Střední škola informatiky, elektrotechniky a řemesel
Rožnov pod Radhoštěm

MATURITNÍ PRÁCE

16bitový procesor ve VHDL

Dominik Salvet

Obor: 26-47-M/002
Třída: PS4

Školní rok: 2016/2017

Střední škola informatiky, elektrotechniky a řemesel
Rožnov pod Radhoštěm

ZADÁNÍ MATURITNÍ PRÁCE

školní rok	jméno žáka a vlastnoruční podpis	třída	obor
2015/2016	Dominik Salvet	PS4.A	EPS

název práce

16 bitový procesor ve VHDL

zaměření práce

Info / elektro – HW

konkretizace zadání:

Cílem práce je návrh 16 bitového procesoru vlastní architektury a konstrukce a jeho následný popis v jazyce VHDL, tak aby ho bylo možno implementovat do hradlového pole.

jednotlivé cíle práce:

- Prozkoumat architekturu vybraných procesorů (x86, ARM, Alpha),
- navrhnout vlastní architekturu procesoru, ALU a sběrnice procesoru,
- navrhnout způsob organizace paměti a případně způsob ochrany paměti,
- vhodně navrhnout instrukční soubor a registry procesoru,
- popsat vnitřní strukturu procesoru v jazyce VHDL,
- zdokumentovat vytvořený procesor,
- ověřit funkčnost procesoru v hradlovém poli,
- vytvořit jednoduchou aplikaci na ověření funkčnosti procesoru a demonstraci jeho možností.

Další formální náležitosti zadání práce podle platného znění vyhlášky MŠMT č. 177/2009 Sb. jsou k dispozici na www adrese: **http://www.roznovskastredni.cz/maturity**

V Rožnově pod Radhoštěm dne: 15.10.2015

Ing. Jan Pilčík
vedoucí práce

ABSTRAKT

Maturitní práce se zabývá logickým návrhem a popisem 16 bitového procesoru, který je popsán v jazyku VHDL. V této práci budou taky okrajově popsány potřebné základy jazyka VHDL a použitý software. Ve hlavní části se bude práce věnovat zprvu shrnutým popisem jednotlivých populárních architektur procesorů a její největší část bude popisovat samotný návrh architektury, instrukční sady a implementace do VHDL jazyka. Celý návrh procesoru je veden v duchu jednoduchosti. Náplní této maturitní práce je taky implementace souboru VHDL s popisem procesoru do fyzického zařízení – hradlové pole, neboli FPGA. K vypracování maturitní práce je použit výukový kit Basys 2 od firmy Digilent. Tento kit obsahuje hradlové pole Spartan 3E-100 od firmy Xilinx. Basys 2 taky obsahuje osmici přepínačů a LED diod a v neposlední řadě taky čtveřici tlačítek s ošetřením zákmitů.

Klíčová slova: procesor, VHDL, FPGA

Rád bych touto cestou poděkoval vedoucímu mé maturitní práce, panu Ing. Janovi Pilčíkovi, za cenné rady, doporučení a především věnovaný čas takřka po celou dobu vypracovávání této práce. Taky bych chtěl poděkovat rodině za toleranci, trpělivost a důvěru nejen po celou dobu vypracovávání této práce.

Prohlašuji, že odevzdaná verze dokumentace maturitní práce a verze elektronická, nahraná do systému MATPRAC, jsou totožné. Při zpracování jsem vycházel z informačních zdrojů uvedených v seznamu na konci dokumentace a také prohlašuji, že je tato práce původní.

V Rožnově pod Radhoštěm -------------------------

podpis žáka

OBSAH

ÚVOD **8**
I TEORETICKÁ ČÁST **9**
1 PROCESOR **10**
1.1 Popis procesoru 10
1.2 Instrukce procesoru 10
1.3 Hlavní části procesoru 11
1.3.1 Aritmeticko-logická jednotka, matematický koprocesor 11
1.3.2 Soubor registrů 11
1.3.3 Řadič 12
1.3.4 Jednotka správy paměti 12
1.4 Architektura procesoru 13
1.4.1 Popis architektury procesoru 13
1.4.2 Typy architektur procesoru 13
2 VÝZNAMNÉ ARCHITEKTURY PROCESORŮ **14**
2.1 Architektura x86 14
2.1.1 Stručný popis 14
2.1.2 Architektura 14
2.1.3 Soubor registrů 14
2.2 Architektura ARM 16
2.2.1 Stručný popis 16
2.2.2 Architektura 16
2.2.3 Instrukční formáty 17
2.2.4 Soubor registrů 17
2.3 Architektura Alpha 19
2.3.1 Instrukční formáty 19
2.3.2 Soubor registrů 19
3 VHDL **20**
3.1 Stručný popis VHDL 20
3.2 Teoretické základy jazyka VHDL 20
3.2.1 Předmluva 20
3.2.2 Příklad 1 – úplná sčítačka (kombinační obvod) 20
3.2.3 Příklad 2 – 4bitový registr (sekvenční obvod) 22
II PRAKTICKÁ ČÁST **24**
4 POUŽITÝ SOFTWARE **25**
4.1 Ubuntu 14.04 25
4.2 Xilinx ISE Webpack 14.7 26
4.3 Digilent Adept 2 27
4.3.1 Popis programu Digilent Adept 2 27
4.3.2 Ovládání programu Digilent Adept 2 27
5 POUŽITÝ HARDWARE **28**
5.1 Digilent Basys 2 28
5.1.1 Popis vývojového kitu Digilent Basys 2 28

5.1.2 Fyzická výbava vývojového kitu Digilent Basys 2 ... 29
6 PROCESOR LIMEN ... 30
6.1 STRUČNÝ POPIS ... 30
6.2 SOUBOR REGISTRŮ ... 30
6.3 INSTRUKČNÍ SADA ... 30
6.3.1 Popis instrukční sady ... 30
6.3.2 Mnemonika instrukcí ... 31
6.4 INSTRUKČNÍ FORMÁTY ... 32
6.4.1 Předmluva k instrukčním formátům ... 32
6.4.2 Výčet instrukčních formátů ... 33
6.4.3 Aritmetika s přímou hodnotou (kladnou) ... 35
6.4.4 Aritmetika s přímou hodnotou (zápornou) ... 36
6.4.5 Logika s přímou hodnotou ... 37
6.4.6 Aritmetika a logika s registry ... 39
6.4.7 Načítaní přímé hodnoty ... 40
6.4.8 Podmíněné skoky s odsazením ... 41
6.4.9 Nepodmíněné skoky s odsazením ... 42
6.4.10 Nepodmíněné skoky s registrem ... 43
6.5 VÝZNAMNÉ MODULY PROCESORU LIMEN ... 45
6.5.1 Schéma jádra ... 45
6.5.2 Aritmeticko-logická jednotka (ALU) ... 46
6.5.3 Tester podmínek (Con) ... 47
6.5.4 Rozšiřovač přímých hodnot (Ext) ... 48
6.6 POPIS PRÁCE JÁDRA PROCESORU LIMEN ... 49
6.6.1 Časování ... 50
6.6.2 Řídící hodinové signály ... 50
6.6.3 Příklad ... 51
6.7 IMPLEMENTACE DŮLEŽITÝCH PROGRAMOVACÍCH KONSTRUKCÍ ... 53
6.7.1 Příznak přenosu ... 53
6.7.2 Podmíněné skoky ... 54
ZÁVĚR ... 55
SEZNAM POUŽITÉ LITERATURY ... 57
SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK ... 59
SEZNAM OBRÁZKŮ A TABULEK ... 61
SEZNAM PŘÍLOH ... 63

ÚVOD

Zpracovávané téma, 16bitový procesor, jsem si vybral, protože mě baví navrhovat, optimalizovat a vymýšlet logické obvody v jakékoli formě. Toto téma jsem zvolit taky z důvodu, že se v budoucnu budu zabývat návrhem procesoru. Není proto divu, že zpracovávané téma mě velmi obohatilo, protože vím, čemu se mám v budoucnu vyhýbat i do čeho mám investovat více mé pozornosti.

K realizaci implementace samotného procesoru jsem využil programovací jazyk VHDL. Při návrhu procesoru jsem kladl důraz především na jednoduchost a pravidelnost. Obě tyto vlastnosti pak podporují navyšování výkonu celkového obvodu. Navzdory tomu, že tento výkon není zde požadován, je dobrým zvykem takto navrhovat jakýkoli obvod, protože složitější obvod lze získat spojením dvou jednoduchých obvodů, ale jednoduchý obvod nelze získat spojením dvou libovolných obvodů, které k tomu při vývoji nebyly speciálně uzpůsobeny.

V této práci se zprvu věnuji teoretickým základům rozebírané tématiky, jako je stručný popis procesoru, charakteristika významných procesorů a velmi okrajově jsou zde popsány na příkladech i základy jazyka VHDL. V druhé části této práce se věnuji použitému softwaru a hardwaru a samotnému popisu a návrhu procesoru. Pro účely této práce jsem procesor nazval „Limen“.

I. TEORETICKÁ ČÁST

1 PROCESOR

1.1 Popis procesoru

Procesor (mnohdy označován CPU) je nejdůležitějším obvodem ve výpočetních systémech nejen obecného použití. Vykonává strojové instrukce, jimiž je tvořen program. Každý procesor definuje vlastní množinu instrukcí (instrukční sada), které podporuje, a ty pak mohou programy využívat k realizaci požadovaných algoritmů. Z předešlého zcela vyplývá, že pokud program využívá strojové instrukce procesoru P1, nemůže ekvivalentně běžet na procesoru P2, pokud tento procesor neimplementuje alespoň celou množinu instrukcí procesoru P1.

Snad všechny procesory obecného použití mají jeden společný cíl – dosáhnout turingovské úplnosti. Turingovsky úplný stroj je takový stroj, který je schopen realizovat jakýkoli algoritmus, který má řešení konečném čase. Samotná turingovská úplnost ovšem nic neříká o efektivitě provedení algoritmu. V definici této úplnosti je ještě uvedeno, že je požadována v podstatě nekonečná paměť, což je vzhledem k doposud známým fyzikálním omezením problém. Možná i to je důvod, proč se tento fakt často opomíjí. Nicméně i v této práci bude turingovská úplnost použita bez uvážení požadavku nekonečné paměti.

1.2 Instrukce procesoru

Instrukce procesoru by měli být navrženy tak, aby dokázaly implementovat co nejefektivněji nejpoužívanější algoritmy. Z důvodů velkého množství programovacích jazyků, technik a faktu, že mnohdy se používají jednoduché algoritmy, by byla pravděpodobně chyba implementovat jako instrukci procesoru operaci, která je specifická jen pro nějaký programovací jazyk, techniku nebo algoritmus. Vedlejším následkem takového návrhu by byl také o poznání složitější výsledný obvod. Ze všech uvedených důvodů se bez nějakých výjimek vždy používají instrukce, které zpravidla mají minimální podíl na provedení celkového algoritmu. Takový způsob se totiž jeví jako vysoce efektivní na hardwarové úrovni a je možno do procesoru implementovat mnoho podpůrných obvodů k celkovému urychlení procesoru. Vzhledem k tomu, že reálné algoritmy většinou nejsou tak triviální, aby je bylo možno vykonat jedinou instrukcí procesoru, je zapotřebí většího množství instrukcí pro realizaci těchto algoritmů. Toto množství se liší v závislosti na instrukčních sadách procesorů.

1.3 Hlavní části procesoru

1.3.1 Aritmeticko-logická jednotka, matematický koprocesor

Velmi důležitou částí procesoru je aritmeticko-logická jednotka (dále už jen ALU). ALU bývá zpravidla realizována jako kombinační obvod a přijímá operační kód, který určuje jaká operace se má mezi vstupními operandy provést. Je důležité poznamenat, že ALU pracuje z pohledu aritmetiky pouze s celými čísly, přičemž podle binární podoby čísel nelze určit, zda jsou jejich hodnoty kladné nebo záporné – tato skutečnost se určuje až podle způsobu přístupu k těmto číslům. Nejnutnější operace pro rozlišení, zda se pracuje s kladnými nebo zápornými čísly, jsou implementovány v podobě instrukcí (například porovnávání).

Absence zpracování čísel s plovoucí desetinnou čárkou jednotkou ALU není způsobena tím, že by ALU nemohla tato čísla zpracovávat, ale několika racionálními důvody. Pro jednoduchost budou uvedeny takové důvody jenom dva. Prvním důvodem je, že aritmetika s plovoucí desetinnou čárkou je natolik náročná na hardwarovou implementaci, že pro ni byla zavedena samostatná jednotka – matematický koprocesor (dále už jen FPU). Druhým, tým podstatnějším, důvodem implementace ALU a FPU zvlášť je využívání výhod spojených se současným využitím těchto dvou jednotek.

1.3.2 Soubor registrů

Registr je zpravidla malé úložiště dat fyzicky umístěno velmi blízko nějakého aktivního prvku, u kterého se předpokládá potřeba rychlého zápisu nebo čtení tohoto registru. Při aplikaci registrů na procesor se jedná o úložiště dat, umístěné uvnitř procesoru, nad kterým lze provádět velmi rychle výpočetní operace. Procesory, které používají registry jako hlavní koncept rychlé vnitřní paměti, implementují většinou těchto registrů více – soubor registrů. Počet takových registrů úzce závisí na instrukční sadě, architektuře a bitové šířce implementujícího procesoru.

Registry ze souboru registrů se pak využívají například jako vstupní operandy ALU, ovšem i jako cílové úložiště výsledku ALU. Soubor registrů většinou nebývá adresovatelný, ale indexovaný. To znamená, že není možno použít hodnotu nějakého registru, ani jakoukoli jinou hodnotu programu a to včetně konstant programu, pro adresaci souboru registrů. Soubor registrů je indexovaný pouze přímo v instrukci, přesněji v instrukčním slově konkrétní instrukce, kde je pro index vyhrazeno místo.

Existují i alternativy registrového souboru. Za zmínku stojí například paměťové úložiště typu LIFO (zásobníková struktura) nebo FIFO (struktura fronty) uvnitř procesoru. Porovnání efektivity mezi těmito třemi metodami z důvodu tématu maturitní práce bude vynecháno.

1.3.3 Řadič

Řadič je hlavní řídicí jednotka procesoru. Bývá realizován jako stavový automat, tudíž se jedná o sekvenční obvod, a řídí činnost celého procesoru. Jedná se zejména o synchronizaci různých stavů procesoru mezi sebou. Velmi často je do řadiče přiveden hodinový signál, který ve výsledku do určité míry ovlivňuje celkový výkon procesoru. Úkolem řadiče je tento signál zpracovat a časovat vnitřní moduly procesoru.

1.3.4 Jednotka správy paměti

Jednotka správy paměti (dále jen MMU) není nezbytným požadavkem k sestavení funkčního turingovsky úplného procesoru. Nicméně se MMU využívá v téměř každém dnešním procesoru obecného použití. Právě z důvodu existence MMU může procesor velmi efektivně implementovat multitasking.

Do doby, než se začala MMU implementovat do procesorů, museli programátoři pracovat při adresování operační paměti (dále jen RWM) v jejich programech s nějakou předem dohodnutou konvencí o využívání adres. Příkladem mohou být programy napsané pro DOS. Důvodem používání takové konvence je fakt, že použitá adresa jakéhokoli programu je ekvivalentní s fyzickou adresou, tj. s adresou, která se opravdu objeví na adresové sběrnici RWM. Sám DOS musel zabírat nějakou paměť v RWM, tudíž ho mohl velmi jednoduše každý program poškodit, protože tyto adresy byly známé.

Implementací MMU do procesorů započala přímo revoluce informatického průmyslu. Nejenže každý program od té chvíle používá virtuální adresu pro adresaci, což mu mimo jiné umožňuje začínat vykonávat instrukce od nulté adresy, ale potenciální operační systémy (dále jen OS) od té doby obdržely velmi efektní hardwarový nástroj ke správě více běžících procesů najednou. MMU totiž chrání OS od všech procesů a zajišťuje taky ochranu procesů mezi sebou. Přímým důsledkem předchozích tvrzení je vytvoření virtuálního adresačního prostoru pro každý proces.

Ochrana paměti je zpravidla implementována jednou ze dvou hlavních metod: segmentace a stránkování. Z důvodu nemalého počtu výhod se dnes téměř vždy používá stránkování.

1.4 Architektura procesoru

1.4.1 Popis architektury procesoru

Architektura procesoru je abstraktní pojem a většinou se jí myslí soubor všech zpravidla neměnitelných vlastností vnitřní struktury a rozhraní procesoru. Vzhledem k tomu, že každá architektura se nějak vyvíjí, jsou architekturou procesoru spíše myšleny hlavní myšlenky návrhu daného procesoru.

1.4.2 Typy architektur procesoru

Architekturu procesoru je možno rozdělit na RISC, CISC a v neposlední řadě ještě VLIW. Pro svou nižší popularitu bude vynechán popis zabývající se architekturou VLIW.

CISC je architektura procesoru, která implementuje ve své instrukční sadě široký okruh funkcí. Původní myšlenkou procesorů CISC architektury zřejmě bylo usnadnění programátorům jejich práci v jazyku symbolických adres. Její existenci lze připisovat i tomu, že v době, kdy se běžně používali CISCové procesory, ještě žádný populární procesor založený na ideách RISC architektury neexistoval – tudíž nebyla možnost porovnat tyto dva druhy architektur. CISC se vyznačuje především instrukcemi, které mají různě dlouhou dobu vykonávání a proměnlivou délku. Zpravidla implementuje registry, kterých většinou není mnoho, a nemají obecné použití. Pro načítání operandů z paměti a ukládání do ní se používá adresovací mód, který je u většiny instrukcí použitelný.

Naproti tomu RISC je architektura procesoru, která se snaží dosáhnout vyššího výkonu jednoduchostí. Opírá se o fakta, že i v samotných CISCových procesorech se v největší míře používají ty nejjednodušší instrukce. Proto zástupce RISC architektury často definuje jen omezené množství instrukcí. Takové instrukce mají všechny pevnou délku a měli by trvat stejně dlouhou dobu. Stejně dlouhou dobu ovšem netrvají všechny. Implementace například násobení a dokonce i načítaní dat z paměti je pro RISCové procesory relativně vážný problém, který může vést k narušení ortogonality. Pro jednoduchost je RISC architektura úzce provázána s „load/store“ modelem přístupu do paměti. Ve zkratce to znamená, že se k přístupu do paměti používají pouze dvě instrukce, které přímo umožňují čtení a zápis mezi pamětí a vnitřním úložištěm procesoru. Adresovací mód pro načtení operandu z paměti by neměl být implementován žádnými ostatními instrukcemi.

2 VÝZNAMNÉ ARCHITEKTURY PROCESORŮ

2.1 Architektura x86

2.1.1 Stručný popis

Architektura x86 byla navržena společností Intel zprvu jako čistě CISCová architektura. Nicméně postupem času, jak různé výzkumy ukázaly vyšší výkon architektury RISC, x86 architektura implementovala RISCové jádro při zachování CISCového rozhraní. Procesory založené na x86 byly vždy zpětně kompatibilní.

Architektura x86 je doposud nejpopulárnější architekturou procesorů pro stolní počítače, notebooky a nějaký podíl má i v menších zařízeních jako jsou například tablety. Její uplatnění lze pozorovat i na výpočetních stanicích. Mezi nynější hlavní výrobce procesorů této architektury se řadí firma Intel, která má většinový podíl na trhu, a firma AMD. Z minulosti rozhodně nelze opomenou firmu Cyrix.

2.1.2 Architektura

Architektura umožnila relativně brzy hardwarovou podporu multitaskingu, a to přidáním ochrany paměti do procesoru Intel 80286, který byl vydán roku 1982, využitím segmentace. K reálnému využití multitaskingu však docházelo až u jeho nástupce, procesoru Intel 80386, který je již 32bitový a podporuje i stránkování.

2.1.3 Soubor registrů

Základní výbavou x86 procesorů jsou přinejmenším aritmetické a segmentové typy registrů. Každý x86 procesory taky obsahuje dva speciální registry.

Mezi aritmetické registry patří:

- AX – akumulátor,
- BX – bázový registr,
- CX – čítač,
- DX – rozšíření akumulátoru,
- SI, DI – registr pro index zdroje a cíle,
- BP – ukazatel na rámec zásobníku aktuální procedury,
- SP – ukazatel na vrchol zásobníku.

Mezi segmentové registry patří:

- CS – kódový segment,
- DS – datový segment,
- ES – extra segment,
- SS – zásobníkový segment.

Speciálními registry jsou:

- IP – ukazatel instrukcí a
- FLAGS – různé příznaky procesoru nebo jeho stavu. [1]

	31	15	7	0
EAX	AX	AH	AL	
EBX	BX	BH	BL	
ECX	CX	CH	CL	
EDX	DX	DH	DL	
ESI				
EDI				
ESP				
EBP				
EIP				
EFLAGS				

Tab. 2.1.3 Soubor registrů 32bitového x86 procesoru

Rozšíření x86 architektury na z původních 16 bitů na 32 bitů navyšuje šířku pracovních registrů na 32 bitů, aby jejich velikost byla v korespondenci s šířkou procesoru. Tabulka 2.1.3 zobrazuje soubor registrů 32 bitového procesoru x86 architektury. Když programátor ví, že nebude potřebovat velikou šířku operandu pro vykonání instrukce, je možno indexovat jenom část registru a mnohdy tak provést tuto operaci rychleji. Například instrukce indexující registr EAX bude pracovat s celým registrem, AX pak s jeho 16 spodními bity, AL pak s nejnižší osmicí bitů registru EAX.

2.2 Architektura ARM

2.2.1 Stručný popis

ARM (původně Acorn RISC Machine, později Advanced RISC Machines) architektura je spravována a vyvíjena firmou ARM Holdings. Paradoxem je, že tato firma čipy postavené na své architektuře sama nevyrábí. Namísto toho firma prodává licence pro výrobu jejich čipů za splnění podmínek, které jsou klíčovými k tomu, aby byl software mezi dvěma procesory ARM stejné verze různých výrobců kompatibilní. ARM architektura se již od svého počátku snažila co nejvíce využít výhod, které nabízí RISCově založený procesor – jednoduchost je rychlost. Při současné implementaci procesorů lze u RISCových typů navíc zpozorovat nižší spotřeba elektrické energie. Bohužel ani to nestačilo k tomu, aby se doposud prosadila tato architektura na trhu s osobními počítači. Nicméně v oblasti zařízení s požadavky na nízkou spotřebu ARM boduje. Téměř každý nynější chytrý telefon je poháněn právě procesorem založeném na architektuře ARM. Velmi rozdílné to není ani u tabletů a poslední dobou se začínají ARMy vybavovat i netbooky, popřípadě menší notebooky. Typickým nynějším příkladem může být Chromebook s výchozím operačním systémem Chrome OS, který je vyráběn pro Google několika světovými výrobci počítačů a celý je postaven na ideji, že mnohé internetové aplikace dokáží být stejně hodnotné jako aplikace nativní.

Ačkoli je ARM nejpočetněji zastoupenou architekturou mezi všemi procesory, vývoj ARMu neprobíhal celou dobu podle představ. Postupem času se začalo opouštět od prvků RISCové architektury a výkon už, ačkoliv se ze začátku tato rozhodnutí zdála výhodná, nebyl takový, jaký mohl být. Nicméně tohoto faktu si všimli i samotní vývojáři této architektury a verze ARMv8 měla opět přinést jednodušší design a hardwarovou implementaci. Ještě je důležité poznamenat, že u ARMu na rozdíl od x86 není zaručená zpětná kompatibilita. Vzhledem k tomuto faktu může ARM architektura pokračovat na vývoji dalších verzí, i když nebyla v minulosti implementována nejefektivněji. Možné nevýhody jsou spojeny s kompatibilitou software mezi jednotlivými ARM verzemi.

2.2.2 Architektura

Jak již bylo zmíněno, ARM byla zamýšlena jako RISCová architektura. Nicméně dalo by se o tom dlouze debatovat. Například zdali je podmíněné vykonání téměř jakékoli instrukce a přítomnost registru s příznaky prvkem procesoru založeného na RISC idejích.

ARM architektura podporuje ochranu paměti pro zajištění běhu více aplikací v odděleném adresovacím prostoru. Používá k tomu techniky známé ze stránkování, které taky implementuje. MMU je nemusí být nutně implementována u každého ARM procesoru, nebo může být softwarově deaktivována.

2.2.3 Instrukční formáty

Architektura ARM používá jako koncept pro dekódování instrukční formáty, jejichž ortogonalita ovšem není naprosto ideální, a to především z důvodu hlavních pilířů ARM architektury. Přesto lze tvrdit, že pokud se rozdělí instrukční sada na instrukční formáty, samotné dekódování instrukcí bude rychlejší než dekódování, které instrukční formáty nepoužívá.

2.2.4 Soubor registrů

<table>
<tr><th>User</th><th>FIQ</th><th>IRQ</th><th>SVC</th><th>Undef</th><th>Abort</th></tr>
<tr><td>R0</td><td rowspan="8">User mode R0 - R7, R15 and CSPR</td><td rowspan="13">User mode R0 - R12, R15 and CSPR</td><td rowspan="13">User mode R0 - R12, R15 and CSPR</td><td rowspan="13">User mode R0 - R12, R15 and CSPR</td><td rowspan="13">User mode R0 - R12, R15 and CSPR</td></tr>
<tr><td>R1</td></tr>
<tr><td>R2</td></tr>
<tr><td>R3</td></tr>
<tr><td>R4</td></tr>
<tr><td>R5</td></tr>
<tr><td>R6</td></tr>
<tr><td>R7</td></tr>
<tr><td>R8</td><td>R8</td></tr>
<tr><td>R9</td><td>R9</td></tr>
<tr><td>R10</td><td>R10</td></tr>
<tr><td>R11</td><td>R11</td></tr>
<tr><td>R12</td><td>R12</td></tr>
<tr><td>R13 (SP)</td><td>R13 (SP)</td><td>R13 (SP)</td><td>R13 (SP)</td><td>R13 (SP)</td><td>R13 (SP)</td></tr>
<tr><td>R14 (LR)</td><td>R14 (LR)</td><td>R14 (LR)</td><td>R14 (LR)</td><td>R14 (LR)</td><td>R14 (LR)</td></tr>
<tr><td>R15 (PC)</td><td></td><td></td><td></td><td></td><td></td></tr>
</table>

<table>
<tr><td>CPSR</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td></td><td>SPSR</td><td>SPSR</td><td>SPSR</td><td>SPSR</td><td>SPSR</td></tr>
</table>

Tab. 2.2.4 Soubor registrů ARM procesoru

Hlavním konceptem registrového souboru procesorů ARM architektury jsou tzv. banky registrů. Aktivní banka je přímo závislá na aktuálním módu procesoru. Počet módů je závislý na verzi architektury, ovšem vždy se tyto módy dají rozdělit do dvou kategorií: privilegované a neprivilegované módy. Neprivilegovaným módem je pouze uživatelský mód, ostatní jsou privilegované. Každá uživatelská aplikace běží v uživatelském módu, a pokud potřebuje komunikovat s hardwarem, vykoná instrukci systémového volání. Tato instrukce způsobí přepnutí procesoru do privilegovaného módu a přesunutí vykonávání instrukcí na adresu, kde má systémové volání vstupní bod, které bývá spravováno OS. Přepnutí do privilegovaného módu mohou způsobit i různé druhy přerušení. Do jakého privilegovaného módu se procesor přepne, pak záleží výhradně na typu aktivního přerušení. Na předchozí straně v tabulce 2.2.4 je vidět rozložení a sdílení částí registrových bank mezi jednotlivými módy.

Téměř v každé situaci může programátor ke všem registrům přistupovat jako k registrům obecného použití. Některé registry ovšem podporují nějakou funkci navíc a jsou až na PC a CPSR implementovány pro každý mód zvlášť. Těmito registry jsou:

- R13 (SP) – zásobníkový registr, který slouží jako ukazatel na vrchol zásobníku pro použití instrukcí PUSH a POP,
- R14 (LR) – odkazovací registr, do kterého je ukládána hodnota PC při volání procedury instrukcí BL (adresa je uložena do tohoto registru v bance aktuálního módu procesoru) a adresa PC ve chvíli, kdy nastane přerušení (adresa je uložena do nového módu procesoru),
- R15 (PC) – programový čítač, udržuje většinou adresu aktuálně poslední rozpracované instrukce,
- CPSR – aktuální registr příznaků a stavů procesoru,
- SPSR – uložený registr příznaků a stavů procesoru každého privilegovaného módu, do kterého je zapsán CPSR ve chvíli, kdy nastane přerušení, které ošetřuje tento mód.

2.3 Architektura Alpha

Alpha je již zaniklou RISCovou architekturou, kterou vyvinula firma Digital Equipment Corporation, a byla to paradoxně nástupce procesorů architektury VAX, jenž byla známá především pro svoji blízkost k čistě CISCovým prvkům. Architektura byla už od začátku vyvíjena jako 64bitová, a tudíž se mohlo vše, co bylo potřeba, přizpůsobit 64bitovému designu už od začátku a vyhnout se tak navýšení šířky procesoru, což v konečném důsledku často vede ke zpomalení výsledné architektury, která má větší bitovou šířku. Mnohými je právem označována za nejčistší RISC architekturu, která kdy byla vyvinuta a implementována. Alpha architektura to s RISCovým návrhem myslela opravdu vážně. Svědčí o tom i fakt, že adresovaní jednotlivých bajtů nebo dvojici bajtů bylo do architektury přidáno až s rozšířením a to na nátlak spotřebitelů. Do té doby bylo možno adresovat paměť pouze po 64bitových nebo 32bitových blocích a z načteného bloku se maskováním dostat k požadovanému bajtu nebo dvojici bajtů. Dalším čistě RISCovým prvkem bylo zavedení tzv. PALCodu. Ve zkratce je to sjednocený nízkoúrovňový přístup ke správě mezipamětí (například softwarově obsluhovaná TLB), řízení ošetření přerušení a výjimek. Pomocí PALCodu lze taky poskytnout systémovému softwaru HAL vrstvu.

2.3.1 Instrukční formáty

Stejně jako architektura ARM, používá i architektura Alpha pro dekódování instrukci instrukční formáty. Nicméně u této architektury, zejména pro její jednoduchost, je využití těchto instrukčních formátů o dost efektivnější a jsou v souladu s potřebnou ortogonalitou procesoru. Je taky důležité zmínit, že vedle ostatních existuje taky PALCode jako instrukční formát.

2.3.2 Soubor registrů

Procesory založené na Alpha architektuře obsahují 32 registrů obecného použití (včetně aritmetiky s celými čísly) a 32 registrů pro použití v aritmetice s plovoucí desetinnou čárkou. V obou případech je registr s indexem 31 vždy roven nule. Při pokusu zapsat do něj nějakou hodnotu, se ve skutečnosti nic nestane. Dále každý procesor založený na Alpha architektuře implementuje PC, který má dva spodní bity vždy rovné nule. Je to z důvodu zarovnání instrukce na 32 bitů v paměti. Toto je mimo jiné důležité pro MMU, která používá stránkování. MMU pak nemusí řešit problémy s instrukcí, která může patřit do dvou stránek najednou. Alpha procesor ještě implementuje několik řídicích registrů.

3 VHDL

3.1 Stručný popis VHDL

VHDL je programovací jazyk, který byl navržen pro popis logického zapojení hardware. Na rozdíl od běžného programovacího jazyku, kterým se popisují algoritmy pro řešení různých problémů sekvencí příkazů, VHDL popisuje zapojení logického obvodu kombinačními a sekvenčními prvky, pomocí kterých se řeší tento algoritmus. VHLD tak pracuje na nejnižší logické vrstvě při řešení jakéhokoli algoritmu – hardware. Není proto divu, že umožňuje návrh i velmi komplexních obvodů jako je třeba i moderní procesor.

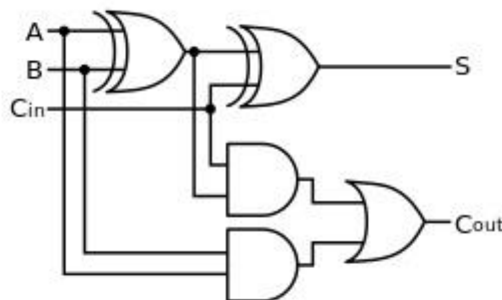
Jazyk VHDL používá koncept modulů. Modul se chová jako černá skříňka, tzn. má jasně definované vstupy a výstupy (blok entity) a je známý vztah mezi vstupem a výstupem. Vnitřní implementace (blok architecture) pak už není důležitá, pokud tento vztah splňuje. Používat takový model má hned několik výhod. Například pokud dojde k vylepšení nějakého modulu, u kterého musí být opravdu prokázáno, že vztah mezi vstupem a výstup je korektní, nemusí se měnit celý systém, ale pouze modul, který byl vylepšen. Moduly taky zvyšují přehlednost a robustnost kódu.

3.2 Teoretické základy jazyka VHDL

3.2.1 Předmluva

V této části bude popsána jenom nezbytná část teoretických základů VHDL jazyka. Celé vysvětlování se bude opírat o konkrétní příklady a na nich bude ukázáno, jak VHDL ve skutečnosti hardware popisuje. Je důležité mít neustále na mysli, že se popisuje zapojení hardware, nikoli chování programu.

3.2.2 Příklad 1 – úplná sčítačka (kombinační obvod)



Obr. 3.2.2.1 Logické schéma úplné sčítačky [2]

```
1  library ieee;
2  use ieee.std_logic_1164.all;
3
4
5  entity uplna_scitacka is
6     port ( a:      in  std_logic;
7            b:      in  std_logic;
8            c_in:   in  std_logic;
9            s:      out  std_logic;
10           c_out:  out  std_logic
11           );
12 end uplna_scitacka;
13
14
15 architecture chovani of uplna_scitacka is
16
17    signal a_xor_b:   std_logic;
18
19 begin
20
21    a_xor_b <= a xor b;
22    s <= a_xor_b xor c_in;
23    c_out <= (a and b) or (c_in and a_xor_B);
24
25 end chovani;
```

Obr. 3.2.2.2 Popis úplné sčítačky ve VHDL

Při porovnání logického schématu úplné sčítačky a VHDL kódu lze mnohé odvodit. Přesto by mělo být určitým částem kódu věnována zvýšená pozornost. Veškeré následující odkazy budou spojené s obrázky č. 3.2.2.1 a 3.2.2.2. Odkazy na konkrétní řádky kódu jsou spojeny jenom s obrázkem č. 3.2.2.2.

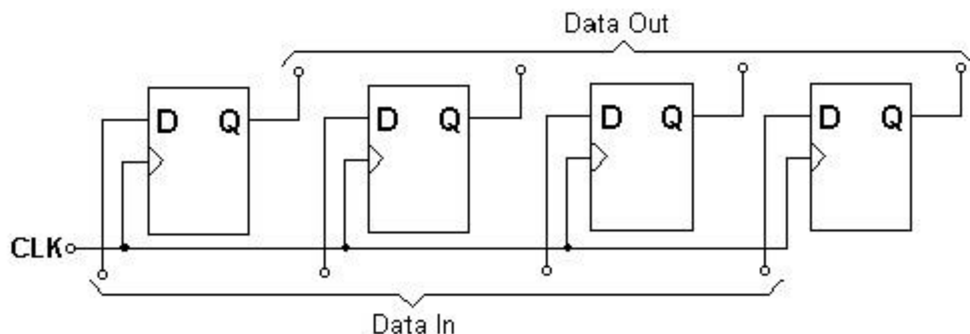
První dva řádky provádí import knihovny, která obsahuje vše potřebné k tomu, aby bylo možno při popisování použít devítistavovou logiku – datový typ std_logic a std_logic_vector. Výhradně dvoustavová logika se moc nepoužívá z důvodu, že úplně neodpovídá reálnému obvodu. Příkladem může být implementace třístavového bufferu, který se používá pro realizaci sběrnic a logikou dvoustavovou by jej nebylo možné implementovat.

Popis je pak rozdělen na dvě části: entita a architektura. Entita popisuje rozhraní modulu s okolním světem, tj. vstupy a výstupy, architektura pak samotnou závislost výstupů na vstupech.

Řádky 5 až 12 jsou definovány identifikátory vstupů a výstupů a jejich datový typ. To je vše s čím pak bude mít možnost okolní svět pracovat.

Řádky 15 až 25 popisují samotné zapojení tohoto kombinačního obvodu. Řádek č. 17 definuje tzv. signál. Při pohledu na schéma jde vidět, že nyní signál slouží pouze jako identifikátor vodiče, který vystupuje z hradla XOR, do kterého jsou navedeny vstupy „a“ a „b“. Hardwarové propojení je ve VHDL jazyku značeno symbolem „<=“. Provádí se vždy tak, že vstup, signál, nebo nějaký výraz je na pravé straně symbolu a cílový výstup nebo signál je na levé straně symbolu a je roven pravé straně.

3.2.3 Příklad 2 – 4bitový registr (sekvenční obvod)



Obr. 3.2.3.1 Logické schéma 4bitového registru [3]

```
 1  library ieee;
 2  use ieee.std_logic_1164.all;
 3
 4
 5  entity registr_4bitovy is
 6     port ( clk:        in   std_logic;
 7            data_in:    in   std_logic_vector(3 downto 0);
 8            data_out:   out   std_logic_vector(3 downto 0)
 9            );
10  end registr_4bitovy;
11
12
13  architecture chovani of registr_4bitovy is
14
15     signal memory: std_logic_vector(3 downto 0);
16
17  begin
18
19     process(clk)
20     begin
21        if(rising_edge(clk)) then
22           memory <= data_in;
23        end if;
24     end process;
25
26     data_out <= memory;
27
28  end chovani;
```

Obr. 3.2.3.2 Popis 4bitového registru ve VHDL

Stejně jako v předchozím příkladu, lze mnohé odvodit. Nicméně na rozdíl od předchozího příkladu je pro realizaci obvod použit sekvenční prvek – registr (sestavěn z D klopných obvodů). Výstup už tedy není závislý pouze na vstupech. Veškeré následující odkazy budou spojené s obrázky č. 3.2.3.1 a 3.2.3.2. Odkazy na konkrétní řádky kódu jsou spojeny jenom s obrázkem č. 3.2.3.2.

V entitě (řádky 5 až 10) je poprvé uvedena vektorová podoba datového typu std_logic, a to std_logic_vector. Konkrétně se jedná o vstup a výstup pro čtené a zapisované data s bitovou šířkou 4.

V architektuře je opět definován signál na řádku č. 15. Tento signál už ovšem není jen pouhým identifikátorem vodiče, ale vzhledem k budoucímu zpracování se jedná o identifikátor klopného obvodu D o šířce 4 bity, který má vlastnosti paměťového úložiště.

Proces je nově uvedenou konstrukcí, která se používá k realizaci sekvenčních obvodů. Důvodem tohoto použití je skutečnost, že proces opravdu probíhá sekvenčně. Ovšem ne vždy je takové sekvenční vykonávání úplně průhledné. Je nutné mít na paměti, že přiřazení signálů je provedeno v celém procesu současně, a to na konci jeho nejdelší sekvence. Pro čistě sekvenční vykonávání procesu se používají proměnné, které jsou velmi podobné signálům, ale:

- definují se lokálně a platí pouze pro daný proces,
- přiřazení není operátorem „<=“, ale operátorem „:=“,
- změna hodnoty je platná okamžitě až do další změny hodnoty. [4]

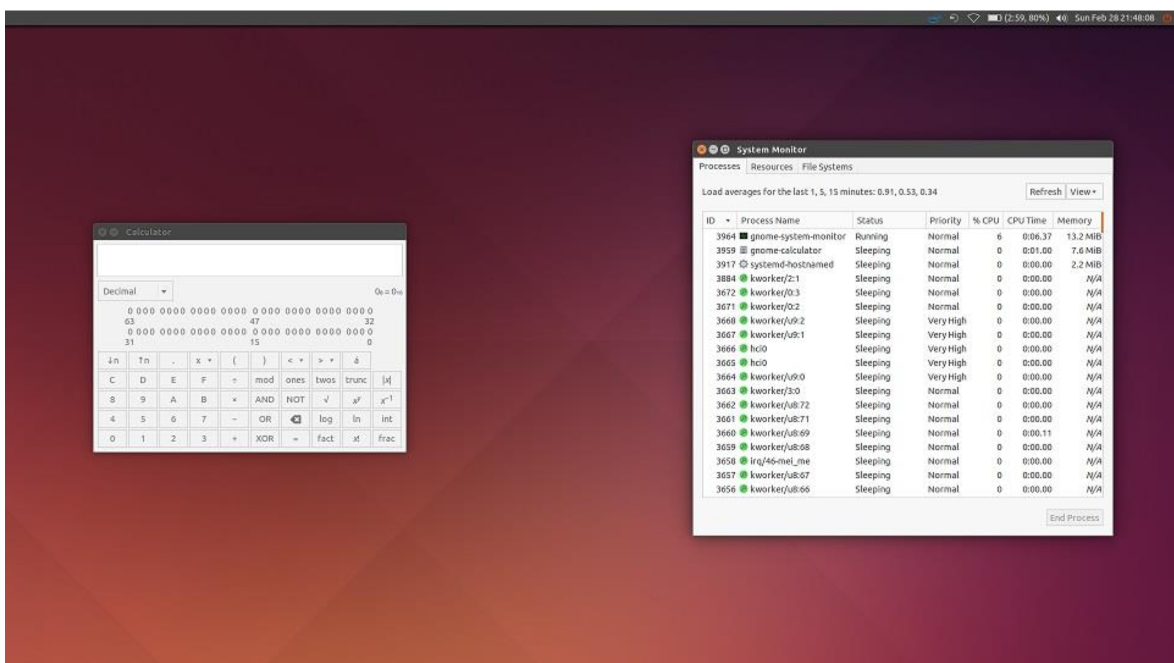
Na řádku č. 19 se definuje v závorkách tzv. sensitivní list. To znamená, že při změně jakéhokoliv signálu uvedeného v tomto listu, se provede to, co je uvedeno v těle procesu. Tento konkrétní proces nastane aktivní ve chvíli, kdy se změní hodnota vstupu clk a podmínka na 21. řádku zajistí, že k zápisu vstupních dat do registru dojde pouze tehdy, je-li tato změna způsobena náběžnou hranou vstupu clk. Z toho vyplývá, že zápis dat je synchronní a kdykoli se objeví na vstupu clk nástupná hrana, uloží se do registru hodnota, která je v tu chvíli na vstupu data_in. Výstup (řádek č. 26) je asynchronní, čili není závislý na žádném hodinovém signálu.

II. PRAKTICKÁ ČÁST

4 POUŽITÝ SOFTWARE

4.1 Ubuntu 14.04

Při popisování použitého softwaru by asi bylo vhodné začít operačním systémem. Na počítači, na kterém se prováděli všechny klíčové operace, tj. logický návrh, hledání zdrojových informací a při nejmenším ještě neméně důležité popisovaní procesoru jazykem VHDL, běží operační systém Ubuntu verze 14.04 LTS. Ubuntu je operační systém, který je založený na Linuxu a je volně ke stažení na oficiálních stránkách jejich vývojáře, společnosti Canoncialu.

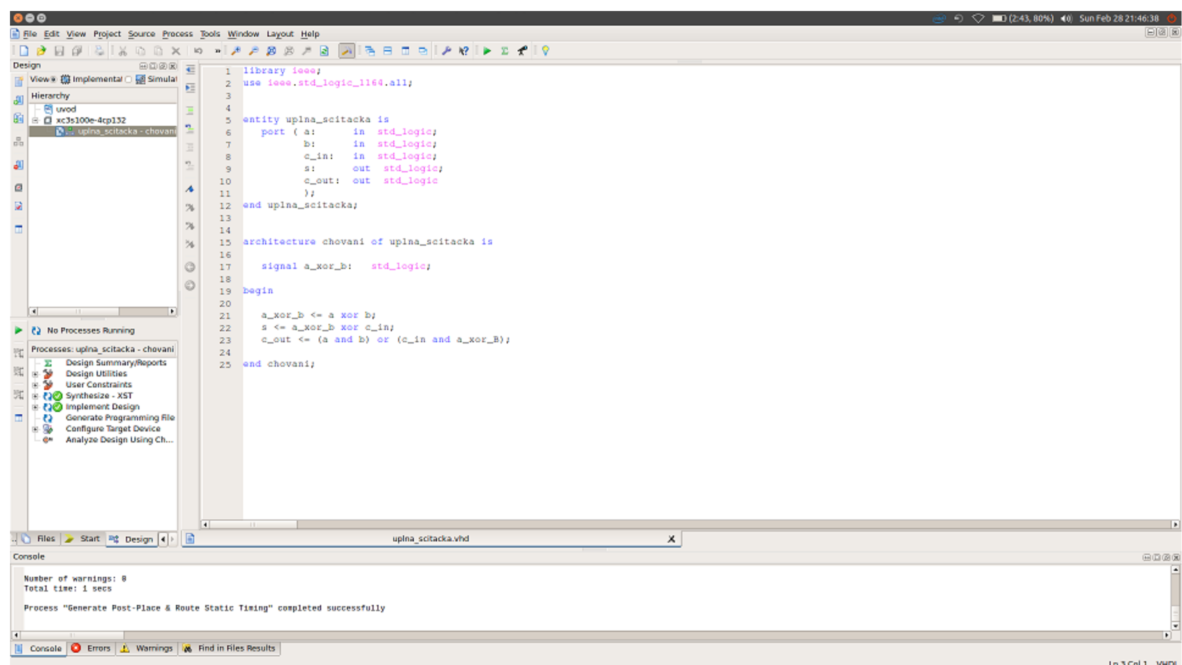


Obr. 4.1 Ukázka GUI modifikovaného Ubuntu 14.04

Důvodem volby tohoto operačního systému nebyl nějaký záměr, nýbrž autor této maturitní práce používá tento operační systém i jako svůj primární operační systém. Těm, kteří se zajímají v jakémkoli směru o informatiku, tento operační systém může pomoci pochopit mnohé skutečnosti z tohoto oboru. Pokud bude uživatel nebo programátor chtít, může pracovat téměř až na úrovni hardwaru. Stačí k tomu jediné klíčové slůvko – sudo. Proto pak není divu, že na Ubuntu existuje veliké množství funkcí ke stažení, které mohou velmi zefektivnit práci při jeho používání. Sám autor si modifikoval Ubuntu k obrazu svému jak z grafické stránky, tak ze stránky použitelnosti.

4.2 Xilinx ISE Webpack 14.7

Při zpracování této práce je použita nejnovější a zároveň poslední verze IDE Xilinx ISE Webpack, který byl vyvíjen firmou Xilinx pro hradlová pole vlastní výroby. Konkrétně se jedná o verzi 14.7, která byla vydána v říjnu roku 2013 a je volně ke stažení na oficiálních stránkách firmy Xilinx. Jeho součástí je celá řada nástrojů ulehčující, nebo umožňující modifikovat výsledný design obvodu. Budiž jedním příkladem za všechny nástroj PlanAhead Design Tool. Placeným dvojníkem ISE Webpack je Xilinx ISE Design Suite. Nástupcem ISE softwaru se stal Xilinx Vivado, který taky vyvíjí Xilinx.



Obr. 4.2 Ukázka programu ISE Project Navigator

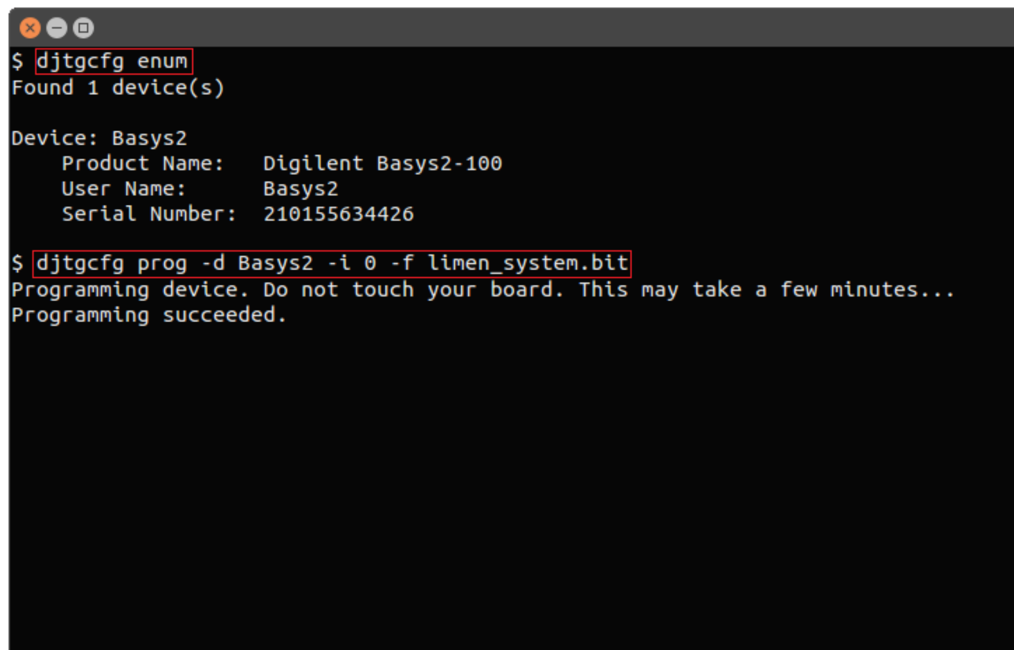
ISE Webpack umožňuje popisovat hardware i ve Verilogu, což je účelem velmi podobný jazyk jazyku VHDL. Jejich popularita je různá v závislosti na lokalizaci. V Evropě se píše především ve VHDL a v Americe je naopak používanější Verilog. Nicméně vždy záleží na pocitech konkrétního jedince. Teoreticky by se měla dát implementovat pomocí obou stejná funkcionalita výsledného obvodu.

4.3 Digilent Adept 2

4.3.1 Popis programu Digilent Adept 2

Digilent Adept 2 je software, který při zpracování této práce umožnil nahrát výstupní programovací soubor z ISE Webpacku do použitého vývojového kitu, Digilent Basys 2. Ve Windows je součástí tohoto programu i GUI, v Linuxu se ovládá příkazy skrze konzoli.

4.3.2 Ovládání programu Digilent Adept 2



Obr. 4.3.2 Ukázka ovládání programu Digilent Adept 2

Následující popis se zabývá obsahem obrázku č 4.3.2. Prvním příkazem programu Digilent Agent 2 lze zjistit, zda je připojeno nějaké Digilent zařízení do počítače, které tento software podporuje. Pokud se provede výpis požadovaného zařízení, jako je vidět na obrázku 4.3.2, je zařízení zapojeno a nakonfigurováno správně. Důležitou částí ve výpisu je identifikátor zařízení, který následuje za „User Name:“. Využívá se pak při použití druhého klíčového příkazu. Nicméně před použitím druhého příkazu je nutné mít na paměti, že se bude relativně adresovat programovací soubor. To znamená, že je vhodné mít Linuxový terminál v adresáři, ve kterém se nachází i programovací soubor. Druhý příkaz ve zkratce nahraje programovací soubor adresovaný relativní cestou do požadovaného Digilent zařízení, které je zvoleno právě pomocí identifikátoru, který byl součástí výpisu po použití příkazu prvního.

5 POUŽITÝ HARDWARE

5.1 Digilent Basys 2

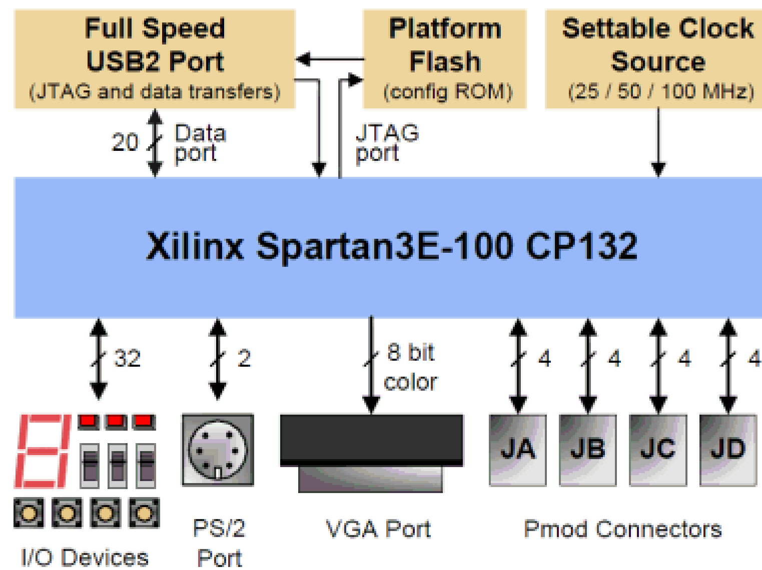
5.1.1 Popis vývojového kitu Digilent Basys 2

Obr. 5.1.1 Ukázka vývojového kitu Digilent Basys 2

Použitým fyzickým zařízením, do kterého byl v průběhu vývoje procesoru nahráván jeho programovací soubor, je vývojový kit Digilent Basys 2, jehož fyzickou podobu lze vidět na obrázku č. 5.1.1. Při svých rozměrech 11,5 x 7,1 mm je velmi malý a při své ceně $149 na oficiálních stránkách je i velmi levný. Nicméně v nynější době už se prodává i jeho nástupce, Digilent Basys 3, za tutéž cenu a s lepším hardwarem. Obecně platí, že řada vývojových kitů Basys je vhodná především pro každého, kdo má zájem začít s digitálním návrhem skrze jazyk VHDL nebo Verilog.

Basys 2 v základu obsahuje mnoho užitečných doplňků, které jsou popsány v sekci 5.1.2, tj. následující. Taky je zde možnost nějakého rozšíření funkcionality pomocí 4 šestipinových konektorů označovaných firmou Digilent Pmod.

5.1.2 Fyzická výbava vývojového kitu Digilent Basys 2



Obr. 5.1.2 Abstraktní schéma zapojení Basys 2 [5]

Obrázek 5.1.2 propaguje abstraktní schéma zapojení fyzického vybavení tohoto vývojového kitu, kterým je:

- hradlové pole Xilinx Spartan 3E-100,
- USB2 port pro napájení, programovaní a komunikaci,
- flash ROM pro uložení konfigurace hradlového pole,
- 8 LED diod,
- 4 sedmisegmentové displeje,
- 4 tlačítka,
- 8 přepínačů,
- PS/2 port,
- 8bitový VGA port,
- uživatelsky nastavitelné hodiny (25/50/100MHz),
- 4 šestipinové Pmod rozšiřující konektory. [6]

6 PROCESOR LIMEN

6.1 Stručný popis

Procesor Limen je 16bitový RISCový procesor, který vznikl modifikací procesoru, jenž měl být původně předmětem této maturitní práce. Data v paměti adresuje po dvojici bajtů, nikoli jak je zvykem po bajtu, a dokáže tak adresovat až 128KB paměti, která je lineárního rozložení. Do tohoto adresního prostoru musí být zahrnuta i všechna vstupní a výstupní zařízení.

Procesor je řízený hodinovým signálem a uvnitř procesoru je tento signál rozdělen do jeho jednotlivých částí. Tyto části by teoreticky mohli pracovat současně a nezávisle na sobě, ovšem pro jednoduchost návrhu není ani pipelining, který by to umožňoval, implementován. Až na instrukci čtení z paměti, je u každé instrukce zaručeno, že bude trvat stejně dlouhou dobu.

6.2 Soubor registrů

Procesor Limen definuje 8 registrů obecného použití. Značí se R0 až R7. Registr R0 udržuje po celou dobu běhu procesoru hodnotu 0. Při pokusu do něj zapsat nějakou hodnotu se nic nestane. To, že je registr R0 roven 0 má hned několik výhod a jejich výčtu bude věnována sekce 6.4.

Procesor Limen taky obsahuje registr speciálního použití – IP, ukazatel instrukcí. V tomto registru je uložena vždy adresa aktuálně vykonávané instrukce a jeho modifikací lze umožnit v programu podmíněné a nepodmíněné skoky. Naopak procesor Limen neimplementuje žádný statusový registr. Namísto toho definuje instrukce, které dokáží jeho funkci plně nahradit, a vylepší tím ortogonalitu procesoru.

6.3 Instrukční sada

6.3.1 Popis instrukční sady

Instrukční sada používá koncept instrukčních formátů. Konkrétní implementace instrukčních formátů je relativně slabá, ovšem dekódování je přesto rychlejší, než kdyby se instrukční formáty nepoužívaly.

6.3.2 Mnemonika instrukcí

Navzdory tomu, že assembler pro Limen procesor neexistuje, budou v této práci použita imaginární mnemonika, jejichž podrobnějšímu zkoumání a popisu je věnována sekce 6.4. Procesor Limen implementuje instrukce těchto mnemonik:

- ST – uložit,
- LD – načíst,
- ADD – sečíst,
- SL – nastavit pokud je menší,
- SLU – nastavit pokud je menší (bez znaménka),
- OR – logická disjunkce,
- NOR – negace logické disjunkce,
- AND – logická konjunkce,
- NAND – negace logické konjunkce,
- XOR – exklusivní logická disjunkce,
- SLL – logický bitový posun vlevo,
- SRL – logický bitový posun vpravo,
- SRA – aritmetický bitový posun vpravo,
- SUB – odečíst,
- LI – načíst 8bitovou hodnotu,
- LIS – načíst 8bitovou hodnotu, bitově posunutou o 8 řádů doleva,
- LIL – načíst spodních 8 bitů,
- LIH – načíst vrchních 8 bitů,
- JNE – skoč, pokud není rovno,
- JE – skoč, pokud je rovno,
- JL – skoč, pokud je menší,
- JLE – skoč, pokud je menší nebo rovno,
- JG – skoč, pokud je větší,
- JGE – skoč, pokud je větší nebo rovno,
- JWL – skoč s odkazem.

6.4 Instrukční formáty

6.4.1 Předmluva k instrukčním formátům

Následující část dokumentace bude používat určité symboly, které z důvodu speciálního použití v imaginárním jazyku symbolických adres, nelze definovat v sekci seznam použitých symbolů a zkratek. Proto význam těchto symbolů vedle několika dodatečných informací bude uveden právě zde.

Instrukční sada používá trojici typů indexů pro registry: Rz, Ry a Rx. Každý z těchto typů může indexovat libovolný registr. Není však podmínkou, aby každá instrukce používala všechny typy těchto indexů najednou. Libovolná instrukce může použít například Rz, Rx a 4bitovou konstantu. Konstanty by se v imaginárním jazyku symbolických adres zapisovaly tak, že se nejprve napíše rovnítko na místo operandu, kde je potřeba konstanta, a poté se napíše požadovaná hodnota s písmenem, které určuje jeho číselnou soustavu (d – desítková (výchozí, nebylo by nutno psát), h – hexadecimální, o – osmičková, b – dvojková). Následující příklad demonstruje proměnný počet operandů a různé typy operandů: LI R5, =45h – uloží hodnotu 45 v hexadecimální soustavě do registru R5. Celá instrukční sada, s výjimkou instrukce s mnemonikou ST (dále už jen ST), byla navržena tak, aby se v imaginárním jazyku symbolických adres operandy, na které je aplikována nějaká funkce, daly zapsat za sebou jako matematická rovnice, ve které je nalevo Rz. Pro pochopení může být uveden další příklad: instrukce OR R4, R5, =5d (která má obecný tvar OR Rz, Ry, zimm4) má stejný význam matematickým vyjádřením jako R4 = R5 | 5 (s obecným tvarem Rz = Ry | zimm4). V jakékoli instrukci, v jejímž instrukčním slově jsou uvedeny operandy Ry nebo Rx se z těchto registrů čte. Naopak, kdekoli je uveden registr Rz, nějaké data se do něho zapisují. Výjimku opět tvoří instrukce ST. Důvodem těchto přinejmenším dvou výjimek ST instrukce je výrazně nízká ortogonalita vzhledem ke všem ostatním instrukcím. Vyžaduje totiž jako jediná instrukce 3 operandy pro načtení.

6.4.2 Výčet instrukčních formátů

<table>
<tr><th></th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th><th></th></tr>
<tr><td>1</td><td>0</td><td>0</td><td>0</td><td colspan="3">func</td><td colspan="4">imm4</td><td colspan="3">Ry</td><td colspan="3">Rz</td><td>Arithmetic (zimm)</td></tr>
<tr><td>2</td><td>0</td><td>0</td><td>1</td><td colspan="3">func</td><td colspan="4">imm4</td><td colspan="3">Ry</td><td colspan="3">Rz</td><td>Arithmetic (oimm)</td></tr>
<tr><td>3</td><td>0</td><td>1</td><td>0</td><td colspan="3">func</td><td colspan="4">imm4</td><td colspan="3">Ry</td><td colspan="3">Rz</td><td>Logic (imm)</td></tr>
<tr><td>4</td><td>0</td><td>1</td><td>1</td><td colspan="4">func</td><td colspan="3">Rx</td><td colspan="3">Ry</td><td colspan="3">Rz</td><td>Arithmetic + logic (reg)</td></tr>
<tr><td>5</td><td>1</td><td>0</td><td>0</td><td colspan="2">func</td><td colspan="8">imm8</td><td colspan="3">Rz</td><td>Load (imm)</td></tr>
<tr><td>6</td><td>1</td><td>0</td><td>1</td><td colspan="7">disp7</td><td colspan="3">Ry</td><td colspan="3">func</td><td>Conditional jump (disp)</td></tr>
<tr><td>7</td><td>1</td><td>1</td><td>0</td><td colspan="10">disp10</td><td colspan="3">Rz</td><td>Unconditional jump (disp)</td></tr>
<tr><td>8</td><td>1</td><td>1</td><td>1</td><td colspan="7">SBZ</td><td colspan="3">Ry</td><td colspan="3">Rz</td><td>Unconditional jump (reg)</td></tr>
</table>

Tab. 6.4.2 Instrukční formáty

Jak již bylo uvedeno, procesor Limen používá instrukční formáty, které jsou hlavním konceptem dekódování instrukcí. Použity jsou zejména z důvodu jednoduchosti hardwarové implementace, jejich flexibility a RISCového charakteru samotného procesoru. Mimo jiné instrukční formáty mají taky podstatný význam při výrobě procesoru s větší šířkou instrukčního slova pro budoucí vývoj jeho instrukční sady při zachování jednoduchosti procesoru.

Procesor Limen rozlišuje 8 instrukčních formátů:

- Aritmetika s přímou hodnotou (kladnou) – zimm
- Aritmetika s přímou hodnotou (zápornou) – oimm
- Logika s přímou hodnotou – imm
- Aritmetika a logika s registry – reg
- Načítání přímé hodnoty – imm
- Podmíněné skoky s odsazením – disp
- Nepodmíněné skoky s odsazením – disp
- Nepodmíněné skoky s registrem – reg

Přídavek za každým instrukčním formátem je zkratkou pro jeho hlavní argument instrukce. Těmito argumenty jsou:

- zimm – hodnota aritmetické operace obsažená přímo ve slově instrukce (přímá hodnota, jejíž bitová šířka je menší než 16), která je před vstupem do ALU rozšířena o nuly
- oimm – přímá hodnota aritmetické operace rozšířená před vstupem do ALU o jedničky
- imm – přímá hodnota rozšířená o nuly
- reg – hlavním operandem je registr
- desp – přímá hodnota v podobě odsazení ve dvojkovém doplňku pro realizaci skoků v programu

Každý instrukční formát sdružuje instrukce, které se do velké míry zpracovávají na hardwarové úrovni stejně. Například instrukce logického součtu a logického součinu dvou registrů potřebují načíst dva operandy z registrů a výsledek do registru uložit. Specifická činnost nastává pouze ve výpočtu těchto logických funkcí v ALU.

Instrukční formát je uložen ve 3 vrchních bitech každého instrukčního slova. Zpracování a typ dat zbylých třinácti bitů pak přímo závisí na tomto instrukčním formátu.

Další alternativou dekódování instrukcí pomocí instrukčních formátů by bylo použití tzv. prefixového bitu. Na rozdíl od předešle použité metody je instrukční formát dán vždy pouze jedním bitem. Návrh instrukční sady postupuje od MSB, pro který zvolíme pevnou hodnotu, a všechny zbylé bity pak popisují funkci konkrétního instrukčního formátu. Poté přiřadíme hodnotu MSB do bitu nižší významnosti a znegujeme MSB. Další instrukční formát pak vznikne za bitem MSB-1. Takto postupujeme až do vyčerpání kapacity šířky instrukčního slova. Výsledkem může být rozsáhlejší (jemnější) využití instrukční sady, ovšem na úkor jednoduchosti.

6.4.3 Aritmetika s přímou hodnotou (kladnou)

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
1	0	0	0	func			imm4				Ry			Rz			Arithmetic (zimm)

Tab. 6.4.3.1 Instrukční slovo Aritmetiky s přímou hodnotou (kladnou)

func			ASM mnemonic	C language
0	0	0	ST Rz, Ry, zimm4	*(Ry + zimm4) = Rz;
0	0	1	LD Rz, Ry, zimm4	Rz = *(Ry + zimm4);
0	1	0	ADD Rz, Ry, zimm4	Rz = Ry + zimm4;
0	1	1	X	X
1	0	0	SL Rz, Ry, zimm4	Rz = sRy < zimm4;
1	0	1	SLU Rz, Ry, zimm4	Rz = Ry < zimm4;
1	1	0	X	X
1	1	1	X	X

Tab. 6.4.3.2 Výčet instrukcí Aritmetiky s přímou hodnotou (kladnou)

Instrukční formát typu aritmetika s přímou hodnotou (kladnou) obsahuje operand přímo ve svém instrukčním slově. Velikost této hodnoty je omezena instrukční sadou na 4 bity a při vstupu do ALU je rozšířena na 16 bitů nulami. To znamená, že pokud programátor tohoto procesoru bude chtít provést nějaký výpočet s konstantou, při kterém by se ovšem použily jen instrukce, jenž tento formát podporují, nemusí načítat hodnotu do jednoho z registrů, pokud její minimální možná bitová šířka nepřesahuje 4 bity – to jsou hodnoty od 0 do 15.

Aritmetika s přímou hodnotou (kladnou) obsahuje tyto instrukce:

- ST Rz, Ry, zimm4
    - ST – store – uložit
    - Uloží hodnotu Rz na adresu (Ry + imm4)
- LD Rz, Ry, zimm4
    - LD – load – načíst
    - Načte hodnotu z adresy (Ry + imm4) do Rz
- ADD Rz, Ry, zimm4
    - ADD – addition – součet
    - Sečte Ry a imm4, výsledek uloží do Rz

- SL(U) Rz, Ry, zimm4
  - SL – set if less (unsigned) – nastav, pokud je menší (bez znaménka)
  - Do Rz se uloží hodnota 1, pokud je hodnota Ry menší než imm4, jinak do Rz bude zapsána hodnota 0 (forma bez znaménka nebere při vyhodnocování ALU komparace ohled na znaménka operandů)

6.4.4 Aritmetika s přímou hodnotou (zápornou)

	15	14	13	12 11 10	9 8 7 6	5 4 3	2 1 0	
2	0	0	1	func	imm4	Ry	Rz	Arithmetic (oimm)

Tab. 6.4.4.1 Instrukční slovo Aritmetiky s přímou hodnotou (zápornou)

func			ASM mnemonic	C language
0	0	0	ST Rz, Ry, oimm4	*(Ry + oimm4) = Rz;
0	0	1	LD Rz, Ry, oimm4	Rz = *(Ry + oimm4);
0	1	0	ADD Rz, Ry, oimm4	Rz = Ry + oimm4;
0	1	1	X	X
1	0	0	SL Rz, Ry, oimm4	Rz = sRy < oimm4;
1	0	1	SLU Rz, Ry, oimm4	Rz = Ry < oimm4;
1	1	0	X	X
1	1	1	X	X

Tab. 6.4.4.2 Výčet instrukcí Aritmetiky s přímou hodnotou (kladnou)

Instrukční formát typu aritmetika s přímou hodnotou (zápornou) je obdobou formátu předešlého. Rozdílem je, že přímá hodnota, která je taktéž součástí instrukčního slova, je před vstupem do ALU rozšířena do 16 bitů o jedničky. Velmi jednoduše je tedy implementována i práce se zápornými konstantami. To je taky důvodem proč tento instrukční formát ani předešlý neimplementují odečítání – odečítání znamená přičítání záporné hodnoty. Samotné rozdělení těchto dvou instrukčních formátů namísto jejich sjednocení je podloženo jednoduchostí a pravidelností. Implementace takto podobných instrukčních formátů odděleně plní návrh pravidelných instrukčních formátů. Kdyby se tyto instrukční formáty spojily a operand by byl ve dvojkovém doplňku, byl by stále instrukční formát uložen ve 3 vrchních bitech instrukčního slova, ovšem tento formát by již nezabíral 1/8 ze všech instrukčních formátů jako každý jiný instrukční formát, a tudíž by se

to v konečném důsledku projevilo mírně zvýšenou náročností hardwaru při dekódování instrukcí. V případě imaginárního jazyka symbolických adres pro tento procesor by byla skutečnost, že neexistuje sjednocený formát pro aritmetické operace s přímým operandem v dvojkovém doplňku, úplně skryta a programátor by mohl standardně pracovat i se zápornými hodnotami. Nicméně i na hardwarové úrovni se ve skutečnosti jedná o přímou hodnotu zapsanou formátem dvojkového doplňku. Jediný vjemový rozdíl je, že v instrukčním slově je znaménkový bit přímé hodnoty, neboli MSB, uložen v LSB operačního kódu pro nulové hodnoty zbylých bitů a nikoli zleva vedle přímé hodnoty.

Aritmetika s přímou hodnotou (zápornou) až na její záporný operand obsahuje stejné instrukce jako aritmetika s přímou hodnotou (kladnou).

6.4.5 Logika s přímou hodnotou

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
3	0	1	0	func			imm4				Ry			Rz			Logic (imm)

Tab. 6.4.5.1 Instrukční slovo Logiky s přímou hodnotou

func			ASM mnemonic	C language
0	0	0	OR Rz, Ry, imm4	Rz = Ry | imm4;
0	0	1	NOR Rz, Ry, imm4	Rz = ~(Ry | imm4);
0	1	0	AND Rz, Ry, imm4	Rz = Ry & imm4;
0	1	1	NAND Rz, Ry, imm4	Rz = ~(Ry & imm4);
1	0	0	XOR Rz, Ry, imm4	Rz = Ry ^ imm4;
1	0	1	SLL Rz, Ry, imm4	Rz = Ry << imm4;
1	1	0	SRL Rz, Ry, imm4	Rz = Ry >> imm4;
1	1	1	SRA Rz, Ry, imm4	Rz = sRy >> imm4;

Tab. 6.4.5.2 Výčet instrukcí Logiky s přímou hodnotou

Instrukční formát typu Logika s přímou hodnotou obsahuje stejně jako dva předešlé instrukční formáty operand přímo v instrukčním slově. Jeho velikost je opět omezena instrukční sadou na 4 bity a při vstupu do ALU je rozšířen na 16 bitů nulami. Za povšimnutí stojí fakt, že mezi instrukčními formáty není žádná obdoba tohoto instrukčního formátu se zápornými hodnotami. Je to především způsobeno požadavky, které se od tohoto instrukčního formátu očekávají. Tyto instrukce jsou určeny pro logické

výrazy s konstantou, jejíž hodnota je v rozsahu od 0 do 15, což pokryje všechny hodnoty šestnáctkové číselné soustavy (0 - F). Část tohoto instrukčního formátu také definuje bitové posuny, jimiž se dá například implementovat s dostatečně nízkou asymptotickou složitostí softwarové násobení libovolných dvou čísel. Bitové posuny se ovšem zpravidla a častěji používají na velmi rychlé násobení, při kterém je hodnota jednoho z operandu rovna n-té mocnině čísla 2. Pro realizaci dělení se taky používá, ovšem musí této n-té mocnině čísla 2 být roven dělitel.

Logika s přímou hodnotou obsahuje tyto instrukce:

- (N)OR Rz, Ry, imm4
    - (N)OR – (not) logical disjunction – (negace) logická disjunkce
    - Provede logickou disjunkci mezi hodnotami jednotlivých bitů Ry a imm a výsledek (popřípadě logicky znegovaný) uloží do Rz
- (N)AND Rz, Ry, imm4
    - (N)AND – (not) logical conjunction – (negace) logická konjunkce
    - Provede logickou konjunkci mezi hodnotami jednotlivých bitů Ry a imm a výsledek (popřípadě logicky znegovaný) uloží do Rz
- XOR Rz, Ry, imm4
    - XOR – logical exclusive disjunction – exklusivní logická disjunkce
    - Provede exklusivní logickou disjunkci mezi hodnotami jednotlivých bitů Ry a imm a výsledek uloží do Rz
- SLL/SRL Rz, Ry, imm4
    - SLL/SRL – shift left/right logical – logický posun doleva/doprava
    - Bitová podoba hodnoty Ry se posune doleva/doprava (opačná strana je doplněna nulami) o hodnotu, kterou udává imm4 a následně je uložena do Rz
- SRA Rz, Ry, imm4
    - SRA – shift right arithmetic - aritmetický posun doprava
    - Bitová podoba hodnoty Ry se posune doprava (zleva doplněna o hodnotu znaménkového bitu Ry) o hodnotu, kterou udává imm4 a následně uložena je do Rz.

6.4.6 Aritmetika a logika s registry

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
4	0	1	1	func				Rx			Ry			Rz			Arithmetic + logic (reg)

Tab. 6.4.6.1 Instrukční slovo Aritmetiky a logiky s registry

func				ASM mnemonic	C language
0	0	0	0	OR Rz, Ry, Rx	Rz = Ry | Rx;
0	0	0	1	NOR Rz, Ry, Rx	Rz = ~(Ry | Rx);
0	0	1	0	AND Rz, Ry, Rx	Rz = Ry & Rx;
0	0	1	1	NAND Rz, Ry, Rx	Rz = ~(Ry & Rx);
0	1	0	0	XOR Rz, Ry, Rx	Rz = Ry ^ Rx;
0	1	0	1	SLL Rz, Ry, Rx	Rz = Ry << (Rx & 0x000F);
0	1	1	0	SRL Rz, Ry, Rx	Rz = Ry >> (Rx & 0x000F);
0	1	1	1	SRA Rz, Ry, Rx	Rz = sRy >> (Rx & 0x000F);
1	0	0	0	X	X
1	0	0	1	X	X
1	0	1	0	ADD Rz, Ry, Rx	Rz = Ry + Rx;
1	0	1	1	SUB Rz, Ry, Rx	Rz = Ry - Rx;
1	1	0	0	SL Rz, Ry, Rx	Rz = sRy < sRx;
1	1	0	1	SLU Rz, Ry, Rx	Rz = Ry < Rx;
1	1	1	0	X	X
1	1	1	1	X	X

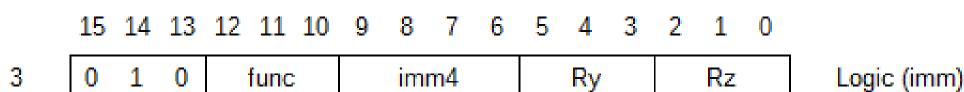
Tab. 6.4.6.2 Výčet instrukcí Logiky s přímou hodnotou

Instrukční formát typu Aritmetika a logika s registry umožňuje provádět základní aritmetické a logické operace výhradně mezi registry procesoru. Tento instrukční formát podporuje doposud všechny zmíněné instrukce u jiných instrukčních formátů a navíc implementuje i instrukci odečítání. Podstatnou změnou je použití instrukcí bitového posunu. Zatímco u instrukčních formátů, které umožňovaly použít 4bitovou konstantu pro bitový posun přímo, zde je na registr, který udává, o kolik se hodnota v jiném registru posune, aplikováno modulo čísla 16. Zajistí se tím, že hodnota registru, kterým se bude posouvat, zůstane stále v požadovaném rozsahu.

Z důvodu, že téměř všechny instrukce v tomto instrukčním formátu již byly předem popsány s operandem přímé hodnoty, bude i pro jejich podobnost popis pro formát s registry vypuštěn. Popis bude omezen pouze na jednu instrukci:

- SUB Rz, Ry, Rx
  - SUB – subtraction – odčítání
  - Odečte hodnotu Rx od Ry, výsledek uloží do Rz

6.4.7 Načítaní přímé hodnoty



Tab. 6.4.7.1 Instrukční slovo Aritmetiky a logiky s registry

func		ASM mnemonic	C language
0	0	LI Rz, imm8	Rz = imm8;
0	1	LIS Rz, imm8	Rz = imm8 << 8;
1	0	LIL Rz, imm8	Rz = (Rz & 0xFF00) + imm8;
1	1	LIH Rz, imm8	Rz = (imm8 << 8) + (Rz & 0x00FF);

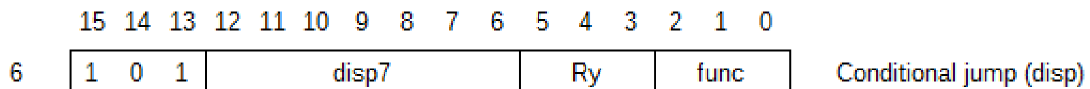
Tab. 6.4.7.2 Výčet instrukcí Logiky s přímou hodnotou

Tento instrukční formát je naprosto základní pro načítání jakékoli konstanty do registru procesoru. Pokud programátorovi například nebude stačit 4bitová konstanta obsažená v instrukčním slově předchozích instrukčních formátů, může využít jeden ze čtyř módů pro načítání konstant a provést instrukci, která pracuje s registry. Ne zřídkakdy se stává, že tyto instrukce jsou používány i pro načítání adresy nějakého prvku (datové struktury, pole a jiné) - následující zpracování pak většinou probíhá bez dalšího načítání těchto konstant. Je důležité poznamenat, že pro načtení některých 16bitových hodnot, vzhledem k RISCové architektuře tohoto procesoru, je zapotřebí sekvence dvou instrukcí. Nicméně tento instrukční formát byl navržen tak, aby bylo umožněno načítat co největší množství konstant jednou instrukcí.

Formát načítání přímé hodnoty obsahuje tyto instrukce:

- LI Rz, imm8
    - LI – load immediate – načíst přímo
    - Do Rz uloží hodnotu imm8
- LIS Rz, imm8
    - LIS – load immediate scaled – načíst přímo rozšířeně
    - Do Rz uloží hodnotu, která je rovna (256 x imm8)
- LIL Rz, imm8
    - LIL – load immediate low – načíst přímo spodní
    - Spodní osmice bitů Rz je nahrazena imm8 a výsledek je uložen do Rz
- LIH Rz, imm8
    - LIH – load immediate high – načíst přímo vrchní
    - Vrchní osmice bitů Rz je nahrazena imm8 a výsledek je uložen do Rz

6.4.8 Podmíněné skoky s odsazením



	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0	
6	1	0	1	disp7							Ry			func			Conditional jump (disp)

Tab. 6.4.8.1 Instrukční slovo Podmíněných skoků s odsazením

func			ASM mnemonic	C language
0	0	0	JNE Rz, disp7	if(Rz != 0) { Rz = IP; IP += disp7; }
0	0	1	JE Rz, disp7	if(Rz == 0) { Rz = IP; IP += disp7; }
0	1	0	JL Rz, disp7	if(sRz < 0) { Rz = IP; IP += disp7; }
0	1	1	JLE Rz, disp7	if(sRz <= 0) { Rz = IP; IP += disp7; }
1	0	0	JG Rz, disp7	if(sRz > 0) { Rz = IP; IP += disp7; }
1	0	1	JGE Rz, disp7	if(sRz >= 0) { Rz = IP; IP += disp7; }
1	1	0	X	X
1	1	1	X	X

Tab. 6.4.8.2 Výčet instrukcí Podmíněných skoků s odsazením

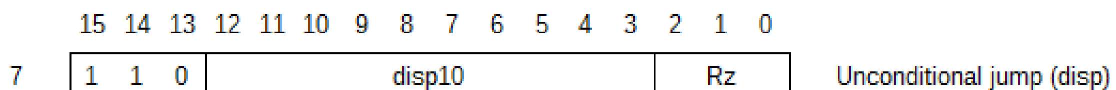
Instrukční formát podmíněné skoky s odsazením slouží k podmíněnému větvení programu. Jakákoli forma podmíněného vykonávání kódu programu je při nejmenším vedle dostatku

paměti klíčovým požadavkem k implementaci všech známých algoritmů, které mají řešení v konečném čase. V procesoru Limen je použit pouze model podmíněných skoků. Instrukce jako podmíněné přiřazení hodnoty z jednoho registru do druhého nebo podmíněné vynechání následující instrukce nebyly z důvodu jednoduchosti implementovány. Pro výpočet adresy, která bude přesunuta do registru IP, se použije odsazení. Odsazení je hodnota ve dvojkovém doplňku, která je uchována přímo v instrukčním slově. Její bitová šířka je 7 bitů a před vstupem do ALU pro součet s hodnotou IP je doplněna hodnotou nejvyššího bitu, čili programátor může provádět relativní podmíněný skok až o 64 instrukcí zpět a až o 63 vpřed vzhledem k adrese aktuální instrukce.

Formát podmíněné skoky s odsazením obsahují tyto instrukce:

- J(N)E Rz, disp7
    - J(N)E – jump if (not) equal – skoč, pokud je/není rovno
    - Provede podmíněný skok, pokud je/není Rz rovno nule na adresu (IP + disp7)
- JL(E) Rz, disp7
    - JL(E) – jump if less (or equal) – skoč, pokud je menší (nebo rovno)
    - Provede podmíněný skok, pokud je Rz menší/menší nebo rovno než nula na adresu (IP + disp7)
- JG(E)
    - JG(E) – jump if greater (or equal) – skoč, pokud je větší (nebo rovno)
    - Provede podmíněný skok, pokud je Rz větší/větší nebo rovno než nula na adresu (IP + disp7)

6.4.9 Nepodmíněné skoky s odsazením



	15	14	13	12–3	2–0	
7	1	1	0	disp10	Rz	Unconditional jump (disp)

Tab. 6.4.9.1 Instrukční slovo Nepodmíněných skoků s odsazením

func	ASM mnemonic	C language
X	JWL Rz, disp10	Rz = IP; IP += disp10;

Tab. 6.4.9.2 Výčet instrukcí Nepodmíněných skoků s odsazením

Instrukční formát nepodmíněné skoky s odsazením umožňuje především implementovat elementární základ k volání procedur. Rozsáhlé využití nalezne taky v přímé změně hodnoty IP, což způsobí změnu adresy vykonávání programu. To se velmi často používá právě v mnoha větvích, do kterých se program dostal z důvodu větvení podmíněnými skoky (používá se například z důvodu stejné návratové adresy pro všechny tyto větve). Na výpočet výsledné adresy uložené do IP se používá taky odsazení, jehož bitová šířka je na rozdíl od předešlého instrukčního formátu větší; a to je 10 bitů. Vzhledem k aktuální adrese IP se dá provést relativní skok až o 512 instrukcí zpět a až o 511 instrukcí vpřed. Aby mohla jediná instrukce implementovat instrukci nepodmíněného skoku i volání procedury, byla navržena následujícím způsobem. Při svém provádění instrukce vždy uloží svou tzv. návratovou adresu (tj. aktuální hodnotu IP) do zvoleného registru – implementace volání procedury. Pokud předáme jako operand registr R0, návratová adresa, jak vyplývá s charakteristiky samotného procesoru, nebude nikde uložena – implementace nepodmíněného skoku.

Formát nepodmíněné skoky s odsazením obsahuje jedinou instrukci:

- JWL Rz, disp10
    - JWL – jump with link – skoč s odkazem
    - Do registru Rz je uložena aktuální hodnota registru IP a provede nepodmíněný skok na adresu (IP + disp10)

6.4.10 Nepodmíněné skoky s registrem

	15	14	13	12–6	5–3	2–0	
8	1	1	1	SBZ	Ry	Rz	Unconditional jump (reg)

Tab. 6.4.10.1 Instrukční slovo Nepodmíněných skoků s registrem

func	ASM mnemonic	C language
X	JWL Rz, Ry	Rz = IP; IP = Ry;

Tab. 6.4.10.2 Výčet instrukcí Nepodmíněných skoků s registrem

Instrukční formát nepodmíněné skoky s registrem umožňuje především implementovat skoky na vzdálená návěští, nebo vzdálená volání procedur (nemá nic společného

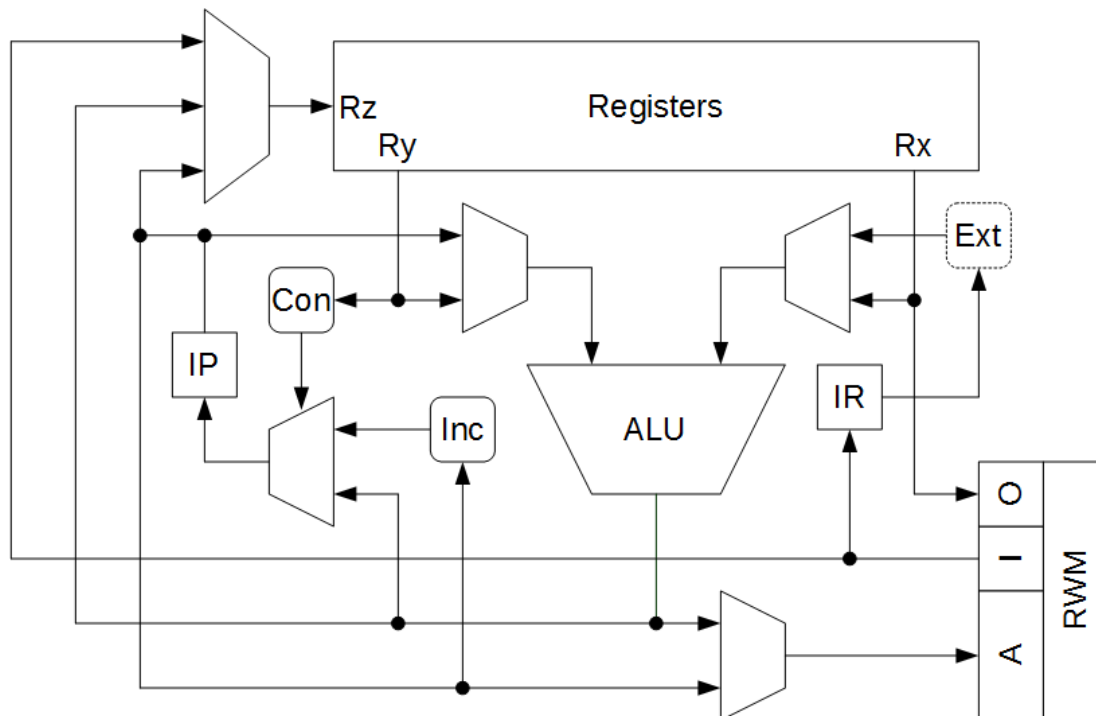
s technologií RPC). Jedinou instrukcí, kterou obsahuje tento instrukční formát, se na rozdíl od předešlého instrukčního formátu provádí skok absolutní. Skoky na vzdálená návěští se často používají v případě, že programátor vytvořil program, který má vzdálenost mezi dvěma částmi kódu, které na sebe navazují skokem (zpravidla nepodmíněným), větší než rozsah hodnot dvojkového doplňku o šířce 10 bitů. Těmto situacím by se měl ovšem moderní kompilátor co nejvíce vyvarovat, ale někdy opravdu jiné východisko není. Vzdálená volání procedur se taktéž používají při volání vlastních procedur programu s podmínkou, že jsou taky mimo rozsah hodnot skoku relativního. Nicméně mezi nejpodstatnější využití patří volání procedur externích knihoven, které byly přidány do programu (zpravidla se totiž do rozsahu relativního skoku nevlezou). Dále se mezi nejpodstatnější využití řadí skoky na adresy, nad kterými je potřeba předem provést nějaké operace. Příkladem může být samotný ukazatel na proceduru, tabulka adres pro skok nebo specifické konstrukce, z nichž lze konkrétně zmínit konstrukci switch z jazyka C, na kterou může být aplikováno hned několik optimalizací, které s využitím tohoto instrukčního formátu úzce souvisí. Pro volání procedury se stejně jako v předchozím instrukčním formátu použije libovolný registr pro uložení návratové adresy. Pokud programátor bude chtít provést pouze skok, předá jako operand pro uložení návratové adresy registr R0.

Formát nepodmíněné skoky s registrem obsahuje jedinou instrukci:

- JWL Rz, Ry
    - JWL – jump with link – skoč s odkazem
    - Do registru Rz je uložena aktuální hodnota registru IP a provede nepodmíněný skok na adresu, která je uložená v Ry

6.5 Významné moduly procesoru Limen

6.5.1 Schéma jádra



Obr. 6.5.1 Schéma jádra procesoru Limen a RWM

Schéma jádra procesoru Limen je velmi jednoduché, přesto si však pro představu, jak přesně funguje, zaslouží určitou pozornost. Schéma podle tvaru obrazce rozlišuje tyto typy logických členů:

- Obdélník – registr, nebo člen s charakteristikou paměti
- Obdélník se zaoblenými vrcholy – kombinační výkonný prvek
- Lichoběžník – s výjimkou ALU se jedná o multiplexer

Je důležité taky poznamenat, že všechny členy, až na multiplexery a prvky označené přerušovanou čárou, jsou implementovány jako samostatný modul.

Neméně důležitý je taky fakt, že procesor pro jednoduchost komunikuje s pamětí RWM po dvou datových sběrnicích (pro každý směr jedna).

6.5.2 Aritmeticko-logická jednotka (ALU)

opcode				operation
0	0	0	0	result <= operand_l or operand_r
0	0	0	1	result <= not (operand_l or operand_r)
0	0	1	0	result <= operand_l and operand_r
0	0	1	1	result <= not (operand_l and operand_r)
0	1	0	0	result <= operand_l xor operand_r
0	1	0	1	result <= operand_l << operand_r
0	1	1	0	result <= operand_l >> operand_r (unsigned)
0	1	1	1	result <= operand_l >> operand_r (signed)
1	0	0	0	result <= operand_l + operand_r
1	0	0	1	result <= operand_l(15 downto 8) & operand_r(7 downto 0)
1	0	1	0	result <= operand_r(15 downto 8) & operand_l(7 downto 0)
1	0	1	1	result <= operand_l – operand_r
1	1	0	0	result <= operand_l < operand_r (signed)
1	1	0	1	result <= operand_l < operand_r (unsigned)
1	1	1	0	result <= operand_l
1	1	1	1	result <= operand_r

Tab. 6.5.2 Výčet operačních kódů aritmeticko-logické jednotky

Operační kód pro ALU lze přirovnat k instrukční sadě pro procesor. ALU je ovšem na rozdíl od procesoru implementována jako čistě kombinační obvod. ALU rozeznává 16 operačních kódů, které většinou přímo souvisí s vykonávanou instrukcí. Na svém vstupu očekává dva 16bitové operandy, mezi kterými pak bude provedena matematická nebo logická operace zvolená na základě vstupního operačního kódu. Výstupem ALU je vždy jen 16bitový výsledek, který je dále v procesoru zpracován v závislosti na několika signálech, které jsou většinou závislé jen na vykonávané instrukci.

Procesor Limen nazývá dva operandy, které vstupují do ALU jako levý a pravý. Je tomu tak z důvodu přehlednosti. Levý operand pak na obrázku schématu č. 6.5.1 opravdu odpovídá operandu, který vstupuje do ALU zleva, pravý operand pak vstupuje analogicky zprava.

Z tabulky č. 6.5.2 lze poznat skrze popis jednotlivých operačních kódů v pseudo-jazyce VHDL jaký je vztah mezi operandy pro obdržení požadovaného výsledku na výstupu ALU. Z toho důvodu bude popsáno využití jen částečně nejasných operačních kódů:

- 1001 – result <= operand_l(15 downto 8) & operand_r(7 downto 0)
    - Vrchní osmice bitů levého operandu se spojí se spodní osmicí pravého operandu
    - Tento operační kód se používá k realizaci instrukce s mnemonikou LIL
- 1010 – result <= operand_r(15 downto 8) & operand_l(7 downto 0)
    - Vrchní osmice bitů pravého operandu se spojí se spodní osmicí levého operandu
    - Tento operační kód se používá k realizaci instrukce s mnemonikou LIH
- 1110 – result <= operand_l
    - Výsledek je roven levému operandu ALU
    - Využívá se k realizaci instrukce s mnemonikou a operandy JWL Rz, Ry, kde je skrze levý operand přeposlána hodnota Ry pro skok na tuto adresu
- 1111 – result <= operand_r
    - Výsledek je roven pravému operandu ALU
    - Využívá se k realizaci instrukcí s mnemonikou LI a LIS

6.5.3 Tester podmínek (Con)

cond_type			operation
0	0	0	jump_ack <= '1' when input_data /= 0 else '0'
0	0	1	jump_ack <= '1' when input_data = 0 else '0'
0	1	0	jump_ack <= '1' when input_data < 0 else '0'
0	1	1	jump_ack <= '1' when input_data <= 0 else '0'
1	0	0	jump_ack <= '1' when input_data > 0 else '0'
1	0	1	jump_ack <= '1' when input_data >= 0 else '0'
1	1	0	jump_ack <= '0'
1	1	1	jump_ack <= '1'

Tab. 6.5.3 Výčet kombinací testeru podmínek

Tester podmínek je základní kombinační element pro realizaci podmíněných skoků. Vstupem je hodnota registru, která se porovnává výhradně s nulou, a typ podmínky ke splnění. Jeho jednobitový výstup je pak závislý na kombinaci zvolených podmínek a při podmíněných skocích i na příznacích, které splňuje hodnota registru k testování. Tento výstup je pak přímo adresovým signálem dvouvstupého multiplexeru, který vybírá následující hodnotu IP. Multiplexer vybírá mezi inkrementovanou hodnotou IP a výstupem ALU. V případě, že je výstup testeru roven jedničce, provede se změna IP na hodnotu, která je výstupem ALU. Pokud je roven nule, změní se hodnota IP na inkrementovanou hodnotu IP, čili program pokračuje ve vykonávání bez skoku.

Tester podmínek umožňuje rozeznat pro vstupní testovanou hodnotu registru konkrétně tyto vlastnosti:

- 000 - není rovna nule
- 001 - rovna nule
- 010 - menší než nula
- 011 - menší nebo rovna nule
- 100 - větší než nula
- 101 - větší nebo rovna nule
- 110 - výstup není závislý na vstupní hodnotě registru, je vždy nula (každá instrukce, která neprovádí skok)
- 111 - výstup není závislý na vstupní hodnotě registru, je vždy jedna (každá instrukce nepodmíněného skoku)

6.5.4 Rozšiřovač přímých hodnot (Ext)

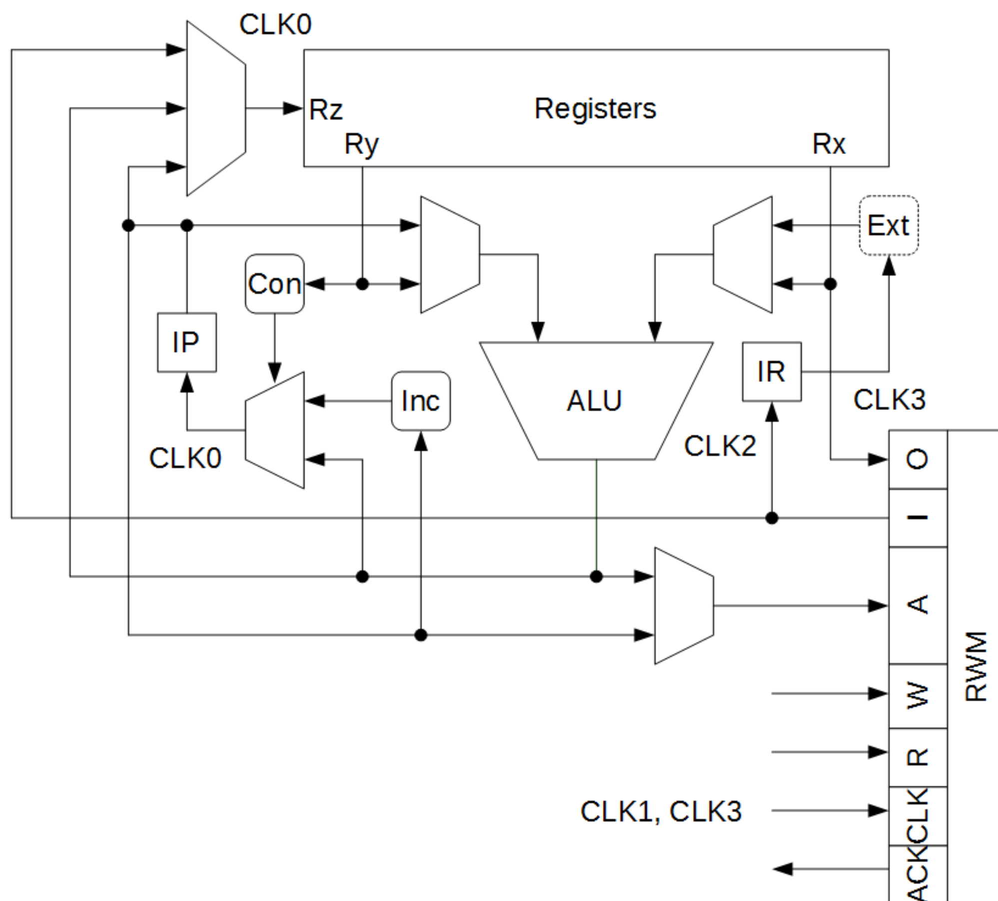
Rozšiřovač přímé hodnoty není implementován jako samostatný modul, jak bylo původně zamýšleno, ovšem je relativně důležitý. I proto zde bude stručně popsán.

Rozšiřovač přímých hodnot plní v procesoru funkci doplňování jedniček, nebo nul do zbylé části načítané konstanty pravého operandu ALU, který je umístěn přímo v instrukčním slově. Tato přímá hodnota je vždy doplňována do 16 bitů shora. To znamená, že její bitová podoba, ať už je obsažena kdekoli v instrukčním slově, se po zpracování rozšiřovače objeví v jeho výstupu ve spodní části.

Podle instrukčního formátu doplňuje hodnoty vyjmuté z instrukčního slova do 16 bitů následovně:

- Aritmetika s přímou hodnotou (kladnou) – bity 6 až 9 instrukčního slova jsou doplněny do 16 bitů nulami
- Aritmetika s přímou hodnotou (zápornou) – bity 6 až 9 jsou doplněny do 16 bitů jedničkami
- Logika s přímou hodnotou – bity 6 až 9 jsou doplněny do 16 bitů nulami
- Načítaní přímé hodnoty – bity 3 až 10 jsou doplněny do 16 bitů nulami
- Podmíněné skoky s odsazením – bity 6 až 12 jsou doplněny do 16 bitů hodnotou bitu 12 (znaménkový bit dvojkového doplňku)
- Nepodmíněné skoky s odsazením – bity3 až 12 jsou doplněny do 16 bitů hodnotou bitu 12 (znaménkový bit dvojkového doplňku)

6.6 Popis práce jádra procesoru Limen



Obr. 6.6 Schéma jádra procesoru Limen a RWM s několika řídícími signály

Obrázek č. 6.6 je do určité míry velmi podobný obrázku č. 6.5.1. Rozdílem je, že na obrázku č. 6.6, který bude popisován následující částí tohoto textu, přibily čtyři řídící signály, navedené mezi jádrem Limen procesoru a RWM. Tyto řídící signály jsou potřebné k vysvětlení práce procesoru Limen. Taky přibyly popisky CLK0, CLK1, CLK2, CLK3, které jsou taktéž klíčovými k vysvětlení práce procesoru Limen.

6.6.1 Časování

Než bude popsána práce procesoru, bylo by vhodné nejprve zmínit jeho časování. Do procesoru vstupuje hodinový signál, jehož čtvrtinová frekvence je rovna frekvenci celého procesoru. To znamená, že 4 hodinové takty jsou rovny 1 procesorovému taktu. Je to způsobeno tím, že instrukce prochází při svém zpracování čtyřmi částmi procesoru, které uloží zpracovávanou hodnotu do nějakého registru a pak vykonají pomocí kombinační logiky vždy jen svůj dílčí krok. Časování těchto částí řídí řadič procesoru, který při každém vstupním hodinovém taktu rotuje jedničku ve 4 bitovém registru, ze kterého jsou pak časované jednotlivé části (resp. proveden zápis na náběžnou hranu do registru před kombinační částí). Pro značení těchto hodinových signálů budou použity názvy CLK0, CLK1, CLK2, CLK3 (dále jen řídící hodinové signály), stejně jako na obrázku č. 6.6.

6.6.2 Řídící hodinové signály

Řídící hodinové signály mají následující vlivy na procesor:

- CLK0
  - probíhá zápis do IP a do registru, který je indexován Rz (v případě, že žádný zápis neprobíhá, je hardwarem zaručeno, že index Rz bude mít hodnotu 0, tudíž se nezmění hodnota žádného registru)
- CLK1
  - v tuto chvíli je již na vodiči R jednička pro čtení další instrukce z paměti RWM
  - nastává nástupní hrana hodinového signálu, který řídí paměť RWM a instrukce bude na datové sběrnici I, vedoucí z RWM
- CLK2
  - v tuto chvíli je již na vstupní datové sběrnici I instrukce
  - probíhá zápis do registru IR a potom začne se provádět hlavní část instrukce v nejrozsáhlejší kombinační části

- CLK3
    - tato poslední část celého strojového taktu je velmi individuální v závislosti na vykonávané instrukci
    - pokud se do RWM zapisuje, kombinační logika předem nastavila na vodič W jedničku, na adresní sběrnici A vstupující do RWM požadovanou adresu a na datovou sběrnici vstupující do RWM hodnotu, která se bude ukládat – registr indexovaný Rz a nástupní hrana pak při tomto taktu způsobit zápis do RWM
    - pokud se z RWM čte
        - kombinační logika předem nastavila vodič R do hodnoty 1, na adresní sběrnici A vstupující do RWM požadovanou adresu a do doby, dokud na libovolný následující takt CLK0 nebude na potvrzovacím vodiči z paměti ACK jednička, nechá tyto signály nastavené a v průběhu čekání na paměť se nic jiného v procesoru neděje
        - jakmile bude na výstupním signálu paměti ACK jednička na nástupnou hranu CLK0, zapíše do registru indexovaného Rz hodnota na vstupní datové sběrnice a pokračuje se ve vykonávání instrukcí
    - pokud se neprovádí žádná operace s RWM, tento takt nemá žádný efekt

6.6.3 Příklad

Pro opravdové pochopení bude uvedena jedna instrukce z instrukční sady a na ní pak bude detailně vysvětlený veškerý doposud rozebíraný postup zpracovávání instrukcí.

Pro příklad byla zvolena instrukce čtení z paměti, jelikož se dá považovat za nejsložitější na zpracování touto metodou.

Mnemonika příkladové instrukce i s operandy: LD R1, R5, =8

Postup:

- CLK0
  - nastane zápis do souboru registrů z předchozí instrukce
  - nastane zápis adresy příkladové instrukce LD do IP registru
  - od doby, kdy nastane náběžná hrana tohoto signálu, provádí se hned několik operací současně:
    - multiplexer vedoucí na adresní sběrnici paměti se nastaví na výstup z IP registru pro čtení instrukce
    - nastavení signálu R do jedničky
- CLK1
  - stejný průběh jako v sekci 6.6.2
- CLK2
  - stejný průběh jako v sekci 6.6.2 a
  - od doby, kdy nastane náběžná hrana tohoto signálu, provádí se hned několik operací současně:
    - multiplexer pro pravý operand se nastaví na přímou hodnotu
    - z instrukčního slova se skrze Rozšiřovač rozšíří hodnota 8 na 16 bitovou podobu pro vstup jako pravý operand do ALU
    - multiplexer pro levý operand se nastaví na registr
    - načítá se registr R5 do levého operandu ALU
    - do ALU vstupuje operační kód součtu
    - multiplexer vedoucí na adresní sběrnici paměti se nastaví na výstup z ALU
    - nastavení signálu R do jedničky
    - do Testeru podmínek vstupuje podmínka k testování 110, což znamená, že se skok neprovede nikdy, čili po této instrukci bude probíhat ta následující
    - multiplexer vedoucí do souboru registrů je nastaven na vstupní datovou sběrnici z paměti RWM
- CLK3
  - stejný průběh jako v sekci 6.6.2

6.7 Implementace důležitých programovacích konstrukcí

6.7.1 Příznak přenosu

U většiny procesorů implementujících statusový registr je získat příznak přenosu otázka jedné instrukce. Limen procesor je však navržený trochu jinak, bude-li potřeba získat příznak přenosu u neznaménkového sčítání, je zapotřebí dvou za sebou jdoucích instrukcí: ADD, SLU. Konkrétními instrukcemi, umožňující toto realizovat, i se značenými operandy jsou:

```
; B = výsledek, D = operand 1, F = operand 2
ADD B, D, F        ; B = D +F
SLU A, B, F        ; A = B < F
; A = 1 při přetečení, jinak 0
```

Pokud se sčítají dvě čísla, jejich výsledek nemůže být nikdy menší než jakékoliv z těchto dvou čísel. Této vlastnosti se dá využít právě ve chvíli, kdy se pracuje s omezeným množstvím hodnot a v případě, kdy se překročením těchto hodnot sečte přesah s nulou.

Tento příklad ukazuje jen jeden způsob použití instrukce SL. Těchto použití je ale veliká spousta. Například přidáním dvou instrukcí k těm stávajícím lze implementovat 32 bitové sčítání:

```
; [A, B] = [C, D] + [E, F], kde každé písmeno představuje 16 bitový registr
ADD B, D, F        ; B = D +F
SLU A, B, F        ; A = B < F
ADD A, A, C        ; A += C
ADD A, A, E        ; A += E
```

První se sčítá spodních 16 bitů, a pokud při jejich součtu dojde k přetečení, do registru A se uloží hodnota 1, jinak nula. Ta se pak sečte s oběma vrchními 16bitovými částmi.

6.7.2 Podmíněné skoky

Limen nabízí pouze podmíněné skoky, které jsou porovnávány s nulou. I když to tak možná nevypadá, více není potřeba. Pomocí instrukční sady procesoru Limen, zejména pak instrukce SL, lze spolu s použitím podmíněných skoků implementovat jakýkoliv podmíněný skok, který porovnává dvě čísla, ať už v podobě hodnoty registru, nebo i konstanty.

Implementace různých podmíněných skoků se značenými operandy pomocí instrukční sady procesoru Limen lze provést:

- JL – Jump if Less – skoč, pokud je A menší než B
  - SL C, A, B
  - JNE C, návěští
- JLE – Jump if Less or Equal – skoč, pokud je A menší než B nebo jsou si rovny
  - SL C, B, A
  - JE A, návěští
- JG – Jump if Greater – skoč, pokud je A větší než B
  - SL C, B, A
  - JNE A, návěští
- JGE – Jump if Greater or Equal – skoč, pokud je A větší než B nebo jsou si rovny
  - SL C, A, B
  - JE A, návěští

ZÁVĚR

Ačkoli byla dokumentace k maturitnímu výrobku spíše teoretického charakteru, nemyslím si, že by tato celková maturitní práce byla vzhledem ke své náročnosti nedostatečně zpracována. Kdyby se práce měla zabývat rozborem samotného kódu VHDL procesoru, který bude mimochodem přiložen k této práci, brzy by přestala být informativní a spadla by k obvyklým stereotypům, které si přečte jen málokdo. Navíc by byla pravděpodobně daleko rozsáhlejší a nebyl by žádný viditelný přínos.

Původně tato maturitní práce měla nabývat mnohem většího rozsahu a i navzdory tomu, že jsem na ní pracoval v průměru větší část každého dne a chtěl jsem, aby postavila základy architektury procesoru, která se v budoucnu bude blížit k dokonalosti, zbylo mi jen 15 dní na vypracování a dokumentaci procesoru Limen, jehož účelem bylo splnit zadání této práce a poskytnout všem čtenářům dokumentace zdroje velmi užitečných informací, které jsem sám shromažďoval několik let.

Původní procesor měl obsahovat většinu prvků, kterými disponuje i procesor Limen, ovšem měl promítat architekturu procesoru budoucnosti do jednoduchosti 16bitového návrhu. Mělo se jednat o procesor, který by se plně držel RISCového návrhu s propracovanou ortogonalitou, implementoval by MMU a cache s vlastním algoritmem nahrazovací politiky. Návrh jeho architektury by taky umožňoval implementaci jeho vícejádrové podoby. Byl by navržen jako asynchronně řízený obvod, který má do budoucna velký potenciál, a v neposlední řadě jsem pro něj chtěl napsat na propagaci jeho možností jednoduché jádro multitaskingového operačního systému. Ale především by byl naprosto jiný než všechny ostatní moderní procesory – jednoduchý!

Nicméně i já jsem si uvědomoval prvotní rozsah této práce, ale pokud nás velké výzvy posouvají vpřed, proč si vybírat ty malé? Nakonec jsem získal v průběhu vypracovávání této maturitní práce mnoho vědomostí a zkušeností. Přečetl jsem stovky různých materiálů popisujících procesory, operační systémy (zejména pak Linux), asynchronně řízené obvody, instrukční sady procesorů, zásobníkové počítače a mnoho dalších takových, které by mi pomohly v budoucnosti vytvořit nejlepší procesor pro osobní počítače. Učinil jsem stovky rozhodnutí, díky kterým jsem tam, kde jsem. Prošel jsem se stovkami neprozkoumaných cest. Položil jsem přes stovku různých otázek svému konzultantovi, panu Ing. Janovi Pilčíkovi, několik i svým spolužákům, kteří se věnují programování. A podařilo se mi najít odpovědi na tisíce otázek.

Nicméně ani to nestačilo k tomu, aby se doposud z mé vize stala realita. Velká část materiálů pro vytvoření tohoto původního procesoru zůstala jen na papíře (několik set stránek formátu A5) a jednou, až nastane správný čas, budou klíčovými nejen pro mě, ale i pro všechny, kteří chtějí využít plný potenciál budoucích procesorů. Děkuji.

SEZNAM POUŽITÉ LITERATURY

[1] Internet, x86 [online], poslední aktualizace 18. 11. 2015 [cit. 2016-02-28], Wikipedia.

Dostupné z WWW:

<https://cs.wikipedia.org/wiki/X86>

[2] Internet, Addition [online], poslední aktualizace 11. 2. 2016 [cit. 2016-02-28], Wikipedia.

Dostupné z WWW:

< https://en.wikipedia.org/wiki/Addition>

[3] Internet, Introduction to combinational and sequentialcircuit, poslední aktualizace 2011 [cit. 2016-02-28], KKHSOU.

Dostupné z WWW:

<http://www.kkhsou.in/main/EVidya2/computer_science/combinational_sequenti al.html>

[4] Internet, Proces, poslední aktualizace 1. 5. 2015 [cit. 2016-02-28], Učíme se VHDL.

Dostupné z WWW:

< http://vhdl.cz/proces>

[5] Internet, Jak se v programování dostat za hranice fyziky, poslední aktualizace 2011 [cit. 2016-02-29], Livejournal.

Dostupné z WWW:

< http://panchul.livejournal.com/184647.html>

[6] Internet, Basys 2 ™ FPGA Board Reference Manual, poslední aktualizace 11. 11. 2011 [cit. 2016-02-29], Digilent.

Dostupné z WWW:

< https://reference.digilentinc.com/_media/basys2:basys2_rm.pdf>

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

VHDL Jazyk pro popis hardware (VHSIC Hardware Description Language)

VHSIC Velmi rychlé integrované obvody (Very High Speed Integrated Circuit)

FPGA Programovatelné hradlové pole (Filed Programmable Gate Array)

LED Dioda emitující světlo (Light-Emitting Diode)

CPU Centrální výpočetní jednotka (Central Processing Unit)

ALU Aritmeticko-logická jednotka

FPU Matematický koprocesor (Floating-Point Unit)

LIFO Zásobníková struktura (Last In First Out)

FIFO Struktura fronty (First In First Out)

MMU Jednotka správy paměti (Memory Management Unit)

PC Programový čítač (Program Counter)

DOS Diskový operační systém

RWM Operační paměť (Read Write Memory)

RISC Redukovaná instrukční sada (Reduced Instruction Set Computing)

CISC Komplexní instrukční sada (Complex Instruction Set Computing)

VLIW Velmi dlouhé instrukční slovo (Very Long Instruction Word)

IP Ukazatel instrukcí (Instruction Pointer)

ARM Pokročilý Riscový Stroj (Advanced RISC Machine)

PAL Privilegovaná knihovna architektury (Privileged Architecture Library)

TLB Mezipaměť překladu virtuálních adres na fyzické (Translation Lookasid Buffer)

HAL Harwarová abstrakční vrstva (Hardware Abstract Layer)

CLK Hodiny (Clock)

LTS Dlouhodobá podpora (Long Term Support)

IDE Vývojové prostředí (Integrated Development Environment)

GUI Grafické uživatelské rozhraní (Graphics User Interface)

MSB Nejvíce významný bit (Most Significant Bit)

LSB Nejméně významný bit (Least Significant Bit)

RPC Vzdálené volání procedur (Remote Procedure Call)

SBZ Měla by být nula (Should Be Zero)

ACK Potvrzení (Acknowledge)

SEZNAM OBRÁZKŮ A TABULEK

Tab. 2.1.3 Soubor registrů 32bitového x86 procesoru 15
Tab. 2.2.4 Soubor registrů ARM procesoru 17
Obr. 3.2.2.1 Logické schéma úplné sčítačky [2] 20
Obr. 3.2.2.2 Popis úplné sčítačky ve VHDL 21
Obr. 3.2.3.1 Logické schéma 4bitového registru [3] 22
Obr. 3.2.3.2 Popis 4bitového registru ve VHDL 22
Obr. 4.1 Ukázka GUI modifikovaného Ubuntu 14.04 25
Obr. 4.2 Ukázka programu ISE Project Navigator 26
Obr. 4.3.2 Ukázka ovládání programu Digilent Adept 2 27
Obr. 5.1.1 Ukázka vývojového kitu Digilent Basys 2 28
Obr. 5.1.2 Abstraktní schéma zapojení Basys 2 [5] 29
Tab. 6.4.2 Instrukční formáty 33
Tab. 6.4.3.1 Instrukční slovo Aritmetiky s přímou hodnotou (kladnou) 35
Tab. 6.4.3.2 Výčet instrukcí Aritmetiky s přímou hodnotou (kladnou) 35
Tab. 6.4.4.1 Instrukční slovo Aritmetiky s přímou hodnotou (zápornou) 36
Tab. 6.4.4.2 Výčet instrukcí Aritmetiky s přímou hodnotou (kladnou) 36
Tab. 6.4.5.1 Instrukční slovo Logiky s přímou hodnotou 37
Tab. 6.4.5.2 Výčet instrukcí Logiky s přímou hodnotou 37
Tab. 6.4.6.1 Instrukční slovo Aritmetiky a logiky s registry 39
Tab. 6.4.6.2 Výčet instrukcí Logiky s přímou hodnotou 39
Tab. 6.4.7.1 Instrukční slovo Aritmetiky a logiky s registry 40
Tab. 6.4.7.2 Výčet instrukcí Logiky s přímou hodnotou 40
Tab. 6.4.8.1 Instrukční slovo Podmíněných skoků s odsazením 41
Tab. 6.4.8.2 Výčet instrukcí Podmíněných skoků s odsazením 41
Tab. 6.4.9.1 Instrukční slovo Nepodmíněných skoků s odsazením 42
Tab. 6.4.9.2 Výčet instrukcí Nepodmíněných skoků s odsazením 42
Tab. 6.4.10.1 Instrukční slovo Nepodmíněných skoků s registrem 43
Tab. 6.4.10.2 Výčet instrukcí Nepodmíněných skoků s registrem 43
Obr. 6.5.1 Schéma jádra procesoru Limen a RWM 45
Tab. 6.5.2 Výčet operačních kódů aritmeticko-logické jednotky 46
Tab. 6.5.3 Výčet kombinací testeru podmínek 47
Obr. 6.6 Schéma jádra procesoru Limen a RWM s několika řídícími signály 49

SEZNAM PŘÍLOH

PŘÍLOHA P I: KONZULTACE

	Datum	Podpis vedoucího práce
Konzultace č. 1	12.11.2015	
Konzultace č. 2	14. 1. 2016	