− − − −
SXT	dst	Rozšíří znaménko z 8.bitu až na 16.bit.		0 * * *
TST(.B)†	dst	Test cíle. Instrukce je emulována pomocí instrukce CMP #0,dst.	dst + 0FFFFh + 1	0 * * 1
XOR(.B)	src,dst	XOR zdroje a cíle	src .xor. dst → dst	* * * *

4.2 Návrh a implementace testu procesoru

Všeobecný základní postup testu celého mikroprocesoru je uveden v literatuře[14]. Postup je následující:

- Reset
- Kontrola čítače instrukcí
- Kontrola adresace registrů
- Kontrola přesunů mezi registry
- Kontrola ukazatele zásobníku
- Kontrola funkce ALU
- Kontrola střadače
- Kontrola dekodéru instrukcí

Jedním z jednoduchých testů může být pseudonáhodná posloupnost instrukcí. Lze určit pouze délku testu a instrukce, které nemají být použity. Nevýhodou je že celkový správný výsledek testu není předem znám. Pokud máme posloupnost určenou a potřebujeme velmi šetřit místem v paměti, můžeme použít nějakou kompresní metodu pro získání výsledků. Třeba počet jedniček nebo nul. Pokud po sekvenci příkazů dojdeme k předpokládanému počtu, prohlásíme výsledek za správný.

Aplikaci pro FITkit jsem rozvinul z aplikace řízení displeje mikrokontrolérem a pojal jsem téma trochu více široce než je v zadání. V zadání je zmíněna pouze aritmeticko logická jednotka, ale já jsem se zaměřil na celkové prověření procesoru v daném mikrokontroléru. FPGA v tomto případě není potřeba na výpočty, ale slouží pouze k zajištění ovládání displeje přes MSP430. Displej zobrazuje výsledky testů. V aplikacích na FITkitu je zajištěna komunikace mezi FPGA a mikrokontrolérem pomocí SPI. Knihovny pro FITkit jsou psány v jazyce C, je to vysoko abstraktní jazyk, a proto jako takový není vhodný pro psaní testů. Překladač nám nezaručí jednotlivé instrukce, které potřebujeme pokud používáme jen vysoko abstraktní příkazy v jazyce C. Avšak existuje možnost volání instrukcí asembleru pomocí klíčového slova `asm`. Syntaxe použití je následující: `asm("mov %1, %0": "=r" (výsledek): "m" (zdroj))`. Příkaz se skládá z několika částí, nejdříve je uvedena požadovaná instrukce jako normální řetězec. Znak % určuje, že následuje číslo udávající pořadí operandu, který zatupuje. Za řetězcem a dvojtečkou následují výstupní operandy, jestliže je jich více, oddělují se čárkou. Před samotný operand se ještě uvádí řetězec značící parametr (platí pro výstup i vstup): r znamená registr, m paměť, i je přímý operand (ve většině případů je to celočíselná konstanta) a I je integer operand. Znak = u parametru operandu znamená výstup. Za další dvojtečkou následují vstupní operandy a jejich parametry. Pak může následovat ještě jedna dvojtečka a seznam registrů, které jsou ovlivněny nepřímo přepisem při provádění dané instrukce. Existuje ještě jeden zápis, kdy v instrukci místo pořadí operandů uvedeme zástupné názvy.

```
asm("mov %src,%res": [res] "=r" (výsledek): [src] "m" (zdroj));
```

Pro testy při provozu není ideální vždy celý systém resetovat, protože tím můžeme ztratit některá data z dynamických pamětí. Na druhou stranu je reset u MSP430 rychlý a samotný zabere 4 takty. Zvolil jsem tedy variantní řešení a to s resetem i bez resetu, kvůli návaznosti ostatních výpočtů, aby nebyly smazány jednotlivé registry procesoru. Pokud se provede reset, následovat bude test. Ten je tedy proveden vždy po startu systému. Reset se volá takto:

```
__asm__ __volatile__(
    "call #0x0fffe"
 );
```

Kde `#0x0fffe` je adresa ve vektoru přerušení. Daný mikrokontrolér má přerušení více, k našemu testování to však nebudeme potřebovat.

Nejdříve otestujeme čítač instrukcí a to pouze velmi jednoduchým skokem:

```
__asm__ __volatile__( "jmp spravny_skok\n" );
```

```
chybka();
__asm__ __volatile__( "spravny_skok:\n" );
```

Pokud se skok nepovede, tedy pokud nejde správně přičítání k programovému čítači, je zavolána chybová funkce.

Pak následují všeobecné přístupné registry. A to tím, že provedeme už zmíněný test, kdy do registrů postupně nahráváme tyto hodnoty:

1111 1111 1111 1111 = ffff

1111 1111 0000 0000 = ff00

1111 0000 1111 0000 = f0f0

1100 1100 1100 1100 = cccc

1010 1010 1010 1010 = aaaa

0000 0000 0000 0000 = 0000

0000 0000 1111 1111 = 00ff

0000 1111 0000 1111 = 0f0f

0011 0011 0011 0011 = 3333

0101 0101 0101 0101 = 5555

Zde je ukázka zadávání a posouvání hodnot po registrech. Nejdříve jsou registry vloženy na zásobník a po testu jsou zase hodnoty ze zásobníku vráceny na své místo:

```
WDG_stop();
__asm__ __volatile__("push r4\n");
__asm__ __volatile__("push r5\n");
__asm__ __volatile__("push r6\n");
__asm__ __volatile__("push r7\n");
__asm__ __volatile__("push r8\n");
__asm__ __volatile__("push r9\n");
__asm__ __volatile__("push r10\n");
__asm__ __volatile__("push r11\n");
__asm__ __volatile__("push r12\n");


__asm__ __volatile__("mov #0xffff,r4\n");

__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0xff00,r4\n");
```

```
__asm__ __volatile__("mov r5,r6\n");
__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0xf0f0,r4\n");

__asm__ __volatile__("mov r6,r7\n");
__asm__ __volatile__("mov r5,r6\n");
__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0xcccc,r4\n");


__asm__ __volatile__("mov r7,r8\n");
__asm__ __volatile__("mov r6,r7\n");
__asm__ __volatile__("mov r5,r6\n");
__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0xaaaa,r4\n");


__asm__ __volatile__("mov r8,r9\n");
__asm__ __volatile__("mov r7,r8\n");
__asm__ __volatile__("mov r6,r7\n");
__asm__ __volatile__("mov r5,r6\n");
__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0x0000,r4\n");
```

Postup dat v registrech rozšiřujeme a do počátečního registru zadáváme nová data. Až se dostaneme ke koncovým registrům, začneme porovnávat chtěnou hodnotu s hodnotou z registru. Pokud se neshodují, je zavolána chybová funkce a displeji se rozsvítí slovo chyba. Tento test nemá velký smysl provádět zvlášť pro byty, pouze s ohledem na adresování se může jevit jako možný, ale poruchy na vodičích či v tištěném obvodu stávající test odhalí.

Zde je ukázka porovnávání, srovnani je proměnná typu uint16_t:

```
  __asm__ __volatile__(
     "mov r15,%0"
   : "=m" (srovnani) );

__asm__ __volatile__(
     "cmp #0xffff,%0"
```

```
   :: "m" (srovnani) );

__asm__ __volatile__(
     "jne chybaV\n" );
```

Chybová funkce se volána na konci při nastalé chybě:

```
__asm__ __volatile__("jmp vseOK");
__asm__ __volatile__("chybaV:\n");
chybka();
__asm__ __volatile__("vseOK:\n");
```

Tímto testem se otestují všechny poruchy v registrech a na datové sběrnici typu trvalá 1 a trvalá 0. Zároveň se prověří většina případů nesprávné adresace registrů. Pokud se některé adresy z důvodu poruchy překrývají, tak jsou data v registrech přepsána a tato chyby je následně zjištěna. Vynechání registru je taky zaznamenáno, ztratí se všechny hodnoty.

Následuje ověření správné funkce ukazatele vrcholu zásobníku. Tento test provedeme pomocí registrů a instrukcí pop a push. Ukazatel vrcholu zásobníku provádí posun minimálně o 2 byty. I když se ukládá pouze jeden byte, je na zásobníku vyhrazeno celé slovo. Ukázka testu vypadá takto:

```
__asm__ __volatile__("mov #0xaaaa,r4\n");
__asm__ __volatile__("mov #0x5555,r5\n");
__asm__ __volatile__("push r4\n");
__asm__ __volatile__("push r5\n");
__asm__ __volatile__("pop r6\n");
__asm__ __volatile__("pop r7\n");

__asm__ __volatile__("cmp r6,r5");
__asm__ __volatile__("jne chybaSP\n" );

__asm__ __volatile__("jmp dobreSP\n" );

__asm__ __volatile__("cmp r7,r4");
__asm__ __volatile__("jne chybaSP\n" );

__asm__ __volatile__("dobreSP:\n" );
```

Také je možné s tímto registrem provádět aritmetické operace, pokud víme, jaká data se na zásobníku nacházejí.

```
__asm__ __volatile__("mov #0xaaaa,r4\n");
__asm__ __volatile__("mov #0x5555,r5\n");
__asm__ __volatile__("push r4\n");
__asm__ __volatile__("push r5\n");

__asm__ __volatile__("add #0x2,r1\n");
```

Tím jsme ztratili na zásobníku poslední záznam.

Následuje kontrola ALU. Zde je mnoho možností, testovat všechny by bylo velmi zdlouhavé a proto si určíme několik testovacích vektorů, které budeme používat. Pro celá slova to budou vektory 10101010 10101010 = 0xaaaa, 11111111 11111111 = 0xffff, 10000000 000000000 = x8000, 00000000 00000000 = 0x0000, 01010101 01010101 = 0x5555. Bytové vektory jsou velmi podobné. V testu jsou prováděny aritmetické či logické operace. Hlídá se výsledek a nastavení 4 stavových bitů. S nutnou kontrolou těchto stavů se také testují podmíněné skoky. Jsou tu uváděny pouze některé testy. Ostatní jsou ve zdrojových souborech.

Začneme sčítáním, je důležité i z důvodu správného adresování. Vyzkoušíme přenos, negativní výsledek a nulový výsledek.

```
__asm__ __volatile__("mov #0xaaaa,r4\n");
__asm__ __volatile__("add #0x8000,r4\n");
__asm__ __volatile__("jnc chybaALU\n");
__asm__ __volatile__("and #0x5555,r4\n");
__asm__ __volatile__("jnz chybaALU\n");
__asm__ __volatile__("mov #0xaaaa,r4\n");
__asm__ __volatile__("bic #0x8000,r4\n");  r4=2AAA
__asm__ __volatile__("cmp #0xaaaa,r4\n");
__asm__ __volatile__("jge chybaALU,r4\n");
__asm__ __volatile__("mov #0x0000,r4\n");
__asm__ __volatile__("dec r4\n");
__asm__ __volatile__("jnc chybaALU,r4\n");
__asm__ __volatile__("pra r4\n");
__asm__ __volatile__("jnc chybaALU,r4\n");
__asm__ __volatile__("swpb r4\n");


__asm__ __volatile__("chybaALU:\n");
chybka();
```

5 Závěr

V této práci jsem se zprvu zabýval vlastnostmi testů. Ohledně výpočetní složitosti je zejména důležité poukázat na nutnost mít testy co nejméně redundantní. Je praktické mít testy málo složité, krátké a v tom případě rychlé. U velmi rizikových systémů je vhodné provádět testy několikrát za sekundu a výpočetní výkon potřebujeme také pro samostatnou aplikaci.

Dále jsme uvedl dělení poruch a různé chybové modely. Různé stupně abstrakce nám dovolují u chybový modelů si vybrat ten nejvíce užitečný pro naše podmínky. Čím je abstrakce modelu nižší, tím složitější je u daného modelu vyjádřit všechny možné poruchy na dané úrovni. Jde také o rychlost, se kterou dokážeme aplikovat některý chybový model na konkrétní řešení. U vysoko abstraktních modelů je samozřejmě rychlost větší. U nižších abstrakcí je možné lépe lokalizovat chybu. Pro mikrokontoléry a potažmo procesory, ve kterém se ALU nachází, kde pro konkrétní aplikaci není známa podrobná vnitřní implementace obvodů, je možné použít funkční chybový model pro procesory. Dále jsem se zabýval možnostmi, jak udělat systémy bezpečné a to pomocí různých metod. Navazuje krátké pojednání o možných poruchách. Následují možnosti generování testů pro ALU. Ze všeobecných modelů je princip testování jasný.

Testy by měly být ze začátku jednoduché a otestovat hlavně samotné registry. Pak se testuje správné provádění ukládání dat na zásobník a následně i instrukce aritmetických a logických operací . V poslední kapitole je nástin konkrétního mikrokontroléru, jeho instrukční sada a ukázky z provedení jeho testů. Knihovna pokrývá většinu poruch na registrech a adresové sběrnici. Protože nelze vycházet z jiných možností, než jen z instrukční sady daného mikrokontroléru, je nemožné pokrýt všechny poruchy a chyby z nich vyplývající. Proto z mého pohledu je současný stav dostačující, ale je možno testy jednoduše rozšířit na úkor zabraného času na procesoru.

Literatura

[1] Jha, N., Sandeep, G. Testing of Digital Systems. Cambridge, Cambridge University Press 2003.

[2] Novák, O., Gramatová, E., Ubar, R. a kolektiv. Handbook of Testing Electronic Systems. Praha, Nakladatelství ČVUT 2005.

[3] Drábek, V. Spolehlivost a diagnostika. Praha, SNTL 1986.

[4] Sobotka, J. Bezpečné aplikace s mikrokontrolery. [Diplomová práce], Vysoké učení technické v Brně 2008

[5] Maněna, M. Simulace logických obvodů v SystemC. [Diplomová práce], České vysoké učení technické v Praze 2008.

[6] Kolektiv autorů. Arithmetic logic unit. Dokument dostupný na URL http://en.wikipedia.org/wiki/Arithmetic_logic_unit (leden 2009)

[7] Hlavička, J., Kottek, E., Zelený, J., Diagnostika elektronických číslicových obvodů. Praha, SNTL 1982

[8] Šimák, J. Implementace prostředků pro vestavěnou diagnostiku. [Bakalářská práce], České vysoké učení technické v Praze 2008. Dokument dostupný na URL https://dip.felk.cvut.cz/browse/pdfcache/simakj1_2008bach.pdf (květen 2009)

[9] Thatte, S.M., Abraham, J.A., Test Generation for Microprocessors, IEEE Transactions on Computers, vol. 29, no. 6, pp. 429-441, June 1980, doi:10.1109/TC.1980.1675602. Dokument dostupný na URL http://www2.computer.org/portal/web/csdl/doi/10.1109/TC.1980.1675602 (květen 2009)

[10] Drábek, V., Způsoby vytvoření OPP. [Učební materiály k předmětu Systémy odolné proti poruchám, FIT VUT]

[11] MSP 430 User's Guide. Dokument dostupný na URL focus.ti.com/lit/ug/slau049f/slau049f.pdf (květen 2009)

[12] Neznámý autor, MSP430 - stručný úvod. Dokument dostupný na URL http://www.bastl.sk/modules/smartsection/item.php?itemid=6 (květen 2009)

[13] Miczo A., Digital Logic Testing and Simulation. New Jersey, John Wiley & Sons 2003

[14] Neumann P., Testování mikroprocesorů [Učební materiály k předmětu Diagnostika číslicových systémů, FAI UTB]

Seznam použitých zkratek

gnd		zem
vdd		napětí
RTL	register transfer level	úrovni přenosu mezi registry a funkčními bloky
SA0	stuck at 0	porucha trvalá logická 1
SA1	stuck at 1	porucha trvalá logická 0
VLSI	very large scale integration	obvody s velmi velkým stupněm integrace
SoC	system on a chip	celý integrovaný obvod na jednom plátku křemíku

Seznam příloh

Příloha 2. Výňatek kódu programu v asembleru s umístěním v paměti s porovnáním se zápisem v jazyce C

Kód v jazyce C

```
__asm__ __volatile__("mov r10,r11\n");
__asm__ __volatile__("mov r9,r10\n");
__asm__ __volatile__("mov r8,r9\n");
__asm__ __volatile__("mov r7,r8\n");
__asm__ __volatile__("mov r6,r7\n");
__asm__ __volatile__("mov r5,r6\n");
__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0x0f0f,r4\n");

__asm__ __volatile__("mov r11,r12\n");
__asm__ __volatile__("mov r10,r11\n");
__asm__ __volatile__("mov r9,r10\n");
__asm__ __volatile__("mov r8,r9\n");
__asm__ __volatile__("mov r7,r8\n");
__asm__ __volatile__("mov r6,r7\n");
__asm__ __volatile__("mov r5,r6\n");
__asm__ __volatile__("mov r4,r5\n");
__asm__ __volatile__("mov #0x3333,r4\n");


  __asm__ __volatile__(
     "mov r12,%0"
   : "=m" (srovnani) );

__asm__ __volatile__(
     "cmp #0xffff,%0"
   :: "m" (srovnani) );

__asm__ __volatile__(
     "jne chybaV\n" );
```

Stejný úsek kódu v asembleru:

```
    4b36:  0b 4a            mov   r10,  r11   ;
    4b38:  0a 49            mov   r9,   r10   ;
    4b3a:  09 48            mov   r8,   r9    ;
```

```
4b3c:  08 47           mov   r7,   r8    ;
4b3e:  07 46           mov   r6,   r7    ;
4b40:  06 45           mov   r5,   r6    ;
4b42:  05 44           mov   r4,   r5    ;
4b44:  34 40 0f 0f     mov   #3855,      r4    ;#0x0f0f
4b48:  0c 4b           mov   r11,  r12   ;
4b4a:  0b 4a           mov   r10,  r11   ;
4b4c:  0a 49           mov   r9,   r10   ;
4b4e:  09 48           mov   r8,   r9    ;
4b50:  08 47           mov   r7,   r8    ;
4b52:  07 46           mov   r6,   r7    ;
4b54:  06 45           mov   r5,   r6    ;
4b56:  05 44           mov   r4,   r5    ;
4b58:  34 40 33 33     mov   #13107,     r4    ;#0x3333
4b5c:  84 4c 00 00     mov   r12,  0(r4) ;
4b60:  b4 93 00 00     cmp   #-1,  0(r4) ;r3 As==11
4b64:  6d 20           jnz   $+220       ;abs 0x4c40
```