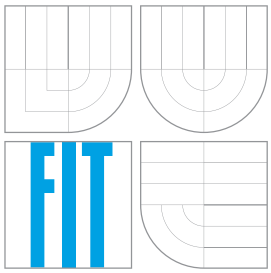
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

SIMULACE ARCHITEKTURY MIKROPROCESORU 8051

SIMULATION OF THE 8051 MICROPROCESSOR ARCHITECTURE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — PETR ŠIMON
AUTHOR

VEDOUCÍ PRÁCE — prof. Ing. TOMÁŠ HRUŠKA, CSc.
SUPERVISOR

BRNO 2010

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav informačních systémů Akademický rok 2009/2010

Zadání bakalářské práce

Řešitel: **Šimon Petr**
Obor: Informační technologie
Téma: **Simulace architektury mikroprocesoru 8051**
Simulation of the 8051 Microprocessor Architecture
Kategorie: Návrh číslicových systémů

Pokyny:

1. Seznamte se s jazykem a návrhovým prostředím pro vývoj mikroprocesorů projektu Lissom.
2. Prostudujte architekturu mikroprocesoru 8051.
3. Vytvořte model mikroprocesoru 8051 v jazyce ISAC.
4. Zvolte sadu algoritmů pro otestování funkcionality modelu. Algoritmy implementuje ve specifickém assemblerovském jazyce mikroprocesoru 8051.
5. Otestujte funkčnost navrženého modelu a programů za pomocí vývojového prostředí projektu Lissom.
6. Ověřte míru odchylky modelu od skutečného mikroprocesoru.

Literatura:

- Hoffmann A. Meyr H., Leupers R.: Architecture Exploration for Embedded Processors with LISA, Kluwer Academic Publishers, 2002, 1-4020-7338-0
- Masařík K.: Systém pro souběžný návrh technického a programového vybavení počítačů, Brno, CZ, FIT VUT, 2008, ISBN 978-80-214-3863-7
- Specifikace a popis instrukční sady mikroprocesoru 8051 (zdroj internet)
- Interní dokumentace projektu Lissom, dle pokynů vedoucího

Při obhajobě semestrální části projektu je požadováno:

- První dva body zadání.

Podrobné závazné pokyny pro vypracování bakalářské práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva bakalářské práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap (20 až 30% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Hruška Tomáš, prof. Ing., CSc.**, UIFS FIT VUT
Konzultant: Přikryl Zdeněk, Ing., FIT VUT
Datum zadání: 1. listopadu 2009
Datum odevzdání: 19. května 2010

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav informačních systémů
612 66 Brno, Božetěchova 2

L.S.

doc. Dr. Ing. Dušan Kolář
vedoucí ústavu

Abstrakt

V dnešní době je více jak 90% procesorů používáno ve vestavěných systémech. Návrh procesorů pro vestavěná zařízení se stává čím dál složitější, a proto je nutné tuto práci co nejvíce automatizovat.
Tato bakalářská práce se věnuje návrhu mikrokontroléru 8051. Návrh je proveden podle dostupné dokumentace a k popisu procesoru je použit jazyk ISAC. Výsledný model je ověřen řadou simulací, které jsou na konci práce analyzovány.

Abstract

More than 90% of processors are used in embedded systems today. Processor design for embedded systems is becoming complicated, so it should be automate as much as possible. This bachelor thesis deals with design of the microcontroller 8051. Design is created according to available documentation and ISAC language is used for model description. Results of model simulations are analyzed at the end of this thesis.

Klíčová slova

jazyk pro popis architektury, ADL, mikrokontrolér, 8051, procesor, CISC, RISC, Harvardská architektura, Von Neumannova architektura, modelování, simulace, ISAC, LISA

Keywords

architecture description language, ADL, microcontroller, 8051, processor, CISC, RISC, Harvard architecture, Von Neumann architecture, modeling, simulation, ISAC, LISA

Citace

Petr Šimon: Simulace architektury mikroprocesoru 8051, bakalářská práce, Brno, FIT VUT v Brně, 2010

Simulace architektury mikroprocesoru 8051

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc.
Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Petr Šimon
17. května 2010

Poděkování

Děkuji vedoucímu mé práce prof. Ing. Tomášovi Hruškovi, CSc., Ing. Karlu Masaříkovi, Ph.D., a Ing. Zdeňkovi Přikrylovi za odborné rady, ochotnou pomoc a čas, který mi věnovali. Dále děkuji celému vývojovému týmu Lissom za kvalitní práci, kterou na projektu odvádějí.

Obsah

Kapitola 1

Úvod

Tato práce se zabývá návrhem a simulací mikrokontroléru 8051. V dnešní době se více než 90% procesorů [5] používá ve vestavěných systémech, a proto je vhodné zabývat se způsoby, jimiž je možno procesory navrhovat a popisovat. Tomu se věnuje kapitola 2.

Cílem této kapitoly je vysvětlit, proč je pro návrh procesorů výhodné použít specializovaný jazyk pro popis architektury. Jedním z těchto jazyků je i jazyk ISAC (kap. 3), který jsem pro popis architektury mikrokontroléru 8051 zvolil.

V následující kapitole jsem procesory rozdělil do kategorií podle instrukční sady a podle paměťového prostoru, protože se na tyto pojmy často odkazuji.

V kapitole 5 se zabývám popisem mikrokontroléru 8051, který je jádrem této práce. Pak již následuje (kap. 6) popis modelu, který jsem v jazyce ISAC vytvořil. Rozebírám zde problémy, na něž jsem při implementaci narazil, a také popisuji odchylky modelu od skutečného mikrokontroléru.

Nakonec jsem s vytvořeným modelem provedl řadu simulací (kap. 7) na testovacích programech a výsledky simulace analyzoval a porovnal s jinými dostupnými simulátory. Na stejných vzorcích programů jsem také provedl simulaci s aktivním profilerem a získaná data analyzoval.

Kapitola 2

Teorie

Informace v této kapitole vycházejí z [5].

V dnešní době se více než 90% procesorů používá ve vestavěných systémech. Vestavěným systémem se rozumí jednoúčelový systém, v němž je řídící počítač zcela zabudován do ovládaného zařízení.

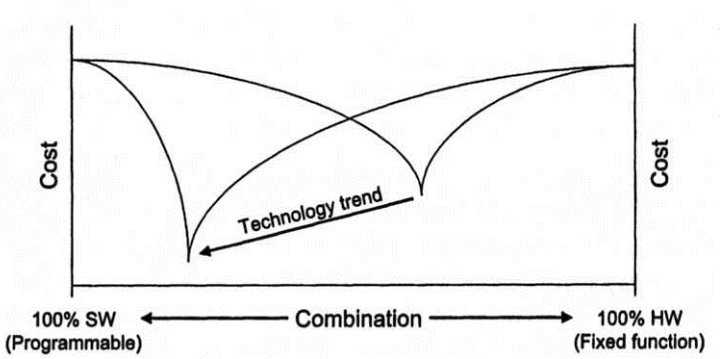
Toto číslo není příliš překvapivé, jelikož v dnešní domácnosti je běžně nejen počítač, ale i celá řada elektrických spotřebičů (televize, video, lednička, pračka...), které jsou také vybaveny a řízeny procesorem.

Od těchto zařízení požadujeme nízkou cenu, nízkou spotřebu, malé rozměry, ale také můžeme požadovat reagování v reálném čase, vysokou spolehlivost (např. v automobilech) apod.

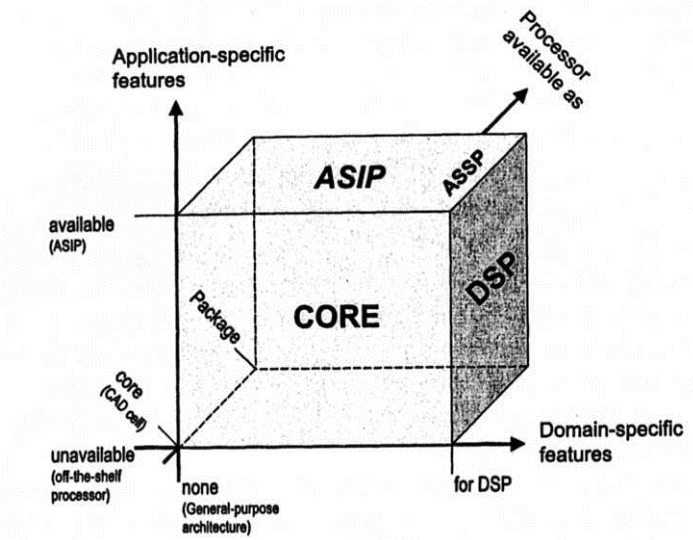
Kritériem hodnocení vestavěných systémů může být:

- Úspěch na trhu - může být dosažen inovativním řešením, kvalitou řešení (dobrý poměr cena/výkon)
- Cena zařízení - je dána cenou návrhu a cenou výroby
- Kvalita návrhu - závisí na délce návrhu. Vzhledem ke stále rostoucí složitosti vestavěných systémů se kvalitní návrh stává stále důležitějším.

Obecně platí, že funkce implementovaná pomocí hardwaru je rychlejší než pomocí softwaru. Návrh hardwaru je však časově i finančně mnohem nákladnější.



Obrázek 2.1: Kombinace hardwaru a softwaru pro dosažení nižších nákladů (cenových, časových apod.). Převzato z [5]



Obrázek 2.2: Graf typů procesorů. Převzato z [5]

V dnešní době se stává čas potřebný pro vývoj důležitější než výkon cílového zařízení (viz obr. 2.1) a tento poměr se stále zvyšuje. Proto se stále více funkcí, jež byly implementovány hardwarově, implementuje pomocí softwaru. Jsou proto vyvíjeny procesory „šité na míru“ konkrétní aplikaci, díky čemuž se dosáhne mnohem vyššího výkonu, než jakého by se dosáhlo použitím univerzálního procesoru.

Příkladem tohoto řešení mohou být např. DSP procesory (Digital Signal Processor), jež jsou používány pro zpracování multimediálních dat, a hlavním požadavkem na ně bývá zpracování velkého množství dat v reálném čase.

Vysoká integrace a miniaturizace hardwaru vede od návrhu více vzájemně propojených, hardwarových komponent (například paměť a procesor) k systému na jednom čipu (SoC). Spojením těchto komponent lze značně ušetřit čas potřebný pro návrh systému, protože získáme platformu, kterou je pak možné programovat ve vyšších programovacích jazycích. Tyto systémy se pak často využívají pro implementaci procesorů ASIP.

2.1 Kategorie procesorů

Procesory můžeme rozdělit do dvou kategorií: univerzální procesory a aplikačně specifické procesory [5, 10].

Univerzální procesory nedisponují příliš vysokým výkonem, na druhou stranu jsou však velmi flexibilní a lze je využít téměř ve všech aplikacích. Musí poskytovat širokou škálu funkcí, protože jejich cílové využití je neznámé. Většina funkcí cílové aplikace se pak implementuje softwarově. Tyto procesory jsou mimo jiné využívány i v PC.

Aplikačně specifické procesory naopak přinášejí velmi vysoký výkon, ale omezenou flexibilitu. Úpravou procesoru přímo pro cílovou aplikaci ale můžeme v konečném důsledku snížit jeho cenu, což je při velkovýrobě vestavěných systémů důležitý parametr. Lze toho dosáhnout např. vypuštěním nepotřebných komponent (násobičky, děličky apod.)

Obrázek 2.2 podle [5] ukazuje rozdělení procesorů podle tří hlavních kritérií. Význam jednotlivých dimenzí je následující:

- Forma, ve které je procesor dodán - Procesor může být dodán kompletně zrekonstruovaný (samostatný čip), či jako integrální součást (označovaný jako core procesor)
- Doména použití - Doménou použití může být digitální zpracování signálů (DSP) nebo řídící aplikace. DSP mohou obsahovat speciální výbavu jako vícenásobnou ALU, specializované sady registrů apod.
- Aplikačně specifická doména - Rozlišujeme procesory s pevnou architekturou a procesory s konfigurovatelnou architekturou. Procesory s pevnou architekturou mívají stavební bloky na čipu extrémně efektivně rozloženy, například podle algoritmu, který bude procesor provádět. Mezi procesory s konfigurovatelnou architekturou patří procesory s aplikačně specifickou instrukční sadou (ASIP).

2.2 Procesory s aplikačně specifikovanou instrukční sadou (ASIP)

ASIP (anglicky Application-Specific Instruction-set Processor) procesory jsou vyvíjeny pro aplikace s odlišnými a často i konfliktními požadavky na jádro. Můžeme například požadovat nízkou energetickou spotřebu, vysoký výpočetní výkon, garantovaný čas odpovědi atd.

K dosažení těchto požadavků můžeme procesor upravit přímo pro cílovou aplikaci. Je možné například upravit velikost kódu, počet cyklů potřebných k provedení jedné instrukce nebo změnit pracovní frekvenci procesoru.

Přidání nových funkcí do ASIP lze realizovat buď přidáním nových instrukcí, nebo formou speciálních funkčních jednotek.

2.2.1 Tradiční návrh ASIP procesorů

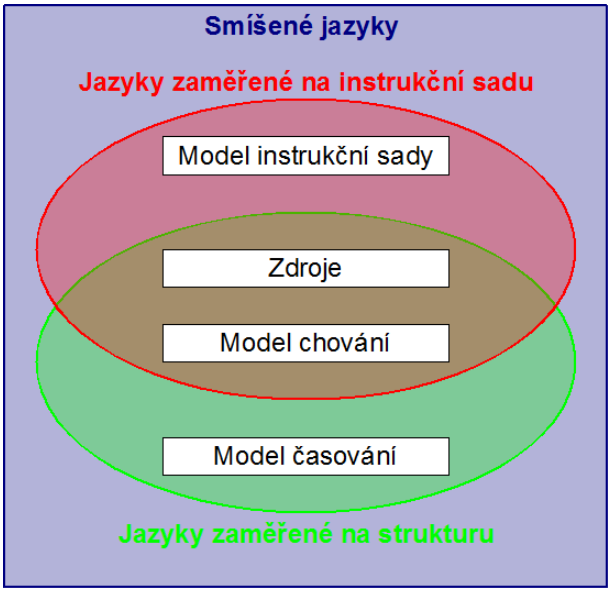
Tato metoda se bohužel i v dnešní době velice často využívá. Procesory ASIP a nezbytné softwarové nástroje jsou vyvíjeny ručně s velmi nízkou úrovní automatizace. Vývoj pak zajišťují velmi zkušení vývojáři.

Průběh vývoje procesoru je následující:

- **Prozkoumání architektury** - Aplikace je mapována do jednoúčelové architektury procesoru. Je nutné zjistit kritické části aplikace, které vyžadují podporu hardwaru v podobě aplikačně specifických instrukcí. Následně se definuje instrukční sada, která je pak implementována pomocí mikroarchitektury.
- **Implementace architektury** - V této fázi se navržený procesor transformuje do syntetizovatelného modelu pomocí jazyka pro popis hardwaru (HDL), jako je např. VHDL nebo Verilog.
- **Návrh softwarové aplikace** - Je nutné vytvořit nástroje pro pohodlné programování architektury. Mezi ně patří překladače assembleru či vyššího programovacího jazyka, disassembler, simulátory apod.

  Návrhář procesoru potřebuje například simulátor založený na cyklech pro HW/SW analýzu, který dává detailní informace o procesoru, ale je velice pomalý. Návrhář aplikace naopak potřebuje velice rychlý simulátor, u něhož nepožaduje tak detailní simulaci.

  Všechny tyto nástroje se vytváří samostatně, což je časově velmi náročné.



Obrázek 2.3: Kategorie ASIP modelů

- **Systémová integrace a verifikace** - Je nutné vytvořit rozhraní pro integraci softwarového simulátoru pro zvolenou architekturu do různých simulačních prostředí.

Každý z těchto kroků vyžaduje použití speciálních nástrojů a často každou z těchto částí provádí oddělený tým vývojářů. V důsledku toho pak vývojáři nemají dostatek času ani potřebné nástroje k nalezení nejlepšího řešení pro vybranou aplikaci.

Proto se jako vhodný krok pro návrh procesorů jeví využití speciálního jazyka pro popis architektury.

2.2.2 Jazyky pro popis procesoru

Jazyky pro popis hardwaru (HDL) jsou vhodné pro modelování a simulaci hardwaru. Simulace je pomalá a neobsahuje některé informace, např. syntaxi assembleru. Navíc se vzrůstající složitostí hardwaru se návrh procesoru pomocí těchto jazyků stává čím dál více obtížným.

Tyto nevýhody překonávají jazyky pro popis architektury (ADL), které se vyznačují vysokou úrovní abstrakce, což umožňuje rychlejší aplikaci změn do modelu. Nevýhodou však je, že částečně ztrácejí spojitost mezi modelem a fyzickými parametry procesoru, jako je frekvence hodin, plocha na čipu apod. [11]

Obrázek 2.3) ukazuje rozdělení těchto jazyků do tří kategorií:

Jazyky zaměřené na instrukční sadu

Tyto jazyky poskytují programátorovi pohled na architekturu přes popis instrukční sady, pomocí které je popsáno její chování. Příkladem jsou jazyky nML, ISDL, Valen-C a CSDL.

Většinou se používají s cílem vyvinout přenositelný překladač vyšších programovacích jazyků. Obsahují informace o instrukční sadě, sekvenci instrukcí a jejich latencí. Neobsahují další informace o mikroarchitektuře, pouze sémantiku instrukcí.

Jazyky zaměřené na strukturu

Zaměřují se na strukturu komponent a jejich propojení. Výhodou je, že stejný popis lze použít pro syntézu do hardwaru i pro generování softwarových nástrojů. Příklady jazyků: MIMOLA, AIDL, COACH.

Nevýhodou těchto nástrojů je pomalost vygenerovaných simulátorů, protože model obsahuje velké množství strukturních detailů, které nejsou důležité ani pro simulaci založenou na cyklech.

Jazyky pro popis instrukční sady a struktury současně (smíšené jazyky)

Přemosťují mezeru mezi předchozími jazyky. Odstraňují nedostatky předchozích jazyků, popis instrukční sady je zjemněn i pro popis procesorů se zřetězenou linkou, zároveň ale umožňují i rychlou simulaci tím, že mohou abstrahovat přílišné detaily. Představiteli těchto jazyků jsou: FlexWare, MDES, PEAS, RADL, EXPRESSION, **LISA** (Language for Instruction-Set Architecture) a **ISAC** (Instruction Set Architecture C).

Kapitola 3

Jazyk ISAC

Informace v této kapitole jsou čerpány z [9].

Jazyk ISAC (Instruction Set Architecture C) je jazyk pro popis architektury (ADL), který spadá do kategorie smíšených jazyků. Umožňuje tedy popis instrukční sady a struktury současně.

Vyvíjí se na Fakultě informačních technologií VUT, a proto jsem se ho pro vytvoření modelu architektury mikrokontroléru 8051 rozhodl využít. Během práce na modelu jsem tak pomohl odhalit mnoho chyb a nedostatků v nástrojích ISAC.

Jazyk ISAC je odvozen od jazyku LISA a používá některá podobná klíčová slova.

Umožňuje popsat časový model (sekce ACTIVATION a STRUCTURE), instrukční model (sekce ASSEMBLER, CODING a CODINGROOT), model chování (sekce BEHAVIOR a EXPRESSION) a strukturální model (sekce RESOURCE a částečně i BEHAVIOR).

3.1 Popis modelu

Popis modelu v jazyce ISAC se skládá ze dvou částí. První část obsahuje deklarace zdrojů, z nichž se model skládá (paměti, registry, pipeline…), druhá část je pak tvořena popisem operací a skupin, které slouží pro popis instrukční sady a mikroarchitektury procesoru.

3.1.1 Zdroje

Sekce zdrojů začíná klíčovým zdrojem RESOURCE. Jsou zde definovány všechny zdroje modelu jako paměť, registry, pipeline apod. Taktéž je možné vytvořit mapování paměti do adresového prostoru a v aktuální verzi lze do adresového prostoru mapovat i registry.

3.1.2 Operace a skupiny

Operace je základním stavebním prvkem při popisu modelu. Může obsahovat některou z následujících sekcí. Některé sekce vyžadují přítomnost jiné a naopak, některé sekce vyžadují nepřítomnost jiné. Více informací lze získat v [9]. Popíšu zde nejdůležitější sekce vzhledem k této práci:

- **CODING** - Popisuje binární obraz instrukce.
- **ASSEMBLER** - Popisuje textovou reprezentaci instrukcí na assemblerovské úrovni. Společně se sekcí CODING tvoří pár a poskytují informace pro konstrukci překladače assembleru, disassembleru a jazyka C.

- **BEHAVIOR** - Obsahuje kód zapsaný v podmnožině jazyka ANSI C, který popisuje chování (sémantiku) operací. Během simulace je kód v této sekci prováděn a jsou měněny hodnoty zdrojů, které jsou definovány v sekci RESOURCE.
- **EXPRESSION** - V této sekci se definuje výraz, který vrací buď konstantu, nebo jakýkoliv zdrojový objekt.
- **ACTIVATION** - Zde je definováno časování jiných operací vzhledem k popisované operaci. Konstrukce tak umožňuje popisovat modely založené na cyklech. Zpoždění se zapisuje pomocí výrazu `%N;`, kde N je číslo vyjadřující počet taktů, o které se má aktivace zpozdit. Pokud je v sekci těchto výrazů více, celková doba zpoždění se sčítá. Lze využít i podmíněné aktivace pomocí řídících konstrukcí IF nebo SWITCH.
- **CODINGROOT** - Slouží pro popis instrukční sady. Popisuje, jak sada vypadá z pohledu jazyka symbolických instrukcí/strojového kódu.
- **STRUCTURE** - Slouží pro popis mikroarchitektury procesoru z pohledu použitých dekodérů.

Speciální funkci zde plní operace *main*. Podobně, jako je tomu v jazyce C, plní i zde funkci vstupního bodu do programu. V některých věcech se však liší. V operaci *main* se aktivují operace (např. načtení instrukce, dekódování…), které mohou následně aktivovat další operace. Po provedení všech naplánovaných operací se znovu aktivuje operace *main* a ta opět začne plánovat další operace uvedené v sekci ACTIVATION.

Výjimku tvoří sekce BEHAVIOR v operaci *main*, která se provádí každý hodinový takt nezávisle na tom, kdy je operace *main* aktivována. Díky tomu můžeme v modelu využívat hodinový signál procesoru. Ten je výhodný v případě, že budeme chtít modelovat např. periferie typu čítač/časovač.

Operace je možné slučovat do skupin (klíčové slovo GROUP). Skupina sdružuje pod jedním jménem operace podobného významu, např. registry, ALU operace apod. Ty pak mohou být využity ve stejném kontextu a skupina potom popisuje společným jménem různé varianty analýzy.

Skupina může obsahovat operace i skupiny. Operace se může ve skupině vyskytovat pouze jednou.

Kapitola 4

Architektury procesorů

Informace v této kapitole vycházejí z [7, 6].

4.1 Rozdělení podle paměťového prostoru

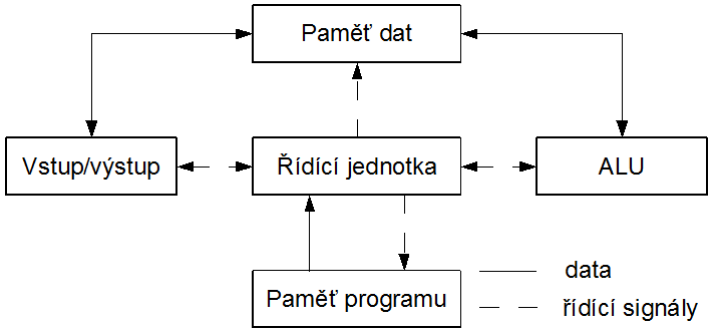
Ve své práci jsem se rozhodl zavést dělení procesorů podle paměťového prostoru na procesory s Harvardskou architekturou a Von Neumannovou architekturou.

4.1.1 Harvardská architektura

V této architektuře je fyzicky oddělena paměť programu a paměť dat. Paměti můžou být naprosto odlišné (například program je uložen v paměti EEPROM, data jsou uložena v dynamické paměti DRAM). Mohou mít odlišnou šířku adresové i datové sběrnice, různou rychlost apod.

Výhodou této architektury je, že je možné současně pracovat s pamětí programu i dat, což výrazným způsobem zrychluje provádění programu a umožňuje efektivní využití proudového zpracování programu (pipelining) - lze např. současně načítat novou instrukci a ukládat výsledek aktuálně prováděné instrukce.

Základní schéma počítače s Harvardskou architekturou je na obrázku 4.1.



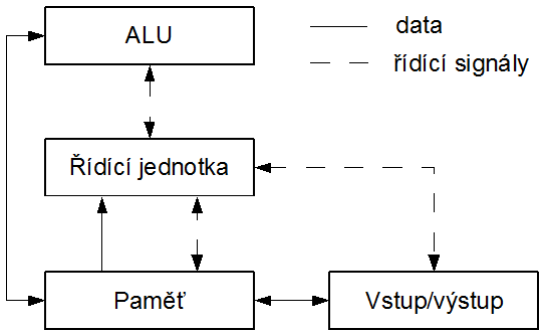
Obrázek 4.1: Základní schéma počítače s Harvardskou architekturou

4.1.2 Von Neumannova architektura

V této architektuře je pro program i pro data použita společná paměť. Výhoda této architektury je její univerzálnost. Bohužel však znesnadňuje proudové provádění programu

(v jedné chvíli nelze např. současně načítat instrukci a zároveň ukládat výsledek po aritmetické operaci do paměti).

Základní schéma počítače podle von Neumannovy architektury je na obrázku 4.2.



Obrázek 4.2: Základní schéma počítače podle von Neumanna

4.2 Rozdělení podle instrukční sady

Procesory můžeme podle instrukční sady rozdělit na procesory s komplexní instrukční sadou (CISC) a procesory s redukovanou instrukční sadou (RISC).

4.2.1 CISC - Complex Instruction Set Computing

Tyto procesory se vyznačují velkým počtem složitých instrukcí. Instrukce mohou mít různou délku, různý počet parametrů a jejich provádění většinou trvá různě dlouhou dobu. Pro přístup do paměti RAM se nevyužívají speciální instrukce LOAD (uložení hodnoty z paměti do registru) a STORE (uložení hodnoty z registru do paměti), ale existují různé varianty konkrétní instrukce podle toho, zda se pracuje s registry nebo s pamětí.

Instrukce jsou prováděny pomocí posloupnosti mikroinstrukcí, jejichž kód je uložen v paměti ROM. Tzn. jedna složitá instrukce procesoru se vykoná provedením posloupnosti několika jednoduchých mikroinstrukcí. V procesoru je většinou malý počet univerzálních registrů (např. 8), takže instrukce ke své činnosti často využívají paměť RAM.

Výhodou tohoto řešení je, že přechod na vyšší verzi mikroprocesoru lze provést doplněním nových instrukcí a vytvořením nových mikroprogramů reprezentujících činnost, kterou mají nové instrukce realizovat. Není proto nutné výrazným způsobem měnit vnitřní architekturu.

4.2.2 RISC - Reduced Instruction Set Computing

V těchto architekturách se klade důraz na jednoduché instrukce stejné délky, jednotný formát a stejnou dobu provádění jedné instrukce, což značným způsobem usnadňuje využívání proudového zpracování programu (pipelining).

Pro přístup do pomalé paměti RAM se využívají instrukce typu LOAD a STORE. Pouze tyto instrukce mohou pracovat s pamětí. Všechny ostatní instrukce využívají ke své činnosti vnitřní registry procesoru. Proto je v procesoru obvykle k dispozici velký počet registrů (např. 32 až 2048, i více), díky čemuž lze značným způsobem omezit přístup do paměti.

Dekódování a provádění instrukcí se řeší hardwarově (například sekvenčním automatem), což je výrazně rychlejší než provádění mikroprogramu. Nevýhodou je, že přidáním nových instrukcí je nutné změnit vnitřní architekturu procesoru (na rozdíl od procesoru CISC nelze pouze přidat nový mikroprogram).

Obecně se dá říci, že to, co se v procesorech CISC provádí mikroprogramem, se v procesorech RISC provádí jako posloupnost jednoduchých instrukcí.

Kapitola 5

Mikrokontrolér 8051

Informace v této kapitole vycházejí z [3, 4, 12, 2]

Mikrokontrolér 8051 vyvinula firma Intel v roce 1980. Byl určen pro použití v oblastech vestavěných systémů (embedded systems). V současné době klony tohoto mikrokontroléru vyrábějí mnohé firmy, které nabízejí různé varianty s různými parametry a vlastnostmi.

Základní vlastnosti mikrokontroléru:

- harvardská architektura
- instrukce typu CISC
- 8-bitová datová sběrnice a ALU (Aritmeticko-logická jednotka)
- 16-bitová adresová sběrnice (umožňuje adresovat 64kB)
- 32 registrů rozdělených do čtyř bank
- bitově adresovatelná paměť a bitově orientované instrukce
- 2 čítače/časovače
- sériová linka
- přerušovací systém s dvouúrovňovou prioritou

5.1 Organizace paměti

Mikrokontrolér 8051 využívá Harvardské uspořádání paměti, takže adresové prostory programu a dat jsou odděleny.

5.1.1 Paměť dat

Paměť dat může být interní a externí. Interní paměť je rozdělena následujícím způsobem:

- 00H - 07H - Banka registrů 0
- 08H - 0FH - Banka registrů 1
- 10H - 17H - Banka registrů 2
- 18H - 1FH - Banka registrů 3

- 20H - 2FH - bitově adresovatelný prostor
- 30H - 7FH - bajtově adresovatelný prostor k univerzálnímu použití
- 80H - FFH - oblast speciálních funkčních registrů (SFR)

V bitově adresovatelném prostoru je možné pomocí bitové adresace adresovat každý ze 128 bitů zvlášť. Instrukční sada obsahuje mnoho instrukcí pro práci s bitovou pamětí. Lze však adresovat i jednotlivé bajty pomocí normálních instrukcí.

V oblasti SFR se nacházejí speciální registry, například registry čítačů/časovačů, sériové linky, vstupně/výstupních portů, ale jsou zde také registry typu Akumulátor, stavové slovo (PSW) apod. Některé z těchto registrů lze adresovat i bitově.

Registr PSW - Program Status Word

Registr PSW uchovává stavové slovo (příznaky) po provedení aritmeticko-logických operací. Jedná se o bitově adresovatelný 8-bitový registr umístěný v oblasti SFR.

- **C (Carry)** - Příznak přenosu. Je nastaven, dojde-li při některé aritmetické operaci k přenosu ze 7. do 8. bitu (počítáno od 1).
- **AC (Auxiliary Carry)** - příznak pomocného přenosu ze 4. do 5. bitu. Používá se při operacích s čísly v BCD kódu.
- **F0** - Uživatelský příznak k volnému využití.
- **RS1, RS0** - Řídící bity pro výběr aktuální banky registrů.
- **OV (Overflow)** - Příznak přetečení. Je nastaven, dojde-li k přetečení ze 7. do 8. bitu a indikuje přetečení při výpočtu s čísly se znaménky.
- **P (Parity)** - Příznak parity. Je svázán se střadačem a doplňuje jeho obsah na sudou paritu.

5.1.2 Paměť programu

Paměť programu může být interní, či externí. Její velikost závisí na konkrétním typu mikrokontroléru. Z paměti programu je také možné číst data pomocí speciálních instrukcí.

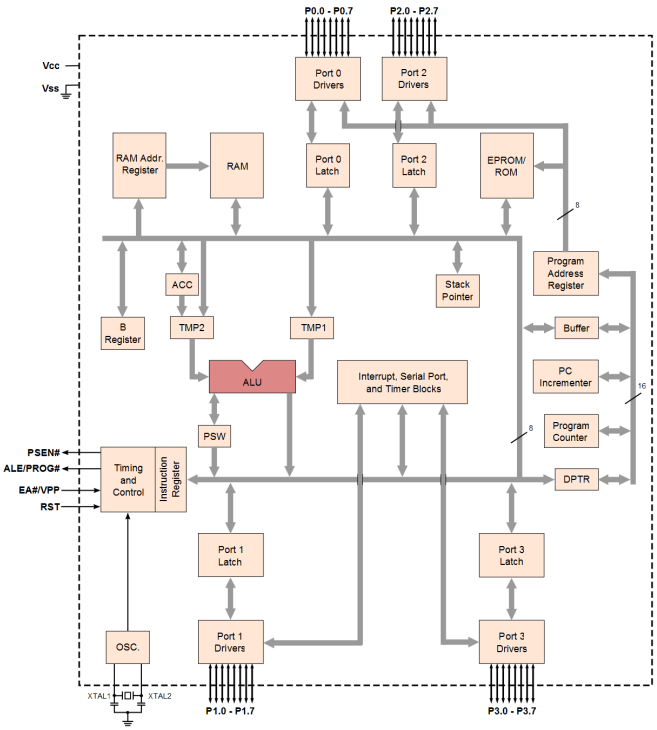
5.2 Architektura

Vnitřní architektura mikrokontroléru je znázorněna na obrázku 5.1

Paměť dat je adresována pouze pomocí registru `RAM_ADDR_REGISTER` (dále označován jako `AR`), paměť programu je adresována registrem `PROGRAM_ADDR_REGISTER` (dále označován jako `PC`). Tento registr adresuje jak interní paměť programu, tak i externí paměť připojenou na porty P0 a P2. Načtená instrukce je uložena v registru `INSTRUCTION_REGISTER` (dále jen `IR`), kde je řadičem dekódována a následně provedena.

Instrukce, které pracují s pamětí programu (instrukce MOVC), využívají k adresaci 16-bitový registr DPTR. Vzhledem k tomu, že vnitřní datová sběrnice je pouze osmibitová, je rozdělen na dva 8-bitové registry DPH a DPL, které se adresují zvlášť.

Na vstupu aritmeticko-logické jednotky (ALU) jsou dva 8-bitové registry TMP1 a TMP2. Vstup registru TMP1 je připojen na sběrnici. Na vstupu registru TMP2 je připojen registr



Obrázek 5.1: Architektura 8051 [14]

ACC nebo sběrnice. Díky tomu je možné současně v jednom taktu na vstup ALU přivést obsah registru ACC a obsah datové sběrnice. Výstup příznaků z ALU je přímo připojen na registr PSW (Program Status Word).

5.3 Instrukce

Šířka instrukcí je 1 až 3 bajty a doba provádění instrukcí je 1 až 4 strojové cykly (zřejmé prvky architektur CISC). První bajt instrukce je vždy operační kód a lze podle něj vždy jednoznačně určit, o jakou instrukci se jedná. Druhý a třetí bajt pak může být adresa nebo přímá hodnota.

Součástí operačního kódu může být i adresa registru. K tomu jsou využity poslední tři bity prvního bajtu. Příklad takovéto instrukce:

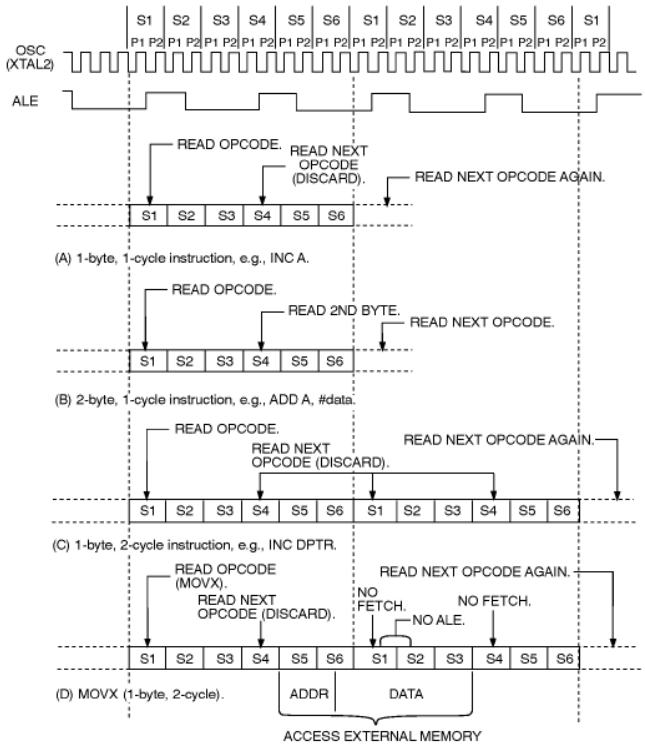
```
MOV Rn,#data
11111RRR   YYYYYYYY
```

Pokud bychom například chtěli do registru R1 uložit hodnotu 20, vypadal by binární kód takto:

```
11111001   00010100
```

V případě instrukcí s 11-bitovou adresou jsou zase nejvyšší tři bity adresy (10-8) uloženy na prvních třech bitech operačního kódu, zbytek adresy (bity 7-0) je uložen v následujícím bajtu:

```
ACALL addr11
AAA10001   AAAAAAAA
```



Obrázek 5.2: Časování řadiče při provádění instrukce. Převzato z [2]

5.4 Časování řídící jednotky

Řídící jednotka generuje synchronizační signály pro časování vnitřních činností mikrokontroléru při provádění instrukcí. Generuje také signály pro spolupráci s externími obvody.

Základní časovou jednotkou je **instrukční cyklus**. Během instrukčního cyklu se provede jedna instrukce. Instrukční cyklus může trvat 1 až 4 **strojové cykly**. Tento strojový cyklus je rozdělen do šesti **stavů** označených S1 až S6. Každý stav je rozdělen na dvě **fáze** P1 a P2. Vykonání jedné fáze trvá jeden **hodinový cyklus** (cyklus oscilátoru). Provedení jednoho strojového cyklu tedy trvá 12 hodinových taktů, které jsou rozděleny do šesti stavů, přičemž každý stav je rozdělen na dvě fáze.

Podle [3] se aritmetické a logické operace provádějí během fáze 1 (P1) a vnitřní přesuny mezi registry se provádějí ve fázi 2 (P2).

Průběh provádění instrukce (viz obr. 5.2) je následující: ve stavu S1P2 dojde k načtení prvního bajtu instrukce (operačního kódu) a uložení do registru instrukcí (`IR`). V následujícím taktu (S2P1) se tato instrukce dekóduje. V dalších taktech se již instrukce provádí. Pokud je délka instrukce větší než 1 bajt (2 nebo 3 bajty), dojde ve stavu S4P2 k načtení dalšího bajtu. V případě tříbajtové instrukce pak dochází k načtení třetího bajtu v následujícím strojovém cyklu ve stavu S1P2.

Většina instrukcí se provádí během jednoho či dvou strojových cyklů. Výjimku tvoří instrukce DIV (dělení) a MUL (násobení). Tyto instrukce vyžadují čtyři strojové cykly.

5.5 Periferie mikrokontroléru 8051

Mikrokontrolér 8051 je vybaven dvěma samostatnými čítači/časovači a sériovou linkou. Pro tyto periferie také poskytuje systém přerušení s dvouúrovňovou prioritou [4]

5.5.1 Čítače/časovače

Mikrokontrolér 8051 má dva 16-bitové čítače/časovače, které mohou pracovat jako interní časovače nebo jako externí čítače. Přístup k nim se provádí pomocí registrů mapovaných v oblasti SFR.

Čítač/časovač je registr, který (podle svého nastavení) počítá časové události od externího zdroje signálu, který se realizuje vstupem Ti (i=0..1), nebo od vnitřního hodinového signálu procesoru. Pokud je tímto signálem vnitřní hodinový signál procesoru, označujeme tento registr jako časovač. Pokud je zdrojem události externí signál, označujeme ho jako čítač.

Pokud obvod pracuje v režimu časovače, je jeho obsah inkrementován v každém strojovém cyklu. Jak bylo vysvětleno v kapitole 5.4, jeden strojový cyklus trvá 12 period hodinového signálu. Kmitočet oscilátoru se tedy dělí v poměru 1:12.

V režimu čítače vnějších událostí se obsah čítače inkrementuje sestupnou hranou impulzu na vstupu Ti. Signál se vzorkuje ve stavu S5P2 každého strojového cyklu. Pokud je v jednom stavu vstup v log. 1 a v následujícím stavu je signál v log. 0, inkrementuje se obsah čítače. Na zjištění hrany signálu jsou tedy potřeba dva strojové takty (24 hodinových taktů). Maximální vzorkovací frekvence je tedy 1/24 frekvence oscilátoru a zároveň musí být minimální délka úrovně na vstupu 12 period hodinových impulzů.

Režimy čítače:

- **Režim 0** - 13-bitový čítač/časovač
- **Režim 1** - 16-bitový čítač/časovač
- **Režim 2** - 8-bitový čítač/časovač s přednastavením
- **Režim 3** - 16-bitový čítač/časovač s dělením

5.5.2 Sériová linka

Mikrokontrolér 8051 obsahuje plně duplexní sériový kanál, který umožňuje komunikaci v synchronním i asynchronním režimu. Pro komunikaci se využívá registr SBUF, který je však ve skutečnosti zdvojený: jeden pro odesílání, druhý pro příjem. Oba jsou však na stejné adrese.

Sériová linka pracuje ve čtyřech režimech. Režim 0 je synchronní. Hodinový signál vystupuje na portu TxD, odesílání nebo přijímání probíhá na portu RxD. Rychlost přenosu je 1/12 frekvence oscilátoru. Tento režim neumožňuje současné odesílání a přijímání.

Režimy 1 a 3 jsou asynchronní, plně duplexní. Jejich rychlost je určena periodou přetečení čítače/časovače 1.

Režim 2 je asynchronní, jeho rychlost je však určena pouze periodou hodinového kmitočtu (1/32 nebo 1/64).

Kapitola 6

Řešení

6.1 Úroveň abstrakce popisu architektury

Architekturu lze popisovat na dvou různých úrovních, každá z nich má své výhody a nevýhody.

6.1.1 Instrukční úroveň

Model je na této úrovni popsán pouze na úrovni instrukční sady. Neřeší se zde vnitřní časování, u modelů s instrukční sadou typu CISC se ani neřeší rozdělení instrukcí na posloupnost mikroinstrukcí.

Výhodou tohoto popisu je jeho relativní jednoduchost. Model se dá popsat velice rychle, snadno se pro něj vytvoří nástroje, například překladač assembleru, disassembler, simulátor, překladač jazyka C a podobně.

Simulátor není příliš detailní, protože nejmenší krok simulace je jedna instrukce, ale díky tomu je velice rychlý.

Nevýhodou ale je, že při tomto popisu chybí detailní informace o vnitřní architektuře. Nemůžeme tedy sledovat vnitřní chování a například optimalizovat provádění instrukcí. Obtížně se také řeší různé periferie typu čítače/časovače, sériová linka apod., které jsou úzce svázány s vnitřní architekturou. Ze stejného důvodu je také problematický export do VHDL kódu.

6.1.2 Úroveň cyklů

Na této úrovni popisu je u modelu popsána nejen instrukční sada, ale i jeho vnitřní časování. To přináší mnoho výhod. U architektur CISC můžeme instrukce rozdělit na posloupnosti mikroinstrukcí, které se provádějí v čase, můžeme lépe namodelovat různé periferie typu čítač/časovač, sériová linka a také přesněji popsat modely se zřetězenou linkou.

Simulace je v tomto případě detailnější, protože nejmenším krokem je jeden hodinový takt. Instrukce se provádí několik hodinových taktů, během nichž se přistupuje ke zdrojům (registry, paměť, ALU). To umožňuje pokročilé optimalizace architektury a instrukční sady. Lze například sledovat rozložení zátěže a podle toho model optimalizovat. Nevýhodou je, že simulátor je pomalejší než v předchozím případě.

Další nevýhodou tohoto řešení je jeho složitost a časová náročnost. Je nutné si detailně promyslet vnitřní architekturu, navrhnout a popsat řadič, případně rozdělit instrukce na posloupnost mikroinstrukcí.

Pokud chceme pro modelovaný procesor pouze sestrojit překladače, popřípadě jednoduchý simulátor založený na instrukční sadě, stačí nám k popisu architektury pouze popis na úrovni instrukcí. Pro pokročilé činnosti je však třeba popsat model na úrovni cyklů.

Pro svůj model jsem zvolil právě tento způsob popisu architektury.

6.2 Vnitřní architektura

Návrh vychází z informací uvedených v kapitole 5.2.

Na obrázku 5.1 je naznačen pomocný 16-bitový registr `Buffer`, který propojuje datovou a programovou sběrnici. V mém modelu je registr `Buffer` rozdělen na dva 8-bitové registry `reg_cache1` a `reg_cache2`. Tento registr je využíván k několika činnostem:

- Dočasné uchování dat během provádění mikroprogramu instrukce (např. výsledek z ALU).
- Dočasné uložení nově načteného bajtu (adresa nebo data). Dekodéry, které dekódují operandy instrukce (druhý a třetí bajt), ukládají načtený operand do tohoto registru.
- Při provádění mikroprogramu instrukce skoku může být obsah tohoto registru uložen do registru PC (Program Counter). Do registru `Buffer` se uloží adresa cíle skoku a při dokončování mikroprogramu je výsledek uložen do registru PC, čímž se provede skok.

6.2.1 Datový adresový prostor

Při návrhu mikroprogramu pro instrukce jsem došel k závěru, že sady registrů R0 až R7 nejsou ve skutečnosti tvořeny registry, ale klasickou pamětí RAM. Pro práci s těmito registry je totiž potřeba stejný čas jako pro práci s klasickou pamětí RAM. K jejich adresování je proto také použit registr AR.

Stejným způsobem se adresují i registry v oblasti SFR. Výjimku tvoří univerzální registr ACC, který je přímo připojen k ALU, a umožňuje tak rychlejší provádění aritmeticko-logických operací. Další výjimkou je ukazatel na vrchol zásobníku (SP) pro urychlení instrukcí pracujících se zásobníkem.

Rovněž bitově adresovatelná oblast je dle mého názoru konstruována jako 8-bitová paměť RAM. Jednak práce s ní trvá stejně dlouho, jednak ji lze využít k bajtovým i bitovým operacím. Při bitových operacích se spodní tři bity použijí k adresaci konkrétního bitu, zbytek adresy adresuje konkrétní bajt. Vzhledem k tomu, že bitový prostor začíná od adresy 20h, je nutné adresu bajtu patřičně upravit:

- adresa bitu = adresa AND 0b00000111
- adresa v paměti = (adresa » 3) OR 20h

Některé registry v oblasti SFR také umožňují bitovou adresaci. Způsob jejich adresace je velice podobný adresaci bitové paměti. Spodní tři bity adresují konkrétní bit v bajtu, zbytek udává adresu celého bajtu v paměťovém prostoru.

- adresa bitu = adresa AND 0b00000111
- adresa v paměti = adresa AND 0b11111000

Pro bitově adresovanou paměť jsou k dispozici speciální instrukce, takže není problém před uložením adresy do registru AR tuto hodnotu změnit. V hardwaru by se tato konverze provedla pomocí kombinační logiky.

Adresový dekodér jsem implementoval jako funkci v jazyce C. K zápisu do paměti slouží funkce `void ram_store(int data)`. Adresa musí být před voláním této funkce uložena v registru AR. Vstupním parametrem je hodnota, kterou chceme do paměti uložit. Podle adresy se rozhodne, zda se bude adresovat oblast paměti RAM, nebo oblast speciálních funkčních registrů.

Ke čtení dat z paměti je určena funkce `int ram_load()`. Princip funkce je podobný jako při zápisu do paměti. Návratová hodnota funkce jsou data uložená na adrese adresované registrem AR.

Adresový dekodér implementovaný ručně v jazyce C přináší mnoho výhod. Je možné pro čtení a pro zápis na stejnou adresu využít různé registry. To se hodí například u registru SBUF používaného sériovou linkou. Tento registr je ve skutečnosti tvořen dvěma registry TBUF a RBUF (souhrně označovány jako SBUF), které jsou mapovány na adresu 99h. Při zápisu do registru SBUF se hodnota uloží do registru TBUF, při čtení z registru SBUF je vrácen obsah registru RBUF.

Další výhodou je, že lze aktivovat události spouštěné čtením/zápisem z/do registru. Tento princip je opět použit u sériové linky. Zápisem do registru SBUF se spustí odesílání zapsaných dat přes sériovou linku.

V současné verzi jazyka ISAC už je možnost mapovat registry do paměťového prostoru (dříve bylo možné do tohoto prostoru mapovat pouze paměti), bohužel však není možné mapovat různé registry pro zápis a pro čtení na stejnou adresu, nelze ani aktivovat události reagující na zápis/čtení do registru. Proto jsem zůstal u vlastního adresového dekodéru.

6.2.2 ALU

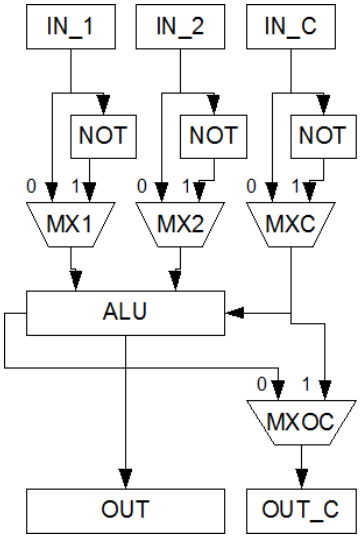
Na vstupu ALU jsou dva 8-bitové registry `ALU_in1` a `ALU_in2` a jednobitový registr `ALU_inC` pro bitové operace nebo vstupní přenos.

O tom, jaká matematická operace bude provedena, se rozhodne pomocí aktivace odpovídající operace v sekci ACTIVATION (ALUadd, ALUand, ALUor apod.). Výsledek je potom uložen do registru `ALU_out` a příznaky (parity, carry, auxiliary carry, overflow) jsou uloženy v pomocných registrech `ALU_outP`, `ALU_outC` apod. Význam jednotlivých příznaků je uveden v kapitole 5.1.1.

Podle [13] je příznak P (parity) svázán s akumulátorem (ACC). Z toho by se dalo usoudit, že příznak parity je za běhu počítán z obsahu akumulátoru. Při návrhu mikroprogramu pro instrukce a z obrázku 5.1 jsem však došel k závěru, že tento příznak je ve skutečnosti počítán v jednotce ALU, která jako jediná zapisuje příznaky do registru PSW. Z toho vyplývá, že i při provádění instrukce `MOV A,#20` (uloží hodnotu 20 do akumulátoru) musí vstupní hodnota 20 projít jednotkou ALU (kde se například provede součet 20+0) a z výsledku se vypočítá parita. Teprve poté se hodnota uloží do akumulátoru a příznak parity se uloží do registru PSW.

Obrázek 6.1 ukazuje můj návrh ALU. Před provedením požadované operace je nutné správně přednastavit multiplexory na vstupu ALU. Tabulka 6.1 ukazuje jejich správnou konfiguraci pro jednotlivé operace.

Tato jednotka je implementována ve funkci `utils_alu_arit(Ealuoper oper)`. Vstupním parametrem této funkce je typ prováděné operace. Podle něj se nastaví multiplexory a operace se provede.



Obrázek 6.1: Blokové schéma ALU

Operace	**MX1**	**MX2**	**MXC**	**MXOC**
Součet	0	0	0	0
Rozdíl	1	0	1	0
Negace	X	1	1	1
Rotace přes carry	X	0	0	0
Ostatní operace	0	0	X	X

Tabulka 6.1: Konfigurace multiplexorů pro zvolenou operaci

Bajtové operace

Abych se přiblížil skutečnému řešení, které by se použilo v hardware, operace „rozdíl“ se ve skutečnosti provede jako součet čísel v doplňkovém kódu. To lze odvodit následujícím způsobem:

$$subb = in2 - in1 - C = in2 + (-in1) - C$$

operand $in1$ se převede na doplňkový kód

$$subb = in2 + (\overline{in1} + 1) - C = in2 + \overline{in1} + (1 - C)$$

podle booleovy algebry lze převést $(1 - C)$ na $(\overline{C})$, čímž získáme konečný výsledek

$$subb = in2 + \overline{in1} + \overline{C}$$

Díky této úpravě je však nutné změnit i výpočet výstupních příznaků Carry a Auxiliary Carry, které se musí znegovat.

Při návrhu řídící jednotky jsem také došel k závěru, že při operaci NOT (negace) se kromě vstupu `ALU_in2` také zneguje vstupní carry. Díky tomu jsem ušetřil jednu operaci (mikroinstrukci) v řídící jednotce.

Speciálním případem jsou operace násobení a dělení. Ty jsem se rozhodl implementovat pomocí standardní funkce násobení a dělení v jazyce C. Vzhledem k tomu, že výsledek po provedení těchto operací je dvojnásobně široký, bylo nutné pro uložení celého výsledku přidat na výstupu ALU druhý pomocný registr `ALU_out2`, do kterého se při násobení uloží vyšší bajt a při dělení se do něj uloží zbytek po celočíselném dělení.

Bitové operace

Vzhledem k tomu, že jsem bitovou paměť implementoval jako klasickou paměť RAM adresovatelnou po bajtech, bylo nutné tomuto řešení přizpůsobit i ALU.

Přidal jsem proto k této jednotce další registr `ALU_rot`. Při provádění instrukce pracující s bitovou pamětí se při zápisu adresy do registru AR (adresový registr) zapisovaná adresa upraví pomocí předřazené kombinační logiky (viz kapitola 6.2.1). Spodní tři bity adresy se uloží do registru `ALU_rot`, zbylá část adresy se uloží do registru AR.

Registr `ALU_rot` adresuje jeden z osmi bitů uložených ve vstupním registru `ALU_in1`. Bitová operace se provede pouze s adresovaným bitem a výsledek se jako celý bajt uloží do výstupního registru `ALU_out`. Dále už se s výsledkem pracuje zcela normálním způsobem.

Právě to, že lze bitové operace řešit tímto způsobem, mě utvrdilo v názoru, že celá paměť v mikrokontroléru 8051 je řešena jako bajtově adresovatelná paměť RAM.

Ostatní operace

Kromě aritmeticko-logických operací jsou v ALU také prováděny speciální operace rotace vlevo a vpravo přes carry, dekadická korekce po sčítání, výměna horní a dolní čtveřice bitů (SWAP) a výměna nižší čtveřice bitů mezi dvěma registry.

Všechny tyto operace využívají standardní nastavení nastavení multiplexorů. V tabulce 6.1 je to řádek „Ostatní operace“.

6.2.3 Vstupně/výstupní rozhraní

Mikrokontrolér 8051 komunikuje se svým okolím pomocí čtyř osmibitových portů označených jako P0 až P3. Každý pin je adresovatelný zvlášť a může být buď vstupní, nebo výstupní. Směr komunikace se určuje pomocí registrů P0 až P3, které jsou mapovány v oblasti speciálních funkčních registrů.

Mikrokontrolér 8051 je postaven na technologii TTL. Tomu také odpovídá chování vstupně výstupních portů. Při zápisu do registru P0 až P3 se zapsaná hodnota objeví také na výstupu.

Zápisem log. 1 do odpovídajícího bitu v registru se tato hodnota objeví i na výstupu. Pokud se nyní tento pin hardwarově vynuluje (připojí se k zemi), je tato hodnota propagována také na vstup, a pokud nějaká instrukce bude číst hodnotu na tomto pinu, přečte log. 0 (i přesto, že v registru je na odpovídající pozici uložena log. 1).

Pokud je v registru zapsána hodnota log. 0, objeví se tato hodnota také na výstupu, ale není možné číst vstupní hodnotu. Vždy by se přečetla log. 0.

Vzhledem k tomu, že jazyk ISAC v současné chvíli neumožňuje modelovat rozhraní mikrokontroléru, vyřešil jsem rozhraní pouze pomocí registrů P0 až P3. Čtení i zápis na vstupně/výstupní port se provádí pouze pomocí těchto registrů. Pokud se v simulátoru do registru zapíše nějaká hodnota, pak je tato hodnota přečtena instrukcí pracující s portem.

6.3 Řídící jednotka

Řídící jednotka zajišťuje činnost procesoru podle pokynů programu: načítá, dekóduje a provádí instrukce, načítá operandy instrukcí a ukládá výsledky po provedení instrukce. Mikrokontrolér 8051 spadá do kategorie procesorů CISC. Každá instrukce je tedy řídící jednotkou provedena jako posloupnost odpovídajících mikroinstrukcí. Viz kapitola 4.2.1.

V dostupné literatuře bohužel nejsou o řídící jednotce uvedeny detailní informace. Tedy ani o tom, jaké mikroinstrukce řadič obsahuje a jak instrukce pomocí těchto mikroinstrukcí provádí. Bylo tedy nutné, abych si mikroprogramy pro instrukce navrhnul sám. Snažil jsem se o co nejjednodušší a nejefektivnější návrh tak, aby vše odpovídalo skutečnému modelu a informacím z dostupné literatury. Vycházel jsem z informací o časování uvedených v kap. 5.4. Výsledné mikroprogramy jsou v uvedeny v příloze D.

6.3.1 Způsoby zápisu řadiče mikroinstrukcí v jazyce ISAC

Pro implementaci řídící jednotky se v jazyce ISAC nabízejí dvě možnosti:

- **Vlastní stavový automat** - V sekci BEHAVIOR pomocí jazyka C implementujeme vlastní stavový automat, který po dekódování instrukce začne provádět odpovídající mikroprogram.
- **Využití sekce ACTIVATION** - Jednotlivé mikroinstrukce se zapíšou jako operace (OPERATION) a ty se v patřičném pořadí a s patřičným zpožděním uvedou v sekci ACTIVATION. Tímto způsobem lze pro každou instrukci vytvořit odpovídající mikroprogram.

Sekce ACTIVATION byla původně určena pro modely se zřetězeným zpracováním (tzv. pipeline modely), kde docházelo k velice malým zpožděním. V mikrokontroléru 8051 však provedení nejkratší instrukce trvá 12 hodinových taktů, takže se běžně využívá zpoždění aktivace operace např. o 12 taktů. Tento způsob však byl při překladu velice náročný na

paměť a strojový čas, protože se při překladu generovaly všechny možné stavy, které mohly nastat.

V současné době je však již práce se sekcí ACTIVATION optimalizována, a tak není k používání vlastního stavového automatu zapsaného v sekci BEHAVIOR důvod. Zápis vlastního automatu je totiž časově i implementačně mnohem náročnější a výsledný kód není příliš přehledný. Při použití sekce ACTIVATION za nás překladač ISAC vytvoří automat sám. Dosáhneme tím vyšší abstrakce a výsledný kód je mnohem přehlednější.

Jednotlivé mikroinstrukce se tedy v jazyce ISAC zapíší jako operace (OPERATION). Pak v každé operaci, která slouží k dekódování a provedení instrukce (obsahují sekce ASSEMBLER a CODING), se v sekci ACTIVATION uvedou s odpovídajícím zpožděním aktivace mikrooperací, čímž pro každou instrukci dostaneme odpovídající mikroprogram.

6.3.2 Implementace

Každá instrukce je v jazyce ISAC řešena jako operace (či skupina operací), která obsahuje sekci ASSEMBLER a CODING. Tyto sekce řeší mapování binárního obrazu instrukce na jazyk symbolických adres (assembler). Pokud je instrukce rozpoznána, provede se kód v sekci BEHAVIOR a naplánují se další operace, které jsou uvedeny v sekci ACTIVATION.

Implementace řídící jednotky začíná v operaci *main*, v jejíž sekci ACTIVATION se aktivuje operace `fetch_instr` se zpožděním jednoho taktu. Ta načte první bajt instrukce s operačním kódem, uloží ho do registru `ir`, uloží aktuální hodnotu registru PC (program counter) do registru `ir_pc` a inkrementuje PC. V sekci ACTIVATION je se zpožděním jednoho taktu naplánována aktivace operace `decode`.

Operace `decode` je složena ze sekcí CODINGROOT a STRUCTURE. Sekce CODINGROOT slouží pro popis instrukční sady a převádí jazyk symbolických instrukcí na strojový kód a obráceně. K tomu využívá skupinu operací `instr_set`, do které patří operace začínající předponou `instr_`.

Sekce STRUCTURE popisuje instrukční sadu z hlediska použitých dekodérů. Všechny instrukce jsou roztříděny podle používaných dekodérů do skupin, které začínají předponou `opc_` a všechny tyto skupiny jsou sjednoceny ve skupině `opc`. Podle operačního kódu načtené instrukce se pomocí podmíněného větvení SWITCH naplánuje použití dekodérů pro vícebajtové instrukce. Dekodéry je možné aktivovat se zpožděním stejným způsobem, jako je tomu v sekci ACTIVATION.

Dekodéry jsou implementovány v operacích začínající předponou `fetch_` a dekódují vždy jeden bajt.

6.3.3 Průběh vývoje dekodéru instrukcí

Šířka instrukcí mikrokontroléru 8051 je 1 až 3 bajty. Problém je, že paměť programu je široká 8 bitů. Pokud je šířka instrukce větší (2 nebo 3 bajty), lze načíst pouze první bajt instrukce. Ten však pro dekódování instrukce v mikrokontroléru 8051 stačí, protože operační kód je vždy široký maximálně jeden bajt.

Například instrukce `MOV A,#20` je široká dva bajty. První bajt je operační kód instrukce, druhý bajt pak přímá hodnota 20. V jazyce ISAC by se dekódování této instrukce teoreticky zapsalo:

```
ASSEMBLER { "MOV" "A" "," "#" data=#U };
CODING { 0b01110100 data=0bx[8] };
```

Tento zápis však nelze použít. Původní verze jazyka ISAC neumožňovala načítat instrukce širší, než je šířka paměti programu. Problém byl vyřešen pomocí rozšíření sekce CODINGROOT. Byla přidána možnost definovat pro vícebajtové instrukce dekodéry, které načítaly další operandy instrukce. Tyto dekodéry se v sekci CODINGROOT uvedly v příkazu SWITCH, kde každý dekodér měl svou vlastní větev CASE. Instrukce pak byla pomocí těchto dekodérů dekódována.

Bylo také možné nastavit zpoždění aktivace dekodéru (podobně, jako je tomu v sekci ACTIVATION), čímž bylo dosaženo reálnějšího chování simulátoru. Například u 8051 se další bajt načítá až po 5 hodinových periodách. Dekódování celé instrukce se provádělo takto (ukázka je bez sekcí ACTIVATION, BEHAVIOR a INSTANCE):

```
OPERATION op_mov_acc_data {
    ASSEMBLER { "mov" "A" "," "#" };
    CODING { 0b01110100 };
}
OPERATION fetch_data {
    ASSEMBLER { imm=#U };
    CODING { imm=0bx[8] };
}
GROUP opc = op_mov_acc_data, ...;

OPERATION decode {
    CODINGROOT {{
        SWITCH(opc(ir,ir_pc)) {
            CASE op_mov_acc_data: {
                %5; fetch_data(program_mem[pc],pc);
            }
        }
    }}
}
```

Ani tento zápis ale nestačil k tomu, abych mohl popsat celou instrukční sadu. Objevily se totiž další problémy.

První problém byl nemožnost dekódovat 16-bitovou hodnotu. Příkladem takové instrukce je `MOV DPTR,#data16`, která do registru DPTR uloží 16-bitovou hodnotu. Každý dekodér, který využíval sekce ASSEMBLER a CODING, umožňoval dekódovat pouze operační kód o velikosti šířky paměti. U mikrokontroléru 8051 je to jeden bajt.

Druhý problém byl prohození pořadí operandů v sekci ASSEMBLER a CODING. Například instrukce `MOV dest,src` má v binární podobě prohozené pořadí bajtů dest a src. První bajt je operační kód instrukce, pak následuje operand src a za ním operand dest.

Oba tyto problémy se vyřešily úpravou sekce CODINGROOT a přidáním nové sekce STRUCTURE. Sekce CODINGROOT nyní popisuje instrukční sadu z pohledu mapování jazyka symbolických instrukcí na strojový kód. Sekce STRUCTURE popisuje mikroarchitekturu z pohledu použitých dekodérů. Kód pro tuto instrukci vypadá takto:

```
OPERATION op_mov_direct_direct {
    ASSEMBLER { "mov" };
    CODING { 0b10000101 };
    ACTIVATION { ... };
```

```
}
OPERATION fetch_direct_direct_1 {
    ASSEMBLER { address=#U "," };
    CODING { address=0bx[8] };
}
OPERATION fetch_direct_direct_2 {
    ASSEMBLER { address=#U };
    CODING { address=0bx[8] };
}

OPERATION instr_direct_direct {
    ASSEMBLER {op_mov_direct_direct daddr_dest=#U "," daddr_src=#U };
    CODING {op_mov_direct_direct daddr_src=0bx[8] daddr_dest=0bx[8] };
}

GROUP instr_set = instr_direct_direct, ... ;
GROUP opc = op_mov_direct_direct, ...;

OPERATION decode {
    CODINGROOT {{instr_set;}};
    STRUCTURE {{
        SWITCH(opc(ir,ir_pc)) {
            CASE op_mov_direct_direct: {
                %5; fetch_direct_direct_1(program_mem[pc],pc);
                %6; fetch_direct_direct_2(program_mem[pc],pc);
            }
        }
    }}
}
```

Z příkladu je patrné, že pro každý bajt (zdrojová i cílová adresa) je definován vlastní dekodér. Mapování celé instrukce z jazyka symbolických adres na strojový kód se provede v sekci CODINGROOT pomocí skupiny operací `instr_set`.

Pro dekódování prvního bajtu instrukce (operačního kódu) se v sekci STRUCTURE využije dekodér `opc` s parametry `ir` (operační kód) a `ir_pc` (adresa, na které se instrukce v paměti nachází).

Po dekódování prvního bajtu se podle rozpoznané instrukce za pomoci větvení SWITCH naplánuje dekódování následujících bajtů. Lze k tomu využít zpožděné aktivace podobně, jako je tomu v sekci ACTIVATION.

Mikroprogram, který realizuje instrukci, je zapsán v sekci ACTIVATION v operaci `op_mov_direct_direct`.

6.3.4 Průběh dekódování instrukce pracující s registry

Některé instrukce pracují s registry R0 až R7. Adresa konkrétního registru je součástí operačního kódu (poslední tři bity prvního bajtu). Následující příklad ukazuje způsob dekódování instrukce `inc Rn`, která inkrementuje obsah registru.

```
...
```

```
OPERATION Rn1 {
    ASSEMBLER {"R1"};
    CODING {0b001};
    EXPRESSION { 1; };
}
OPERATION Rn0 {
    ASSEMBLER {"R0"};
    CODING {0b000};
    EXPRESSION { 0; };
}
GROUP Rn = Rn0, Rn1, Rn2, Rn3, Rn4, Rn5, Rn6, Rn7;

OPERATION op_inc_Rn {
    INSTANCE Rn;
    ASSEMBLER { "inc" Rn };
    CODING { 0b00001 Rn };
    BEHAVIOR { ... };
    ACTIVATION { ... };
}
```

Skupina `Rn` sdružuje operace `Rn0` až `Rn7`, které slouží pro dekódování registrů z operačního kódu.

Dekódování celé instrukce provádí operace `op_inc_Rn`. Ta zajišťuje mapování celého kódu v assembleru na binární obraz. Instrukce má však několik variant, protože může pracovat s registry R0 až R7. Pro dekódování správného registru využívá skupinu operací `Rn`. Ta dekóduje registr a jeho číslo (0 až 7) vrátí pomocí sekce EXPRESSION zpět do operace `op_inc_Rn`. V sekci BEHAVIOR pak lze s touto hodnotou pracovat a např. inkrementovat rozpoznaný registr.

6.3.5 Odchylky instrukční sady od skutečného modelu

Instrukce HALT

V instrukční sadě mikrokontroléru 8051 neexistuje instrukce HALT. Tato instrukce není v jazyce ISAC povinná, ale velice usnadňuje simulaci modelu. Při dekódování instrukce lze provést různé sémantické akce (např. výpis paměti na obrazovku). Lze ji pochopitelně nahradit tzv. breakpointem (zarážka, na které se simulace pozastaví), ale toto řešení není vždy nejvhodnější a nejrychlejší.

Vzhledem k tomu, že v instrukční sadě zbylo jedno volné místo s operačním kódem A5h, rozhodl jsem instrukci HALT s tímto operačním kódem přidat do instrukční sady.

Instrukce pracující s externí pamětí (MOVX)

Problém těchto instrukcí je spojen s rozhraním mikrokontroléru, které bylo popsáno v kap. 6.2.3.

Tyto instrukce komunikují s externí pamětí pomocí čtení/zápisu z/do registrů portů P2 (vyšší část adresy) a P0 (nižší část adresy/data).

Práci s externí pamětí simuluji pouze pomocí práce s registry portů. Během simulace instrukce se do registrů P0 a P2 zapíše adresa, následně se z/do registru P0 přečtou/zapíšou

data. Při čtení z externí paměti se v průběhu simulace této instrukce musí ve správný okamžik do registru P0 zapsat čtená data, jinak dojde k přečtení adresy, jež byla do registru zapsána.

Instrukce ACALL a AJMP

Tyto instrukce bohužel nelze v současné verzi nástrojů popsat. Důvodem je 11-bitový operand (adresa), který je rozdělen na dvě části, které spolu nesousedí. Viz následující příklad:

- Zápis v assembleru: `ACALL 11BitAddress`
- Zápis v binární podobě: `aaa10001 aaaaaaaa`

kde 11-bitová adresa je rozdělena na dvě části. Spodních 8 bitů je uloženo jako druhý bajt instrukce, horní 3 bity jsou uloženy v operačním kódu, avšak neleží vedle zbylé části adresy.

V jazyce ISAC se s tímto problémem počítalo a lze ho vyřešit následujícím zápisem:

```
ASSEMBLER { "ACALL" addr11=#U };
CODING { addr11=0bx[10..8] 0b10001 addr11=0bx[7..0] };
```

V současné chvíli však není tento zápis podporován v generátorech nástrojů, a proto tyto instrukce v současné chvíli podporují pouze 8-bitovou adresu. Horní 3 bity adresy se ignorují.

6.4 Periferie

I přes to, že jazyk ISAC není přímo určen pro modelování periferních obvodů, ale spíše pro popis architektury CPU, rozhodl jsem se periferie sériové linky a čítačů/časovačů namodelovat také. Název této práce zní „Simulace architektury mikroprocesoru 8051“ a periferie jsou důležitou součástí mikrokontroléru.

Všechny tyto periferie jsou nezávislé na CPU, což znamená, že mohou pracovat i bez účasti procesoru. Je proto nezbytně nutné zajistit spolehlivý hodinový signál. K tomu jsem v jazyce ISAC využil sekci BEHAVIOR v hlavní operaci *main*. Tato sekce je totiž jako jediná prováděna každý hodinový takt.

Je zde přítomna proměnná `ISAC_CYCLE_CNT`, která se každý takt inkrementuje. Podle informací uvedených v kapitole 5.4 tuto proměnnou dělím hodnotou 12. Zbytek po dělení nabývá hodnot 0 až 11, což odpovídá stavům S1P1 až S6P2.

Každou periferii jsem naprogramoval jako samostatnou funkci v jazyce C, jejímž jediným parametrem je aktuální stav procesoru. Při každém provedení sekce BEHAVIOR v operaci *main* je tento stav změněn. Každá periferie je implementována jako stavový automat s vlastní stavovou proměnnou, která je odlišná od stavu procesoru, ale je pomocí něj synchronizována.

6.4.1 Sériová linka

Sériová linka je implementována ve funkci `handle_serial(int state)`. Je řízena pomocí stavového automatu. Bity `SM0` a `SM1` rozhodnou o prováděném režimu.

Režim 0

V režimu 0 může být aktivní buď příjem nebo odesílání.

Pokud dojde v režimu příjmu k vynulování bitu RI, sériová linka se aktivuje a do registru RBUF se uloží hodnota FEh. Ve stavu procesoru S5P2 se vzorkuje vstupní bit a ve stavu S6P2 se navzorkovaný bit přidá na začátek registru RBUF a jeho obsah se posune doleva. Po přijmutí všech 8 bitů se na 9. pozici dostane log. 0 (nejpravější bit v přednastavené hodnotě FEh), čímž je detekován konec příjmu a dojde k nastavení bitu RI.

Obdobně funguje i odesílání, které je aktivováno zápisem do registru SBUF. Ve stavu S1P1 je odeslán bit a obsah registru se posune doprava. Po odeslání 8 bitů je obsah registru TBUF prázdný. To se detekuje ve stavu S6P2 a v případě nulovosti se nastaví bit TI.

V obou případech se na výstupu pinu TxD vysílá hodinový signál. Ve stavu S3P1 nabývá hodnotu log. 0, ve stavu S6P1 log. 1.

Režim 1, 2 a 3

Tyto režimy jsou řízeny dalším stavovým automatem, který slouží k odeslání start bitu, dat, případného devátého bitu a stop bitu. Rychlost odesílání je ovlivňována předděličkou.

V režimu je 1 a 3 je rychlost ovlivněna rychlostí čítače/časovače 1. Při jeho přetečení dojde k inkrementaci registru předděličky. Při přetečení předděličky se začne odesílat další bit. V režimu 2 předdělička pouze dělí hodinový signál.

6.4.2 Čítače/časovače

Čítače/časovače jsou implementovány ve funkci `handle_timer(int state)`. Ve stavu S5P2 se neustále vzorkují hodnoty na vstupech T0 a T1, které jsou součástí portu P3. Činnost čítačů/časovačů probíhá ve stavu S3P1.

V tomto stavu se v režimu časovače inkrementuje obsah volně běžícího čítače (registr). V režimu čítače se ověřuje, zda předchozí hodnota na vstupu T0 nebo T1 (zjištěná ve stavu S5P2) byla log. 1 a aktuální hodnota je log. 0. Pokud ano, dojde k inkrementaci registru čítače.

Pokud došlo k přetečení registru čítače/časovače, nastaví se bit TF0 nebo TF1 indikující přetečení. V režimu 2 se navíc do registru TLx vloží obsah registru THx (přednastavení).

6.4.3 Přerušovací systém

Přerušovací systém je implementován ve funkci `handle_interrupt(int state)`. Ve stavu S5P2 se vzorkují žádosti o přerušení. Pokud bylo nějaké přerušení zjištěno a je v registru `IE` povoleno, je nastaven bit `INT_REQ`.

Registr `INT_PRI0` indikuje provádění obsluhy přerušení s prioritou 0. Podobně registr `INT_PRI1` indikuje provádění obsluhy přerušení s prioritou 1. Registr `INT_LCALL` indikuje právě prováděný skok na obsluhu přerušení.

Ve stavu S6P2 se ověřuje, zda bylo detekováno přerušení (`INT_REQ == 1`). Pokud *neprobíhá žádný skok* na obsluhu přerušení (`INT_LCALL == 0`), začnou se žádosti o přerušení vyhodnocovat. Nejdříve se zkontroluje, zda není signalizován požadavek na přerušení s prioritou 1. Pokud takový požadavek existuje a není aktuálně prováděno žádné přerušení s prioritou 1 (`INT_PRI1 == 0`), nastaví se příznak probíhajícího skoku na obsluhu přerušení (`INT_LCALL = 1`) a příznak, že právě probíhající přerušení má prioritu 1 (`INT_PRI1 = 1`). Také se do registru `INT_ADDR` uloží adresa, na které leží rutina pro obsluhu přerušení.

V případě, že žádný požadavek na přerušení s úrovní priority 1 neexistuje, ověří se, zda byl detekován požadavek na přerušení s úrovní 0. Pokud tento požadavek existuje a není právě prováděn skok na obsluhu přerušení (`INT_LCALL == 0`), neprovádí se obsluha přerušení s úrovní 1 (`INT_PRI1 == 0`) ani s úrovní 0 (`INT_PRI0 == 0`), přerušení je akceptováno, nastaví se příznak probíhajícího skoku na obsluhu přerušení (`INT_LCALL = 1`) a příznak, že právě probíhající přerušení má prioritu 0 (`INT_PRI0 = 1`). Do registru `INT_ADDR` se uloží adresa, na které leží rutina pro obsluhu přerušení.

Pokud má právě prováděná obsluha přerušení prioritu 0 (`INT_PRI0 == 1`), neprovádí se skok na obsluhu přerušení (`INT_LCALL == 0`) a detekuje se požadavek na obsluhu přerušení s úrovní 1, přerušení se obslouží. Opět se nastaví příznak obsluhy přerušení s úrovní 1 (`INT_PRI1 = 1`) a obsluha se provede.

Jakmile dojde k nastavení příznaku skok na obsluhu přerušení (`INT_LCALL == 1`), dokončí se právě prováděná instrukce. Po jejím dokončení se v operaci *main* ve stavu S1P2 místo aktivace načtení další instrukce aktivuje operace skok na obsluhu přerušení. Toho lze dosáhnout podmíněnou aktivací:

```
ACTIVATION {
    %1; IF (INT_LCALL == 0) fetch_instr;
        ELSE interrupt_lcall;
}
```

Je však nutné zajistit, aby byl příznak `INT_LCALL` nastaven ještě před vyhodnocením sekce ACTIVATION (tedy před skončením právě prováděné instrukce). Pokud by byl příznak nastaven až ve stavu S1P1, nebyl by proveden skok na obsluhu přerušení, i když se začne provádět teprve ve stavu S1P2.

Během provedení instrukce RETI se musí zrušit příznak o prováděném přerušení. To znamená, že se zkontroluje, zda bylo prováděno přerušení s prioritou 1 (`INT_PRI1 == 1`). Pokud ano, je tento příznak vynulován. Jinak se vynuluje příznak obsluhy přerušení s úrovní 0 (`INT_PRI0 = 0`).

Kapitola 7

Ověření modelu

7.1 Knihovna pro ověření modelu

Pro ověření modelu jsem naprogramoval knihovnu, která umožňuje provádět s obecně n-bajtovými čísly tyto operace:

- součet
- rozdíl
- násobení - Násobení probíhá pomocí operací rotace a součet. K výpočtu je potřeba dvakrát větší šířka proměnných. Pokud například chceme provádět operace s dvoubajtovými čísly a jednou z operací bude i násobení, je nutné provádět všechny operace s dvojnásobnou šířkou (tedy 4 bajty).
- Fibonacciho posloupnost
- faktoriál - Vychází z rovnice $f(n) = f(n-1) + f(n-2)$, avšak pro výpočet využívá iterační algoritmus. Vyžaduje proto dvě pomocné proměnné, ve kterých jsou uloženy dva poslední výsledky $f(n-1)$ a $f(n-2)$
- inkrementace
- dekrementace
- rotace vlevo přes carry
- rotace vpravo přes carry
- přesun čísla
- vynulování čísla

Šířka čísel je definována v proměnné DELKA a může nabývat hodnot mocniny 2 (1, 2, 4, 8, 16 ...). Čísla jsou uložena ve formátu little-endian, tedy nejméně významný bajt (LSB) je uložen na nejnižší adrese. Při volání knihovních funkcí se předávají adresy na LSB vstupních čísel a jejich šířka (ta se vždy předává v registru R7).

Každá funkce před samotným výpočtem uloží na zásobník všechny registry, které bude nějakým způsobem měnit. Po skončení výpočtu pak tyto registry ze zásobníku obnoví.

Knihovna je vytvořena s ohledem na co největší univerzálnost, takže její kód není příliš rychlý ani efektivní. To však ani nebylo mým záměrem. Chtěl jsem především vytvořit dostatečně dlouhý a složitý kód pro ověření svého modelu.

7.2 Simulace

Pro porovnání výkonnosti vygenerovaného simulátoru mého modelu jsem se rozhodl využít dva známé simulátory mikrokontrolérů 8051.

Prvním z nich je simulátor EASY51 od firmy EasySoft, která mi laskavě dovolila jejich simulátor k tomuto účelu využít. Jedná se o DOSovskou aplikaci, jejíž vývoj byl sice ukončen před cca 10 lety, nicméně se stále jedná o jeden z nejlepších simulátorů mikrokontroléru 8051.

Druhým testovaným simulátorem je AdSim ve verzi 3.650 od firmy Analog Devices určený pro operační systém Microsoft Windows. Simulátor je volně ke stažení na jejich internetových stránkách. Program jsem nastavil tak, aby během simulace nevypisoval obsahy registrů, ale aby vše zobrazil až po skončení simulace. Simulace tak není ovlivněna voláním systémových funkcí pro překreslování oken apod.

Ve verzi nástrojů 0.61.0, na které byla simulace prováděna, byl konečně zprovozněn profiler, což je nástroj, který umožňuje detailně sledovat chování uvnitř modelu a získávat informace o nejčastěji používaných instrukcích, době provádění instrukcí, úrovni využití částí kódu apod. Tyto informace jsou velice důležité při návrhu ASIP procesorů. Lze díky nim optimalizovat konkrétní instrukce, jejichž provádění zabíralo nejvíce času, optimalizovat úseky kódu, které se nejčastěji provádějí apod.

Porovnávání probíhalo na stroji s těmito parametry:

- Procesor Intel Core 2 Duo T7200 2.00 GHz, 4MB L2 Cache, FSB 667 MHz
- Paměť 2 GB, 667 MHz
- Operační systém Windows 7, 32 bit

Pro porovnání jsem vytvořil několik programů využívajících moji matematickou knihovnu, která byla uvedena v kapitole 7.1. Simulaci jsem provedl na několika vzorcích programu. Každý vzorek jsem několikrát odsimuloval a celkový čas potřebný pro simulaci jsem zprůměroval.

U mého vygenerovaného simulátoru jsem navíc vyzkoušel rychlost simulace, pokud byly/nebyly simulovány i externí periferie (čítače/časovače, sériová linka, přerušení).

Pro porovnání simulátorů jsem využil tyto testovací programy:

1. Jednoduchý program se třemi zanořenými cykly, které provedou $256 \cdot 256 \cdot 256 = 16777216$ skoků. Příloha A.1
2. Fibonacciho posloupnost z čísla 51455 se šířkou čísel 8 bajtů. Příloha A.2
3. 100x výpočet faktoriálu z čísla 15 a šířkou čísel 16 bajtů. Příloha A.3
4. Obsluha časovačů 0 a 1 a využití přerušení. Časovač 0 inkrementuje registr R0, při jeho přetečení inkrementuje registr R1. Časovač 1 kontroluje, zda registr R1 je roven 0. Pokud je roven 0, program skončí. Příloha A.4

7.3 Vyhodnocení simulace

Jako výchozí hodnotu pro porovnání výsledků simulace z tabulky 7.1 používám vygenerovaný simulátor se simulací periferií.

simulace	EASY51	ADSIM	bez periferií	s periferiemi		profiler s perif.	
	[s]	[s]	[s]	[s]	[MHz]	[s]	[MHz]
1	116	158	31	57,10	7,3	7558	0,05
2	526	145	20	34	6,8	4039	0,06
3	235	123	17	32	6,5	3327	0,06
4	96	62	-	19	8,4	1987	0,08
průměr	243,25	122	22,67	35,53	7,25	4227	0,06
rychlost	0,15	0,29	1,57	1		0,0084	

Tabulka 7.1: Porovnání simulátorů

7.3.1 Porovnání simulátorů

Z výsledků je patrné, že vygenerovaný simulátor mého modelu je z porovnávaných simulátorů nejrychlejší. Pokud simulátor nesimuluje periferie čítač/časovač, sériová linka a přerušení, pak je rychlost simulace průměrně $1,57\times$ rychlejší.

Nejhůře dopadl simulátor EASY51. Ten dosahuje 15% základní průměrné rychlosti, takže je cca $7\times$ pomalejší. Toto zpomalení je dle mého názoru z větší části způsobeno tím, že se během simulace na obrazovce neustále přepisovaly obsahy registrů. Toto chování bohužel nelze ovlivnit.

O něco lépe dopadl simulátor ADSIM. Ten dosahuje 29% základní průměrné rychlosti, takže je cca $3,4\times$ pomalejší.

Zajímavé je, že na prvním testovaném programu (jednoduché zanořené cykly) dosahoval simulátor EASY51 vyšších rychlostí než simulátor ADSIM.

Průměrná frekvence simulátoru modelu s periferiemi byla 8,4 MHz. Vzhledem k tomu, že pracovní frekvence mikrokontroléru 8051 se pohybuje v rozmezí 6 - 24 MHz, je tento výsledek poměrně slušný. Dá se říct, že simulátor je tento model schopen simulovat v reálné rychlosti.

7.3.2 Simulace s profilerem

Simulace modelu se zapnutým profilerem byla značně pomalá. Vykonání programu trvalo průměrně 120 minut! Rychlost simulace byla průměrně $119\times$ pomalejší než simulace bez profileru, což odpovídá frekvenci procesoru 0,06 MHz. Optimalizovat program při této rychlosti simulace by tak bylo dosti obtížné.

Výsledky simulace však splnily má očekávání. V prvním testovaném programu profiler neukázal nic neobvyklého. Zajímavější informace jsem obdržel až ve druhém a třetím testovaném programu, kde se ani jednou neprovedlo cca 60% celého kódu. Matematická knihovna, kterou jsem v těchto příkladech použil obsahovala mnoho funkcí, které nebyly využity. Navíc se při vykonávání programu využilo pouze 20% instrukcí z celé instrukční sady.

Zřejmě nejzajímavější informací však je, že jedny z nejpoužívanějších instrukcí jsou instrukce PUSH a POP. Testovaná knihovna není příliš optimalizována pro rychlost a každá volaná funkce si nejdříve na zásobník uloží všechny registry, jejichž hodnoty bude měnit. To značným způsobem zpomaluje vykonávání programu, a pokud bychom požadovali zrychlení aplikace, bylo by vhodné se zaměřit na optimalizaci tohoto problému.

V obou případech pak z hlediska zpoždění vede instrukce DJNZ (cca 18%). Zrychlení

celé aplikace by tedy bylo možné dosáhnout i optimalizací instrukční sady, zejména pak optimalizací instrukce DJNZ.

Ve čtvrtém programu pak může být překvapivé, že se nevyužilo celkem 53% programu, i když program se ve skutečnosti provádí celý. Důvodem jsou „prázdná místa“ na začátku programu, kam se skáče při obsluze přerušení. Na těchto místech je vždy pouze instrukce skoku na obsluhu přerušení a zbytek je prázdný.

Profiler ukázal mnoho zajímavých informací, které by pomohly při optimalizaci procesoru pro cílovou aplikaci. To je velice důležité při návrhu a ladění ASIP procesorů, není to však cílem této práce. Kompletní výstup z profileru je v příloze C.

Kapitola 8

Závěr

V jazyce ISAC se mi povedlo vytvořit kompletní model mikrokontroléru 8051 založený na cyklech. Je to první model založený na cyklech, jenž byl v tomto prostředí vytvořen. Veškeré modely byly doposud modelovány pouze na úrovni instrukcí. Díky tomu jsem ověřil mnoho aspektů, které dosud nebyly důkladně ověřeny.

Během vytváření modelu v jazyce ISAC jsem objevil mnoho chyb v generovaných nástrojích. Jejich nahlášením do systému pro hlášení chyb [1] jsem napomohl k zlepšení stability nástrojů a celého vývojového prostředí.

Také jsem odhalil několik slabin v návrhu jazyka ISAC. Především to byla nepřipravenost na modely založené na cyklech. Překladač nezvládal aktivace operací s tak vysokým zpožděním, jaké je u těchto modelů potřeba.

Dále nebylo možné dekódovat instrukce, jejichž šířka operačního kódu byla širší, než je šířka paměti programu. Také nebylo možné popsat některé instrukce, které vyžadovaly složitější mapování jazyka symbolických adres na strojový kód.

Všechny tyto problémy vedly k razantní změně v jazyce ISAC. Došlo k úpravě sekce CODINGROOT a zavedení nové sekce STRUCTURE, které nyní umožňují zvlášť popsat mapování assembleru na strojový kód a použité dekodéry pro jednotlivé části operačního kódu.

Porovnání simulace prokázalo, že vygenerovaný simulátor je v porovnání s ostatními testovanými simulátory mnohem rychlejší a rychlost simulace na testovaném PC téměř dosahovala rychlosti reálného mikrokontroléru. Také jsem vyzkoušel simulátor s aktivovaným profilerem a výstupní data z něj analyzoval.

Komentáře ve zdrojovém kódu jsem psal v souladu s bakalářskou prací „Generátor manuálu instrukční sady“ [8]. Cílem této práce bylo vytvořit generátor manuálu zdrojů a instrukční sady modelů. V současné chvíli lze generovat manuál pouze pro zdroje. Vygenerovaný manuál se nachází na přiloženém CD (viz B).

Pro modelování mikrokontrolérů by bylo vhodné mít možnost v jazyce ISAC definovat rozhraní mikrokontrolérů (vstupně/výstupní porty). To bohužel v současné době není možné. Pokud by se navíc do vývojového prostředí přidala možnost navrhovat vlastní externí obvody (například klávesnice, LCD displej apod.) a tyto obvody prezentovat v grafickém uživatelském prostředí, bylo by možné simulovat model, jako by byl zapojen do konkrétní aplikace. Podobné možnosti některé komerční simulátory nabízejí.

Literatura

[1] Bugzilla. http://lissom.aps-brno.cz:8666, systém pro hlášení chyb.

[2] Atmel: Atmel 8051 Microcontrollers Hardware Manual. http://www.atmel.com/dyn/resources/prod_documents/doc4316.pdf, datasheet.

[3] Babák, M.; Chládek, L.: *Architektura a technické vlastnosti jednočipových mikrořadičů 8051*. 1993, iSBN 80-7102-031.

[4] Fronc, V.: *Mikrokontroléry ATMEL s jádrem 8051*. Nakladatelství BEN, 1999, iSBN 80-7300-008-2.

[5] Hoffmann, A.; Meyer, H.; Leupers, R.: *Architecture Exploration for Embedded Processors with LISA*. Kluwer Academic Publishers, 2002, iSBN 1-4020-7338-0.

[6] Kotásek, Z.: Materiály k předmětu IPZ [online]. https://www.fit.vutbr.cz/study/courses/IPZ/public/, 2010 [cit. 2010-04-15].

[7] Kotásek, Z.: Materiály k předmětu ITP [online]. https://www.fit.vutbr.cz/study/courses/ITP/public/, 2010 [cit. 2010-04-15].

[8] Křen, M.: *Generátor manuálu instrukční sady*. FIT VUT v Brně, 2010, bakalářská práce.

[9] Lissom: ISAC Language Manual, interní dokumentace projektu.

[10] Masařík, K.: Jazyky pro popis architektury počítačových systémů. 2010 [cit. 2010-04-17], materiály k předmětu IPP.

[11] Novotný, T.: *Transformace popisného jazyka mikroprocesoru do jazyka pro popis hardware*. FIT VUT v Brně, 2007, diplomová práce.

[12] Skalický, P.: *Mikroprocesory řady 8051 / 2. rozš. vyd.* BEN - technická literatura, 1998, iSBN 80-86056-39-2.

[13] Vacek, V.: *Učebnice programování ATMEL s jádrem 8051*. BEN - technická literatura, 2001, iSBN 80-7300-043-1.

[14] Wikipedia: Intel 8051. http://cs.wikipedia.org/wiki/Intel_8051, [cit. 2010-04-16].

Dodatek A

Testovací programy pro simulaci

A.1 Jednoduché zanořené cykly

```
        mov R7,#0
        mov R6,#0
        mov R5,#0
skok:
        djnz R7,skok
        djnz R6,skok
        djnz R5,skok
        halt
```

A.2 Fibonacciho posloupnost

```
.equiv TEMP,0x60
.equiv DELKA,8
.equiv PRVNI,0x30
.equiv DRUHY,0x40
.equiv TRETI,0x50
.equiv VYSL,0x20

        mov R0,#PRVNI
        mov R1,#DRUHY
        mov R2,#VYSL
        mov R3,#TRETI
        mov TRETI,#255
        mov TRETI+1,#200
        mov R7,#DELKA

        lcall fibonaccin
        halt
```

A.3 Faktoriál

```
.equiv TEMP,0x60
```

```
.equiv DELKA,16
.equiv PRVNI,0x30
.equiv DRUHY,0x40
.equiv TRETI,0x50
.equiv VYSL,0x20

        mov R7,#DELKA
        mov R5,#100
skok:
        mov R0,#PRVNI
        lcall clrn
        mov PRVNI,#15
        mov R1,#DRUHY
        mov R2,#VYSL
        lcall factorialn
        djnz R5,skok
        halt
```

A.4 Obsluha časovačů s obsluhou přerušení

```
.start:
        ljmp zacatek

.section intvec0, 8, "x"
.org 0x0B
        ljmp timer0

.section intvec1, 8, "x"
.org 0x1B
        ljmp timer1

.section code, 8, "x" 'tahle sekce se uz umisti podle linkeru tak, kde bude misto
zacatek:
        mov IE,#0xFF    'enable all interrupts
        mov TL0,#0x40
        mov TH0,#0x40
        mov TL1,#0x90
        mov TH1,#0x90
        mov TMOD,#0x22    'samoplnici rezim
        mov TCON,#0x50    'spustim casovace 0 a 1
        mov R1,#1
pok:    ljmp pok

'obsluha preruseni casovace 0
timer0: inc R0
        mov A,R0
        jz pricti
        reti
```

```
pricti: inc R1
        reti

'obsluha preruseni casovace 1
timer1: mov A,R1
        jz konec
        reti
konec:  halt
```

Dodatek B

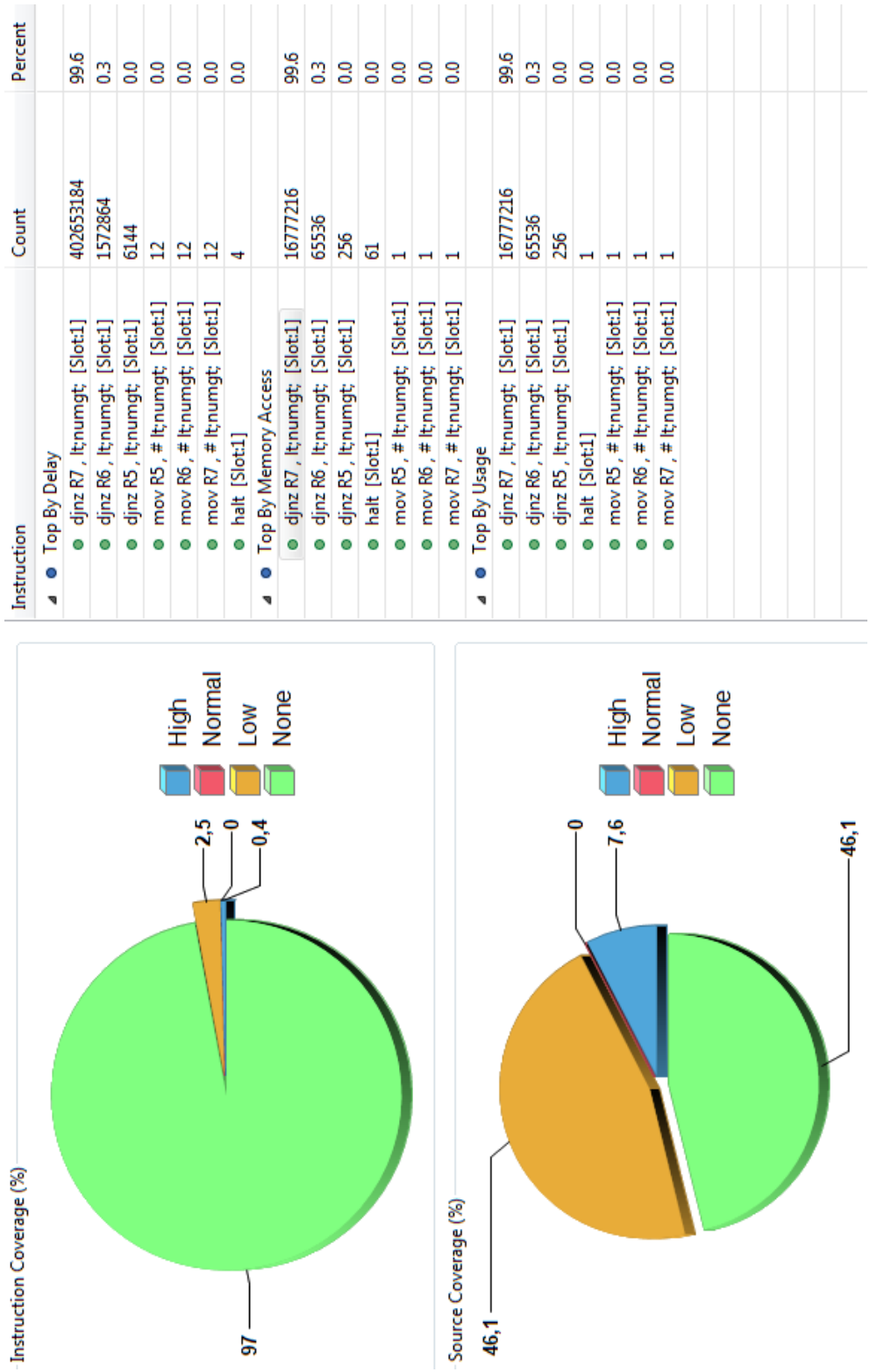
Obsah CD

- `8051ca`
  - `model`
    - `8051.isac` - model mikrokontroléru
    - `8051_defs.h` - hlavičkový soubor s definicemi adres registrů apod.
    - `8051_utils.cpp` - knihovna s implementací ALU, paměťového dekodéru, čítačů/časovačů, sériové linky a obsluhy přerušení
    - `8051_utils.h` - hlavičkový soubor knihovny
  - `src`
    - `math.asm` - matematická knihovna vytvořená v assembleru pro testování a simulaci modelu
    - `priklad1.asm` - 1. testovaný program v assembleru
    - `priklad2.asm` - 2. testovaný program v assembleru
    - `priklad3.asm` - 3. testovaný program v assembleru
    - `priklad4.asm` - 4. testovaný program v assembleru
- `mikroprogramy` - adresář obsahuje dokumenty s návrhem mikroprogramů pro instrukce
  - `mikroprogramy.ods` - OpenOffice Calc formát
  - `mikroprogramy.pdf` - PDF formát
- `thesis`
  - `src` - adresář se zdrojovým kódem technické zprávy v LaTeXu
  - `thesis.pdf` - technická zpráva ve formátu PDF (tento dokument)
- `gen_doc.rtf` - vygenerovaný manuál zdrojů modelu
- `instalace.txt` - návod k instalaci a zprovoznění prostředí pro otestování modelu

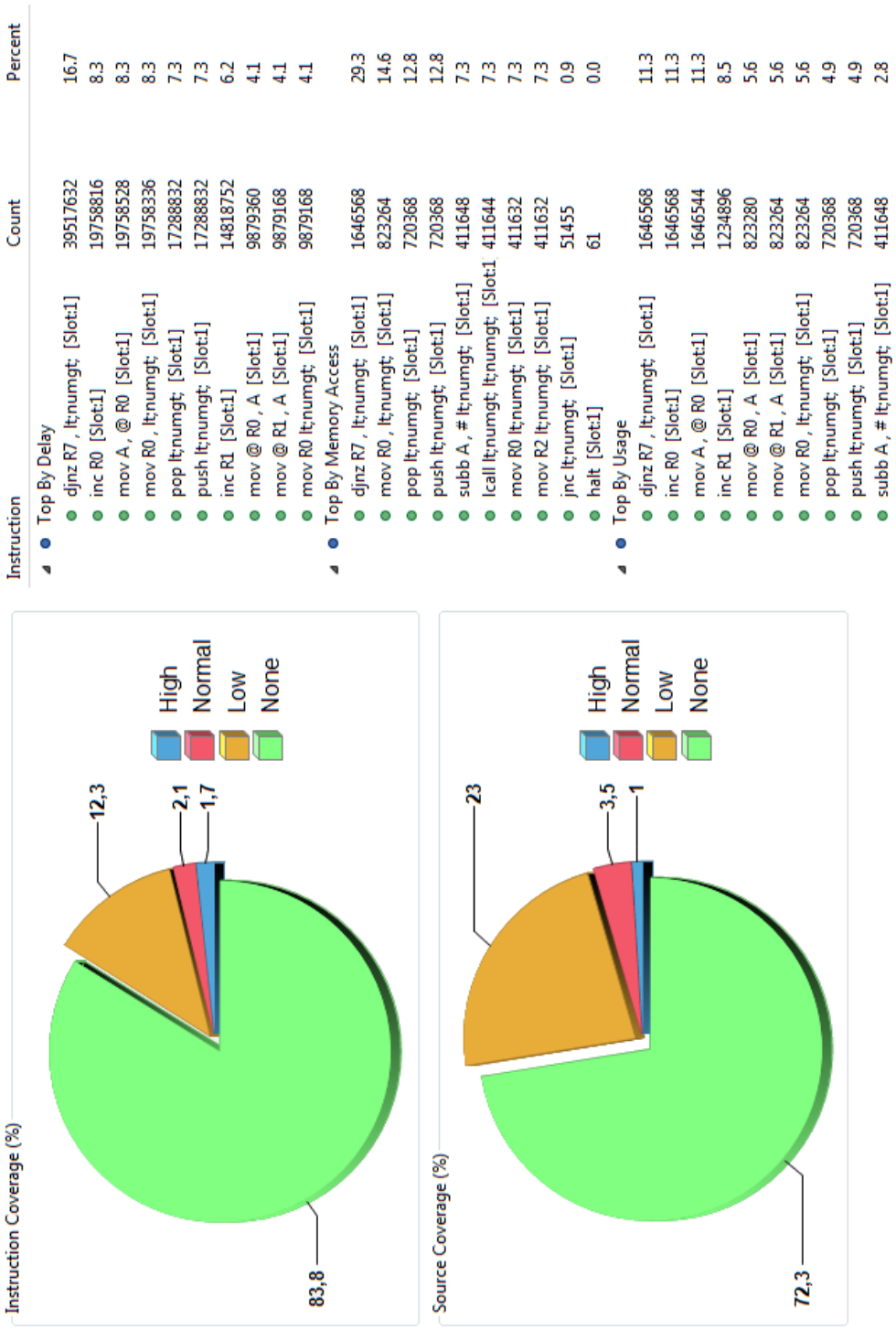
Dodatek C

Výstup z profileru

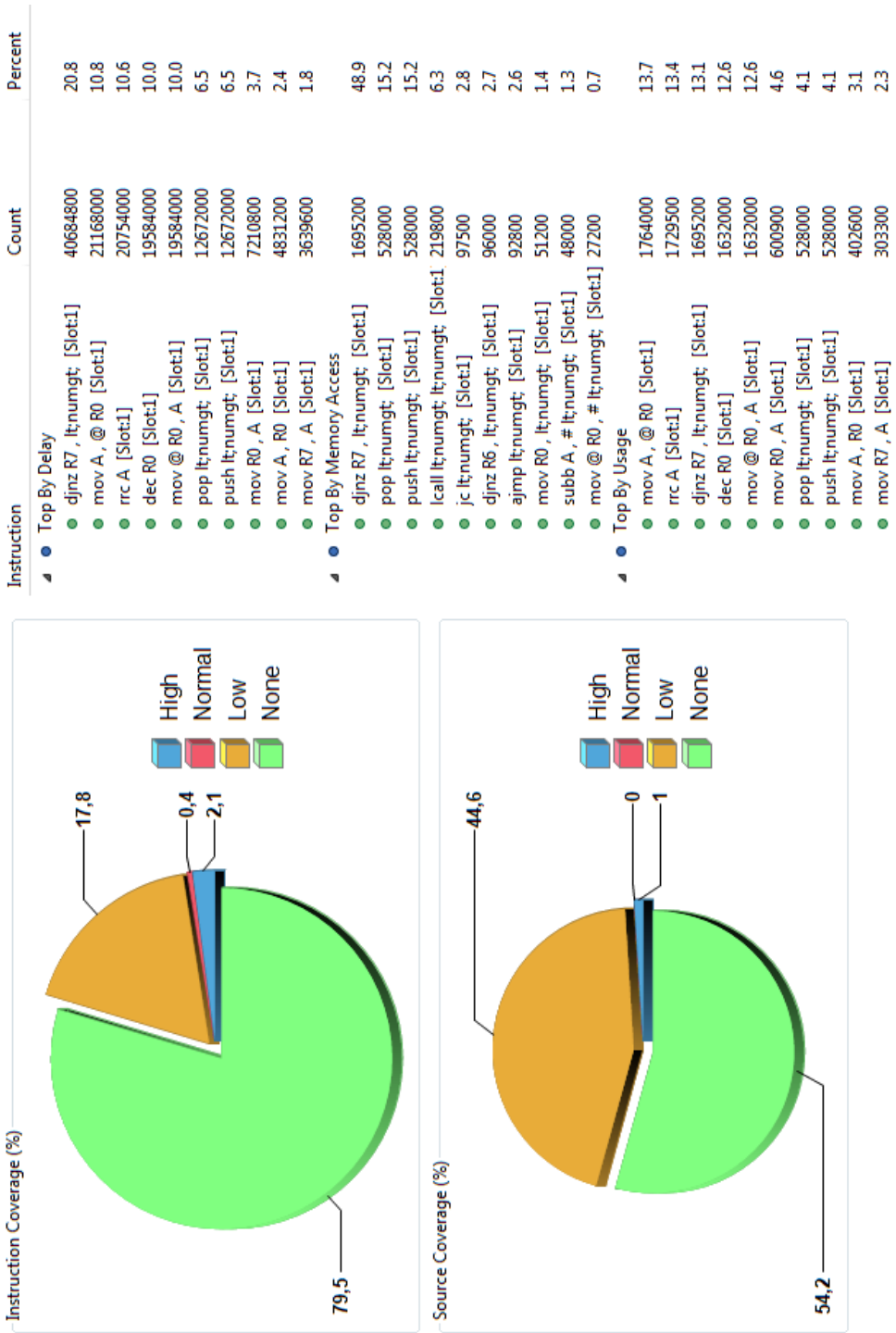
C.1 Příklad 1


Instruction
Count
Percent
Top By Delay
djnz R7, lt;numgt; [Slot:1] 402653184 99.6
djnz R6, lt;numgt; [Slot:1] 1572864 0.3
djnz R5, lt;numgt; [Slot:1] 6144 0.0
mov R5, # lt;numgt; [Slot:1] 12 0.0
mov R6, # lt;numgt; [Slot:1] 12 0.0
mov R7, # lt;numgt; [Slot:1] 12 0.0
halt [Slot:1] 4 0.0
Top By Memory Access
djnz R7, lt;numgt; [Slot:1] 16777216 99.6
djnz R6, lt;numgt; [Slot:1] 65536 0.3
djnz R5, lt;numgt; [Slot:1] 256 0.0
halt [Slot:1] 61 0.0
mov R5, # lt;numgt; [Slot:1] 1 0.0
mov R6, # lt;numgt; [Slot:1] 1 0.0
mov R7, # lt;numgt; [Slot:1] 1 0.0
Top By Usage
djnz R7, lt;numgt; [Slot:1] 16777216 99.6
djnz R6, lt;numgt; [Slot:1] 65536 0.3
djnz R5, lt;numgt; [Slot:1] 256 0.0
halt [Slot:1] 1 0.0
mov R5, # lt;numgt; [Slot:1] 1 0.0
mov R6, # lt;numgt; [Slot:1] 1 0.0
mov R7, # lt;numgt; [Slot:1] 1 0.0
Instruction Coverage (%)
High
Normal
Low
None
2,5
0
0,4
97
Source Coverage (%)
High
Normal
Low
None
0
7,6
46,1
46,1

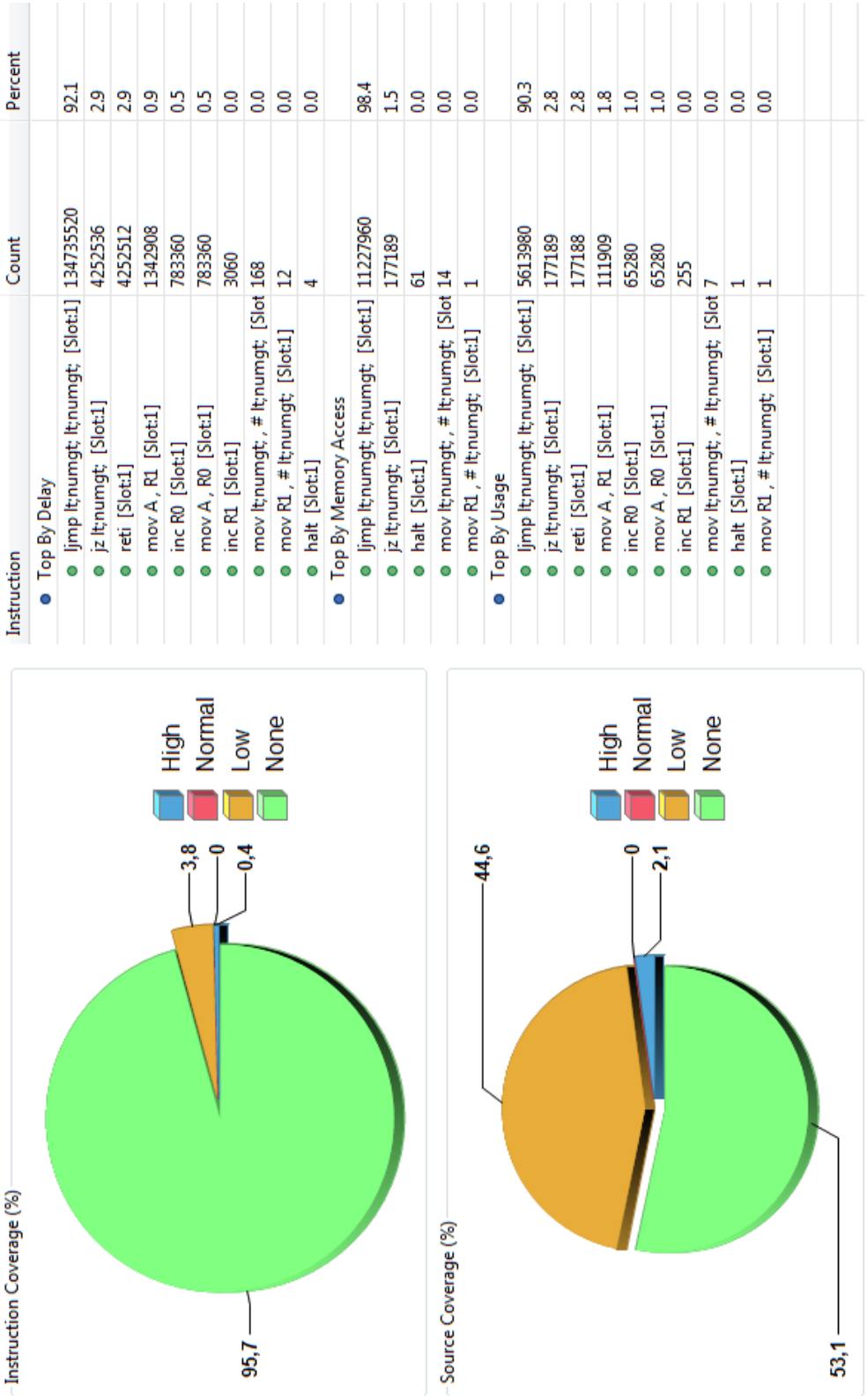
C.2 Příklad 2


Instruction
Count
Percent
Top By Delay
djnz R7 , lt;numgt; [Slot:1] 39517632 16.7
inc R0 [Slot:1] 19758816 8.3
mov A , @ R0 [Slot:1] 19758528 8.3
mov R0 , lt;numgt; [Slot:1] 19758336 8.3
pop lt;numgt; [Slot:1] 17288832 7.3
push lt;numgt; [Slot:1] 17288832 7.3
inc R1 [Slot:1] 14818752 6.2
mov @ R0 , A [Slot:1] 9879360 4.1
mov @ R1 , A [Slot:1] 9879168 4.1
mov R0 lt;numgt; [Slot:1] 9879168 4.1
Top By Memory Access
djnz R7 , lt;numgt; [Slot:1] 1646568 29.3
mov R0 , lt;numgt; [Slot:1] 823264 14.6
pop lt;numgt; [Slot:1] 720368 12.8
push lt;numgt; [Slot:1] 720368 12.8
subb A , # lt;numgt; [Slot:1] 411648 7.3
lcall lt;numgt; lt;numgt; [Slot:1 411644 7.3
mov R0 lt;numgt; [Slot:1] 411632 7.3
mov R2 lt;numgt; [Slot:1] 411632 7.3
jnc lt;numgt; [Slot:1] 51455 0.9
halt [Slot:1] 61 0.0
Top By Usage
djnz R7 , lt;numgt; [Slot:1] 1646568 11.3
inc R0 [Slot:1] 1646568 11.3
mov A , @ R0 [Slot:1] 1646544 11.3
inc R1 [Slot:1] 1234896 8.5
mov @ R0 , A [Slot:1] 823280 5.6
mov @ R1 , A [Slot:1] 823264 5.6
mov R0 , lt;numgt; [Slot:1] 823264 5.6
pop lt;numgt; [Slot:1] 720368 4.9
push lt;numgt; [Slot:1] 720368 4.9
subb A , # lt;numgt; [Slot:1] 411648 2.8
Instruction Coverage (%)
High
Normal
Low
None
12,3
2,1
1,7
83,8
Source Coverage (%)
High
Normal
Low
None
23
3,5
1
72,3

C.3 Příklad 3


Instruction
Count
Percent
Top By Delay
djnz R7 , lt;numgt; [Slot:1] 40684800 20.8
mov A , @ R0 [Slot:1] 21168000 10.8
rrc A [Slot:1] 20754000 10.6
dec R0 [Slot:1] 19584000 10.0
mov @ R0 , A [Slot:1] 19584000 10.0
pop lt;numgt; [Slot:1] 12672000 6.5
push lt;numgt; [Slot:1] 12672000 6.5
mov R0 , A [Slot:1] 7210800 3.7
mov A , R0 [Slot:1] 4831200 2.4
mov R7 , A [Slot:1] 3639600 1.8
Top By Memory Access
djnz R7 , lt;numgt; [Slot:1] 1695200 48.9
pop lt;numgt; [Slot:1] 528000 15.2
push lt;numgt; [Slot:1] 528000 15.2
lcall lt;numgt; lt;numgt; [Slot:1 219800 6.3
jc lt;numgt; [Slot:1] 97500 2.8
djnz R6 , lt;numgt; [Slot:1] 96000 2.7
ajmp lt;numgt; [Slot:1] 92800 2.6
mov R0 , lt;numgt; [Slot:1] 51200 1.4
subb A , # lt;numgt; [Slot:1] 48000 1.3
mov @ R0 , # lt;numgt; [Slot:1] 27200 0.7
Top By Usage
mov A , @ R0 [Slot:1] 1764000 13.7
rrc A [Slot:1] 1729500 13.4
djnz R7 , lt;numgt; [Slot:1] 1695200 13.1
dec R0 [Slot:1] 1632000 12.6
mov @ R0 , A [Slot:1] 1632000 12.6
mov R0 , A [Slot:1] 600900 4.6
pop lt;numgt; [Slot:1] 528000 4.1
push lt;numgt; [Slot:1] 528000 4.1
mov A , R0 [Slot:1] 402600 3.1
mov R7 , A [Slot:1] 303300 2.3
Instruction Coverage (%)
High
Normal
Low
None
17,8
0,4
2,1
79,5
Source Coverage (%)
High
Normal
Low
None
44,6
0
1
54,2

C.4 Příklad 4


Instruction
Count
Percent
Top By Delay
ljmp lt;numgt; lt;numgt; [Slot:1] 134735520 92.1
jz lt;numgt; [Slot:1] 4252536 2.9
reti [Slot:1] 4252512 2.9
mov A , R1 [Slot:1] 1342908 0.9
inc R0 [Slot:1] 783360 0.5
mov A , R0 [Slot:1] 783360 0.5
inc R1 [Slot:1] 3060 0.0
mov lt;numgt; , # lt;numgt; [Slot 168 0.0
mov R1 , # lt;numgt; [Slot:1] 12 0.0
halt [Slot:1] 4 0.0
Top By Memory Access
ljmp lt;numgt; lt;numgt; [Slot:1] 11227960 98.4
jz lt;numgt; [Slot:1] 177189 1.5
halt [Slot:1] 61 0.0
mov lt;numgt; , # lt;numgt; [Slot 14 0.0
mov R1 , # lt;numgt; [Slot:1] 1 0.0
Top By Usage
ljmp lt;numgt; lt;numgt; [Slot:1] 5613980 90.3
jz lt;numgt; [Slot:1] 177189 2.8
reti [Slot:1] 177188 2.8
mov A , R1 [Slot:1] 111909 1.8
inc R0 [Slot:1] 65280 1.0
mov A , R0 [Slot:1] 65280 1.0
inc R1 [Slot:1] 255 0.0
mov lt;numgt; , # lt;numgt; [Slot 7 0.0
halt [Slot:1] 1 0.0
mov R1 , # lt;numgt; [Slot:1] 1 0.0
Instruction Coverage (%)
High
Normal
Low
None
3,8
0
0,4
95,7
Source Coverage (%)
High
Normal
Low
None
44,6
0
2,1
53,1

Dodatek D

Mikroprogramy pro instrukce

					S1		S2		S3		S4		S5		S6		S1		S2		S3		S4		S5		S6	
		bajty	takty	příznaky	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2	P1	P2
					1	2	3	4	5	6	7	8	9	10	11	12	1	2	3	4	5	6	7	8	9	10	11	12
ACALL	addr11	2	2			F	D	cache2=adrOp		ALU_inC=1 ALU_in1=0 ALU_in2=SP	ALUadd	cache1=PMEM		SP=ALU_out		ALU_in2=SP		AR=SP		RAM=PC_low	ALUadd	SP=ALU_out		AR=SP		RAM=PChigh TMP_bit=1		TMP_bit? PC=CACHE
ADD	A,Rn	1	1	C, AC, OV, P		F	D	AR=Rn						ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADD	A,direct	2	1	C, AC, OV, P		F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADD	A,@Ri	1	1	C, AC, OV, P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADD	A,#data	2	1	C, AC, OV, P		F	D					cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADDC	A,Rn	1	1	C, AC, OV, P		F	D	AR=Rn						ALU_inC=C ALU_in1=RAM ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADDC	A,direct	2	1	C, AC, OV, P		F	D					AR=PMEM		ALU_inC=C ALU_in1=RAM ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADDC	A,@Ri	1	1	C, AC, OV, P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=C ALU_in1=RAM ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
ADDC	A,#data	2	1	C, AC, OV, P		F	D					cache2=PMEM		ALU_inC=C ALU_in1=cache2 ALU_in2=A	ALUadd	A=ALU PSW=C,AC,OV,P												
AJMP	addr11	2	2			F	D	cache2=adrOp				cache1=PMEM														TMP_bit=1		TMP_bit? PC=CACHE
ANL	A,Rn	1	1	P		F	D	AR=Rn						ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUand	A=ALU PSW=P												
ANL	A,direct	2	1	P		F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUand	A=ALU PSW=P												
ANL	A,@Ri	1	1	P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUand	A=ALU PSW=P												
ANL	A,#data	2	1	P		F	D					cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=A	ALUand	A=ALU PSW=P												
ANL	direct,A	2	1			F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUand	RAM=ALU												
ANL	direct,#data	3	2			F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM				cache2=PMEM								ALU_in2=cache2	ALUand	RAM=ALU_out
ANL	C,bit	2	2	C		F	D					AR(bit)=PMEM ALU_rot=PMEM& 0x7														ALU_inC=C ALU_in1=RAM	ALUbitand	PSW=C
ANL	C,/bit	2	2	C		F	D					AR(bit)=PMEM ALU_rot=PMEM& 0x7		ALU_in1=RAM	ALUbitnot	cache2=ALU										ALU_inC=C ALU_in1=cache2	ALUbitand	PSW=C
CJNE	A,direct,rel	3	2	C		F	D	ALU_in2=0	ALUnot			AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=ACC cache1=ALU_out	ALUsubb	C=ALU_outC ALU_in1=cache1 ALU_in2=ALU_out	ALUadd	TMP_bit=ALU_outC cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
CJNE	A,#data,rel	3	2	C		F	D	ALU_in2=0	ALUnot			cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=ACC cache1=ALU_out	ALUsubb	C=ALU_outC ALU_in1=cache1 ALU_in2=ALU_out	ALUadd	TMP_bit=ALU_outC cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
CJNE	Rn,#data,rel	3	2	C		F	D	AR=Rn ALU_in2=0	ALUnot			cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=RAM cache1=ALU_out	ALUsubb	C=ALU_outC ALU_in1=cache1 ALU_in2=ALU_out	ALUadd	TMP_bit=ALU_outC cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
CJNE	@Ri,#data,rel	3	2	C		F	D	AR=Ri ALU_in2=0	ALUnot	cache1=RAM		AR=cache1 cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=RAM cache1=ALU_out	ALUsubb	C=ALU_outC ALU_in1=cache1 ALU_in2=ALU_out	ALUadd	TMP_bit=ALU_outC cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
CLR	A	1	1	P		F	D							ALU_inC=0 ALU_in1=0 ALU_in2=0	ALUadd	A=ALU PSW=P												
CLR	C	1	1	C		F	D							ALU_inC=0 ALU_in1=0 ALU_in2=0	ALUadd	PSW=C												
CLR	bit	2	1			F	D					AR(bit)=PMEM ALU_rot=PMEM& 0x7		ALU_inC=1 ALU_in1=RAM	ALUbitand	RAM=ALU												
CPL	A	1	1			F	D							ALU_in2=A	ALUnot	A=ALU												
CPL	C	1	1	C		F	D							ALU_inC=C	ALUnot	PSW=C												

CPL	bit	2	1			F	D					AR(bit)=PMEM ALU_rot=PMEM&0x7		ALU_in1=RAM	ALUbitnot	RAM=ALU												
DA	A	1	1	C, P		F	D	TMP_C=C ALU_inC=AC ALU_in2=ACC	ALUdaLow	ALU_inC=0 ALU_in1=ALU_out	ALUadd	C=orALU_outC ALU_inC=TMPC ALU_in2=ALU_out	ALUdaHigh	ALU_inC=0 ALU_in1=ALU_out	ALUadd	ACC=ALU_out C=orALU_outC PSW=P												
DEC	A	1	1	P		F	D							ALU_inC=1 ALU_in1=0 ALU_in2=A	ALUsubb	A=ALU PSW=P												
DEC	Rn	1	1			F	D	AR=Rn						ALU_inC=1 ALU_in1=0 ALU_in2=RAM	ALUsubb	RAM=ALU												
DEC	direct	2	1			F	D					AR=PMEM		ALU_inC=1 ALU_in1=0 ALU_in2=RAM	ALUsubb	RAM=ALU												
DEC	@Ri	1	1			F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=1 ALU_in1=0 ALU_in2=RAM	ALUsubb	RAM=ALU												
DIV	AB																										
DJNZ	Rn,rel	2	2			F	D	AR=Rn ALU_in2=0	ALUnot	ALU_in2=ALU_out		cache1=rel		ALU_inC=0 ALU_in1=RAM	ALUadd	RAM=ALU ALU_in1=ALU_out	ALUadd	TMP_bit=ALU_outC		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
DJNZ	direct,rel	3	2			F	D	ALU_in2=0	ALUnot	ALU_in2=ALU_out		AR=PMEM		ALU_inC=0 ALU_in1=RAM	ALUadd	RAM=ALU ALU_in1=ALU_out	ALUadd	TMP_bit=ALU_outC		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
INC	A	1	1	P		F	D							ALU_inC=1 ALU_in1=0 ALU_in2=A	ALUadd	A=ALU PSW=P												
INC	Rn	1	1			F	D	AR=Rn						ALU_inC=1 ALU_in1=RAM ALU_in2=0	ALUadd	RAM=ALU												
INC	direct	2	1			F	D					AR=PMEM		ALU_inC=1 ALU_in1=RAM ALU_in2=0	ALUadd	RAM=ALU												
INC	@Ri	1	1			F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=1 ALU_in1=RAM ALU_in2=0	ALUadd	RAM=ALU												
INC	DPTR	1	2			F	D					AR=&DPL		ALU_inC=1 ALU_in1=RAM ALU_in2=0	ALUadd	DPL=ALU							AR=&DPH		ALU_inC=ALU_outC ALU_in1=RAM ALU_in2=0	ALUadd	DPH=ALU
JB	bit,rel	3	2			F	D					AR(bit)=PMEM ALU_rot=PMEM&0x7		ALU_inC=0 ALU_in1=RAM	ALUbitor	TMP_bit=ALU_outC		cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
JBC	bit,rel	3	2			F	D					AR(bit)=PMEM ALU_rot=PMEM&0x7		ALU_inC=0 ALU_in1=RAM	ALUbitor	TMP_bit=ALU_outC	ALUbitand	RAM=ALU_out cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
JC	rel	2	2			F	D					cache1=rel				TMP_bit=C				ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
JMP	@A+DPTR	1	2			F	D					AR=&DPL		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUadd	TMP_bit=ALU_outC cache1=ALU_out		AR=&DPH		ALU_inC=TMP_bit ALU_in1=RAM ALU_in2=0	ALUadd	cache2=ALU				TMP_bit=1		TMP_bit? PC=CACHE
JNB	bit,rel	3	2			F	D					AR(bit)=PMEM ALU_rot=PMEM&0x7		ALU_inC=0 ALU_in1=RAM	ALUbitor	ALU_inC=ALU_outC	ALUnot	TMP_bit=ALU_outC cache1=PMEM		ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
JNC	rel	2	2			F	D					cache1=rel		ALU_inC=C	ALUnot	TMP_bit=ALU_outC				ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
JNZ	rel	2	2			F	D	ALU_in2=0	ALUnot	ALU_inC=0 ALU_in1=ALU_out ALU_in2=A	ALUadd	cache1=rel				TMP_bit=ALU_outC				ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
JZ	rel	2	2			F	D	ALU_in2=0	ALUnot	ALU_inC=0 ALU_in1=ALU_out ALU_in2=A	ALUadd	cache1=rel		ALU_inC=ALUoutC	ALUnot	TMP_bit=ALU_outC				ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
LCALL	addr16	3	2			F	D	ALU_inC=1 ALU_in1=SP ALU_in2=0	ALUadd	SP=ALU_out		cache2=PMEM		ALU_in1=SP		AR=SP		cache1=PMEM		RAM=PC_low	ALUadd	SP=ALU_out		AR=SP		RAM=PChigh TMP_bit=1		TMP_bit? PC=CACHE
LJMP	addr16	3	2			F	D					cache2=PMEM						cache1=PMEM								TMP_bit=1		TMP_bit? PC=CACHE
MOV	A,Rn	1	1	P		F	D	AR=Rn						ALU_inC=0 ALU_in1=RAM ALU_in2=0	ALUadd	A=ALU PSW=P												

MOV	A,direct	2	1	P		F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=0	ALUadd	A=ALU PSW=P												
MOV	A,@Ri	1	1	P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=0 ALU_in1=RAM ALU_in2=0	ALUadd	A=ALU PSW=P												
MOV	A,#data	2	1	P		F	D					cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=0	ALUadd	A=ALU PSW=P												
MOV	Rn,A	1	1			F	D	AR=Rn								RAM=A												
MOV	Rn,direct	2	2			F	D					AR=PMEM						cache1=RAM		AR=Rn								RAM=cache1
MOV	Rn,#data	2	1			F	D	AR=Rn				cache2=PMEM				RAM=cache2												
MOV	direct,A	2	1			F	D					AR=PMEM				RAM=A												
MOV	direct,Rn	2	2			F	D	AR=Rn		cache1=RAM		cache2=PMEM														AR=cache2		RAM=cache1
MOV	direct,direct	3	2			F	D					AR=PMEM		cache1=RAM				AR=PMEM										RAM=cache1
MOV	direct,@Ri	2	2			F	D	AR=Ri		cache1=RAM		cache2=PMEM		AR=cache1		cache1=RAM										AR=cache2		RAM=cache1
MOV	direct,#data	3	2			F	D					AR=PMEM						cache2=PMEM										RAM=cache2
MOV	@Ri,A	1	1			F	D	AR=Ri		cache1=RAM				AR=cache1		RAM=A												
MOV	@Ri,direct	2	2			F	D					AR=PMEM						cache1=RAM		AR=Rn		cache2=RAM		AR=cache2				RAM=cache1
MOV	@Ri,#data	2	1			F	D	AR=Ri		cache1=RAM		cache2=PMEM		AR=cache1		RAM=cache2												
MOV	C,bit	2	1			F	D					AR(bit)=PMEM ALU_rot=PMEM& 0x7		ALU_inC=0 ALU_in1=RAM	ALUbitor	C=ALU_outC												
MOV	bit,C	2	2			F	D					AR(bit)=PMEM ALU_rot=(PMEM &0x7)		ALU_inC=0 ALU_in1=RAM	ALUbitand	RAM=ALU										ALU_inC=C ALU_in1=RAM	ALUbitor	RAM=ALU_out
MOV	DPTR,#data16	3	2			F	D					cache2=PMEM		AR=DPH		RAM=cache2		cache1=PMEM								AR=DPL		RAM=cache1
MOVC	A,@A+DPTR	1	2	P		F	D	AR=&DPL		ALU_inC=0 ALU_in1=RAM ALU_in2=ACC	ALUadd	cache1=ALU TMPbit=ALUoutC		AR=DPH		ALU_inC=TMPbit ALU_in1=RAM ALU_in2=0	ALUadd	cache2=ALUout						A=PMEM[cache]		ALU_inC=0 ALU_in1=0 ALU_in2=A	ALUadd	A=ALU P=ALUoutP
MOVC	A@A+PC	1	2	P		F	D	cache=PC		ALU_inC=0 ALU_in1=cache1 ALU_in2=ACC	ALUadd	cache1=ALU TMPbit=ALUoutC				ALU_inC=TMPbit ALU_in1=cache2 ALU_in2=0	ALUadd	cache2=ALUout						A=PMEM[cache]		ALU_inC=0 ALU_in1=0 ALU_in2=A	ALUadd	A=ALU P=ALUoutP
MOVX	A,@Ri	1	2	P		F	D	AR=Ri		cache1=RAM		AR=cache1		P0=RAM								ALUinC=0 ALUin1=0 ALUin2=P0	ALUadd					ACC=ALU P=ALUoutP
MOVX	A,@DPTR	1	2	P		F	D							P0=DPL P2=DPH								ALUinC=0 ALUin1=0 ALUin2=P0	ALUadd					ACC=ALU P=ALUoutP
MOVX	@Ri,A	1	2			F	D	AR=Ri		cache1=RAM		AR=cache1		P0=RAM		P0=ACC												
MOVX	@DPTR,A	1	2			F	D							P0=DPL P2=DPH		P0=ACC												
MUL	AB	1	4	C, OV,P		F	D	ALU_in1=B ALU_in2=A	ALUmul					B=ALU_out2		A=ALU PSW=C,P,OV												
NOP		1	1			F	D																					
ORL	A,Rn	1	1	P		F	D	AR=Rn						ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUor	A=ALU PSW=P												
ORL	A,direct	2	1	P		F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUor	A=ALU PSW=P												
ORL	A,@Ri	1	1	P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUor	A=ALU PSW=P												
ORL	A,#data	2	1	P		F	D					cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=A	ALUor	A=ALU PSW=P												
ORL	direct,A	2	1			F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUor	RAM=ALU												
ORL	direct,#data	3	2			F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM				cache2=PMEM								ALU_in2=cache2	ALUor	RAM=ALU_out
ORL	C,bit	2	2	C		F	D					AR(bit)=PMEM ALU_rot=PMEM& 0x7														ALU_inC=C ALU_in1=RAM	ALUbitor	PSW=C
ORL	C,/bit	2	2	C		F	D					AR(bit)=PMEM ALU_rot=PMEM& 0x7		ALU_in1=RAM	ALUbitnot	cache2=ALU										ALU_inC=C ALU_in1=cache2	ALUbitor	PSW=C
POP	direct	2	2			F	D	AR=SP		cache1=RAM		cache2=PMEM		AR=cache2		RAM=cache1										ALU_inC=1 ALU_in1=0 ALU_in2=SP	ALUsubb	SP=ALU_out
PUSH	direct	2	2			F	D					AR=PMEM		cache1=RAM						ALU_inC=1 ALU_in1=0 ALU_in2=SP	ALUadd	SP=ALU_out		AR=SP				RAM=cache1
RET		1	2			F	D	AR=SP		cache2=RAM				ALU_inC=1 ALU_in1=0 ALU_in2=SP	ALUsubb	SP=ALU_out		AR=SP		cache1=RAM		ALU_in2=SP	ALUsubb	SP=ALU_out		TMP_bit=1		TMP_bit? PC=CACHE
RETI																												
RL	A	1	1			F	D			ALU_inC=0 ALU_in2=A	ALUrlc			ALU_inC=ALU_o utC ALU_in2=A	ALUrlc	A=ALU												
RLC	A	1	1	C, P		F	D							ALU_inC=C ALU_in2=A	ALUrlc	A=ALU PSW=C, P												

RR	A	1	1			F	D			ALU_inC=0 ALU_in2=A	ALUrrc			ALU_inC=ALU_outC ALU_in2=A	ALUrrc	A=ALU														
RRC	A	1	1	C, P		F	D							ALU_inC=C ALU_in2=A	ALUrrc	A=ALU PSW=C, P														
SETB	C	1	1	C		F	D							ALU_inC=1 ALU_in1=0 ALU_in2=0	ALUsubb	C=ALU_outC														
SETB	bit	2	1			F	D					AR(bit)=PMEM ALU_rot=PMEM&0x7		ALU_inC=1 ALU_in1=RAM	ALUbitor	RAM=ALU														
SJMP	rel	2	2			F	D	ALU_in2=0	ALUnot			cache1=rel				TMP_bit=ALU_outC				ALU_inC=0 ALU_in1=cache1 ALU_in2=PClow	ALUadd	cache1=ALU		ALU_inC=ALU_outC ALU_in1=cache2 ALU_in2=PChigh	ALUadd	cache2=ALU		TMP_bit? PC=CACHE
SUBB	A,Rn	1	1	C, AC, OV, P		F	D	AR=Rn						ALU_inC=C ALU_in1=RAM ALU_in2=A	ALUsubb	A=ALU PSW=C,AC,OV,P														
SUBB	A,direct	2	1	C, AC, OV, P		F	D					AR=PMEM		ALU_inC=C ALU_in1=RAM ALU_in2=A	ALUsubb	A=ALU PSW=C,AC,OV,P														
SUBB	A,@Ri	1	1	C, AC, OV, P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=C ALU_in1=RAM ALU_in2=A	ALUsubb	A=ALU PSW=C,AC,OV,P														
SUBB	A,#data	2	1	C, AC, OV, P		F	D					cache2=PMEM		ALU_inC=C ALU_in1=cache2 ALU_in2=A	ALUsubb	A=ALU PSW=C,AC,OV,P														
SWAP	A	1	1			F	D							ALU_in2=A	ALUswap	A=ALU														
XCH	A,Rn	1	1	P		F	D	AR=Rn		cache1=RAM ALU_inC=0 ALU_in1=0 ALU_in2=A				ACC=RAM	ALUadd	RAM=ALU_out P=ACCparity														
XCH	A,direct	2	1	P		F	D			ALU_inC=0 ALU_in1=0 ALU_in2=A		AR=PMEM		ACC=RAM	ALUadd	RAM=ALU_out P=ACCparity														
XCH	A,@Ri	1	1	P		F	D	AR=Ri		cache1=RAM ALU_inC=0 ALU_in1=0 ALU_in2=A		AR=cache1		ACC=RAM	ALUadd	RAM=ALU_out P=ACCparity														
XCHD	A,@Ri	1	1	P		F	D	AR=Ri		cache1=RAM ALU_in2=A		AR=cache1		ACC(low)=RAM ALU_in1=RAM	ALUxchd	RAM=ALU_out P=ACCparity														
XRL	A,Rn	1	1	P		F	D	AR=Rn						ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUxor	A=ALU PSW=P														
XRL	A,direct	2	1	P		F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUxor	A=ALU PSW=P														
XRL	A,@Ri	1	1	P		F	D	AR=Ri		cache1=RAM		AR=cache1		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUxor	A=ALU PSW=P														
XRL	A,#data	2	1	P		F	D					cache2=PMEM		ALU_inC=0 ALU_in1=cache2 ALU_in2=A	ALUxor	A=ALU PSW=P														
XRL	direct,A	2	1			F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM ALU_in2=A	ALUxor	RAM=ALU														
XRL	direct,#data	3	2			F	D					AR=PMEM		ALU_inC=0 ALU_in1=RAM				cache2=PMEM								ALU_in2=cache2	ALUxor	RAM=ALU_out