Testování mikrokontrolérů

IMP Projekt 2020/2021

Autor: Jan Lorenc (xloren15)
Datum: 25. listopadu 2020

Anotace

Vzhledem k pandemickým omezením nebylo možné projekt vypracovat zcela, neboť jsem nemohl obdržet ARM-FITKit3, na kterém se měl program vyvíjet. Projekt je proto zaměřen spíše na teoretickou část problematiky. Rozebírány budou metody samočinného testování registrů procesoru, paměti RAM a digitálního vstupu/výstupu. Tyto testy jsou i naprogramované, program lze přeložit, avšak pravděpodobně nebude z výše zmíněného důvodu funkční. Slouží spíše k demonstraci toho, jakým způsobem či principy by se mohlo výsledku dosáhnout.

Obsah

1 Zadání

Cílem projektu bylo navrhnout dva až tři mechanismy pro samočinné testování vybraných komponent mikrokontrolérů, následně je implementovat, funkčnost ověřit a demonstrovat. Vzhledem k nemožnosti vyzvednutí platformy ARM-FITKit3 z důvodu pandemických opatření a po konzultaci s vedoucím práce došlo k úpravě praktické části zadání. Testy stále musí být naimplementované, avšak již se nepočítá s funkčností řešení. Důležitá je idea za tvorbou realizace a musí být jasně zdokumentováno, co bylo implementací zamýšleno.

2 Testované komponenty

Mikrokontroléry obsahují mnoho komponent, jejichž funkcionalitu lze testovat. K nim patří například paměť flash nebo RAM, registry, zásobník, hodiny, vstup/výstup a další. Pro tento projekt jsem zvolil testování standardních registrů procesoru, paměti RAM a digitálního vstupu a výstupu, tedy pinů mikrokontroléru.

2.1 Registry CPU

Registry procesoru slouží pro ukládání a zpracování dočasných dat. Tyto paměti jsou obecně malé, ale tím, že jsou přímo v CPU, se k nim přistupuje velmi rychle. V mikrokontroléru Cortex-M0+ se nachází 16 standardních registrů o velikosti 32 bitů, tyto jsou předmětem mých testů, a speciální registry.

Registry R0-R12 jsou tzv. registry pro obecné použití a dále se dělí na nižší (R0-R7) a vyšší (R8-R12). S těmito registry mohou instrukce běžně pracovat. Vyšší registry však nemusí být přístupné některým Thumb instrukcím. Registr R13 neboli SP slouží k přístupu na zásobník. Obsahuje adresu vrcholu zásobníku. V Cortex-M0+ se dělí ještě na MSP (Main SP) používaný u privilegovaných aplikací a PSP (Process SP) využívaný běžnými aplikacemi. R14 nebo také LR registr uchovává hodnotu adresy, kam se má program vrátit po vykonání podprogramu. Posledním registrem je Program counter (PC) udržující adresu aktuálně prováděné instrukce.

2.2 Paměť RAM

Jedná se o variabilní paměť pro ukládání dat, jakými jsou zejména zásobník a haldy. V základním nastavení je pro ni v adresovém prostoru vyhrazeno 512MB na adresách 0x20000000-0x3fffffff. Velikost tohoto regionu se může měnit.

2.3 Digitální vstup a výstup

Tento projekt řeší pouze digitální vstupy/výstupy. Fyzicky jsou řešeny vývody z mikrokontroléru neboli piny, na kterých buď je přiveden potenciál vůči zemi a je na něm nastavena logická 1, nebo je potenciál stejný, to potom značí logickou 0. Typicky je tam samozřejmě určitá tolerance. Dále může nastat situace, kdy na pinu není připojeno nic a je tedy ve stavu vysoké impendance. To se řeší pull up/down rezistory nastavujícími výchozí úroveň.

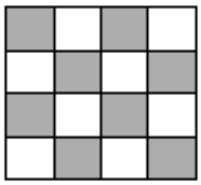
3 Návrh samočinných testů

Všechny testy mají společný záměr. Na jednotlivých komponentách testují poruchu stuck-at. Jedná se o výrobní vadu integrovaných obvodů, kdy jsou jednotlivé piny či signály zaseknuté na hodnotě logické 0, 1 nebo ve stavu vysoké impendance a nelze jim měnit hodnotu. V následujících sekcích jsem čerpal inspiraci v [1, 2, 3, 4].

3.1 Registry CPU

Pro testování registrů používám algoritmus Checkerboard. Algoritmus spočívá v tom, že se do registru nahraje určitý vzor, konkrétně 0xAAAAAAAA a ověří se, zda je hodnota správně nahraná.

V druhém kroku se nahraje jeho inverze, tedy 0x55555555 a opět se ověří. Název plyne z toho, že tyto dva vzory spolu vytváří šachovnici. Tato inverze slouží i k ověření, zda se sousedi neovlivňují.



Write		Read	
0	1	0	1
1	0	1	0
0	1	0	1
1	0	1	0

Obrázek 1: Vzory checkerboard algoritmu (převzato z [2])

3.2 Paměť RAM

K dostačujícímu otestování RAM paměti na poruchu stuck-at stačí provést test statické paměti, pro který jsem zvolil algoritmus MarchX.

Jeho kroky jsou následující:

1. Do pole zapíši samé nuly.
2. Iteruji polem, načítám nuly a zapisuji hodnoty jedna.
3. Iteruji polem zpětně, načítám jedničky a opět nastavuji nuly.
4. Načtu z pole samé nuly.

Typicky testuji jenom část paměti, kterou si předem vymezím. Tato je zmíněné pole v algoritmu. Pokud program neobsahuje pouze tento test, je vhodné si hodnoty testované části paměti před začátkem provedení algoritmu MarchX zazálohovat. Po jeho dokončení se paměť opět obnoví.

3.3 Digitální vstup a výstup

Princip testování pinů je vcelku jednoduchý. Chci-li ověřit hodnotu na některém pinu, nastavím jeho výstup na vstup jiného pinu a tyto hodnoty porovnám. Další možností je test provádět pouze na jednom pinu využitím zpětné vazby.

4 Implementace testů

Pro vývoj jsem využil prostředí KDS verze 3.2.0, kde jsem si vytvořil projekt dle postupů z přednášek a cvičení. Mnoho souborů, zejména tedy hlavičkové, mi KDS vygenerovalo samo. Já pak doplnil pouze soubory *main.c*, odkud řídím program, *test_registers_asm.S* a *test_ram_asm.S*, které obsahují assemblerovké funkce pro testování registrů a paměti RAM, a *test_functions.h* s *test_functions.c* implementující jednotné rozhraní pro spouštění testů. Modul obsahuje funkci testující piny a obalující funkce pro volání assemblerovských testů. U všech testů platí, že pokud skončil úspěšně, kit upozorní uživatele pípnutím, pokud nikoliv, dojde ke třem rychlejším zapípáním. Zdroji informací pro implementaci mi byly [1, 5, 6].

4.1 Registry CPU

Test je implementovaný funkcí *testRegisters* v souboru *test_registers_asm.S* v jazyce strojových instrukcí. Kdyby program nesloužil pouze pro samotné testování, muselo by být řešeno i zálohování registrů, aby tato funkce neovlivnila jejich hodnoty na globální úrovni. V tomto případě by se mimo funkci v souboru nacházela i datová proměnná pro zálohu aktuálně testovaného registru. Jedinou

výjimkou by byly registry R0-R3, které přenáší parametry funkcí, proto se u nich nepočítá se zachováním původní hodnoty. Vzhledem k tomu, že jinou než testovací funkci celý program nemá, přepsání hodnot registrů nám nevadí a není třeba se zálohou v implementaci zabývat.

Postup implementace testu konkrétního registru:

1. Do referenčních registrů si uložím vzory 0xAAAAAAAA a 0x55555555. K tomu využiji instrukci LDR, neboť MOV, byť rychlejší, umí nastavit maximálně 8b hodnotu.
2. Do testovaného registru načtu instrukcí LDR vzor 0xAAAAAAAA.
3. Porovnám s referenčním registrem.
4. Pokud se nerovnají, končím test s chybou.
5. Do testovaného registru načtu instrukcí LDR vzor 0x55555555.
6. Porovnám s druhým referenčním registrem.
7. Pokud se nerovnají, končím test s chybou.

Tento postup provádím u všech nižších standardních registrů, tedy R0-R7. Další registry program netestuje, neboť do nich nelze zapsat instrukcemi Thumb a prostředí KDS mi nedovolí program sestavit. Pro jejich testování by však stačilo pouze používání Thumb instrukcí zakázat. Pro účely projektu to nicméně není potřeba, protože test by vypadal naprosto shodně, jako u předchozích registrů, tedy pro demonstraci principů nám nižší registry stačí.

Speciální test vyžaduje až registr PC, pro který není výše popsaný test aplikovatelný, neboť se jeho hodnota mění s každou instrukcí, tedy i instrukcí samotného testu tohoto registru. Test jsem proto navrhl jednoduše tak, že si uložím hodnotu PC registru, poté k této hodnotě přičtu číslo 8, které značí dvě 32-bitové instrukce, a následně porovnám hodnotu v PC s mou uloženou hodnotou. Tyto hodnoty by se měly rovnat, neboť od uložení probíhá již druhá instrukce, první byla na přičtení k uložené hodnotě, druhá pak samotné porovnání. Registr PC by tedy měl být navýšen o 8, což je přesně hodnota, kterou jsem přičetl jeho původní hodnotě.

V případě neúspěchu některého z testů se vykonávání programu přesune na chybové návěští a funkce končí s návratovou hodnotou 1. Pokud vše proběhne bez problémů, funkce vrací 0.

V modulu *test_functions.c* je funkce *testRegisters* pouze obalena funkcí *runRegisterTests*, která zpracuje výstup z *testRegisters* a dle toho zapípá. V *main* části programu pak stačí volat jen tuto funkci a kód je tak mnohem čistší.

4.2 Paměť RAM

Paměť RAM se opět testuje assemblerovskou funkcí, konkrétně *testRam* v souboru *test_ram_asm.S*. Tato funkce očekává jeden parametr udávající adresu paměti, odkud má provádět test. Blok paměti, který testuje, je veliký 32B.

Funkce implementuje MarchX algoritmus následujícím způsobem:

1. Registru R1 nastaví hodnotu 0. Registr bude nadále sloužit jako čítač v průchodu 32B blokem paměti.
2. Ve smyčce *zeroLoop* načte aktuální hodnotu z paměti, uloží ji do zásobníku jako zálohu a místo ní zapíše 0. Zapsáním 0 do 4B bloku paměti načteného do registru logicky zapíše 0 do každého bytu. Poté hodnotu registru zapíše do paměti.
3. Cyklem *zeroTestLoop* opět prochází blokem paměti, načítá a ověřuje nuly, načež zapíše vzor 0x01010101. Opět vzhledem k načítání do registru drží 4B z paměti. Zapsáním tohoto vzoru budou v jednotlivých bytech hodnoty 0x01, tedy 1 dekadicky.
4. Smyčkou *oneTestLoop* znovu prochází blokem paměti, tentokrát však sestupně. Čte a ověřuje hodnoty 1 v jednotlivých bytech (opět přes vzor 0x01010101) a zapisuje nuly.
5. V poslední smyčce *backupRestoreLoop* načítá nuly a zároveň už rovnou obnovuje původní hodnoty ze zásobníku. Z tohoto důvodu je cyklus opět sestupný.

6. Pokud vše proběhlo v pořádku, funkce vrací hodnotu nula. V případě předčasného ukončení skokem na chybové návěští končí s návratovou hodnotou jedna.

V modulu *test_functions.c* ji podobně jako tomu bylo u registrů obaluje funkce *runRamTests*. Ta ve výchozí podobě volá funkci *testRam* opakovaně v cyklu, dokud se neotestuje celkem 16kB paměti, tedy rozmezí 0x20000000-0x00003fff. Tento region je však nastavitelný právě v počátečních a koncových podmínkách smyčky. Dále pak už jen zapípá dle toho, jestli byla či nebyla odhalena chyba.

4.3 Digitální vstup a výstup

Kompletní test pinů se nachází ve funkci *runPinTests* modulu *test_functions.c*. Funkce testuje piny 0-12 portů A i B mikrokontroléru. Poslední čtrnáctý pin jsem již v testu vynechal z toho důvodu, že na portu B je tento připojen k bzučáku, jenž využívám k signalizaci výsledků testů. Pro jeho testování by však šlo použít stejný princip, jen by se výstup musel převést například na diodu.

Před samotným testem je ještě potřeba funkcí *pinInit* piny zprovoznit. To se udělá tím, že se na PORTA i PORTB přivede hodinový signál. Dále se všem nastaví konfigurace GPIO pro jejich snazší používání. Na závěr se aktivuje pin bzučáku.

Nyní již konečně k samotnému testu. Jak jsem již popsal v teoretické části, správnou funkcionalitu ověřím tím, že výstup pinu přivedu na vstup jiného a porovnám. Fyzicky však jsou už piny napojeny k vlastním komponentám, takže nemůžu výstup jednoho přivést přímo na vstup druhého, avšak softwarově lze tento princip stále použít. Na PORTA piny přivedu logické jedničky a PORTB pinům předám hodnotu PORTA pinů. Následně porovnám hodnotu registru PDO portu B s hodnotou 0x1fff, neboli posloupností 13 jedniček, které by měly být přivedeny od PORTA pinů. PORTA piny zde tedy slouží de facto jako meziproměnná a musí být funkční, aby byl předpoklad splněn. V druhé polovině funkce provádím to samé, jen naopak, tedy z PORTB na PORTA. Funkce opět vyvolá jedno pípnutí, pokud test proběhne v pořádku nebo tři, jestli nikoliv.

5 Ověření funkcionality testů

Tato část práce je ryze teoretická a zabývá se tím, jak ověřit, že mé testy skutečně naleznou chybu. V obecnosti je princip jednoduchý, chybu vytvořím, test spustím a čekám selhání.

Prakticky lze však poměrně těžko vložit poruchu do registrů nebo paměti za běhu programu. Nicméně jak popisuje pan Denk ve své práci [1], lze změnit obsah registru/paměti v debug módu. Funkcionalitu testů registrů CPU a paměti RAM bych tedy prováděl tak, že bych si pozastavil vykonávání programu, změnil hodnotu registru/paměti, nechal program pokračovat a očekával chybový výstup.

Simulovat chybu u pinů by pak šlo zkratem sousedních pinů, rozpojením vodičů nebo uzemněním výstupního pinu. [1]

Další možností by bylo vystavit mikrokontrolér nepříznivým fyzikálním podmínkám (teplota, tlak...). V nich je MCU náchylnější na chyby, a proto tím lze dosáhnout přirozeného náhodného vzniku chyby.

Alternativou je i generování přerušení, která by měnila hodnoty testovaných komponent. Vzhledem k tomu, že tyto nastávají asynchronně vůči testům, dochází k injekci chyby, jež by měly testy odhalit.

6 Využití testů

Již jsem zmínil, že všechny testy mají za úkol odhalit chybu stuck-at, jež je výrobní vada integrovaných obvodů. Z toho důvodu by je šlo využít pro testování kvality výrobního procesu mikrokontrolérů.

Závěr

Vzhledem k teoretické povaze práce z již zmíněných důvodů bohužel nelze vyvozovat smysluplné závěry. Za běžných podmínek by se mohla diskutovat třeba rychlost a optimalizace testů nebo porovnání výsledků s jinými algoritmy. Alespoň tu však zmíním, že naopak právě teoretický charakter projektu mě donutil si o to více nastudovat danou problematiku a tedy alespoň v některých aspektech ho považuji za přínosnější než jeho praktický ekvivalent.

Autoevaluace

Zde rozeberu můj pohled na splnění zadání dle hodnotícího klíče.

E - přístup

Na projektu jsem začal včas, o FitKit3 jsem dokonce žádal ještě před zákazem osobních konzultací. Z důvodu nemožnosti experimentování s kitem jsem o to více času dal samostudiu problematiky a využil jsem i konzultace. Projekt jsem zvládl dokončit do konce listopadu. Proto považuji za splněno a dal bych si celý bod.

F - funkčnost

Program sice s největší pravděpodobností nebude funkční, nicméně je přeložitelný a myslím, že dané instrukce dávají smysl. Hodnocení této části se má údajně přesunou do D-Dokumentace, ve které jsem pak, ale i v komentářích, o to více popisoval, co je čím zamýšleno. Přestože si myslím, že v rámci možností jsem to udělal správně, svědomí mi nedá a z důvodu, že to pravděpodobně na první dobrou funkční nebude, jelikož některé konstrukce zřejmě budou špatně, bych si zde dal jen polovinu bodů, tedy 2.5b.

Q - kvalita

Uživatelskou přívětivost opět nešlo dobře ověřit, ale vzhledem k tomu, že zde není žádná uživatelova interakce potřeba, jen se to spustí a oznámí výsledek testů, tak bzučák mi přijde stejně dobrá indikace, jako cokoliv jiného (dioda, terminál), proto tu asi nemám problém si dát celý bod. Za přehlednost kódu a dekompozici problému bych si taktéž dal po bodu. Celé to mám rozdělené do modulů, kód je přehledný a komentovaný.

D - dokumentace

Na začátku mám úvod do problému ve formě zadání s popsanou úpravou kvůli pandemickým opatřením a seznámením s testovanými komponentami. Rozsah této části je necelá stránka A4, což splňuje zadání. Dále mám na 2 a půl stranách popsaný návrh a implementaci testů. To zadání přesahuje, avšak přišlo mi fér tuto část udělat rozsáhlejší z důvodu již zmíněných problémům s funkčností. Ty mají mimo jiné i za následek nemožnost ověření funkcionality a absenci informací ke zhodnocení výsledků projektu. Dohromady si však myslím, že rozsah i obsah byl splněn. Přesto však počítám s půl až bodem dolů za někde nějakou špatnou formu něčeho.

P - prezentace

Z mého pohledu prezentace proběhla kvalitně a byl jsem za ni i na místě pochválen, proto si myslím, že si bod zasloužím.

Výsledek

Skutečnost, že se počítá s nefunkčním řešením, asi ruší význam koeficientů K_1 a K_2. Proto bych se ohodnotil jako E + F + Q + D + P, tedy 1 + 2.5 + 3 + 3.5 + 1 = 11b (/14). Záleží však na způsobu hodnocení té "teoretické funkčnosti".

Reference

[1] F. Denk: *Samočinné testování mikrokontrolerů.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Josef Strnadel, Ph.D.

[2] M. Priyadarshi: *Memory Testing - An Insight into Algorithms and Self Repair Mechanism* [Online; navštíveno 6.11.2020].
https://www.design-reuse.com/articles/45915/memory-testing-self-repair-mechanism.html

[3] P. Krenek: *IEC 60730B Safety Routines for Kinetis MCUs* [Online; navštíveno 6.11.2020].
https://www.nxp.com/docs/en/application-note/AN4873.pdf

[4] Wikipedia contributors: *"Stuck-at fault," Wikipedia, The Free Encyclopedia* [Online; navštíveno 6.11.2020].
https://en.wikipedia.org/wiki/Stuck-at_fault

[5] Azeria Labs: *Introduction to ARM assembly basics* [Online; navštíveno 6.11.2020].
https://azeria-labs.com/writing-arm-assembly-part-1/

[6] Arm Limited: *Calling assembly functions from C and C++* [Online; navštíveno 6.11.2020].
https://developer.arm.com/documentation/100748/0615/Using-Assembly-and-Intrinsics-in-C-or-C---Code/Calling-assembly-functions-from-C-and-C--