Středoškolská technika 2011

Setkání a prezentace prací středoškolských studentů na ČVUT

Experimentální procesorová architektura AttoWPU a 2D WPU

Tomáš Mariančík

Střední průmyslová škola elektrotechniky, informatiky a řemesel
Křižíkova 1258, Frenštát pod Radhoštěm

ANOTACE

attoWPU je první z připravované řady experimentálních procesorových architektur (tzv. WPU), které se pokoušejí o netradiční, nové, originální a zábavné náhledy na to, jak je strojový kód zpracováván a také tvořen programátorem pomocí programovacích jazyků. attoWPU zahrnuje specifikaci funkce tohoto procesoru, speciálního programovacího jazyka attoASM, dále překladač pro jazyk attoASM a simulátor, na kterém je možné programy napsané pro attoWPU otestovat.

AttoWPU a pro něj určený jazyk attoASM dává programátorům možnost vyzkoušet si méně tradiční způsob programování, lépe pochopit způsoby, důvody a výhody optimalizace programu, umožňuje také pomocí attoASM kódu nadefinovat funkci procesoru pro běh programu v libovolném formátu v paměti programu a simulovat tak teoreticky libovolný procesor.

Klíčová slova: programovací jazyk, architektura procesoru, experiment, assembler, netradiční programování, strojový kód

OBSAH

PŘEHLED

AttoWPU je experimentální procesorová architektura ze série WPU (Weird Processing Unit - Podivná procesorová jednotka) zahrnující příslušené programovací jazyky. Série WPU se snaží o netradiční přístup k programování v assembleru i programování a způsobu zpracování programu obecně pro rozličné účely: výuka, zvědavost, zábava a dokonce v jisté míře i umělecký záměr.

Tato experimentální big-endian procesorová jednotka umožňuje programátorovi definovat funkci procesoru pomocí speciálního jazyku - attoassembleru. Samotný procesor nemá přímo žádnou funkci či možnost řízení jeho jednotek použitím konvenčních instrukcí, neboť v něm nejsou přímo nadefinovány opkódy pro různé operace. Procesor však rozumí třem opkódům speciálních attoinstrukcí, pomocí kterých lze vytvořit attokód. Tyto instrukce umožňují měnit stavy vždy jednoho z 64 bitů na sběrnici procesoru, tedy nastavit bit do logické nuly, jedničky, či jej invertovat.

Tato 64bitová sběrnice je přímo řízena jádrem procesoru a je rozdělena na čtyři logické sběrnice: adresní sběrnici, řídicí sběrnici, datovou sběrnici a Quick aJump sběrnici. Tyto sběrnice umožňují výměnu dat mezi různými jednotkami procesoru a řídit jejich funkci patřičnou změnou bitů pomocí tří attoinstrukcí.

Základem procesoru je attojádro (attocore), během každého cyklu je načtena attoinstrukce z paměti attokódu, dekódována na číslo bitu a operaci, načež je příslušný bit patřičně změněn. Následně je atto program counter (atto programový čítač, zkráceně aPC) inkrementován, čímž dojde k přesunu na další attoinstrukci. Tento jednoduchý proces se opakuje v neustálém cyklu.

Programátor může definovat funkci procesoru vytvořením attokódu a uložením do paměti attokódu: specifickou sekvencí změny bitů na sběrnici může ovládat jednotlivé jednotky procesoru dle svých potřeb, obvykle takto vytvoří attokód, který bude zpracovávat běžný program vytvořený v konvenčním jazyku symbolických adres, uloženým v paměti programu a dat. Attokód vždy dekóduje kód instrukce (navržený programátorem) z paměti programu a dat a poté provede patřičné operace s jednotkami procesoru.

Tato speficikace se zabývá pouze definicí architektury z pohledu programátora, neuvádí příklad možné realizace pomocí logických hradel a dalších obvodů, dále definuje dva programovací jazyky určené pro attoWPU.

ATTOINSTRUKCE

Attoinstrukce slouží k ovládání stavů individuálních bitů na sběrnici a umožňují tedy nastavit libovolné hodnoty na těchto sběrnicích. K dispozici jsou tři attoinstrukce, popřípadě čtvrtá volitelná, která ale není pro samotné programování důležitá:

- Nastavit - změní bit do stavu logické 1 nezávisle na aktuálním stavu
- Resetovat - změní bit do stavu logické 0 nezávisle na aktuálním stavu
- Invertovat - invertuje aktuální stav příslušeného bitu
- Zastavit - zastaví procesor, volitelná

Attoinstrukce musí být zkombinována s číslem (adresou) bitu, který mění: každý z bitů má unikátní adresu. Pro adresaci 64 bitů je potřeba 6 bitů, dva bity jsou pak potřeba pro zvolení patřičné operace. Každá attoinstrukce má tedy velikost jednoho bajtu - osmi bitů, které jsou uspořádány v následujícím formátu:

IH	**IL**	**BN5**	**BN4**	**BN3**	**BN2**	**BN1**	**BN0**
0	**1**	**2**	**3**	**4**	**5**	**6**	**7**

Kód operace je zakódován pomocí bitů IH (Instruction High bit) a IL (Instruction Low bit) dle následující tabulky:

IH	**IL**	**Instrukce**
0	0	Resetovat
0	1	Nastavit
1	0	Invertovat
1	1	Zastavit/nedefinováno

Bity BN0 až BN5 (Bit Number - Číslo Bitu) pak určují číslo bitu: od 0 až po 63 v dekadické soustavě.
Speciální attoinstrukce Zastavit zastaví běh procesoru až do jeho resetování uživatelem, význam má zejména v simulátoru procesoru, neboť umožňuje kupříkladu indikovat dokončení programu, což lze využít při měření optimalizace, dle potřeby je však možné tuto attoinstrukci ignorovat u implementací kde nemá velký význam.

SBĚRNICE

Sběrnice jsou nejdůležitější součástí tohoto procesoru, neboť veškeré ostatní součásti jsou k této sběrnici paralelně připojeny a mohou být pomocí nich řízeny a vyměňovat si navzájem data. Procesor v podstatě obsahuje jednu 64bitovou sběrnici, která je ale logicky rozdělena na čtyři sběrnice s různým účelem.

ADRESOVÁ SBĚRNICE 8BITOVÁ (BITY 0 AŽ 7)

Neboť procesor obsahuje různé jednotky, které může programátor využít k realizaci svých programů, je potřeba vždy vybrat pouze jedinou jednotku předtím, než bude řízena řídicími kódy. Všechny ovladatelné jednotky jsou připojeny k adresové sběrnici, proto výběr probíhá právě pomocí této sběrnice.

Je-li adresa na sběrnici shodná s adresou jednotky, začne daná jednotka přijímat řídící kódy od řídící sběrnice (viz níže), ostatní jednotky budou tyto řídící kódy ignorovat. Neboť je sběrnice 8bitová, je teoretické maximum 256 jednotek, což poskytuje dostatečnou rezervu pro budoucí verze, neboť současná nevyužívá ani čtvrtinu z dostupných adres.

Rozložení bitů na sběrnici:

A7	A6	A5	A4	A3	A2	A1	A0
0	1	2	3	4	5	6	7

A0 - A7 jsou adresní bity, reprezentují 8 bitovou adresu. Hodnota po resetu je 0.

ŘÍDÍCÍ SBĚRNICE 8 BITOVÁ (BITY 8 AŽ 15)

Jakmile je jednotka naadresovaná, může přijímat příkazy od řídící sběrnice, které ji řeknou, co má provést. Kupříkladu příkaz může jednotce přikázat, aby načetla hodnotu z datové sběrnice a uložila ve vnitřním registru, či naopak tuto hodnotu vypustit na datovou sběrnici, provést s hodnotou nějakou operaci a podobně.

Ačkoli je sběrnice 8bitová, maximální počet přímých příkazů je 128. Toto je z toho důvodu, že nejméně významný bit sběrnice (bit 15) je použit k indikaci, kdy má dojít ke spuštění příkazu, tento bit se nazývá spouštěcí bit. Programátor nejprve připraví 7bitový kód na řídící sběrnici a poté změní hodnotu 15 bitu na logickou 1 a zpět na logickou 0. Jednotka ignoruje příkazy, dokud neregistruje změnu z logické 0 na logickou 1 u spouštěcího bitu.

Teprve po zaregistrování změny je příkaz vykonán. Toto je potřeba, neboť jádro procesoru mění vždy pouze jeden bit, takže ke změně všech sedmi bitů je potřeba sedm cyklů. Kdyby jednotka nečekala na změnu spouštěcího bitu, tak by během každého cyklu, kdy je nastavován řídící kód, aktuální nevyžádaný řídící kód spustila.

Jak již bylo řečeno, každá jednotka zcela ignoruje řídící sběrnici, když se její adresa neshoduje s adresou datové sběrnice. Tohoto je dosaženo binárním násobením (operace AND) všech bitů řídicí sběrnice vstupujících do dané jednotky s logickou 1 v případě, že se adresa shoduje, nebo s logickou 0 v případě, že se adresa neshoduje. Jestliže programátor ponechá spouštěcí bit nastavený na logickou 1 a změní hodnotu na adresové sběrnici, nově naadresovaná jednotka spustí řídící kód okamžitě, neboť se vstupující hodnoty řídící sběrnice vynásobí s logickou 1 namísto logické 0, čímž jednotka zaregistruje změnu u spouštěcího bitu z logické 0 na logickou 1, i když na řídící sběrnici bit změněn nebyl. Tohoto faktu lze využít při extrémní optimalizaci.

Každá z jednotek má vlastní sadu řídících kódů, které jsou uvedeny v kapitole Jednotky. Navíc některé jednotky mohou ignorovat některé bity řídící sběrnice, pokud nevyžadují všech sedm, neboť je počet řídících kódů malý. Počet platných bitů je u každé jednotky uveden, neplatné bity jsou jednotkou ignorovány a považovány, jako by byly v logické 0. Tento fakt opět pomůže při optimalizaci attokódu.

Rozvržení řídící sběrnice je následující:

CC6	CC5	CC4	CC3	CC2	CC1	CC0	EB
8	9	10	11	12	13	14	15

CC0 až CC6 - řídící kód
EB - spouštěcí bit (execution bit)

Datová sběrnice 32bitová (bity 16 až 47)

Tato logická sběrnice je největší a umožňuje výměnu dat v libovolné formě (dle typu jednotky), až po 32 bitech najednou. Díky 32bitové šířce je možné snadno přenášet standardní typy jako 32bitový integer, reálné číslo s plovoucí řádovou čárkou s jednoduchou přesností (float), či 32 bitové adresy, které umožňují teoreticky adresovat až 4 GB dat po jednotlivých bajtech, ačkoli takové množství se v praxi u attoWPU zřejmě nevyužije a nebude ani implementováno.

Jednotky mohou některé bity datové sběrnice ignorovat podle toho, kolik jich pro výměnu dat potřebují, někdy se dokonce u jednotky počet ignorovaných bitů mění. Podobně jako u řídící sběrnice lze tohoto faktu využít při optimalizacích, neboť není třeba nastavovat všech 32 bitů.

Na rozdíl od adresové a řídící sběrnice mohou jednotky na tuto sběrnici data i vypouštět, nejenom číst a předat tak hodnotu jiné jednotce. Tímto je umožněna výměna dat mezi dvěma, někdy i více jednotkami připojenými k této sběrnici. Některé jednotky data ze sběrnice pouze čtou, nebo zapisují podle jejich účelu. Většina jednotek však data čte i zapisuje.

Jestliže některá jednotka bude vypouštět data na sběrnici, musí attokód nastavit všechny platné bity do logické 1, neboť se logické 0 chovají stejně jako zem v elektrických obvodech: uzemní jakékoli napětí, podobně by jiná jednotka přečetla logickou 0, i kdyby jiná jednotka vypouštěla na sběrnici logickou 1. Podobně i ostatní, momentálně nekomunikující jednotky musí vypouštět logické 1, či se od sběrnice odpojit (režim vysoké impedance), aby nenarušovaly datové přenosy.

Rozvržení datové sběrnice:

D32	D30	D29	D28	…	D03	D02	D01	D00
16	17	18	19	…	44	45	46	47

D00 až D32 slouží k přenosu až 32 bitových libovolných hodnot, bit D32 je v některých případech obvykle vnímán jako znaménkový bit.

QUICK AJUMP SBĚRNICE 16BITOVÁ (BITY 48 AŽ 63)

Tato speciální sběrnice je použita pouze jedinou jednotkou procesoru: aPC Write. V kombinaci s touto jednotkou umožňuje rychlý nepodmíněný skok v attokódu bez nutnosti měnit ostatní sběrnice. Ačkoli je sběrnice 16bitová, adresa je pouze 15bitová, neboť je nejméně významný bit použit podobně jako u řídící sběrnice a nazývá se skokový bit. Jestliže jednotka aPC write detekuje změnu skokového bitu z logické 0 na logickou 1, zapíše najednou 15 bitovou adresu do spodních 15 bitů aPC, čímž vytvoří programový skok.

Tímto lze provádět rychlé skoky uvnitř 32 kB bloku attokódu, avšak protože je paměť attokódu obvykle větší jak 32 kB, kupříkladu 1 MB, není možné pomocí této sběrnice

skočit kamkoliv v paměti. U 1 MB paměti je celkem 32 bloků po 32 kB, skok pomocí Quick aJump sběrnice lze tedy provést vždy pouze v daném bloku.

Rozvržení sběrnice je následující:

AAE	AAD	AAC	AAB	AAA	AA9	AA8	AA7	AA6	AA5	AA4	AA3	AA2	AA1	AA0	JB
48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63

AA0 až AAE - 15 bitová adresa
JB - skokový bit (jump bit)

JEDNOTKY PROCESORU

Na logické sběrnice procesoru je paralelně připojeno několik jednotek, které jsou programátorovi k dispozici pro vykonání různých činností. Jednotka, která bude přijímat řídící kódy je vybrána pomocí adresové sběrnice, proto má každá jednotka svou unikátní adresu. Každá jednotka má také svou vlastní sadu řídících kódů, přičemž maximum je 127 kódů (viz Řídící sběrnice), jednotky však mohou ignorovat některé bity řídící sběrnice, pokud je nevyžadují: nemají tolik řídících kódů, čímž se eliminuje nutnost nastavovat tyto neplatné bity, neboť jsou danou jednotkou vždy ignorovány a nijak nemění její činnost.

Veškeré jednotky využívají spouštěcí bit (nejméně významný bit řídící sběrnice) stejným způsobem: řídící kód je spuštěn při detekci změny spouštěcího bitu z logické 0 na logickou 1. Maximální počet jednotek je 256, avšak tolik jich procesor zdaleka neobsahuje a ponechává tak dostatečnou rezervu pro další verze. Je-li adresována neexistující jednotka, žádná z jednotek nebude přijímat řídící kódy a nedojde tedy k žádné akci.

Každá jednotka má v závorce uvedenu svou adresu. Většina jednotek má své vlastní interní registry, které slouží k řízení jejich operace. K těmto registrům nelze přímo přistupovat, avšak pomohou pochopit, jak daná jednotka pracuje a naprogramovat tak adekvátně řídící attokód. Pro specifikaci změněných registrů a bitů je použita syntaxe ve stylu jazyka C.

CLOCK - HODINOVÝ GENERÁTOR (---)

Hodinový generátor generuje hodinový signál o libovolné frekvenci (závisí na použitých obvodech, či výkonu počítače, na kterém běží simulátor), který určuje, kdy má jádro procesoru zpracovat jednu attoinstrukci. Hodinový generátor nemůže být ovládán přímo procesorem, proto tato jednotka nemá svou adresu ani řídící kódy. Zpomalením hodin v

simulátoru lze krokovat program. Pulz je nejprve poslán do aPC, který inkrementuje svou hodnotu a teprve poté je předán jádru, který zpracuje attoinstrukci odkazovanou aPC jednotkou. Technicky dochází k inkrementaci aPC před provedením attoinstrukce při každém pulzu, neboť po provedení instrukce odkazuje aPC na právě provedenou instrukci, v některých případech (po skoku a na startu) je však inkrementování přeskočeno, takže z praktického hlediska funguje programový čítač stejně, jako by byl inkrementován po provedení instrukce.

ATTOCORE - JÁDRO (---)

Jádro procesoru zpracovává při každém pulzu jednu attoinstrukci: nejprve načte instrukci z paměti attokódu, která je k tomuto jádru přímo připojena, následně instrukci dekóduje a změní jeden z bitů na sběrnici dle instrukce. Mimo této činnosti nemá žádnou jinou funkci, proto nemůže být tato jednotka adresována ani řízena pomocí sběrnic. Jako jediná může tato jednotka zapisovat do všech logických sběrnic.

APC WRITE (PŘÍMÝ PŘÍSTUP POMOCÍ QUICK AJUMP SBĚRNICE)

Tato speciální jednotka umožňuje rychlé lokání skoky v attokódu bez nutnosti měnit ostatní sběrnice. Programátor musí nejprve nastavit 15 bitů Quick aJump sběrnice na cílovou adresu lokálního skoku a následně změnit šestnáctý bit Quick aJump sběrnice z logické 0 na logickou 1. Jakmile tato jednotka detekuje tuto změnu, zapíše 15 bitů do spodních 15 bitů adresy aPC, čímž způsobí okamžitý nepodmíněný skok uvnitř 32 kB bloku attokódu. Ke skoku dochází pouze při změně z logické 0 na logickou 1, trvale nastavená jednička neustálý skok způsobovat nebude.

APC (0X00)

Attokódový programový čítač ukládá adresu attoinstrukce v paměti attokódu, která bude spuštěna jádrem. Adresa instrukce je přímo předána paměti attokódu, která poté předá attoinstrukci na příslušné adrese jádru ke zpracování. Pokaždé, když aPC obdrží impulz od hodinového generátoru, zvýší svou hodnotu o jedničku a předá pulz jádru, které zpracuje attoinstrukci.

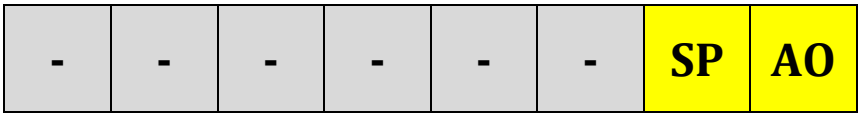
Je možné zapsat libovolnou hodnotu jako novou adresu pomocí adresové, řídící a datové sběrnice pro vykonání dlouhého skoku, či přikázat této jednotce, aby vypustila aktuální adresu na datovou sběrnici. Při zápisu nové hodnoty je jeden impulz z hodinového generátoru ignorován, jinak by došlo k okamžitému inkrementování nové hodnoty.

VNITŘNÍ REGISTRY

- IA (instruction address - adresa instrukce)
    - 24bitový, hodnota po resetu: 0x000000

    - Obsahuje adresu aktuální attoisntrukce
- CR (Control register - řídící registr)
    - 8bitový, hodnota po resetu: 0x02
    - Bity:
        - AO (address output), je-li v logické 1, hodnota registru IA bude vypuštěna na datovou sběrnici (24 bitů, zbylé bity sběrnice budou nezměněny)
        - SP (skip pulse), je-li v logické 1, hodnota IA nebude inkrementována po přijetí hodinového pulzu, namísto toho bude hodnota tohoto bitu změněna na logickou 0, takže při příštích pulzech dojde i inkrementování

Rozvržení registru CR:



ŘÍDICÍ KÓDY (2 PLATNÉ BITY)

Kód	Symbol	Akce	Změna registrů a bitů
0x00	STOP	Zastavit výstup dat	AO = 0
0x01	APC_W	Zapsat novou adresu	IA = (24b)DATA; SP = 1
0x02	APC_O	Vypustit hodnotu IA na datovou sběrnici	AO = 1; (24b)DATA &= IA
0x03	APC_R	Resetovat	IA = 0x00000; SP = 1;

ATTOKÓD MEMORY - PAMĚŤ ATTOKÓDU (0X01)

Jedná se o RWM RAM paměť s maximální velikostí 16 MB, která ukládá attokód procesoru, který je zpracováván jádrem. Maximální možná kapacita 16 MB při 8bitových paměťových buňkách umožňuje uložit přesně 16 777 216 attoinstrukcí, což by mělo být více než dostatečné pro většinu účelů. Paměť attokódu čte adresu instrukce z aPC a vypouští přímo do jádra příslušnou attoinstrukci.

Dále je možné číst i zapisovat data v této paměti pomocí datové sběrnice, což umožňuje měnit attokód za běhu a dosáhnout tak sebemodifikujícího se procesoru, ačkoli taková technika bude zřejmě zřídka využívána, důležitější je možnost při tvorbě attokódu uložit do této paměti potřebné konstanty a další data a během běhu programu je číst.

INTERNÍ REGISTRY

- AD (address - adresa)
    - 24bitový, hodnota po resetu: 0x000000

- obsahuje adresu právě adresované paměťové buňky pro čtení či zápis, tato adresa je nezávislá na buňce adresované pomocí aPC

- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - AO (address output), je-li v logické 1, hodnota AD registru bude vypuštěna na datovou sběrnici (24 bitů)
    - DO (data output), je-li v logické 1, hodnota buňky adresované AD registrem bude vypuštěna na datovou sběrnici (8 bitů)

Rozvržení registru CR:

-	-	-	-	-	-	**DO**	**AO**

ŘÍDÍCÍ KÓDY (4 PLATNÉ BITY)

Kód	**Symbol**	**Akce**	**Změněné registry a bity**
0x00	STOP	Zastavit výstup dat	AO = 0; DO = 0
0x01	AM_AD	Zapsat novou adresu	AD = (24b)DATA
0x02	AM_OA	Vypustit aktuální adresu	AO = 1; DO = 0; (24b)DATA &= AD
0x03	AM_OD	Vypustit naadresovaná data	DO = 1; AO = 0; (8b)DATA &= *AD
0x04	AM_WR	Zapsat data z datové sběrnice	*AD = (8b)DATA
0x05	AM_WN	Zapsat data z datové sběrnice a přesunout se na další buňku	*AD = (8b)DATA; AD++
0x06	AM_WP	Zapsat data z datové sběrnice a přesunout se na předchozí buňku	*AD = (8b)DATA; AD--
0x07	AM_NX	Přesunout se na následující buňku	AD++
0x08	AM_PR	Přesunout se na předchozí buňku	AD--
0x09 … 0x0F		Bez akce	

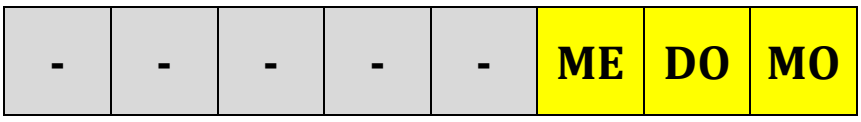
TEMP REGISTER - TEMP REGISTR (0X02)

Jedná se o samostatný nezávislý registr, který umožňuje uložit 32 bitovou hodnotu (tedy celou hodnotu datové sběrnice). Je také přímo připojen k jednotkám ALU a FPU, neboť slouží k uložení jednoho operandu. Často bude používán pro dočasné uložení různých typů dat, včetně adres. Registr neobsahuje pouze data, ale také 32bitovou masku, která určuje, které bity budou ignorovány při čtení či zápisu dat.

VNITŘNÍ REGISTRY

- DT (data)
  - 32bitový, hodnota po resetu: 0x00000000
  - obsahuje samotná uložená data
- MK (mask - maska)
  - 32bitový, hodnota po resetu: 0xFFFFFFFF
  - Vstupně výstupní maska. Bity v logické 0 určují bity, které budou při zápisu či čtení z registru ignorovány, tedy nedojde k jejich změně u DT registru v případě zápisu a datové sběrnice v případě čtení.
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - MO (mask output), je-li v logické 1, bude hodnota registru MK vypuštěna na datovou sběrnici
    - DO (data output), je-li v logické 1, bude hodnota registru DT vypuštěna na datovou sběrnici (popřípadě jen bity udržené MK registrem, viz ME bit)
    - ME (mask enable), je-li v logické 1, při vypuštění dat z registru DT na sběrnici jsou vypuštěny pouze ty bity, u kterých je u odpovídajícího bitu v registru MK nastavena logická 1, ostatní bity jsou v režimu vysoké impedance, či vždy 1 (nestahují hodnotu na sběrnici dolů)

Rozvržení registru CR:



ŘÍDÍCÍ KÓDY (4 PLATNÉ BITY)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	MO = 0; DO = 0
0x01	TMP_WRM	Zapsat hodnotu (s použitím masky)	DT = (DATA&MK) | (DT&~MK)
0x02	TMP_ODM	Vypustit hodnotu (s použitím masky)	DO = 1; ME = 1; MO = 0; DATA = (DT&MK) | (DATA&~MK)
0x03	TMP_WR	Zapsat hodnotu (bez masky)	DT = DATA
0x04	TMP_OD	Vypustit hodnotu (bez masky)	ME = 0; DO = 1; MO = 0; DATA = DT
0x05	TMP_WM	Zapsat masku	MK = DATA
0x06	TMP_OM	Vypustit masku na datovou sběrnici	MO = 1; DO = 0;

0x07	TMP_ME	Aktivovat masku	ME = 1
0x08	TMP_MD	Deaktivovat masku	ME = 0
0x09	TMP_CLR	Vynulovat	DT = 0x00000000
0x0A	TMP_FLL	Vyplnit	DT = 0xFFFFFFFF
0x0B … 0x0F		Bez akce	

REGISTER MEMORY - PAMĚŤ REGISTRŮ (0X03)

Speciální nezávislá malá paměť, umožňující uložit až 256 32bitových hodnot (tedy 1 kB dat), které mohou být použity jako přídavné registry. Tato paměť je nezávislá na paměti programu a dat, a proto poskytuje prostor pro různé registry, které bude využívat attokód bez toho, aby byla ovlivněna paměť programu a dat. Použití této paměti je však zcela na programátorovi, může ji použít pro libovolné účely, či ji zcela ignorovat a další registry vytvořit kupříkladu ve zmíněné paměti programu a dat.

VNITŘNÍ REGISTRY

- AD (address - adresa)
  - 8bitový, hodnota po resetu: 0x00
  - obsahuje adresu právě naadresované paměťové buňky
- PA (previous address - předchozí adresa)
  - 8bitový, hodnota po resetu: 0x00
  - obsahuje předchozí hodnotu AD registru
- MK (mask - maska)
  - 32bitový, hodnota po resetu: 0xFFFFFFFF
  - Vstupně výstupní maska. Bity v logické 0 určují bity, které budou při zápisu či čtení z paměti ignorovány, tedy nedojde k jejich změně u naadresované buňky paměti v případě zápisu a datové sběrnice v případě čtení.
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - MO (mask output), je-li v logické 1, je obsah registru MK vypuštěn na datovou sběrnici
    - DO (data output), je-li v logické 1, je obsah paměťové buňky naadresovaný pomocí AD registru vypuštěn na datovou sběrnici
    - ME (mask enable), je-li v logické 1, při vypuštění dat z naadresované buňky na sběrnici jsou vypuštěny pouze ty bity, u kterých je u odpovídajícího bitu v registru MK nastavena logická 1,

ostatní bity jsou v režimu vysoké impedance, či vždy 1 (nestahují hodnotu na sběrnici dolů)

- AO (address output), je-li v logické 1, je obsah registru AD vypuštěn na datovou sběrnici

ŘÍDÍCÍ KÓDY (5 PLATNÝCH BITŮ)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	MO = 0; DO = 0; AO = 0
0x01	RG_AD	Zapsat novou adresu	PA = AD; AD = (8b)DATA
0x02	RG_AO	Vypustit aktuální adresu na datovou sběrnici	AO = 1; DO = 0; MO = 0; (8b)DATA &= AD
0x03	RG_ODM	Vypustit obsah naadresované buňky (s použitím masky)	DO = 1; AO = 0; MO = 0; ME = 1; DATA &= (*AD&MK) | (DATA&~MK)
0x04	RG_WRM	Zapsat data z datové sběrnice (s použitím masky)	*AD = (DATA&MK) | (*AD&~MK)
0x05	RG_WNM	Zapsat data z datové sběrnice a přejít na další buňku (s použitím masky)	*AD = (DATA&MK) | (*AD&~MK); AD++;
0x06	RG_WPM	Zapsat data z datové sběrnice a přejít na předchozí buňku (s použitím masky)	*AD = (DATA&MK) | (*AD&~MK); AD--;
0x07	RG_NX	Přejít na další buňku	AD++
0x08	RG_PR	Přejít na předchozí buňku	AD--
0x09	RG_WM	Zapsat novou masku	MK = DATA;
0x0A	RG_OM	Vypustit masku na datovou sběrnici	MO = 1; DO = 0; AO = 0; DATA &= MK;
0x0B	RG_ME	Aktivovat masku	ME = 1;
0x0C	RG_MD	Deaktivovat masku	ME = 0;
0x0D	RG_OD	Vypustit obsah naadresované buňky (bez masky)	DO = 1; AO = 0; MO = 0; ME = 0; DATA &= *AD;
0x0E	RG_WR	Zapsat data z datové sběrnice (bez masky)	*AD = DATA;
0x0F	RG_WN	Zapsat data z datové sběrnice a přejít na další buňku (bez masky)	*AD = DATA; AD++;
0x10	RG_WP	Zapsat data z datové sběrnice a přejít na předchozí buňku (bez masky)	*AD = DATA; AD--;
0x11	RG_RES	Navrátit předchozí adresu	AD = PA
0x12 ... 0x1F		Bez akce	

ALU (0x04)

Tato jednotka umožňuje provádění aritmetických a logických operací na 32bitových datech, jako je sčítání odečítání, násobení, dělení, operace AND, OR, XOR, NOT a podobně. Neboť je potřeba dvou 32bitových operandů, ALU používá hodnotu na datové sběrnici a

hodnotu uloženou v TEMP registru. Hodnota z TEMP registru je přímo předána jednotce ALU nezávisle na tom, v jakém módu TEMP registr je a není na ni aplikována maska. Výsledek operace je zapsán do registru OUT.

Najednou může být provedena pouze jediná operace a výsledek je vždy uložen pouze v OUT registru, nejsou zde žádné speciální bity pro indikaci přenosu, či další registr pro uložení 64bitových výsledků. Namísto toho musí být každá z těchto operací provedena v jednotlivých oddělených krocích s použitím různých řídících kódů. Je tedy zcela na programátorovi, jestli kupříkladu využije přenosový bit, jestli ho bude ignorovat, popřípadě jak s ním bude zacházet. ALU také obsahuje speciální funkce pro rozhodovací logiku.

INTERNÍ REGISTRY
Neobsahuje.

ŘÍDÍCÍ KÓDY (6 PLATNÝCH BITŮ)

Kód	Symbol	Akce	Změněné registry a bity
0x00	ZERO	Nula	OUT = 0
0x01	ADD	ADD (sčítání)	OUT = (uint)DATA + TEMP
0x02	SUB	SUB (odčítání)	OUT = (uint)DATA - TEMP
0x03	MULL	MUL_LOW (násobení – nižších 32 bitů)	OUT = (lower 32b) (uint)DATA*TEMP
0x04	MULH	MUL_HIGH (násobení – vyšších 32 bits)	OUT = (higher 32b) (uint)DATA*TEMP
0x05	DIV	DIV (celočíselné dělení)	OUT = (uint)DATA/TEMP
0x06	REM	REM (zbytek po dělení - modulo)	OUT = (uint)DATA % TEMP
0x07	CR	Indikace přenosu při sčítání	OUT = (bool)CARRY(DATA+TEMP)
0x08	BO	Indikace přenosu při odečítání	OUT = (bool)BORROW(DATA-TEMP)
0x09	SADD	SADD (sčítání se znaménkovým bitem)	OUT = (int)DATA + TEMP
0x0A	SSUB	SSUB (odečítání se znaménkovým bitem)	OUT = (int)DATA - TEMP
0x0B	SMULL	SMUL_LOW (násobení se znaménkovým bitem – nižších 32 bitů)	OUT = (lower 32b) (int) DATA*TEMP
0x0C	SMULH	SMUL_HIGH (násobení se znaménkovým bitem - vyšších 32 bitů)	OUT = (higher 32b) (int) DATA*TEMP
0x0D	SDIV	SDIV (dělení se znaménkovým bitem)	OUT = (int)DATA/TEMP
0x0E	SREM	SREM (zbytek po dělení se znaménkovým bitem)	OUT = (int)DATA%TEMP
0x0F	SCR	Indikace přenosu při sčítání se znaménkovým bitem	OUT = (bool)sCARRY(DATA+TEMP)
0x10	SBO	Indikace přenosu při odečítání se znaménkovým bitem	OUT = (bool)sBORROW(DATA-TEMP)
0x11	ANDB	Binární AND	OUT = DATA & TEMP

0x12	ORB	Binární OR	OUT = DATA | TEMP
0x13	NOTB	Binární NOT	OUT = ~DATA
0x14	XORB	Binární XOR	OUT = DATA ^ TEMP
0x15	RL	Rotace vlevo (s přenosem)	OUT = (DATA << TEMP) | (DATA >> (32-TEMP))
0x16	RR	Rotace vpravo (s přenosem)	OUT = (DATA >> TEMP) | (DATA << (32-TEMP))
0x17	ANDL	Logická AND	OUT = DATA && TEMP
0x18	ORL	Logická OR	OUT = DATA || TEMP
0x19	NOTL	Logická NOT	OUT = !DATA
0x1A	XORL	Logická XOR	OUT = (bool)DATA ^ (bool)TEMP
0x1B	SL	Posun vlevo (bez přenosu)	OUT = DATA << TEMP
0x1C	SR	Posun vpravo (bez přenosu)	OUT = DATA >> TEMP
0x1D	NAND	Binární NAND	OUT = ~(DATA & TEMP)
0x1E	NOR	Binární NOR	OUT = ~(DATA | TEMP)
0x1F	BOOL	Převést na boolean hodnotu (0 nebo 1)	OUT = (bool)DATA
0x20	MAX	Zvolit větší číslo	OUT = MAX(DATA, TEMP)
0x21	MAXN	Indikovat, které z čísel je větší DATA => 0, TEMP =>1	OUT = TEMP > DATA
0x22	MIN	Zvolit menší číslo	OUT = MIN(DATA, TEMP)
0x23	MINN	Indikovat, které z čísel je menší DATA => 0, TEMP =>1	OUT = TEMP < DATA
0x24	SMAX	Zvolit větší číslo (se znaménkovým bitem)	OUT = MAX((signed)DATA, (signed)TEMP)
0x25	SMAXN	Indikovat, které z čísel je větší (se znaménkovým bitem) DATA => 0, TEMP =>1	OUT = (signed)TEMP > (signed)DATA
0x26	SMIN	Zvolit menší číslo (se znaménkovým bitem)	OUT = MIN((signed)DATA, (signed)TEMP)
0x27	SMINN	Indikovat, které z čísel je menší (se znaménkovým bitem) DATA => 0, TEMP =>1	OUT = (signed)TEMP < (signed)DATA
0x28	EQL	Indikovat, zdali se čísla rovnají	OUT = DATA == TEMP
0x29	ZSET	Zkopírovat hodnotu z DATA do OUT pouze pokud je TEMP nula	If(!TEMP) OUT = DATA
0x2A	NZSET	Zkopírovat hodnotu z DATA do OUT pouze pokud je TEMP nenulový	If(TEMP) OUT = DATA
0x27 ... 0x3F		Bez akce	

OUT REGISTER - VÝSTUPNÍ REGISTR (0X05)

Tento samostatný 32bitový nezávislý registr je přístupný pouze pro čtení z datové sběrnice, je využíván jednotkami ALU a FPU pro ukládání výsledků operací. Výsledek uložený v tomto registru může být vypuštěn na datovou sběrnici kdykoliv je potřeba.

VNITŘNÍ REGISTRY

- DT (data)

- 32bitový, hodnota po resetu: 0x00000000
- obsahuje samotná data registru

- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - DO (data output), je-li v logické 1, pak je obsah registru DT vypuštěn na datovou sběrnici

Rozvržení registru CR:

-	-	-	-	-	-	-	**DO**

ŘÍDÍCÍ KÓDY (1 PLATNÝ BIT)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	DO = 0
0x01	OUT_D	Aktivovat výstup dat	DO = 1; DATA = DT

FPU (0x06)

Tato jednotka umožňuje provádění operací s reálnými čísly s plovoucí desetinnou čárkou s jednoduchou přesností a ukládání výsledků do OUT registru. Hodnoty na datové sběrnici a v TEMP registru jsou považovány za hodnoty ve formátu float, nedochází k žádnému rozlišování datových typů, takže je na programátorovi, aby připravil data ve správném formátu.

INTERNÍ REGISTRY

Nejsou.

ŘÍDÍCÍ KÓDY (5 PLATNÝCH BITŮ)

Kód	Symbol	Akce	Změněné registry a bity
0x00	ZERO	Nula	OUT = 0.0F
0x01	FADD	Sčítání	OUT = (float)DATA + (float)TEMP
0x02	FSUB	Odečítání	OUT = (float)DATA – (float)TEMP
0x03	FMUL	Násobení	OUT = (float)DATA * (float)TEMP
0x04	FDIV	Dělení	OUT = (float)DATA / (float)TEMP
0x05	FSIN	Sinus (úhel je radiánech)	OUT = sin((float)DATA)
0x06	FTAN	Tangens	OUT = tan((float)DATA)
0x07	FEXP	Exponenciální funkce	OUT = exp((float)DATA)
0x08	FSQRT	Druhá odmocnina	OUT = sqrt((float)DATA)

0x09	FLOG2	Logaritmus při základu 2	OUT = log2((float)DATA)
0x0A	FLOG10	Logaritmus při základu 10	OUT = log10((float)DATA)
0x0B	FLN	Přirozený logaritmus	OUT = ln((float)DATA)
0x0C	FISINF	Detekce, jestli je hodnota nekonečno	OUT = ((float)DATA == 1.#INF) || ((float)DATA == -1.#INF)
0x0D	FTOINT	Převedení float na integer	OUT = (int)DATA
0x0E	FTOFLT	Převedení integeru na float	OUT = (float)DATA
0x0F	FMAX	Vybrat větší číslo	OUT = max((float)DATA, (float)TEMP)
0x10	FMAXN	Indikovat, které číslo je větší	OUT = ((float)TEMP>(float)DATA)
0x11	FMIN	Vybrat menší číslo	OUT = min((float)DATA, (float)TEMP)
0x12	FMINN	Indikovat, které číslo je menší	OUT = ((float)TEMP<(float)DATA)
0x13	FABS	Absolutní hodnota	OUT = abs((float)DATA)
0x14	FPOW	Mocnina	OUT = pow((float)DATA, (float)TEMP)
0x15 ... 0x1F		Bez akce	

MEMORY CONTROLLER A - PAMĚŤOVÝ ŘADIČ A (0X07)

Tato jednotka umožňuje přístup k operační paměti, která může ukládat vlastní program (nikoliv attokód) a data. Pomocí této jednotky dochází k adresování paměťových buněk, čtení a zápisu dat. Paměťový řadič je obecný a v podstatě stejný, jako je paměťový řadič B, jediným rozdílem je, s jakou pamětí pracuje. Řadič umožňuje teoreticky až 64bitové adresování, toto však v praxi nebude využito, 32 bitové adresování poskytuje obrovské množství paměťového prostoru, doporučené množství paměti při běžném použití je však 16 MB, avšak množství závisí čistě na implementaci a potřebách.

Maximální velikost paměťové buňky je 32bitů, avšak lze použít i menší hodnoty: 24 bitů, 16 bitů a 8 bitů. Kupříkladu u paměťového řadiče B bude mnohem častěji použito 8 bitů pro přístup k externí paměti po bajtech, zatímco u operační paměti je vhodné 32 bitů, což odpovídá šířce datové sběrnice. Velikost buňky také určuje kolik bitů je při čtení a zápisu z datové sběrnice platných.

VNITŘNÍ REGISTRY

- AD (address - adresa)
  - 64bitový, hodnota po resetu: 0x0000000000000000
  - adresa právě naadresované paměťové buňky
- SZ (size - velikost)
  - 64bitový

- obsahuje vždy velikost paměti v bajtech (nikoli paměťových buňkách, což je proměnlivá hodnota dle nastavené velikosti buňky)

- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0xC0
  - Bity:
    - ALO (address low output), je-li v logické 1, je na datovou sběrnici vypuštěno spodních 32 bitů AD registru
    - DO (data output), je-li v logické 1, hodnota naadresované paměťové buňky je vypuštěna na datovou sběrnici
    - AHO (address high output), je-li v logické 1, je na datovou sběrnici vypuštěno vrchních 32 bitů AD registru
    - SO (size output), je-li v logické 1, je na datovou sběrnici vypuštěn obsah registru SZ, čímž může attokód zjistit, kolik paměti má k dispozici
    - CO (cell size output), je-li v logické 1, je na datovou sběrnici vypuštěna právě nastavená velikost paměťové buňky (2 bitová hodnota složená z bitů CEL a CEH)
    - CEL a CEH (cell size low, cell size high), kombinace těchto dvou bitů určuje velikost paměťové buňky, dle následující tabulky:

CEH	CEL	Velikost buňky	Násobič adresy
0	0	8 bit	1
0	1	16 bit	2
1	0	24 bit	3
1	1	32 bit	4

Rozložení registru CR:

CEH	**CEL**	**-**	**CO**	**SO**	**AHO**	**DO**	**ALO**

ŘÍDÍCÍ KÓDY (5 PLATNÝCH BITŮ)

Kód	**Symbol**	**Akce**	**Změněné registry a bity**
0x00	STOP	Zastavit výstup dat	ALO = DO = AHO = SO = CO = 0;
0x01	M_WRL	Zapsat spodní adresu	(low 32b)AD = DATA;
0x02	M_OAL	Vypustit spodní adresu	DO = AHO = SO = CO = 0; ALO = 1; DATA = (low 32b)AD;
0x03	M_OD	Vypustit naadresovaná data	ALO = AHO = SO = CO = 0;

			DO = 1; (cell size)DATA = *AD;
0x04	M_WR	Zapsat data z datové sběrnice	*AD = (cell size)DATA;
0x05	M_WN	Zapsat data z datové sběrnice a přejít na další buňku	*AD = (cell size)DATA; AD++;
0x06	M_WP	Zapsat data z datové sběrnice a přejít na předchozí buňku	*AD = (cell size)DATA; AD--;
0x07	M_NX	Přejít na další buňku	AD++
0x08	M_PR	Přejít na předchozí buňku	AD--
0x09	M_WRH	Zapsat vrchní adresu	(higher 32b)AD = DATA;
0x0A	M_OAH	Vypustit vrchní adresu	ALO = DO = SO = CO = 0; AHO = 1; DATA = (higher 32b)AD;
0x0B	M_SZ	Vypustit kapacitu paměti v bajtech	ALO = DO = AHO = CO = 0; SO = 1; DATA = SZ;
0x0C	M_CL	Nastavit velikost buňky	CEL = (1st bit)DATA; CEH = (2nd bit)DATA;
0x0D	M_32	Nastavit velikost buňky na 32b	CEL = 1; CEH = 1;
0x0E	M_24	Nastavit velikost buňky na 24b	CEL = 0; CEH = 1;
0x0F	M_16	Nastavit velikost buňky na 16b	CEL = 1; CEH = 0;
0x10	M_8	Nastavit velikost buňky na 8b	CEL = 0; CEH = 0;
0x11	M_OCL	Vypustit velikost paměťové buňky na datovou sběrnici	ALO = AHO = DO = SO = 0; CO = 1; (2b)DATA = CEL | (CEH << 1)
0x12 **…** **0x1F**		Bez akce	

Paměťový řadič B (0x08)

Funkčně je shodný s paměťovým řadičem A, avšak obvykle bývá připojen k externímu paměťovému médiu, nebo nepoužit. V simulátoru to může být kupříkladu soubor na pevném disku, díky 64bitovému adresování je možné takto adresovat i velké soubory.

SmallQueue - MaláFronta (0x09)

Tato speciální paměť uloží až 32 32bitových hodnot z datové sběrnice. Hlavní předností je schopnost automaticky vypustit či načíst hodnotu do/z datové sběrnice v předdefinovaných intervalech, což programátorovi umožňuje jednoduše instruovat ostatním jednotkám vypuštění různých dat, bez nutnosti neustále adresovat tuto paměť a přikazovat uložení dat. Tímto lze výrazně zrychlit některé operace, zvláště při zpracování větších množství dat.

Operace je řízena odečítacím registrem: jakmile dosáhne hodnoty nula, SmallQueue začne číst či zapisovat data, dokud AD registr nedosáhne maximální hodnoty. Každá čtecí či zapisovací operace proběhne za N attocyklů, kde N je hodnota předem nastavená

programátorem ve speciálním registru. Po každé operaci je hodnota N použita k naplnění odečítacího registru. Tak je možné odečítacímu registru nastavit jinou prodlevu, než je mezi čtecími či zápisovými operacemi.

INTERNÍ REGISTRY

- AD (address register - adresový registr)
  - 8bitový, hodnota po resetu: 0x00
  - obsahuje právě naadresovanou buňku SmallQueue paměti (je použito jen 5 bitů)
- MK (mask - maska)
  - 32bitový, hodnota po resetu: 0xFFFFFFFF
  - Vstupně výstupní maska. Bity v logické 0 určují bity, které budou při zápisu či čtení z paměti ignorovány, tedy nedojde k jejich změně u naadresované buňky paměti v případě zápisu a datové sběrnice v případě čtení.
- CD (countdown - odečítání)
  - 16bitový, hodnota po resetu: 0x0000
  - obsahuje počet attocyklů před dalším čtením/zápisem
- FL (fill - naplnění)
  - 16bitový, hodnota po resetu: 0x0000
  - obsahuje hodnotu, která bude použita k naplnění CD registru při dosažení nuly
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - DO (data output), je-li v logické 1, je obsah aktuálně naadresované paměťové buňky vypuštěn na datovou sběrnici
    - DW (data write), je-li v logické 1, tak pokaždé, když CD dosáhne nuly, je hodnota z datové sběrnice uložena do právě naadresované buňky a AD je inkrementován
    - ME (mask enable), je-li v logické 1, při vypuštění dat z naadresované buňky na sběrnici jsou vypuštěny pouze ty bity, u kterých je u odpovídajícího bitu v registru MK nastavena logická 1, ostatní bity jsou v režimu vysoké impedance, či vždy 1 (nestahují hodnotu na sběrnici dolů)
    - QR (queue run), je-li v logické 1, je CD registr dekrementován při každém attocyklu a v případě, že dosáhne nuly, jsou provedeny příslušené operace: zápis či čtení dat (záleží na DO a DW, pokud

jsou oba v logické 0, je manipulace s daty přeskočena), CD je naplněn hodnotou z FL a AD je inkrementován. Jestliže AD dosáhne maximální hodnoty, je QR automaticky vynulován, čímž se běh zastaví

Rozložení registru CR:

QR	-	-	-	ME	DW	DO	AO

ŘÍDÍCÍ KÓDY (5 PLATNÝCH BITŮ)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit čtení či zápis dat	AO = DO = DW = 0;
0x01	SQ_AD	Zapsat novou adresu	AD = (8b)DATA;
0x02	SQ_OA	Vypustit aktuální adresu	DO = DW = 0; AO = 1;
0x03	SQ_ODM	Vypustit naadresovaná data (s použitím masky)	AO = DW = 0; DO = 1; ME = 1; DATA = (*AD & MK) | (DATA & ~MK);
0x04	SQ_WRM	Zapsat data z datové sběrnice (s použitím masky)	*AD = (DATA & MK) | (*AD & ~MK);
0x05	SQ_NX	Přejít na další buňku	AD++
0x06	SQ_PR	Přejít na předchozí buňku	AD--
0x07	SQ_OD	Vypustit naadresovaná data (bez masky)	AO = DW = 0; DO = 1; ME = 0; DATA = *AD;
0x08	SQ_WR	Zapsat data z datové sběrnice (bez masky)	*AD = DATA;
0x09	SQ_ME	Aktivovat masku	ME = 1;
0x0A	SQ_MD	Deaktivovat masku	ME = 0;
0x0B	SQ_O	Aktivovat mód výstupu dat (čtení)	AO = DW = 0; DO = 1;
0x0C	SQ_I	Aktivovat mód vstupu dat (zápis)	AO = DO = 0; DW = 1;
0x0D	SQ_CD	Nastavit odpočet	CD = (16b)DATA;
0x0E	SQ_FL	Nastavit naplnění	FL = (16b)DATA;
0x0F	SQ_R	Spustit	QR = 1;
0x10	SQ_S	Zastavit	QR = 0;
0x11 .. 0x1F		Bez akce	

LED CONTROL - VÝSTUP LED (0X0A)

Toto je nejjednodušší jednotka pro výstup dat, umožňuje binárně zobrazovat 32 bitové hodnoty pomocí LED diod, obsahuje 4 řady těchto diod pro čtyři hodnoty. Zobrazovanou hodnotu ukládá ve svých vlastních registrech, jakmile je tedy hodnota z datové sběrnice zapsána, zůstane stejná, dokud není přepsána jinou.

INTERNÍ REGISTRY

- ROW0 (řada 0), 32bitový, hodnota po resetu: 0x00000000
- ROW1 (řada 1), 32bitový, hodnota po resetu: 0x00000000
- ROW2 (řada 2), 32bitový, hodnota po resetu: 0x00000000
- ROW3 (řada 3), 32bitový, hodnota po resetu: 0x00000000

ŘÍDÍCÍ KÓDY (2 PLATNÉ BITY)

Kód	Symbol	Akce	Změněné registry a bity
0x00	LED_R0	Zapsat do řady 0	ROW0 = DATA;
0x01	LED_R1	Zapsat do řady 1	ROW1 = DATA;
0x02	LED_R2	Zapsat do řady 2	ROW2 = DATA;
0x03	LED_R3	Zapsat do řady 3	ROW3 = DATA;

TEXT DISPLAY CONTROLLER - ŘADIČ TEXTOVÉHO DISPLEJE (0x0B)

Tato jednotka slouží k rychlému zobrazení textových informací použitím textového displeje s 40x4 znaky. Text je zapsán nepřímo: řadič je vybaven 160 bajtovou pamětí a periodicky aktualizuje samotný displej (který není součástí procesoru), nezávisle na ostatních součástech procesoru. Text musí být odeslán ve formátu ASCII o sedmi bitech, hodnoty nad 128 nemají přiřazeny žádné znaky).

VNITŘNÍ REGISTRY

- AD (address - adresa)
  - 8bitový, hodnota po resetu: 0x00
  - aktuálně naadresovaný znak, je-li hodnota větší jak 160, bude vynulován
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - AO (address output), je-li v logické 1, je obsah registru AD vypuštěn na datovou sběrnici
    - DO (data output), je-li v logické 1, je kód aktuálně naadresovaného znaku vypuštěn na datovou sběrnici

ŘÍDÍCÍ KÓDY (4 PLATNÉ BITY)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	AO = DO = 0;
0x01	TX_ADR	Zapsat adresu znaku	AD = (8b)DATA;
0x02	TX_WR	Zapsat znak	*AD = (8b)DATA;
0x03	TX_WN	Zapsat znak a přesunout se na další	*AD = (8b)DATA; AD++;
0x04	TX_NX	Další znak	AD++;
0x05	TX_PR	Předchozí znak	AD--;
0x06	TX_OA	Vypustit adresu znaku na datovou sběrnici	DO = 0; AO = 1;

			(8 valid bits)DATA = AD;
0x07	TX_OD	Vypustit kód naadresovaného znaku na datovou sběrnici	AO = 0; DO = 1; (8b)DATA = AD;
0x08	TX_R	Resetovat adresu	AD = 0;
0x09	TX_CLR	Vymazat paměť	AD = 0; Clear *AD;
0x0A ... 0x0F		Bez akce	

LCD Display Controller - Řadič LCD (0x0C)

Pomocí této jednotky lze zobrazovat bitmapová data s maximálním rozlišením 128x128 pixelů v 24bitové barevné hloubce, lze jej tedy použít pro zobrazení libovolného druhu informací. Podobně jako textový řadič má svou vlastní paměť pro uložení dat k zobrazení, takže může zobrazovat data nezávisle na procesoru a také zpětně číst zapsané hodnoty.

Každý pixel obsahuje 24 bitů dat, 8 bitů pro každý barevný kanál. Bitmapová data jsou adresována lineárně, od levého vrchní rohu směrem doprava. Jestliže programátor potřebuje adresovat pixely dle jejich XY souřadnic, musí sám provést nezbytné kalkulace adresy.

Je také možné aktivovat double buffering, kde jsou použity dvě paměti, každá schopná uložit 128x128x24bpp obrázek. Jedna paměť bude použita k zápisu nových dat, druhá bude použita k zobrazení těchto dat na displeji. Jakmile programátor dokončí zápis nových dat, použije řídící kód pro výměnu rolí pamětí: nová data budou zobrazena, zatímco druhá paměť bude dostupná pro zápis nových dat. Toto zabraňuje uživateli, aby viděl, jak se data vykreslují, pixel po pixelu.

Vnitřní registry

- AD (address - adresa)
  - 24bitový, hodnota po resetu: 0x0000
  - aktuálně naadresovaný pixel, adresuje 24bitové buňky, je-li hodnota větší jak 16383, bude vynulován. Při použití double bufferingu umožňuje přístup pouze k paměti, která se zrovna nevykresluje na displej
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - AO (address output), je-li v logické 1, je obsah registru AD vypuštěn na datovou sběrnici
    - DO (data output), je-li v logické 1, je RGB kód aktuálně naadresovaného pixel vypuštěn na datovou sběrnici

- BE (buffer enable), je-li v logické 1, je aktivován double buffering
- BM (buffer memory), je-li v logické 0, je první paměť použita pro zápis dat a druhá pro zobrazování, je-li v logické 1, jsou funkce pamětí prohozeny

ŘÍDÍCÍ KÓDY (5 PLATNÝCH BITŮ)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	AO = DO = 0;
0x01	LCD_AD	Zapsat adresu pixelu	AD = (16b)DATA;
0x02	LCD_WR	Zapsat pixel	*AD = (24b)DATA;
0x03	LCD_WN	Zapsat pixel a přesunout se na další	*AD = (24b)DATA; AD++;
0x04	LCD_NX	Další pixel	AD++;
0x05	LCD_PR	Předchozí pixel	AD--;
0x06	LCD_AO	Vypustit adresu aktuálního pixelu na datovou sběrnici	DO = 0; AO = 1; (16b)DATA = AD;
0x07	LCD_DO	Vypustit RGB kód aktuálního pixelu na datovou sběrnici	AO = 0; DO = 1; (24b)DATA = AD;
0x08	LCD_R	Přejít na první pixel	AD = 0;
0x09	LCD_CLR	Vymazat paměť	AD = 0; Clear *AD;
0x0A	LCD_SB	Single buffer	BE = 0; BM = 0;
0x0B	LCD_DB	Double buffering	BE = 1;
0x0C	LCD_BS	Přehodit roli pamětí	BM = !BM
0x0D ... 0x0F		Bez akce	

INPUT CONTROLLER - ŘADIČ VSTUPU (0X0D)

Tento řadič nabízí přístup k několika metodám vstupu uživatelských dat do procesoru za běhu programu. Nejjednodušší formou vstupu jsou 4 řady přepínačů, každý přepínač může být pouze ve dvou stavech, takže každý přepínač odpovídá jednomu bitu. Další metodou je zjednodušená numerická klávesnice, kde má každá klávesa svůj vlastní kód, který odpovídá číslici+1, desetinná čárka má kód 11.

Poslední metodou, jejíž implementace však není povinná, je alfanumerická klávesnice: řadič vstupu umožňuje čtení skenovacích kódů kláves odpovídajících písmenům, je dokonce možné detekovat několik stištěných kláves najednou přeskočením několika kláves během skenování.

ROZLOŽENÍ NUMERICKÉ KLÁVESNICE

Každá klávesa má skenovací kód odpovídající číslici plus jedna. To znamená, že nula má kód 1, jednička kód 2 a podobně. Skenovací kód 0 znamená, že není stisknuta žádná klávesa.

Čtení několika skenovacích kódů

Klávesy jsou skenovány v sekvenčním vzestupném pořadí dle jejich skenovacího kódu. Nejjednodušší řídící kód navrací kód první nalezené klávesy, avšak další kód umožňuje přeskočit určitý počet nalezených kláves při skenování, zadaný 8bitovým číslem na datové sběrnici. Jestliže je toto číslo nenulové, bude zadaný počet zmáčknutých kláves během skenování přeskočen. Je-li například toto číslo 2, jsou první dvě zmáčknuté klávesy ignorovány a je vrácena buď třetí v pořadí, nebo nula, v případě že nejsou zmáčknuty tři klávesy.

Interní registry

- TD (temporary data - dočasná data)
  - 32bitový, hodnota po resetu: 0x00000000
  - registr slouží k dočasnému uložení dat, která budou vypuštěna na datovou sběrnici. Programátor nejprve odešle příkaz, který uloží vstupní hodnotu z některého vstupu do tohoto registru a teprve poté je vypuštěna na datovou sběrnici. To znamená, že pouze jediný registr a jediný bit je nutný pro výstup dat.
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00
  - Bity:
    - DO (data output), je-li v logické 1, je obsah registru TD vypuštěn na datovou sběrnici
    - BM (byte mode), je-li v logické 1, je při výstupu dat platných pouze 8 bitů, což je využito u skenovacích kódů, kde není třeba 32 bitů

Rozložení registru CR:

-	-	-	-	-	-	**BM**	**DO**

Řídící kódy (4 platné bity)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	DO = 0;
0x01	IN_DO	Aktivovat výstup dat	DO = 1;
0x02	IN_R0	Přečíst řadu přepínačů 0	TD = SW0; BM = 0; DO = 1;
0x03	IN_R1	Přečíst řadu přepínačů 1	TD = SW1; BM = 0; DO = 1;

0x04	IN_R2	Přečíst řadu přepínačů 2	TD = SW2; BM = 0; DO = 1;
0x05	IN_R3	Přečíst řadu přepínačů 3	TD = SW3; BM = 0; DO = 1;
0x06	IN_RN	Přečíst kód numerické bez přeskočení	TD = get_key(NUM, 0); BM = 1; DO = 1;
0x07	IN_SN	Přečíst kód numerické klávesy s přeskočením	TD = get_key(NUM, (8b)DATA); BM = 1; DO = 1;
0x08	IN_RK	Přečíst kód alfanumerické klávesy bez přeskočení	TD = get_key(KEYB, 0); BM = 1; DO = 1;
0x09	IN_SK	Přečíst kód alfanumerické klávesy s přeskočením	TD = get_key(KEYB, (8b)DATA); BM = 1; DO = 1;
0x0A ... 0x0F		Bez akce	

TIMER CONTROLLER - ŘADIČ ČASOVAČŮ (0X0E)

K přesnému měření počtu uběhlých attocyklů či milisekund může programátor využít řadiče časovačů. Tato jednotka nabízí přístup ke čtyřem 16bitovým časovačům čítajícím nahoru a dalšímu speciálnímu časovači, který měří uběhlé milisekundy, čímž umožňuje tvorbu aplikací pracujících v reálném čase.

Časovače lze spouštět a zastavovat, nastavovat předvyplňovací hodnotu, číst jejich aktuální hodnotu a také počet přetečení. Přečtením počtu přetečení se počet přetečení automaticky resetuje.

VNITŘNÍ REGISTRY

- TD (temp data - dočasná data)
  - 32bitový, hodnota po resetu: 0x00000000
  - použit pro zachycení dat z časovačů pro jejich výstup na datovou sběrnici
- T0, T1, T2, T3 (timer - časovač)
  - 16bitové, hodnota po resetu: 0x0000
  - obsahují hodnoty časovačů
- TF0, TF1, TF2, TF3 (timer fill - naplnění časovače)
  - 16bitové, hodnota po resetu: 0x0000
  - obsahují hodnoty, kterými budou vyplněny časovače při přetečení
- OC0, OC1, 0C2, 0C3 (overflow count - počet přetečení)
  - 16bitové, hodnota po resetu: 0x0000
  - obsahují počet přetečení časovačů
- RT (real time - reálný čas)
  - 32bitový, hodnota po resetu: 0x00000000
  - obsahuje počet milisekund uběhlých od startu procesoru
- CR (control register - řídící registr)
  - 8bitový, hodnota po resetu: 0x00

- Bity:
  - DO (data output), je-li v logické 1, je obsah registru TD vypuštěn na datovou sběrnici
  - WM (word mode), je-li v logické 1, je platných pouze 16 bitů při výstupu dat na sběrnici
  - TR0, TR1, TR2, TR3 (timer run), je-li v logické 1, pak je daný časovač spuštěn a inkrementuje se každý attocyklus

ŘÍDÍCÍ KÓDY (5 PLATNÝCH BITŮ)

Kód	Symbol	Akce	Změněné registry a bity
0x00	STOP	Zastavit výstup dat	DO = 0;
0x01	TI_DO	Aktivovat výstup dat	DO = 1;
0x02	TI_TR0	Spustit časovač 0	TR0 = 1;
0x03	TI_TR1	Spustit časovač 1	TR1 = 1;
0x04	TI_TR2	Spustit časovač 2	TR2 = 1;
0x05	TI_TR3	Spustit časovač 3	TR3 = 1;
0x06	TI_TS0	Zastavit časovač 0	TR0 = 0;
0x07	TI_TS1	Zastavit časovač 1	TR1 = 0;
0x08	TI_TS2	Zastavit časovač 2	TR2 = 0;
0x09	TI_TS3	Zastavit časovač 3	TR3 = 0;
0x0A	TI_TF0	Naplnit časovač 0	T0 = (16b)DATA;
0x0B	TI_TF1	Naplnit časovač 1	T1 = (16b)DATA;
0x0C	TI_TF2	Naplnit časovač 2	T2 = (16b)DATA;
0x0D	TI_TF3	Naplnit časovač 3	T3 = (16b)DATA;
0x0E	TI_AF0	Nastavit vyplnění časovače 0	TF0 = (16b)DATA;
0x0F	TI_AF1	Nastavit vyplnění časovače 1	TF1 = (16b)DATA;
0x10	TI_AF2	Nastavit vyplnění časovače 2	TF2 = (16b)DATA;
0x11	TI_AF3	Nastavit vyplnění časovače 3	TF3 = (16b)DATA;
0x12	TI_OC0	Vypustit počet přetečení časovače 0	TD = OC0; DO = 1; WM = 1; OC0 = 0; (16b)DATA = TD;
0x13	TI_OC1	Vypustit počet přetečení časovače 1	TD = OC1; DO = 1; WM = 1; OC1 = 0; (16b)DATA = TD;
0x14	TI_OC2	Vypustit počet přetečení časovače 2	TD = OC2; DO = 1; WM = 1; OC2 = 0; (16b)DATA = TD;
0x15	TI_OC3	Vypustit počet přetečení časovače 3	TD = OC3; DO = 1; WM = 1; OC3 = 0; (16b)DATA = TD;
0x16	TI_OV0	Vypustit hodnotu časovače 0	TD = T0; DO = 1; WM = 1; (16b)DATA = TD;
0x17	TI_OV1	Vypustit hodnotu časovače 1	TD = T1; DO = 1; WM = 1; (16b)DATA = TD;
0x18	TI_OV2	Vypustit hodnotu časovače 2	TD = T2; DO = 1; WM = 1; (16b)DATA = TD;
0x19	TI_OV3	Vypustit hodnotu časovače 3	TD = T3; DO = 1; WM = 1; (16b)DATA = TD;
0x1A	TI_ORT	Vypustit počet uběhlých milisekund	TD = RT; DO = 1; WM = 0; DATA = TD;
0x1B ...		Bez akce	

0x1F			

PROGRAMOVÁNÍ

ATTOASSEMBLER (ATTOASM)

K vytvoření attokódu je použit jazyk attoassembler, zkráceně attoASM a poté přeložen attoassemblerem (překladačem), čímž vznikne strojový kód (attokód), který může být spuštěn jádrem attoWPU.

CELÁ ČÍSLA

Pro specifikaci celých čísel (integer) mohou být použity čtyři číselné soustavy: binární, osmičková, desítková a hexadecimální. Soustava je určena dle symbolu za číslem: B pro binární, O pro osmičkovou, D pro desítkovou (volitelné, neboť je desítková výchozí) a H pro hexadecimální. Je také možno před číslo dát znaménko mínus pro zápornou hodnotu.

Celé číslo je 32bitová hodnota, jestli se jedná o signed integer nebo unsigned integer je určeno automaticky: pro kladné hodnoty menší či rovny kladnému maximu 32b signed integeru (2 147 483 647) není třeba určovat typ, neboť je v obou případech stejný. Pro hodnoty větší než tato hodnota je číslo automaticky uloženo jako unsigned integer, pro záporné hodnoty je vždy použit signed integer. Pro záporné hodnoty je použit dvojkový doplněk.

Celá čísla se používají pro specifikaci startovního bitu a k převodu číselné hodnoty na sérii attoinstrukcí (více níže). Jestliže hexadecimální číslo začíná symboly A až F, musí být před číslo připsána nula, jinak bude považováno za symbol. Je také množné použít jednoduchou matematiku použitím symbolů plus a mínus, čímž dojde k sečtení či odečtení několika čísel, čímž se vyprodukuje jedno číslo, které je zpracováno, jako by bylo v kódu zapsáno přímo.

Syntax:

```
<number>[base]
```

Příklad:

```
30923           // celé číslo (signed i unsigned)
-2440           // celé číslo v osmičkové soustavě
0C5809H         // celé číslo v hexadecimální soustavě
-1010111111B    // celé číslo v binární soustavě jako signed int
```

```
3879330290      // unsigned int
309H+1101B-VAL  // jednoduchá matematika
```

Čísla s plovoucí řádovou čárkou

Je také možné specifikovat čísla s plovoucí řádovou čárkou s jednoduchou přesností (typ float - 32 bitů), avšak je potřeba počítat s limitacemi. Protože sčítání a odečítání funguje pouze pro hodnoty typu celé číslo, nelze reálná čísla používat ve výrazech. Proto je syntaxe pro definici reálného čísla jiná, čímž se zabrání použití těchto čísel ve výrazech. Reálné číslo musí začínat symbolem &, vše za tímto symbolem je zpracováno jako reálné číslo, až do nalezení první mezery či nalezení jiného znaku.

Pokud je reálné číslo specifikováno samostatně, je v daném místě uloženo do attokódu jako přímá 32bitová hodnota, podobně jako u specifikace libovolných dat, může být také specifikováno za číslem bitu, kdy je automaticky převedeno na 32 attoinstrukcí odpovídajícím binární hodnotě daného čísla.

Příklad:

```
&2.222388       // uloží binární formu přímo do attokódu
DATA &3.141592  // převede na adekvátní sekvenci attoinstrukcí
```

ASCII znaky

Uvedením jednoho znaku v jednoduchých závorkách je možné získat číselnou hodnotu znaku dle ASCII tabulky, tato hodnota se chová zcela stejně jako celé číslo, může být použita i ve výrazech. Ačkoli jsou ASCII kódy obvykle 8bitové, jsou rozšířeny na 32 bitů jako ostatní celá čísla: bity navíc jsou převedeny na nuly.

Příklad:

```
DATA+23 ['f', 8] // zapsat kód znaku f na datovou sběrnici
```

Attoinstrukce

Attokód je vytvořen použitím vždy jedné ze čtyř (většinou tří) attoinstrukcí v kombinaci s číslem prvního bitu od kterého se aplikují. Každá skupina attoinstrukcí tedy začíná celým číslem od 0 do 63, které specifikuje první bit na sběrnici, toto číslo je následováno jedním nebo více symboly představujícími attoinstrukce. Je-li použit více jak jeden symbol, každý další symbol platí pro následující bit od startovního. Toto umožňuje specifikovat sekvenci souvislých attoinstrukcí bez nutnosti pro každou specifikovat startovní bit. Pro zjednodušení jsou předdefinovány symboly pro startovní bity všech čtyř logických sběrnicí.

Předdefinované symboly jsou následující:

```
ADDR = 0   začátek adresové sběrnice
CTRL = 8   začátek řídící sběrnice
DATA = 16  začátek datové sběrnice
AJMP = 48  začátek Quick aJump sběrnice
```

Attoinstrukce jsou reprezentovány následujícími symboly:

```
0     resetovat
1     nastavit
!     invertovat
|     zastavit
-     přeskočit (pseudoinstrukce)
```

Také je možné přičíst či odečíst číslo od startovního bitu pro posunutí startovního bitu relativně ke startu dané logické sběrnice, jedná se v podstatě o jednoduché matematické výrazy s celými čísly.

Kupříkladu následující kód změní třetí, čtvrtý a pátý bit řídící sběrnice na nulu, šestý pak na jedničku.

```
CTRL+2 0001
```

Pseudoinstrukce "-" umožňuje programátorovi zapsat sekvenci attoinstrukcí najednou, kde se jeden nebo více bitů přeskočí, bez nutnosti zapisovat dva výrazy. Kupříkladu:

```
DATA 110-11001
```

Je ekvivalentem.

```
DATA 110
DATA+4 1101
```

SESKUPOVÁNÍ ATTOINSTRUKCÍ

Pro zkrácení zápisu skupiny stejných attoinstrukcí lze specifikovat počet opakování dané attoinstrukce v závorce okamžitě za danou attoinstrukcí. Překladač automaticky vygeneruje specifikovaný počet samostatných attoinstrukcí. Předchozí příklad tedy může být zapsán takto:

```
CTRL+2 0(3)1
```

Dále je možné opakovat i celou skupinu attoinstrukcí zapsáním počtu opakování do závorky ihned za startovní bit. Toto je stejné jako zkopírování stejné skupiny attoisntrukcí v kódu několikrát za sebou.

```
CTRL+7(2) !
```

Je zkráceným zápisem:

```
CTRL+7!
CTRL+7!
```

Syntaxe attoinstrukcí

Obecná forma zápisu attoinstrukcí je následující (části v hranatých závorkách jsou volitelné):

```
<startovní bit> [(<opakování skupiny>)] <instrukce 1>[(<počet opakování instrukce>)]  … <instrukce n>[(<počet opakování instrukce >)]
```

Skupiny instrukcí jsou odděleny mezerami, každá nová specifikace startovního bitu je považována za novou skupiny attoinstrukcí, takže je možné zapsat několik skupin na jeden řádek, avšak pro přehlednost je doporučeno oddělit skupiny odřádkováním.

Startovní bit může být jakékoli číslo od 0 do 63, je také možné použít předdefinované symboly uvedené výše, stejně jako jednoduché matematické operace. Však je, aby výsledné číslo nepřekročilo limit. Pokud je například použito číslo 63, může být následováno pouze jednou instrukcí, neboť druhá by odpovídala neexistujícímu bitu 64, což způsobí chybu při překladu.

Komentáře

Vzhledem k povaze attoASM je důrazně doporučeno kód bohatě komentovat, aby byl jednodušeji pochopitelný. AttoASM používá komentáře ve stylu jazyka C. Symboly // indikují začátek jednořádkového komentáře, kdy je ignorováno vše za těmito znaky do konce řádku. Symboly /* a */ indikují začátek a konec víceřádkového komentáře, kdy je ignorováno vše mezi těmito symboly.

```
DATA 0(7)1 // naadresovat paměť attokódu
/* Následující kód spustí předem připravený řídící kód,
který přikazuje zastavení výstupu dat z dané jednotky, takže
datová sběrnice může být použita pro komunikaci s ostatními
zařízeními */
CTRL(2) !
```

Převod celých čísel na attoinstrukce

Programátor může převést 32bitová celá čísla na sérii attoinstrukcí, například může zapsat celé číslo v dekadické soustavě a nechat překladač vytvořit adekvátní sérii attoinstrukcí. Samozřejmě je možné použít i čísla v binární, osmičkové a hexadecimální soustavě, či ASCII znaky, které jsou také považovány za celá čísla.

K převodu musí dané číslo (či výraz) uzavřít do hranatých závorek, volitelně může specifikovat kolik attoinstrukcí chce vytvořit, přičemž výchozí hodnota je zároveň maximum: 32 attoinstrukcí. Binární reprezentace celých čísel vychází ze způsobu interpretace celých čísel dle jejich velikosti: zdali se jedná o signed či unsigned integer je určeno automaticky (viz Celá čísla).

Počet vygenerovaných attoinstrukcí je rovněž zapsán ve hranatých závorkách za číslem k převedení, oddělený čárkou. Za další čárkou je také možné volitelně specifikovat, kolik bitů bude přeskočeno. Počet převedených a přeskočených bitů se počítá od nejméně významného bitu.

Syntaxe (části v hranatých závorkách jsou volitelné):

```
[<číslo, či výraz> (,<počet bitů k převedení>,<počet bitů k přeskočení>)]
```

Examples:

```
DATA [15550]          // zapsat číslo 15550 na datovou sběrnici
ADDR [0CH,8,0]        // naadresovat jednotku na adrese 0CH
```

NÁVĚŠTÍ

Jestliže programátor potřebuje vytvořit skok na určitou lokaci v attokódu, nebo zjistit adresu určitého místa v kódu, může využít návěští. Návěští je vytvořeno zapsáním jména návěští následovaném dvojtečkou, specifikace návěští musí být oddělena mezerami. Jakmile je specifikováno, může být použito kdekoliv v kódu jako běžný symbol, který je při překladu nahrazen číslem vyjadřujícím adresu v attokódu odpovídající místu definice návěští. Adresa sestává z 20 bitů, je však doplněna nulami na 32 bitů a je ji možné převést na sérii attoisntrukcí. Název návěští musí začínat písmenem či podtržítkem a může obsahovat pouze písmena, číslice a podtržítko. Návěští může být definováno pouze jednou, počet použití je libovolný.

Příklad nekonečné smyčky:

```
ADDR 0(8)         // naadresovat aPC
CTRL 0(6)01       // připravit řídící kód pro zápis nové adresy
DATA+11 [Somelabel, 20]     // zapsat adresu návěští na datovou sběrnici
Somelabel:
CTRL+7(2) // zapsat adresu do aPC
```

Návěští mají také další účel: při překladu zdrojového kódu je vygenerován také speciální textový soubor, obsahující seznam všech návěští a příslušných adres v attokódu, což může pomoci při tvorbě vlastních instrukcí.

Specifikace libovolných dat - datové bloky

Do attokódu je také možné zahrnout libovolná binární data, která budou v nezměněné formě uložena do attokódu, ačkoliv nepředstavují žádné smysluplné attoinstrukce. Neboť je paměť attokódu přístupná pro čtení, může programátor tyto data libovolně využít za běhu programu. Datové bloky jsou vždy zarovnány na celé bajty, jinak by došlo k problému u následujících attoinstrukcí, které by začínaly například v polovině bajtu a končily v polovině následujícího bajtu. Datové bloky mohou být specifikovány pomocí hexadecimální soustavy, či jako ASCII řetězce.

Hexadecimální specifikace

Datový blok specifikovaný pomocí hexadecimální soustavy začíná symbolem $, za kterým následuje libovolný počet bajtů zapsaných v hexadecimální formě. Datový blok je ukončen mezerou. Bajty jsou do attokódu uloženy v pořadí, ve kterém jsou přečteny. Jestliže datový blok končí v polovině bajtu, je poslední bajt doplněn o čtyři nuly.

```
$00A3                  // dva bajty dat
$1E892A8C881F DATA+2 0010  // datový blok následovaný
attoinstrukcí
```

ASCII řetězec

Druhá forma datového bloku je ASCII řetězec, kde je každý znak převeden na jeden bajt odpovídající příslušené hodnotě dle ASCII tabulky. Podporovány jsou pouze základní únikové sekvence a řetězec není ukončen nulou: programátor musí dle potřeby nulu uvést sám. Znaky musí být uzavřeny ve dvojitých uvozovkách a musí být od ostatního kódu odděleny mezerou.

```
DATA 0(31)1 “Test string”        // řetězec následující skupinu
attoinstrukcí
“This is a longer string.\nTerminated with zero manually” 0x00
// řetězec ukončený nulou
```

Symbol	Význam
\n	Nový řádek
\”	Dvojité uvozovky
\t	Horizontální mezera
\0	Null znak
****	Zpětné lomítko
\f	Form feed
\r	Carriage return
\b	Backspace

Návěští

Je také možné uložit 32bitovou hodnotu návěští přímo do attokódu pro různé účely, kupříkladu pro vytvoření pole ukazatelů na důležité bloky kódu. Převod se provede uvedením jména návěští v závorkách za symbolem $.

Syntaxe

```
$(<jmenonavesti>)
```

Příklad

```
<0>
$(NAVESTI) // toto zapíše data typu integer s hodnotou 8 do
attokódu
DATA [8034]
NAVESTI:
…
```

DEFINICE SYMBOLŮ

Pro definici libovolného symbolu, což je v podstatě zástupné jméno pro libovolnou část zdrojového kódu o libovolné velikosti jsou použity složené závorky. Programátor nejprve specifikuje unikátní jméno symbolu, následované otevírací složenou závorkou. Poté může zapsat attoASM kód o libovolné délce, obsahující libovolné platné attoASM výrazy. Poté zapíše uzavírací složenou závorku, čímž ukončí definici symbolu. Jméno symbolu musí začínat písmenem a může obsahovat písmena, číslice a podtržítko.

Jakmile je symbol definován, může být použit kdekoliv v kódu, přičemž při překladu bude nahrazen kódem uvedeným ve složených závorkách, který bude zpracován jako obvykle. Tímto lze zjednodušit psaní opakujících se částí kódu, nebo jako kód v symbolu použít kupříkladu jen číslo, které bude použito jako argument v jiných výrazech. Není však běžně možné definovat symbol uvnitř symbolu: došlo by k redefinici symbolu, pokud by byl symbol použit více než jednou.

Syntaxe:

```
<symbol name> { <Kód> }
```

Příklad:

```
EXECUTE { CTRL+7(2) ! }     // definice celé skupiny attoinstrukcí
EXE { EXECUTE }       // je možné použít symbol uvnitř symbolu
Default { [440, 16, 0] }    // definovat pouze attoinstrukce
TMP_to_aPC {                // defining definice celého bloku kódu
ADDR [0x02, 8, 0]
CTRL [0x04, 7, 0]
EXE
ADDR [0x00, 8, 0]
```

```
CTRL [0x01, 7, 0]
EXE }

DATA+4 Default        // použití symbolu
```

REDEFINICE SYMBOLU

Běžně není možné symboly definovat více než jednou, toto chování je však možné cíleně obejít přidáním vykřičníku ihned za otevírací složenou závorku. Tímto se předchozí definice symbolu (pokud existuje) nahradí novou, která bude platná od místa její definice. Tímto lze kupříkladu předávat argumenty do symbolů. Návěští není možné redefinovat.

Syntaxe:

```
<jmeno symbolu> {! <kod> }
```

Příklad:

```
ARG { 0 }
NECO { DATA [ARG, 32] }

NECO  // zapíše 0 na datovou sběrnici
ARG {! 255 }     // redefinovat symbol
NECO  // zapíše 255 na datovou sběrnici
```

LOKÁLNÍ SYMBOLY A NÁVĚŠTÍ

Symboly a návěští by neměly být definovány uvnitř jiného symbolu, neboť by došlo k redefinici symbolu (vynucenou redefinici ovšem lze použít, ale ne pro návěští). To však znamená, že nelze vytvořit návěští pro tvorbu lokálních skoků či smyček uvnitř kódu definovaného v symbolu. Toto však lze vyřešit použitím lokálního symbolu či návěští.

Lokální symbol či návěští je vytvořen vložením znaku % do jména symbolu, což automaticky vygeneruje unikátní jméno pro každé použití symbolu, ve kterém je lokální symbol či návěští definován a použit. Znak % musí být součástí jména jak při definici, tak při použití symbolu či návěští.

Příklad:

```
SYMBOL
{
AJMP [LABEL%, 15]
LABEL%:
AJMP+15(2) !
}
```

ORGANIZACE ATTOKÓDU

Je možné specifikovat startovní adresu v attokódu, od které se budou následující attoinstrukce zapisovat. Tímto lze vytvořit mezery vyplněné nulami, obvykle jako

rezervaci místa pro specifické účely. Technicky je možné specifikovat adresu před aktuální pozicí, to však znamená, že nově přeložený kód přepíše již přeložený kód na dané lokaci, proto je při překladu v takové situaci vygenerováno varování. Startovní adresa vložena mezi symboly < a >. Adresou může být výraz, samostatné číslo, symbol, avšak nikoliv návěští.

Syntaxe (proměnná je mezi hranatými závorkami):

```
<[adresa]>
```

Příklad:

```
// kód
<800H>
// další kód…
```

CUSTOMIZABLE ASSEMBLER (CUSTASM)

Attokód je obvykle použit pouze pro specifikaci funkce procesoru, avšak ne k jednoduchému psaní samotných programů. Ačkoli je technicky možné programy psát přímo v attokódu, není to doporučeno. Pro větší programy také paměť attokódu neposkytuje dostatečný prostor, ačkoliv je tento faktor závislý na dané implementaci.

Programátoři by proto měli samotné programy ukládat do paměti programu a dat (skrze paměťový řadič A), přičemž attokód bude sloužit ke zpracování instrukcí v této paměti. Pro vytvoření programu pro paměť programu a dat potřebuje programátor jazyk symbolických adres, který mu umožní nadefinovat vlastní formát strojového kódu.

Pro tento účel je nabízen speciální jazyk: customizable assembly (přizpůsobitelný jazyk symbolických adres), zkráceně custASM. Ten umožňuje programátorovi specifikovat jeho vlastní instrukce spolu s jejich opkódy a rozvržením argumentů. Překladač jazyka custASM poté tento kód přeloží na strojový kód použitím definicí instrukcí poskytnutých programátorem. Tento program je poté načten do paměti programu a dat a spuštěn attokódem. Jak přesně ovšem programátor využije attoASM spolu s custASM záleží čistě na jeho záměru. Kupříkladu se může rozhodnout paměť programu a dat zcela ignorovat a kupříkladu zpracovávat instrukce vložené pomocí některé vstupní jednotky. Navíc je možné použít i jiný existující překladač pro vytvoření programu pro paměť programu a dat, pokud k danému strojovému kódu programátor napíše adekvátní attokód. V tomto ohledu neexistuje žádný limitující faktor a jazyk custASM je při vývoji pro AttoWPU zcela volitelný, avšak jeho použití je doporučeno, neboť poskytuje rychlý způsob, jak vytvořit vlastní instrukční sadu bez nutnosti programovat svůj vlastní překladač.

Přizpůsobitelný jazyk symbolických adres poskytuje programátorovi způsob, jak nadefinovat své vlastní instrukce definicí otisku instrukce a použitím speciálních znaků pro určení, kam budou vloženy argumenty. Překladač se poté pokusí spojit použitou instrukci v kódu s její definicí a vygenerovat příslušný strojový kód. Pokud není nalezena příslušná definice, je vypsána chyba a překlad je ukončen.

DATOVÁ JEDNOTKA

Binární data jsou použita prakticky všude v jazyku custASM: pro definici opkódů, argumentů, či libovolných binárních dat. Existuje několik způsobů jak specifikovat data: jako hodnotu typu integer o libovolné velikosti, jako reálné číslo s plovoucí řádovou čárkou a jako ASCII znaky či řetězce. Celá čísla mohou být specifikována v několika číselných soustavách.

Datová jednotka je jeden kousek dat, obvykle 32 bitů velký, který je nejčastěji předán jako argument instrukcím.

Celé číslo (integer)

Jakákoli číselná hodnota bez řádové tečky je považována za celé číslo, s libovolnou velikostí v bitech, která je specifikována programátorem. Pokud není velikost specifikována, je použita výchozí: 32 bitů. Záporné hodnoty jsou automaticky uloženy jako signed integer, hodnoty větší jak maximum pro signed integer pro danou šířku jsou pak automaticky uloženy jako unsigned integer. Není tedy třeba o tomto rozhodovat manuálně, protože ve své čisté binární formě nezáleží, o jaký datový typ se jedná: záleží na programátorovi, jak jej interpretuje. Čísla jsou defaultně považována za dekadická, avšak je možné specifikovat jinou číselnou soustavu přidáním symbolu za číslo:

B	binární	(11001101B)
O	osmičková	(13673O)
D	dekadická	(923950D)
H	hexadecimální	(0FF39A8CH)

Veškeré hodnoty jsou ve výchozím stavu považovány za 32bitové, avšak toto lze změnit přidáním čísla za písmeno specifikující číselnou soustavu. Toto číslo určuje, kolik bitů bude použito pro reprezentaci daného čísla. Také je možné místo počtu bitů uvést znak "x", který překladači říká, aby sám zvolil potřebnou velikost dle hodnoty daného čísla. Při počítání počtu vyžadovaných bitů se počítají i nuly na začátku čísla, takže je možné v kombinaci s použitím hexadecimální soustavy uložit libovolné množství libovolných binárních dat. Kompletní syntaxe pro uložení celého čísla je následující:

```
<číslo>[<soustava>[<počet bitů>]]
```

Příklady:

```
11001000B8              // 8b integer v binární soustavě
35601O                  // 32b integer v osmičkové soustavě
32D8                    // 8b integer v dekadické soustavě
083AE09C933H64          // 64b integer v hexadecimální soustavě
2085442                 // 32b integer v dekadické soustavě
0000FD380B838A0245CAF0Hx   // data o libovolné velikosti v
hexadecimální soustavě
```

Reálné číslo

V programu je také možné specifikovat reálná čísla s plovoucí řádovou čárkou, která ve výchozím stavu používají jednoduchou přesnost vyžadující 32 bitů. Je však možné použít dvojitou přesnost přidáním znaku D za dané číslo. Reálná čísla mohou být zapsána pouze v desítkové soustavě a musí obsahovat řádovou tečku. Syntaxe je proto velmi jednoduchá:

```
<číslo s řádovou tečkou>[D]
```

Příklady:

```
438.429                 // 32bit - jednoduchá přesnost
12.0941229955093D       // 64bit - dvojitá přesnost
```

Znaky a řetězce

Jakýkoliv ASCII znak je možné vložit mezi dvojité uvozovky, hodnota jednoho znaku je vždy 8bitová. Pouze nižších 128 ASCII znaků je oficiálně podporováno, vyšší znaky specifické pro kódování sice budou také zakódovány přímým přečtením jejich hodnoty v textovém souboru. Mezi dvojité uvozovky je také možno vložit více než jeden znak, čímž se vytvoří řetězec. Řetězce nejsou automaticky ukončeny nulou, programátor musí tento znak přidat sám, potřebuje-li jej . Je-li řetězec předán instrukci, bude automaticky ořezán na maximální velikost argumentu instrukce, počínaje prvním znakem. Pokud v řetězci naopak není dostatek znaků na vyplnění celého argumentu, je volné místo za řetězcem vyplněno nulami, na rozdíl od číselných hodnot, kde je nulami vyplněno místo před daným číslem.

Podporovány jsou i základní únikové sekvence:

Symbol	Význam
\n	Nový řádek
\”	Dvojitá závorka
\t	Horizontální tab
\0	Null znak

****	Zpětné lomítko
\f	Form feed
\r	Carriage return
\b	Backspace

Syntaxe:

```
“<jeden nebo více znaků>”
```

Příklady:

```
“f”
“This string is not fluffy.”
“Neither this one, but it's at least zero terminated\0”
```

Uložení libovolných binárních dat

Pokud programátor potřebuje do strojového kódu uložit binární data bez úprav, použije k tomuto datové jednotky. V podstatě při každém zapsání datové jednotky samy o sobě, mimo jakýkoli jiný výraz, je binární forma dané jednotky uložena přímo do strojového kódu na místě její definice. Hodnoty však dle nastavení (viz níže) nemusí být zarovnány na celé bajty, což může způsobit problémy.

JEDNODUCHÉ VÝRAZY

Číselné datové jednotky je možné použít ve výrazech, výsledkem je nová datová jednotka. Pokud se ve výrazu vyskytuje alespoň jedno reálné číslo, bude výsledkem datová jednotka typu reálné číslo. Jinak je výsledkem celé číslo, jehož datová šířka odpovídá největší datové šířce u datových jednotek použitých ve výrazu. Jestliže je u reálných čísel alespoň jedno s dvojitou přesností, bude výsledkem datová jednotka s dvojitou přesností, v opačném případě datová jednotka s jednoduchou přesností.

Programátor může využít sčítání, odečítání, násobení a dělení kdekoliv v programu, kde je platné zapsání datových jednotek, ve výrazech je také možné použít symboly a návěští za předpokladu, že obsahují číselnou datovou jednotku. Pořadí operací lze změnit pomocí závorek.

Příklady:

```
JMP loop-2
ADD 2*123
FADD 2.0/3 // výsledkem bude datová jednotka typu reálné číslo
```

DEFINICE INSTRUKCÍ A JEJICH POUŽITÍ

Instrukce jsou definovány použitím klíčových slov **def** a **as** dle následující syntaxe:

```
def <jméno instrukce> [<otisk argumentů>] as <rozložení
strojového kódu>
```

Jméno instrukce může obsahovat jakékoli písmena, číslice, podtržítko a mezeru. Jako jméno instrukce však není možné použít slovo "as", neboť je rezervováno. Pouze mezery mezi slovy jsou součástí jména instrukce, takže jakékoli mezery za klíčovým slovem "def" stejně jako mezery mezi posledním slovem a klíčovým slovem "as" jsou ignorovány.

Otisk argumentů může obsahovat libovolný počet argumentů, včetně žádného. Argument v podstatě rezervuje místo pro proměnnou hodnotu, která bude do strojového kódu vložena při překladu dle konkrétní hodnoty předané instrukci při jejím použití. Pro specifikaci jednoho argumentu v otisku argumentů je použita následující syntaxe:

```
{<číslo argumentu>:<velikost>[:<přepis hodnoty>]}
```

Strojový kód odpovídající instrukci se skládá z libovolných binárních dat, které zahrnují argumenty, jejichž hodnota závisí na hodnotě předané instrukci při jejím použití. Pro vložení hodnoty argumentu do strojového kódu v definici rozložení strojového kódu je použita následující syntaxe:

```
{<číslo argumentu>:<počáteční bit>:<koncový bit>}
```

Při definici rozložení strojového kódu je důležité číslo argumentu, neboť umožňuje programátorovi změnit pořadí argumentů ve strojovém kódu oproti zdrojovému kódu. Může také vybrat pouze určité bity z dané hodnoty a kupříkladu argument rozdělit na více částí: například u 8bitového argumentu vložit první bit před opkód instrukce, zbylých 7 pak za opkód.

Samotný opkód lze specifikovat použitím libovolné datové jednotky, která může být o libovolné šířce, překladač ji v závislosti na nastavení (viz níže) nemusí zarovnat na celé bajty, takže lze vytvořit i atypické velikosti, kupříkladu 13bitové opkódy. Programátor může také v definici otisku argumentů použít různé symboly, které se stanou součástí tohoto otisku a při použití instrukce musí být na stejných lokacích přítomny, jinak by se jednalo o jinou instrukci. Argumenty musí být vždy nějak odděleny, kupříkladu mezerou, neboť samotné datové jednotky musí být při definici také odděleny. Je také možné přetěžovat instrukce: vytvořit instrukce se stejným otiskem, ale různými velikostmi argumentů, jako demonstruje následující příklad:

```
def ADD {0:32}, {1:32}      as 2AH{1:32}{0:32}
def ADD {0:64}, {1:8}       as 2CH{1:8}{0:64}
def ADD {0:32}, #{1:32}     as 2BH{1:32}{0:32}
def ADD #{0:32}, {1:32}     as 2BH{0:32}{1:32}
def INC A                   as 35H
```

Při každém použití instrukce se překladač pokusí najít souhlasnou definici instrukce pro vytvoření příslušného strojového kódu. Jestliže instrukce vyžaduje argumenty, musí být specifikovány také použitím datových jednotek. Pokud je datová jednotka větší jak argument, jsou bity navíc ořezány, pokud je naopak menší, je volné místo vyplněno nulami.

Překladač se vždy snaží najít nejpřesnější definici, což znamená, že u dvou podobných definicí se stejným otiskem, ale rozdílnými velikostmi argumentů, kupříkladu instrukce s 32bitovým argumentem a shodné instrukce s 64bitovým argumentem překladač pro datové jednotky větší jak 32bitů použije 64bitovou verzi. Obecně se překladač snaží vždy použít variantu, která zachová co největší množství (ideálně všechna) předaných dat, ale zároveň není větší než je potřeba: snaží se, aby bylo ořezáno co nejméně bitů a zároveň, aby výsledný strojový kód byl co nejmenší. Programátor tedy může použít instrukce definované výše následovně:

```
ADD 40H, 12AA56H
ADD 255, #0FFFFFFH
ADD #0EEEEEE, 255
INC
ADD “This is a test”, 8
```

Při překladu je vygenerován následující strojový kód (zobrazen v hexadecimálním formátu):

```
2A 0012AA56 00000040
2B 00FFFFFF 000000FF
2B 00EEEEEE 000000FF
35
2C 08 5468697320697320
```

KOMENTÁŘE

custASM používá komentáře ve stylu jazyka C/C++. // indikuje jednořádkový komentář, symboly /* a */ pak víceřádkový komentář. Veškerý text označený jako komentář je ignorován, pokud se nachází mimo výraz. Je-li kupříkladu komentář zapsán uvnitř řetězce, je považován za znaky řetězce.

Syntaxe:

```
// jednořádkový
/* víceřádkový
komentář */
```

SYMBOLY

CustASM umožňuje také definici různých symbolů přiřazením libovolné části kódu určitému jménu, přičemž tuto část lze opakovaně použít. Na rozdíl od jiných jazyků symbol obsahuje textová data, která jsou vložena na místě použití symbolu. Tímto lze vytvořit nejenom symboly, které ukládají jednoduché číselné hodnoty, ale i větší kousky kódu.

Symbol je definován použitím klíčového slova **equ** následovaném textovými daty ve složených závorkách. Uvnitř závorek je možné specifikovat libovolný platný custASM kód, pokud bude platný na místě použití symbolu. Symbol je použit vložením jména symbolu kdekoliv v kódu, avšak po definici symbolu. Jméno symbolu musí začínat písmenem a může obsahovat pouze písmena, číslice a podtržítko.

Syntaxe:

```
<jméno symbolu> equ { <kód> }
```

Příklady:

```
Value equ { 34C8H }   // symbol obsahuje pouze číselnou hodnotu
Kód equ {        // symbol obsahuje sekvenci instrukcí
 ADD 30, 80H
 INC
 INC
}
```

NÁVĚŠTÍ

Návěští je speciální variantou symbolu, která je definována zapsáním jména symbolu následovaném dvojtečkou. Překladač automaticky tomuto symbolu přiřadí číselnou hodnotu, která obsahuje adresu následujícího bajtu či bitu (dle nastavení) ve strojovém kódu v místě definice návěští. Problém nastává při nestandardní velikosti opkódu, když může následující výraz začínat například uprostřed bajtu a bajtová adresa by tak byla nepřesná. Aktivováním bitových adres se však problém vyřeší, neboť adresa v podstatě udává pořadí následujícího bitu ve strojovém kódu, taková adresa je 35bitová.

Návěští jsou jediné symboly, které mohou být použity před jejich definicí: pokud je v kódu nalezen neznámý symbol, je automaticky považován za návěští a jeho zpracování je odloženo na později, kdy bude známa jeho hodnota. Jméno návěští musí začínat písmenem

a může obsahovat písmena, číslice a podtržítko. Návěští může být definováno pouze jednou a může být použito kdekoliv v kódu, jako běžný symbol.

Existuje také speciální návěští, které vždy obsahuje adresu právě zpracovávaného bloku (datový blok či instrukce). Toto návěští je specifikováno znakem $ a může být použito jako běžný symbol, kupříkladu pro instrukci skoku, která takto vytvoří skok na sebe samu, čímž zastaví program, využití také najde při výpočtu relativních adres skoku.

Syntaxe:

```
<jméno návěští>:
```

Příklad:

```
loop:
INC A
JMP loop

JMP $      // zastavit program
```

PŘEPIS HODNOTY ARGUMENTU

Ve výchozím stavu jsou binární data argumentu přímo odvozena od hodnoty předané instrukci při použití instrukce. Programátor však může chtít tuto hodnotu nějak upravit, proto je k dispozici způsob, jak tuto hodnotu změnit, a to pomocí volitelného třetího výrazu uvnitř složených závorek při definici argumentu u rozložení opkódu. Změna hodnoty je provedena zapsáním výrazu do této části definice argumentu, přičemž výraz může obsahovat různé operace (sčítání, odečítání, násobení, dělení a modulo) s okamžitými hodnotami a symboly (včetně návěští). Speciální symbol **val** obsahuje hodnotu předanou argumentu, programátor jej může použít kdekoliv ve výrazu dle potřeby, či úplně vynechat, použít může i speciální symbol $.

Příklad:

```
def SJMP {0:8:$-val} as 80H{0:8} // relativní skok
```

VLOŽENÍ SOUBORŮ

Pro lepší správu vývoje je vhodné větší projekty rozložit do více souborů a tyto soubory pak vložit do hlavního zdrojového kódu. Také je možné vložit obsah souboru v jeho binární formě přímo do strojového kódu. Jakmile překladač narazí na výraz include, je obsah daného souboru vložen do hlavního zdrojového kódu, jako by byl v něm přímo zapsán.

Syntaxe:

```
include(“<soubor>”)   // vložit zdrojový kód
binclude(“<soubor>”)  // vložit soubor binárně
```

Příklady:

```
include(“definice.casm”)
binclude(“obrazek.bmp”)
```

NASTAVENÍ PŘEKLADU

Proces překladu lze upravit pomocí speciálních nastavení, které mohou výrazně ovlivnit generování strojového kódu. Nastavení se mění pomocí pseudoinstrukce **__set** následované názvem nastvení a hodnotou. Kupříkladu je možné změnit bajtový mód na bitový mód.

Syntaxe:

```
__set <jméno nastavení> <hodnota>
```

Příklady:

```
__set BITADDRESS 1    // aktivovat bitové adresy
__set DATACHUNK_PADDING 1        // aktivovat zarovnávání
datových kousků na bajty
```

Seznam podporovaných nastavení a hodnot

Nastavení	Hodnota	Význam
BITADDRESS	0 def.	Návěští produkují 32bitovou adresu, odkazující na celé bajty
	1	Návěští produkují 35bitovou adresu, odkazující na konkrétní bity
DATACHUNK_PADDING	0	Datový blok není zarovnán na celý bajt
	1 def.	Datový blok je zarovnán na celý bajt doplněním nul
INSTRUCTION_PADDING	0	Opkód instrukce není zarovnáván na celé bajty
	1 def.	Opkód instruke je dle potřeby doplněn nulami, aby byl zarovnán na celý bajt
ATOMIC_SIZE	uint def. 8	Velikost datového elementu pro endianitu
LITTLE_ENDIAN	0 def.	Hodnoty jsou uloženy jako big endian
	1	Hodnoty jsou uloženy jako little endian
LOGFILE	string	Jméno a adresa souboru pro uložení logu
OUTPUT	string	Umožňuje specifikovat jméno výstupního souboru
LIBRARY	string	Cesta k adresáři obsahujícím knihovny

ZÁVĚR

Funkčnost konceptu procesoru a programovacího jazyka attoASM jsem ověřil na naprogramovaném překladači a simulátoru vytvořením a simulováním jednoduchých příkladů využívajících různé jednotky i naprogramováním složitějšího programu: hry Pong, která funguje dle předpokladů. Vývoj procesoru tedy směřuje správným směrem a navíc poskytuje zajímavé zkušenosti během programování, neboť tvorba této hry v jazyce attoASM podnítila myšlení o způsobu funkce a výhodách jiných programovacích jazyků a zároveň donutila vymyslet nové řešení některých programových konstruktů.

POUŽITÁ LITERATURA A SOFTWARE

Software pro vývoj:
[1] Microsoft Visual Studio C++ 2008 Express
[2] Qt SDK a Qt Creator
[3] Notepad++
Vytvoření loga:
[4] GIMP2
[5] Blender 2.6.2
Literatura:
[6] C++: The Complete Reference, third edition, Herbert Schildt
[7] Qt Dokumentace, http://doc.qt.nokia.com