Střední průmyslová škola a Vyšší odborná škola, Liberec, příspěvková organizace

Implementace procesoru

Ročníková práce

Autor **Jiří Černohorský**

Obor **Informační technologie**

Vedoucí práce **Ing. Petr Socha, Ph.D.**

Školní rok **2024/2025**

Počet stran **19**

Počet slov **1116**



Anotace

Tato ročníková práce se zabývá návrhem jednoduchého jednocyklového procesoru založeného na architektuře RISC-V. Procesor je vytvořen pomocí jazyka VHDL a implementován na vývojové desce FPGA.

Summary

This year-end project focuses on the design of a simple single-cycle processor based on the RISC-V architecture. The processor is developed using the VHDL language and implemented on an FPGA development board.

Čestné prohlášení

Prohlašuji, že jsem předkládanou ročníkovou práci vypracoval sám a uvedl jsem veškerou použitou literaturu a bibliografické citace. Také prohlašuji, že jsem v průběhu příprav a psaní závěrečné práce použil nástroje umělé inteligence. Vygenerovaný obsah jsem ověřil. Stvrzuji, že jsem si vědom, že za obsah závěrečné práce plně zodpovídám.

V Liberci dne

Jiří Černohorský

Obsah

[Úvod 1](#_Toc198475135)

[1 Návrh 2](#_Toc198475136)

[1.1 Instrukční sada tzv. picoRISC-V 2](#_Toc198475137)

[1.2 Použité prostředky 2](#_Toc198475138)

[2 Implementace 3](#_Toc198475139)

[2.1 Kombinační obvody 3](#_Toc198475140)

[2.1.1 Řadič 3](#_Toc198475141)

[2.1.2 Extraktor přímého operandu 4](#_Toc198475142)

[2.1.3 Aritmeticko-logická jednotka 4](#_Toc198475143)

[2.2 Sekvenční obvody 5](#_Toc198475144)

[2.2.1 Registr 5](#_Toc198475145)

[2.2.2 Datová paměť 6](#_Toc198475146)

[2.2.3 Instrukční paměť 7](#_Toc198475147)

[2.3 Top level architektura 8](#_Toc198475148)

[3 Testování 9](#_Toc198475149)

[3.1 Simulace 9](#_Toc198475150)

[3.2 Reálné FPGA 10](#_Toc198475151)

[Závěr 12](#_Toc198475152)

[Seznam zkratek a odborných výrazů 13](#_Toc198475153)

[Seznam obrázků 14](#_Toc198475154)

[Použité zdroje 16](#_Toc198475155)

[A. Seznam přiložených souborů I](#_Toc198475156)

Úvod

Dnes jsou počítače součástí celé řady věcí, ať už se jedná o lednici, chytrou matraci, nebo jen jednoduchou LED žárovku. Srdcem každého z nich je procesor neboli CPU (Central Processing Unit). Pro většinu lidí se jedná pouze o „černou skříňku“, i když právě on je zodpovědný za vykonávání instrukcí programů a rozhoduje o tom, co, kdy a jak se v zařízení bude dít. Návrh CPU není triviální úkol. Kombinují se při něm znalosti z elektrotechniky a informatiky. Vyžaduje znalosti digitální logiky a architektury počítače.

Cílem této práce je navrhnout jednoduchý jednocyklový procesor, který bude schopný provádět základní instrukce a demonstrovat činnost CPU. Popíše strukturu jednotlivých částí procesoru jako je řadič, registr, nebo paměť a jejich spolupráce při zpracování instrukcí. Důležitou součástí cíle je implementace procesoru na programovatelné hradlové pole (FPGA) s použitím hardwarového popisného jazyka VHDL. V této části ověřím funkčnost procesoru nejen v simulovaném prostředí ale i na fyzické FPGA desce.

V kapitole 1 analyzuji požadavky kladené na CPU a uvádím použité postupy. V kapitole 2 popisuji implementaci. V kapitole 3 se věnuji testování hotového systému, poukazuji, na jaké problémy jsem při testování narazil a navrhuji jejich řešení.

# Návrh

Tato práce implementuje jednoduchý jednocyklový procesor s Harvardskou architekturou (rozdělená paměť programu a dat) na otevřené architektuře RISC-V. Bude však implementovat jen 11 základních instrukcí po vzoru přednášky od pana Ing. Michala Štěpanovského Ph.D., od které se tato práce odráží (1). Procesor bude mít narozdíl od návrhu v přednášce k dispozici i 4 DIP Switche a 8 ledek.

## Instrukční sada tzv. picoRISC-V

* Čtení a zápis (lw, sw)
* Aritmeticko-logické instrukce (add, addi, sub, and, or, slt)
* Podmíněný skok (beq)
* Volání podprogramu (jal, jalr)

## Použité prostředky

FPGA, na kterém celý procesor otestuji, je SPARTAN-3E, protože je poměrně velké a pan Ing. Petr Socha Ph.D. mi ho mohl zapůjčit na testování. Z to se odvíjí, jaký program používám na implementování, a to Xilinx ise, protože toto FPGA podporuje. Na simulaci jsem používal iSIM, který je součástí tohoto programu, má však poměrně zastaralý editor, takže jsem na psaní kódu používal Sigasi Visual HDL extension do VS Code.

# Implementace

V implementaci nejprve podrobně popíšu jednotlivé části procesoru, jejich funkci a způsob, jakým vzájemně spolupracují a jakým způsobem se propojují se zbytkem obvodu. Na konec kapitoly popíšu komunikaci v rámci celého CPU.

## Kombinační obvody

### Řadič

Řadič podle aktuální instrukce vytváří ovládací signál, který řídí celý procesor. Jedná se o vnořené přepínače case. Prvním vnoření přečte 0. - 6. bit instrukce tzv. Opcode, který stačí na rozpoznání většiny instrukcí. Při druhém vnoření kontroluje u zbytku instrukcí 12. - 14. bit tzv. Funct3 a konečně při posledním vnoření na 25. - 31. tzv. Funct7 kde v našem procesoru už rozpozná každou jednotlivou funkci a je schopen vytvořit ovládací signál. Samotný ovládací signál má 13 bytů a je složen z částí které můžeme vidět na obrázku 1

Obrázek 1 Řídicí signály procesoru.

Pro každou instrukci je jejich hodnota určena podle následující tabulky

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | ImmControl | ALUControl | ALUSrc | MemW | MemToReg | RegW | BraJarl | BraJal | BraBeq |
| lw | 110 | 111 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| sw | 101 | 111 | 1 | 1 | X | 0 | 0 | 0 | 0 |
| add | 111 | 111 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| addi | 110 | 111 | 1 | 0 | 0 | 1 | 0 | 0 | 0 |
| sub | 111 | 110 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| and | 111 | 101 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| or | 111 | 100 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| slt | 111 | 011 | 0 | 0 | 0 | 1 | 0 | 0 | 0 |
| beq | 100 | 110 | 0 | 0 | X | 0 | 0 | 0 | 1 |
| jal | 010 | XXX | X | 0 | 0 | 1 | 0 | 1 | 0 |
| jalr | 110 | 111 | 1 | 0 | 0 | 1 | 1 | 0 | 0 |

Podrobný popis jednotlivých částí kontrolního signálu je uveden v příslušných částech práce dle jejich konkrétního využití.

### Extraktor přímého operandu

Obsah obrázku text, snímek obrazovky, Písmo, řada/pruh

Obsah vygenerovaný umělou inteligencí může být nesprávný.Extraktor přímého operandu z aktuální instrukce extrahuje přímí operand podle signálu immControl, který je součástí ovládacího signálu, který mu pošle řadič. Extraktor je v kódu velmi podobný řadiči jedná se také o přepínač case, pomocí kterého ze signálu immControl zjistí, o jaký typ instrukce se jedná (R [111], I [110], S [101], B [100], U [011], J [010]). Typ R nemá přímý operand, takže neposílá nic. U ostatních typů je extrahován přímý operand podle tabulky na obrázku 2.

Obrázek 2 immediate operand dekódování

Poznámka: RISC-V používá Doplňkový kód na ukládaní čísel, proto se sign bit kopíruje na doplnění horních bytů.

Příklad extrakce:

 when "110" => imm(31 downto 11) <= (others => inst(31));

               imm(10 downto 0) <= inst(30 downto 20); -- I

### Aritmeticko-logická jednotka

Díky tomu že implementuji jen jednoduché aritmetické a logické operace tak je již většina implementovaná v numeric\_std, takže se z většiny zase jedná o přepínač case tentokrát závislém na ALUControl části ovládacího signálu (+ [111], - [110], & [101], | [100], < [011]).

 case ALUControl is

            when "111" => ALU\_Result <= std\_logic\_vector(signed(ScrA) + signed(ScrB));

            when "110" => ALU\_Result <= std\_logic\_vector(signed(ScrA) - signed(ScrB));

            when "101" => ALU\_Result <= ScrA and ScrB;

            when "100" => ALU\_Result <= ScrA or ScrB;

            when "011" => if (signed(ScrA) < signed(ScrB)) then

                            ALU\_Result <= (31 downto 1 => '0') & '1';

                          else

                            ALU\_Result <= (others => '0');

                          end if;

           when others => null;

        end case;

ALU kromě toho ještě nastavuje signál Z\_Flag (zero flag), který se používá na kontrolu při podmíněném skoku.

if ALU\_Result = (31 downto 0 => '0') then

                Z\_Flag <= '1';

          else

                Z\_Flag <= '0';

          end if;

## Sekvenční obvody

### Registr

Registr tohoto procesoru má 32 registrů, které slouží k uchování mezivýsledků během vykonávání instrukcí. Šířka registru je také 32. První 0. registr je hard-wired nula pro referenci v programu a zápis do něj je ignorován. Má 2 porty pro čtení. Adresu, z které má číst, načte z instrukce na bitech 15–19 a 20-24. Obsahuje i 1 port pro zápis, který čte adresu z instrukce na bitech 7-11. Zápis je řízený pomocí RegW.

architecture behavioral of registr32x4 is

    type REG\_ARRAY is array (31 downto 0) of std\_logic\_vector (31 downto 0);

    signal REG: REG\_ARRAY := (others => (others => '0'));

begin

    RegProcess : process (clk) is

    begin

        if rising\_edge(clk) then

            if rst = '1' then

                REG <= (others => (others => '0'));

            else

                if WE3 = '1' and A3 /= "00000" then

                    REG(to\_integer(unsigned(A3))) <= WD3;

                end if;

            end if;

        end if;

    end process RegProcess;

    RD1 <= REG(to\_integer(unsigned(A1)));

    RD2 <= REG(to\_integer(unsigned(A2)));

end architecture behavioral;

### Datová paměť

Procesor disponuje 512 bajtovou datovou paměti. V současné implementaci lze do paměti zapisovat pouze 4 bity najednou, ale kód je navržen tak, aby v budoucnu podporoval zápis po jednotlivých bitech. Každý zápis je momentálně kontrolován, zda je zarovnán na čtyřbytovou hranici, Tím se zajišťuje ochrana před nechtěným přepsáním jiných dat.

architecture behavioral of RAM512x1 is

    type RAM\_ARRAY is array (511 downto 0) of std\_logic\_vector (7 downto 0);

    signal RAM: RAM\_ARRAY := (others => (others => '0'));

begin

    RAMProcess : process (clk) is

    begin

        if rising\_edge(clk) then

            if rst = '1' then

                RAM <= (others => (others => '0'));

            else

                if WE1 = '1' then

                    RAM(to\_integer(signed(A))) <= WD1;

                end if;

                if WE4 = '1' and unsigned(A) mod 4 = 0 then

                    RAM(to\_integer(unsigned(A))) <= WD4(31 downto 24);

                    RAM(to\_integer(unsigned(A))+1) <= WD4(23 downto 16);

                    RAM(to\_integer(unsigned(A))+2) <= WD4(15 downto 8);

                    RAM(to\_integer(unsigned(A))+3) <= WD4(7 downto 0);

                end if;

            end if;

            RD1 <= RAM(to\_integer(unsigned(A)));

            if  unsigned(A) mod 4 = 0 then

                RD4 <= RAM(to\_integer(unsigned(A))) & RAM(to\_integer(unsigned(A))+1) & RAM(to\_integer(unsigned(A))+2) & RAM(to\_integer(unsigned(A))+3);

            end if;

        end if;

    end process RAMProcess;

end architecture behavioral;

Část paměťového prostoru není přístupná, protože je využita pro vstupní a výstupní zařízení. Konkrétně jsou na adresu 0 připojeny LED diody a na adresu 4 DIP přepínače.

Zápis do paměti je řízen pomocí jednoduchého adresového dekodéru. Ten přijímá adresu z ALU, která ji vypočítá podle aktuálně vykonávané instrukce, a zároveň signál MemW. Pokud je MemW na hodnotě 1, dekodér podle výsledné adresy určí, zda se jedná o přístup k I/O zařízení (např. LED na adrese 0) nebo k běžné paměti. Na základě toho pak nastaví příslušný řídicí signál write enable. Obdobný princip se používá i při čtení, kdy se podle adresy rozhoduje, zda se data načítají z paměti nebo z I/O portů.

WELed <= '1' when A = "00000000000" and WE = '1' else '0';

WERAM <= '1' when A > "00000000100" and WE = '1' else '0';

Sel <= '1' when A = "00000000100" else '0';

### Instrukční paměť

Také instrukční paměť o velikosti 1 Kb (256 instrukcí). Instrukce jsou hard-coded, stačí jen referovat adresou a posílat ven instrukci. Nevýhodou tohoto přístupu je, že se při každé změně kódu musí znova kód sestavit. Kdybych v práci na procesoru pokračoval v maturitní práci, tak připojím paměť na svojí vlastní sběrnici, kde do ní půjde nahrávat, nebo použiji přímo externí flash paměť s instrukcemi, ale to je již nad rámec mé ročníkové práce. Na překlad assembler kódu jsem používal externí stránku uvedenou ve zdrojích (2).

architecture behavioral of InstrMem is

type RAM\_ARRAY is array (0 to 1023) of std\_logic\_vector (7 downto 0);

    signal RAM: RAM\_ARRAY :=(

        x"00",x"16",x"86",x"93", --addi x13,x13,1

        x"00",x"56",x"06",x"13", --addi x12,x12,5

        x"00",x"40",x"25",x"03", --lw x10,4(x0)

        …

        );

begin

    RD <= RAM(to\_integer(signed(A))) & RAM(to\_integer(signed(A))+1) & RAM(to\_integer(signed(A))+2) & RAM(to\_integer(signed(A))+3);

end architecture behavioral;

## Top level architektura

Na obrázku můžeme vidět, jak je celý systém zapojený. Diagram je z většiny inspirovaný přednáškou od pana Ing. Michala Štěpanovského Ph.D. (1).

Obsah obrázku text, diagram, Plán, schématické

Obsah vygenerovaný umělou inteligencí může být nesprávný.

Obrázek 3 Diagram procesoru

# Testování

## Simulace

Pro otestování funkčnosti jsem napsal jednoduchý program v assembleru, který bliká LED diodou, pokud je DIP přepínač vypnutý. Když ho zapnu, dioda přestane svítit.

addi x13,x13,1

addi x12,x12,5

lw x10,4(x0)

beq x10,x0,-4

addi x11,x11,1

beq x11,x12,8

beq x0,x0,-8

beq x14,x13,20

sw x13,0(x0)

sub x11,x11,x12

addi x14,x14,1

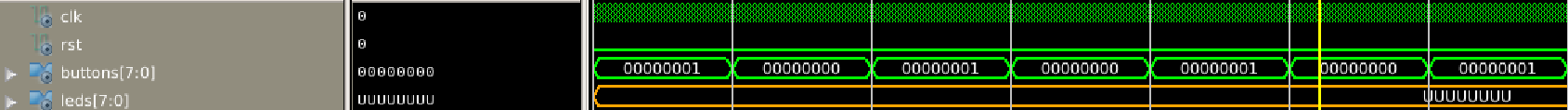
beq x0,x0,-28

sw x0,0(x0)

sub x11,x11,x12

sub x14,x14,x13

beq x0,x0,-56

Kód jsem následně otestoval v simulátoru ISIM a dostal tento výsledek 

Obrázek 4 nefunkční graf

V tento moment jsem začal testovat každou komponentu jednotlivě a začal hledat chyby:

* Při psaní instrukční paměti jsem narazil na problém, že jsem paměť četl odzadu jelikož používám v celém kódu logiku „downto“. Takže jsem místo toho použil logiku „to“ aby se dalo do paměti pohodlněji zapisovat.
* V top hierarchii jsem zjistil že jsem měl vyměněné vstupní signály u led diody

Což ale situaci nevyřešilo a v tento moment jsem měl již každou jednotlivou komponentu v izolaci funkční. Pan Socha mi poradil na začátku simulace poslat signál reset což problém vyřešilo a už jen stačilo opravit assembler kód (upravit délku skoků) a dostal jsem tento výsledek:

Tato simulace ukazuje, že LED dioda bliká, když je vstupní tlačítko vypnuté (00000000), a zhasne, když je tlačítko zapnuté (00000001). Což odpovídá dřívějšímu popisu chování.

Obrázek 5 funkční graf

## Reálné FPGA

Obsah obrázku text, snímek obrazovky, číslo, Písmo

Obsah vygenerovaný umělou inteligencí může být nesprávný.Po syntéze návrhu při implementaci se ukázalo že se mi procesor na FPGA nevešel, jak můžete vidět na obrázku 6 tak zabíral až 300 % velikosti.

Obrázek 6 přeplněné FPGA

Problém byl ve způsobu implementace RAM paměti. Původně byla RAM definována jako čistě logická struktura (pomocí array of std\_logic\_vector), což znamenalo, že se syntetizovala do běžných logických prvků (LUT), místo aby využila blokovou paměť (Block RAM), kterou FPGA nativně nabízí.

Takže jsem místo své napsané paměti nakonec použil paměť přímo od Xilinx ISE, kterou jsem jen zapojil do implementace místo té svojí. Původní návrh jsem v dokumentu ponechal, protože jsem mu věnoval značné množství času a chtěl jsem jej prezentovat jako součást vývoje. Po opětovné syntéze a implementace se již návrh, jak můžeme vidět na obrázku 7 na FPGA vešel i když teda těsně. Určitě by šel návrh dále optimalizovat, ale to je již nad rámec mé ročníkové práce.

Obsah obrázku text, snímek obrazovky, číslo, Písmo

Obsah vygenerovaný umělou inteligencí může být nesprávný.

Obrázek 7 namapované FPGA

Po nahrání programu na desku bohužel nefungoval správně. Pokusil jsem se problém vyřešit pomocí synchronizéru, což však nepomohlo. Bohužel jsem již nestihl vyzkoušet časově náročnější varianty opravy, a proto jsem CPU na FPGA zatím neotestoval. V nejbližším možném termínu se pokusím problém opravit.

Závěr

V rámci této ročníkové práce jsem si stanovil cíl navrhnout a implementovat jednoduchý jednocyklový procesor, který bude schopen provádět základní instrukce a umožní názorně demonstrovat činnost CPU. Během práce jsem se musel podrobně seznámit s architekturou RISC-V a jejím způsobem zpracování instrukcí, což zahrnovalo nejen pochopení formátu instrukcí. Důkladné prostudování těchto základů bylo nezbytným předpokladem pro návrh jednotlivých bloků procesoru.

Jedním z hlavních kroků při návrhu byla tvorba jednotlivých komponent, jako jsou řadič, ALU, extraktor operandu nebo registr. V této části jsem se zaměřil nejen na teoretický návrh, ale i na praktickou realizaci v jazyce VHDL, přičemž jsem čelil několika technickým problémům – například s prací s pamětí nebo s logikou směru bitových polí („downto“ vs. „to“), které se mi však podařilo vyřešit.

Následně jsem se pokusil implementovat procesor na FPGA desce SPARTAN-3E, což se mi však nepodařilo. V následujících dnech se budu snažit chybu odstranit.

Cíle práce se podařilo naplnit – vytvořený procesor dokáže vykonávat zadanou množinu instrukcí a jeho fungování bylo úspěšně otestováno jak simulací, tak na fyzickém zařízení. Proces navrhování CPU mi umožnil lépe porozumět architektuře počítačů, vnitřnímu chodu procesorů a principům práce s jazykem VHDL. V budoucnu by bylo možné projekt rozšířit například o implementaci vícecyklové architektury, větší paměť, více instrukcí, nebo podpora přerušení, ale to až v případné Maturitní práci.

Seznam zkratek a odborných výrazů

CPU

Central processing unit – hlavní jednotka počítače, která zpracovává instrukce a řídí chod všech operací.

VHDL

VHSIC Hardware Description Language – jazyk pro popis a simulaci digitálních obvodů, často používaný při návrhu čipů a FPGA.

FPGA

Field-Programmable Gate Array – programovatelný integrovaný obvod, který lze po výrobě nakonfigurovat pro různé digitální funkce.

LED

Light Emitting Diode – je polovodičová dioda, která při průchodu elektrického proudu vyzařuje světlo.

DIP Switch

Dual In-line Package Switch – je sada malých přepínačů v jednom pouzdře, která umožňuje manuální nastavení konfigurace elektronických zařízení.

Seznam obrázků

[Obrázek 1 Řídicí signály procesoru. 3](#_Toc198495566)

[Obrázek 2 immediate operand dekódování 4](#_Toc198495567)

[Obrázek 3 Diagram procesoru 8](#_Toc198495568)

[Obrázek 4 nefunkční graf 9](#_Toc198495569)

[Obrázek 5 funkční graf 10](#_Toc198495570)

[Obrázek 6 přeplněné FPGA 10](#_Toc198495571)

[Obrázek 7 namapované FPGA 11](#_Toc198495572)

Použité zdroje

1. **Ph.D., Ing. Michal Štěpanovký.** *Architektura počítačových systémů (BI-APS), Přednáška č.3 Návrh jednocyklové RISC mikroarchitektury.* Faculty of Information Technology, Czech Technical University in Prague. Praha : autor neznámý, 2022. str. 57.

2. **LupLab.** RISC-V Instruction Encoder/Decoder. [Online] University of California, 2023. https://luplab.gitlab.io/rvcodecjs/.

1. Seznam přiložených souborů

Na přiloženém datovém nosiči se nacházejí následující soubory a složky:

* **Implementace\_CPU.docx** – editovatelná verze dokumentace ročníkové práce
* **CPU.zip**– zdrojové kódy