Procesor s jednoduchou instrukční sadou

Datum zadání: 3.10.2023
Datum a forma odevzdání: do 27.11.2023 23:59, POUZE přes IS VUT, 4 soubory
Počet bodů: max. 23 bodů
Dotazy ohledně projektu: v případě nejasností využijte osobní konzultace (Zdeněk Vašíček, L326)

1 Úvod

Cílem tohoto projektu je implementovat pomocí VHDL procesor, který bude schopen vykonávat program napsaný v rozšířené verzi jazyka BrainF*ck [1, 2]. Jazyk v základní verzi používá pouze osm jednoduchých příkazů (instrukcí) a jedná se o výpočetně úplnou sadu, pomocí které je možné implementovat libovolný algoritmus. Pro základní ověření korektní funkce jsou k dispozici automatizované testy.

Činnost procesoru

Jazyk používá příkazy kódované pomocí tisknutelných 8-bitových znaků. Implementovaný procesor bude zpracovávat přímo tyto znaky (tzn. operační kód procesoru bude sestávat vždy z osmi bitů). Pro jednoduchost budeme uvažovat pouze 10 příkazů uvedených v tabulce níže. Program sestává ze sekvence těchto příkazů. Neznámé příkazy jsou ignorovány, což umožňuje vkládat textové komentáře přímo do programu. Vykonávání programu začíná první instrukcí a končí jakmile je detekován konec sekvence (znak s ASCII hodnotou 0). Program i data jsou uložena ve stejné paměti mající kapacitu 8192 8-bitových položek. Program je uložen od adresy 0 a je vykonáván nelineárně (tzn. může obsahovat skoky). Obsah paměti nechť je pro jednoduchost inicializován na hodnotu nula. Pro přístup do paměti se používá ukazatel (ptr), který je možné přesouvat o pozici doleva či doprava. Paměť je chápána jako kruhový buffer uchovávající 8-bitová čísla bez znaménka. Posun doleva z adresy 0x0000 tedy znamená přesun ukazatele na konec paměti odpovídající adrese 0x1FFF.

Implementovaný procesor nechť podporuje příkazy definované v následující tabulce. Operační kódy, které se v tabulce nenacházejí jsou procesorem ignorovány.

příkaz	operační kód	význam	ekvivalent v C
`>`	0x3E	inkrementace hodnoty ukazatele	ptr += 1;
`<`	0x3C	dekrementace hodnoty ukazatele	ptr -= 1;
`+`	0x2B	inkrementace hodnoty aktuální buňky	*ptr += 1;
`-`	0x2D	dekrementace hodnoty aktuální buňky	*ptr -= 1;
`[`	0x5B	je-li hodnota aktuální buňky nulová, skoč za odpovídající příkaz] jinak pokračuj následujícím znakem	while (*ptr) {
`]`	0x5D	je-li hodnota aktuální buňky nenulová, skoč za odpovídající příkaz [jinak pokračuj následujícím znakem	}
`~`	0x7E	ukončí právě prováděnou smyčku while	break;
`.`	0x2E	vytiskni hodnotu aktuální buňky	putchar(*ptr);
`,`	0x2C	načti hodnotu a ulož ji do aktuální buňky	*ptr = getchar();
@	0x40	oddělovač kódu a dat způsobí zastavení vykonávání programu	 return;

V případě příkazů manipulujících s ukazatelem do programového kódu (instrukčním čítačem PC), kterými jsou `[` a `]`, je zapotřebí detekovat odpovídající pravou, respektive levou, závorku. Možností je několik, nejjednodušší je postupně inkrementovat (respektive dekrementovat) ukazatel a počítat počet závorek (viz dále). Ukazatel ptr je nutné před zahájením vykonávání programu přesunout na adresu bezprostředně následující po znaku @. Každý program musí obsahovat minimálně jeden znak @.

Kód 1: Příklad kódu, který vytiskne na výstup řetězec Hello World!

```
[.>]@Hello World!
```

Mikrokontroler

Aby bylo možné vykonávat smysluplný program, je procesor nutné doplnit o paměť programu, paměť dat a vstupně-výstupní rozhraní umožňující načítat a vypisovat data. V našem případě budeme uvažovat společnou pamět programu a dat o celkové kapacitě 8kB.

Vstup by bylo možné v praxi řešit pomocí maticové klávesnice obsahující typicky znaky 0-9 a dva speciální symboly. Jakmile procesor narazí na instrukci načtení hodnoty (operační kód 0x2C), vykonávání se pozastaví do té doby, než je stisknuto některé z tlačítek klávesnice. Klávesnici by obsluhoval řadič, který by na vstup procesoru dával 8-bitovou hodnotu, která odpovídá ASCII kódu stisknuté klávesy.

Výstup dat lze řešit pomocí LCD displeje, kam se postupně vypisují znaky. Posun kurzoru na displeji by byl řešen automaticky.

Z důvodu usnadnění vývoje kódu máte připraveno prostředí emulující výše uvedené periferie a sadu základních testů.

2 Úkoly

1. Seznamte se s jednotlivými instrukcemi procesoru uvedenými v části 1. Obsah souboru login.b obsahujícího program v jazyce BrainF*ck, který tiskne řetězec xlogin01, zkopírujte do debuggeru na adrese [3]. Tlačítkem "Start debugger" spusťte krokování a sledujte, co způsobuje která instrukce, jak se pohybuje programový čítač a ukazatel do paměti dat. Vytvořte program, který vytiskne na displej *Váš login* (na velikosti písmen nezáleží). Snažte se v programu využít všechny dostupné příkazy s výjimkou příkazu načtení. Pokuste se vytvořit co nejkratší program tisknoucí Váš login. Za znak `@` si můžete uložit pomocná data, avšak nanejvýše 4 nenulové položky.

2. Seznamte se se způsobem zprovoznění testovacího prostředí na serveru fitkit-build a spusťte automatické testy pomocí příkazu *make*. Všechny testy by měly skončit chybou, neboť architektura popisující činnost procesoru v souboru *cpu.vhd* je prázdná.

3. Do souboru *cpu.vhd* doplňte vlastní VHDL kód, který bude **syntetizovatelným** způsobem popisovat implementaci procesoru vykonávajícího program zapsaný v jazyce BrainF*ck.

   Rozhraní procesoru je pevně dané a skládá se z čtyř skupin signálů: synchronizace, rozhraní pro paměť programu a dat, vstupní rozhraní a výstupní rozhraní.

   Synchronizační rozhraní tvoří tři signály. **CLK** - hodinový synchronizační signál. Procesor pracuje vždy při vzestupné hraně hodinového signálu. **RESET** - asynchronní nulovací signál. Je-li RESET=1, procesor zinicializuje svůj stav (PTR=0, PC=0). **EN** - povolení činnosti procesoru. Pokud je signál RESET uvolněn (RESET=0) a EN=1, procesor postupně s každou vzestupnou hranou hodinového signálu začne vykonávat program. Vykonávání začíná nalezením znaku `@` a nastavením PTR za tento znak. Poté se postupně začnou vykonávat příkazy od adresy 0 dále.

   Rozhraní *synchronní* paměti, která slouží k uchování programu a dat je tvořeno třemi datovými a dvěma řídicími signály. Signál **DATA_ADDR** o šířce 13 bitů slouží k adresaci konkrétní buňky paměti. Signál **DATA_RDATA** (načtená data) obsahuje 8-bitovou hodnotu buňky na adrese DATA_ADDR. Signál **DATA_WDATA** (zapisovaná data) nechť obsahuje 8-bitovou hodnotu, kterou se má přepsat buňka na adrese DATA_ADDR. Rozhraní pracuje následovně. Pokud **DATA_EN**=1 (povolení činnosti paměti) a **DATA_RDWR**=0 (volba režimu čtení), signál DATA_RDATA je aktualizován hodnotou buňky na adrese DATA_ADDR. Je-li DATA_EN=1 a DATA_RDWR=1 (volba režimu zápis), hodnota buňky na adrese DATA_ADDR je přepsána hodnotou signálu DATA_WDATA a signál DATA_RDATA je

aktualizován hodnotou DATA_WDATA. K aktualizaci hodnoty signálu DATA_RDATA dochází pouze pokud **DATA_EN** = 1 (tj. aktivním signálu povolení činnosti). Signály DATA_ADDR, DATA_EN a DATA_WDATA jsou čteny a signál DATA_RDATA aktualizován při vzestupné hraně hodinového signálu CLK.

Vstupní rozhraní pracuje následovně. Při požadavku na data procesor nastaví signál **IN_REQ** na 1 a čeká tak dlouho, dokud signál **IN_VLD** (input valid) není roven 1. Jakmile se tak stane, může procesor přečíst signál **IN_DATA**, který obsahuje ASCII hodnotu načteného znaku.

Výstupní rozhraní pracuje následovně. Při požadavku na zápis dat procesor nejdříve musí otestovat stav signálu **OUT_BUSY**. Tento signál indikuje, že je periferie zaneprázdněna vyřizováním předchozího požadavku. Pouze pokud je OUT_BUSY=0, procesor inicializuje signál **OUT_DATA** zapisovanou ASCII hodnotou a současně na jeden hodinový takt nastaví signál **OUT_WE** (povolení zápisu) na 1. V opačném případě musí čekat tak dlouho, dokud nebude OUT_BUSY=0.

Mimo výše uvedené signály rozhraní obsahuje dva stavové signály – **READY**, který bude aktivován v okamžik, kdy dojde k inicializaci ukazatele ptr na počáteční hodnotu (tj. poté, co je detekována pozice znaku @) a **DONE**, který bude aktivován ve chvíli, kdy dojde k vykonání poslední instrukce programu (tj. narazí se na instrukci @).

Činnost procesoru ověřte nejen pomocí testů ale taktéž pomocí simulace. K důkladnějšímu ověření je doporučeno použít složitější testovací programy. Předpokládejte, že na vstupu bude vždy validní kód.

3 Odevzdává se

Do IS VUT se odevzdávají následující **4 soubory** (nikoliv ZIP či jiný archiv).

1. Soubor **login.b** obsahující program v jazyce BrainF*ck vypisující Váš login (jedná se o výsledek bodu 1 zadání).
2. Soubor **cpu.vhd** obsahující implementaci procesoru (jedná se o výsledek bodu 3 zadání).
3. Soubor **log.txt** obsahující report z dodaných automatických testů získaný voláním **make > log.txt**. Výstup může obsahovat dodatečné testy (plus pro Vás), avšak testy dodané se zadáním musí zůstat bezezměny.
4. Soubor **login.png** obsahující printscreen ze simulace vykonávání programu login.b (tj. výstup volání příkazu **TESTCASE=test_login make questa**) zachycující stav signálů v okamžik zápisu posledního a předposledního znaku na výstup. Není nutné uvádět všechny signály detailně, avšak na obrázku by měl být vidět stav automatu, hodinový signál a signál OUT_WE, OUT_DATA zachycující znak zapisovaný na výstup. Z obrázku by mělo být dekódovatelné, že procesor narazil na instrukci HALT (tj. zastav vykonávání programu). Signál CLK bude mít nastavenu barvu Orange a signál OUT_DATA bude přepnut do režimu výpisu ASCII znaků (položka Radix ve vlastnostech signálu). V obrázku vyznačte (např. červeným oválem) hodinové hrany, kdy je vystaven požadavek na zapsání předposledního a posledního znaku. Dále vyznačte hodinovou hranu, při které dojde ke zpracování instrukce HALT.

4 Hodnocení

Za kompletní implementaci procesoru (tj. splnění bodu 3) lze získat až 17 bodů. Implementace procesoru podporujícího pouze jednoduché while/do cykly (tj. nepodporující vnořené cykly) lze získat až 12 bodů. Odevzdání po termínu je penalizováno bodovou srážkou ve výši 10 bodů a to za každý započatý den.

5 Upozornění

Pracujte samostatně, nikomu nedávejte svoji práci k opsání.

Plagiátorství se hodnotí 0 body, neudělením zápočtu a případným dalším adekvátním postihem dle platného disciplinárního řádu VUT v Brně. Přejmenování názvu identifikátorů, změna pořadí jednotlivých bloků či modifikace komentářů není považováno za autorské dílo a na řešení bude nahlíženo jako na plagiát.

6 Revize

1.10.2023 - první verze dokumentu

Testovací prostředí

Součástí zadání je testovací prostředí umožňující na základní úrovni ověřit korektní činnost kódu pomocí behaviorální simulace. K ověření se používá projekt GHDL a simulátor Questa. Testovací prostředí je doporučeno spouštět na stroji fitkit-build.fit.vutbr.cz, kde již je k dispozici veškerý software. Jedinou podmínkou je zprovoznit připojení na tento server včetně tunelování protokolu X11 pro vzdálený přenos obrazu. V Linuxu funguje nativně pomocí příkazu `ssh -X xlogin01@fitkit-build.fit.vutbr.cz`. Pro přístup z prostředí Windows je nutné nainstalovat aplikaci Xming X Server a Putty, kde je nutné povolit tunelování protokolu X11, viz např. [6].

Po připojení na fitkit-build.fit.vutbr.cz (mimo VUT síť je nutné se připojovat přes merlin.fit.vutbr.cz nebo využít VPN FIT) vytvořte nový adresář a do něj stáhněte zadání.

Kód 2: Stažení zadání a inicializace simulačního prostředí

```
mkdir inp23-projekt1
cd inp23-projekt1
python3 -m venv env
curl https://www.fit.vutbr.cz/~vasicek/inp23/zadani.zip | jar xv
. env/bin/activate
pip install -r zadani/requirements.txt
```

Od této chvíle máte k dispozici kompletní prostředí. V adresáři `inp23` se nachází složka `src` obsahující zdrojové kódy `cpu.vhd` a `login.b`. Prostředí je možné kdykoliv v budoucnu aktivovat voláním skriptu `env/bin/activate`, který nastaví korektní cesty k Python interpretu.

Automatické testy lze spouštět voláním příkazu `make` ve složce test.

Kód 3: Aktivace prostředí a spuštění automatických testů

```
cd inp23-projekt1
. env/bin/activate
cd zadani/test
make
```

Příklad výstupu v případě prázdného kódu cpu.vhd:

```
**********************************************************************************************
** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
**********************************************************************************************
** cpu.test_reset                FAIL          51.00           0.00      18464.35  **
** cpu.test_init                 FAIL        1055.00           0.04      27270.99  **
** cpu.test_increment            FAIL        1055.00           0.04      28223.48  **
** cpu.test_decrement            FAIL        1055.00           0.04      28532.86  **
** cpu.test_move                 FAIL        1055.00           0.04      28477.78  **
** cpu.test_print                FAIL        3555.00           0.12      30007.85  **
** cpu.test_input                FAIL        5155.00           0.17      29963.38  **
** cpu.test_while_loop           FAIL        5055.00           0.17      30216.56  **
** cpu.test_break                FAIL        5055.00           0.17      30218.93  **
** cpu.test_login_vasicek        FAIL      250055.00           6.92      36121.12  **
**********************************************************************************************
** TESTS=10 PASS=0 FAIL=10 SKIP=0             273146.00           8.02      34039.92  **
**********************************************************************************************
```

Budete-li chtít spustit pouze jeden konkrétní test (vhodné zejména na začátku, kdy většina testů selhává), lze k tomu využít systémovou proměnnou TESTCASE, kterou nastavíte na název testu z výpisu, který začíná prefixem `test_`.

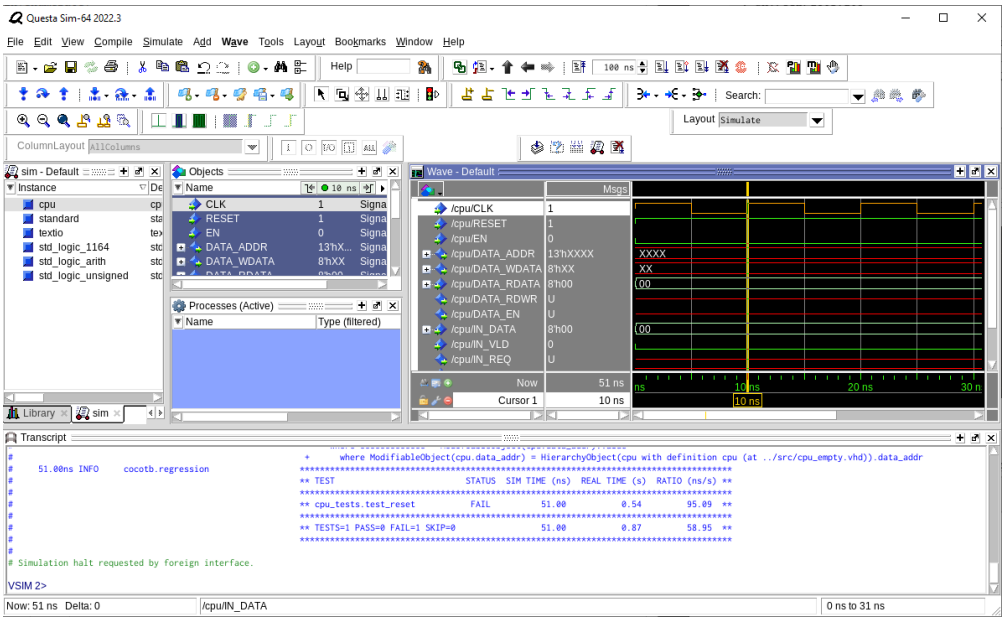
Kód 4: Spuštění konkrétního testu

```
cd zadani/test
TESTCASE=test_reset make
```

V případě, že Vám některý test selže a potřebujete zjistit příčinu, je možné spustit simulátor Mentor Questa, který umožní detailně sledovat stav signálů.

Kód 5: Spuštění simulátoru pro jeden test

```
cd zadani/test
TESTCASE=test_reset make questa
```



Obrázek 1: Simulační okno pro test test_reset

Do okna s průběhy signálu jsou automaticky přidávány všechny dostupné signály, viz soubor **runsim.do**. Vysvětlení jednotlivých částí skriptu pro simulátor Questa bylo podáno v rámci cvičení INP.

Upozornění

Testy nejsou kompletní ani komplexní. V případě, že budete kód tvořit čistě skrze tzv. test driven development, není zaručeno, že bude hodnocen plným počtem bodů, ačkoliv Vám vše bude svítit zeleně. Testy *nejsou izolované* a předchozí test může v případě nevhodné implementace ovlivnit následující. Je proto dobré si vyzkoušet každý test samostatně (izolovaně) přes `TESTCASE=test_xxxx make`.

Vlastní testy je možné dopsat do souboru `cpu.py`. Do existujících testů není dovoleno zasahovat. Stejně tak není dovoleno zasahovat do pomocných skriptů.

Na řešení pracujte průběžně, na nemožnost vypracovat projekt z důvodu přetížení serveru v důsledku současného přístupu mnoha studentů nebude brán ohled. Alternativně je možné nainstalovat GHDL a Questa na svůj PC. Instalátor Questa se nachází v adresáři `/mnt/data/install/Questa_Core_2022.3`.

Odkazy

[1] `https://esolangs.org/wiki/Brainfuck` - popis instrukční sady

[2] `https://esolangs.org/wiki/Extended_Brainfuck`

[3] `http://www.fit.vutbr.cz/~vasicek/inp23` - online debugger Brainlove / BrainFuck

[4] `http://www.hevanet.com/cristofd/brainfuck/` - několik programů napsaných v jazyce BrainFuck

[5] `https://esolangs.org/wiki/Brainfuck_algorithms` - různé algoritmy

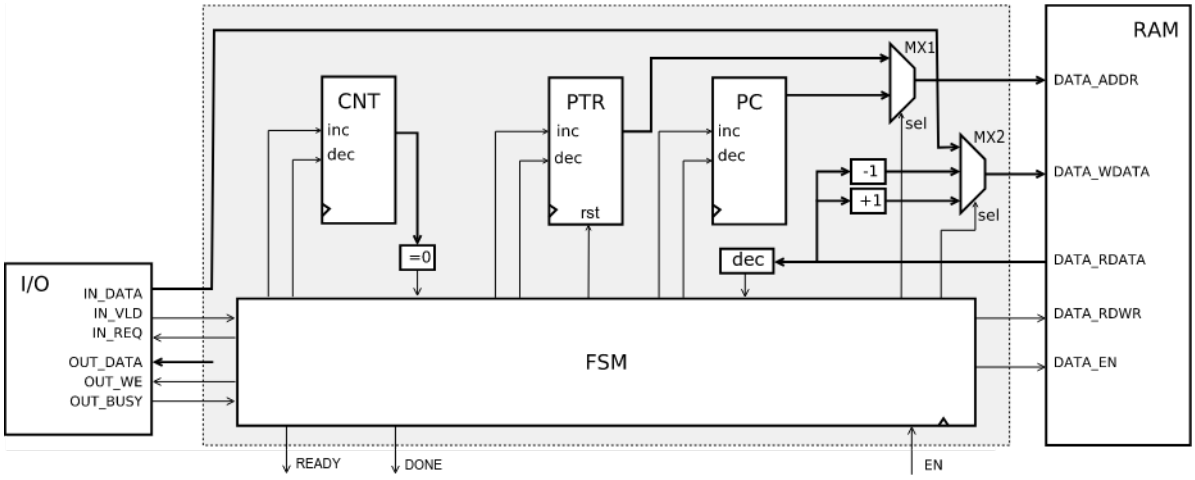
[6] `https://helpdesk.engr.arizona.edu/support/solutions/articles/33000203663-how-do-i-tunnel-x11-windows-using-ssh-`

Návod

Následující řádky jsou určeny těm, kteří doposud netuší jak procesor naimplementovat. Obecně platí, že procesor se skládá z datové cesty obsahující registry, ALU, apod. a řídicí cesty obsahující automat. Stejně tak je tomu i v tomto případě. Blokové schema možné implementace je uvedeno na obrázku 2.

Abychom mohli vykonávat program, obsahuje datová cesta tři registry (čítače) s možností inkrementace a dekrementace. Registr PC slouží jako programový čítač (tj. ukazatel do paměti programu), registr PTR jako ukazatel do paměti dat a registr CNT slouží ke korektnímu určení odpovídajícího začátku/konce příkazu while (počítání otevíracích / uzavíracích závorek, viz. popis instrukční sady). Mimo to datová cesta obsahuje multiplexor MX1, pomocí kterého lze během čtení z paměti definovat, zda-li se jedná o adresu programu nebo adresu dat a multiplexor MX2, který určuje hodnotu zapisovanou do paměti. Zapsat je možné buď hodnotu načtenou ze vstupu, hodnotu v aktuální buňce sníženou o jedničku nebo hodnotu aktuální buňky zvýšenou o jedničku. V případě, že se rozhodnete NEimplementovat podporu vnořených while cyklů, registr CNT nepotřebujete.

Všechny řídicí signály jsou ovládány automatem, tak jak je uvedeno ve schematu. Při tvorbě VHDL kódu se inspirujte procesorem probíraným na cvičeních. V prvním kroku implementujte registry (konstrukce process) a multiplexory (dataflow popis) a poté postupujte od jednodušších instrukcí ke složitějším. Implementaci smyček si ponechte až úplně na závěr. Korektní činnost ověřte pomocí simulace a vlastních či přiložených programů.



Obrázek 2: Blokové schema mikrokontroleru

Nevíte-li jak implementovat automat pomocí VHDL, podívejte se do materiálů ke cvičením INP. Jako návod pro implementaci automatu by měl posloužit pseudokód popisující chování jednotlivých instrukcí uvedený v tabulce 1. Máte-li s implementací problémy, vytvořte jednodušší verzi automatu, který nepodporuje vnořené smyčky. Pseudokód je uveden v tabulce 2.

příkaz / stav	**pseudokód**
výchozí stav	PC ← 0, PTR ← 0, CNT ← 0, READY ← 0, DONE ← 0
inicializace	PTR ← x + 1, READY ← 1, přičemž platí mem[x]='@' nutné vymyslet, inspirace viz [
`>`	PTR ← PTR + 1 % 0x2000, PC ← PC + 1
`<`	PTR ← PTR – 1 % 0x2000, PC ← PC + 1
`+`	DATA_RDATA ← mem[PTR] mem[PTR] ← DATA_RDATA + 1, PC ← PC + 1
`-`	DATA_RDATA ← mem[PTR] mem[PTR] ← DATA_RDATA - 1, PC ← PC + 1
`.`	while (OUT_BUSY) {} OUT_DATA ← mem[PTR], PC ← PC + 1
`,`	IN_REQ ← 1 while (!IN_VLD) {} mem[PTR] ← IN_DATA, PC ← PC + 1

`[`	PC ← PC + 1
	if (mem[PTR] == 0)
	  CNT ← 1
	  while (CNT != 0)
	  c ← mem[PC]
	  if (c == '[') CNT ← CNT + 1 elsif (c == ']') CNT ← CNT - 1
	  PC ← PC + 1

`]`	if (mem[PTR] == 0)
	  PC ← PC + 1
	else
	  CNT ← 1, PC ← PC - 1
	  while (CNT != 0)
	  c ← mem[PC]
	  if (c == ']') CNT ← CNT + 1 elsif (c == '[') CNT ← CNT - 1
	  if (CNT == 0) PC ← PC + 1 else PC ← PC - 1

`~`	nutné vymyslet, inspirace viz [
`@`	PC ← PC, DONE ← 1
ostatní	PC ← PC + 1

Tabulka 1: Popis chování (pseudokód) jednotlivých intrukcí procesoru.

příkaz / stav	**pseudokód**

`[`	PC ← PC + 1
	if (mem[PTR] == 0)
	  do
	  c ← mem[PC]
	  PC ← PC + 1
	  until (c == ']')

`]`	if (mem[PTR] != 0)
	  do
	  PC ← PC - 1
	  c ← mem[PC]
	  until (c == '[')
	PC ← PC + 1

Tabulka 2: Zjednodušená implementace instrukcí procesoru nepodporující vnořené smyčky