NP1 - MCU matrioška

Úvod

Keď sa pred nejakou dobou sčerili tuzemské vody okolo tzv. NP (nepoužiteľné počítače*) projektom QR6 [1], inšpiroval ma tento počin k uvažovaniu o vlastnej koncepcii NP. Neašpiroval som o prekonanie QR6, pretože minimálne v jednom ohľade by to bolo zbytočné – a tým je minimálny počet inštrukcií – ale snažil som sa vytvoriť niečo, čo tu ešte nebolo, resp. ja som to nikde nevidel.
Hneď zo začiatku som zavrhol stavbu CPU z logických obvodov nízkej úrovne (hradlá, čítače, klopné obvody, atď.), pretože som na zdĺhavý návrh a nemenej zdĺhavé spájkovanie jednoducho nemal chuť. QR6-tka, podobne ako aj niekoľko „šialených“ projektov zozbieraných z webu dokazujú, že to možné je (koniec-koncov v počiatkoch počítačovej techniky pred príchodom prvých integrovaných CPU v sedemdesiatych rokoch ani vhodnejšie prostriedky na realizáciu počítačov neboli), ja som s týmto faktom uspokojil a nechal som túto možnosť vývoja bokom.
V súčasnej dobe je viacero lacných spôsobov, ako úlohu realizovať. Jedným z nich je implementácia do nejakého obvodu programovateľnej logiky – mimochodom je to ideovo prakticky to isté riešenie (ale technická realizácia je odlišná) ako predošlý prípad, pričom aj tieto riešenia sú dostupné na webe v pomerne dostatočnom množstve [2]. Je asi zrejmé, že som si tú svoju možnosť nechal nakoniec – áno, ide o implementáciu (emulovanie) počítača v počítači, resp. MCU v MCU. Projektov tohto typu je už na sieti pomenej (ak odčítame nespočetné množstvo emulátorov bežiacich na PC), ale niečo predsa len existuje (link). Rozhodol som sa pridať do tejto neveľkej zbierky svoj kúsok, pomenovaný NP1 (Nepoužiteľný Počítač číslo 1**).

Realizácia, časť prvá – ideový návrh

Najprv som si musel samozrejme určiť základné východiská pre návrh jadra NP1. Lákalo ma netradičné riešenie, ako napríklad to, ktoré ponúkal pred časom CPU CDP1802 [3] od firmy RCA (COSMAC – **CO**mplementary **S**ilicon **Met**a**l**-oxide **C**onductor, čiže iné pomenovanie pre CMOS), ktorý bol mimochodom jedným z prvých CMOS CPU na svete a preto bol hojne využívaný vo vojenských zariadeniach a v kozmickom výskume – napríklad vesmírna sonda Voyager má na sebe hneď po tri kusy tohto podareného kusu kremíku. V jeho prípade neexistoval klasický PC (program counter), X (index register), ale mal skupinu 16-tich univerzálnych 16-bitových registrov, z ktorých sa jeden zvolil ako PC (inštrukciou SEP) a jeden ako X (inštrukciou SEX). Tým pádom neexistovala obdoba inštrukcie JUMP a CALL, resp. RETURN (BRA, JSR, resp. RTS) tak, ako v prípade typu 8080 od Intelu (alebo 6800 od Motoroly).
Ukázalo sa, že emulovať niečo takéto by síce bolo možné, ale neumožňovalo by to takú rýchlosť emulácie, akú som si stanovil (milión emulovaných inštrukcií za sekundu), tak som si zvolil inú architektúru, ktorá má tiež netradičné rysy, a okrem iného aj minimálny počet inštrukcií.

Realizácia, časť druhá – jadro

Pod honosným názvom jadro počítača sa skrýva nepríliš obsiahla tabuľka inštrukcií a adries, ale je to módne slovo, tak som ho použil. NP1 je počítač Van Neumann-ovskej architektúry, teda počítač nerozlišuje pamať programu a pamäť dát, obe pamäte sú rovnakej šírky (16b).
V prvom rade by som vymenoval registre NP1 (ich detailný popis bude neskôr). NP1 obsahuje klasický programový čítač (program counter - PC), ukazovateľ zásobníka (stack pointer - SP), register príznakov (flags - F) a trojicu registrov A,B a C (ako ukážem neskôr, sú takmer zbytočné). Všimnime si chýbajúci register nepriameho adresovania.
Inštrukčný súbor je minimalistický a ťažkopádny, aby ladil s pomenovaním NP1. Všetky inštrukcie sa pre jednoduchosť vykonajú v rovnakom čase, ktorý je tvorený 37-mi pracovnými cyklami emulujúceho MCU (v prípade taktovania na 37MIPS je to 1MIPS emulovaných inštrukcií).
Obsahuje 16 inštrukcií, rozdelených do šiestich skupín.

V prvej skupine ide o inštrukcie presunu dát:

mvi (move immediate)
: vloží konštantnú hodnotu do daného pamäťového miesta

mov (move)
: presun dát z jedného pamäťového miesta na druhé

V druhej skupine ide o aritmetické a logické operácie na operandoch:

rol (rotate left)
: rotuje obsah daného pamäťového miesta o jeden bit doľava

ror (rotate right)
: rotuje obsah daného pamäťového miesta o jeden bit doprava

add (add)
: pričíta operand k obsahu pamäťového miesta

and (logical and)
: vykoná operáciu AND obsahu pamäťového miesta s operandom

ior (inclusive or)
: vykoná operáciu OR obsahu pamäťového miesta s operandom

xor (inclusive or)
: vykoná operáciu XOR obsahu pamäťového miesta s operandom

Potom nasleduje inštrukcia pre prácu s bitmi, jediný priamy spôsob podmieneného vetvenia programu.

bts (bit test)
: otestuje hodnotu pamäťového miesta na danom bite a v prípade zhody s danou polaritou skočí na miesto dané v inštrukcii

Operácie vetvenia programu

jmp (jump)
: vloží priamu hodnotu do PC – teda skok v programe

jsr (jump to subroutine)
: aktuálnu hodnotu PC vloží na vrchol zásobníka, potom vloží priamu hodnotu do PC, inkrementuje SP

rts (return from subroutine)
: vloží hodnotu z miesta adresovaného vrcholom SP do PC, dekrementuje SP

rti (return from interrupt)
: to isté ako rts, ale povolí prerušenia

Operácie na manipulovanie so zásobníkom

psh (push)
: vloží obsah pamäťového miesta na zásobník, inkrementuje SP

pop (pop)
: vloží hodnotu z vrcholu zásobníku do pamäťového miesta, dekrementuje SP

A nakoniec nastavenie offsetu

off (set offset)
: nastaví dve pamäťové miesta do funkcie offsetových registrov

Práve posledná menovaná inštrukcia a offsetové registre s ňou spojené sú zvláštnosťou NP1. Určuje, ktoré dva adresové miesta sa stanú ukazovateľom na hodnotu offsetu niektorých inštrukcií (v tabuľke inštrukcií sú operandy, na ktoré sa vzťahuje offset označené šikmým písmom). Tento offset sa potom pripočíta k hodnote (ktorá je adresou) obsiahnutou v inštrukcii.

Napríklad ak hodnota na adrese 0x1000 je 0x0000, na adrese 0x1002 je 0x0000, vykonanie inštrukcie

off 0x1000, 0x1002

spôsobí, že tieto dve hodnoty budú považované za offset pre inštrukcie (v tomto prípade nula), u ktorých je to relevantné. Napríklad inštrukcia

mov 0x0900, 0x0950

spôsobí skutočne presun dát z adresy 0x900 na 0x950 (pretože offset je nulový). Ale ak na adrese 0x1000 je hodnota 0x0010 a na adrese 0x1002 je 0x0012, táto istá inštrukcia spôsobí presun hodnôt z adresy 0x0910 na 0x0962 (spôsobené offset-om).
Tento mechanizmus umožňuje nepriamu adresáciu operandov, takisto ako aj odskoky na vypočítané adresy (computed jump), pretože sa vzťahuje aj na inštrukcie vetvenia programu = to umožňuje napríklad vytváranie tabuliek.

Realizácia, časť tretia – spájkovačku do ruky!

Vzhľadom k nemalým požiadavkam na software-ovú emuláciu MCU je treba použiť relatívne výkonný MCU. Ako vhodný kandidát sa ukázal byť niektorý z dsPIC MCU od firmy Microchip, Aby nebolo potrebné spomaľovať emuláciu prácou s externou pamäťou (a aby sa zároveň aj nekomplikoval HW), použil som typ s najväčšou dostupnou pamäťovou dotáciou v puzdre SDIP28 – dsPIC33FJ128GP802. Má na čipe 128kB FLASH, 16kB RAM, všetky štandardné „features“ ako IIC, SPI, UART, ale aj chuťovky ako CAN, 1,1Msps ADC, audio DAC, CRC generátor a mapovateľné piny periférií (u výrobcu za 5,88USD pri odbere 1ks). Žiaľ, väčšina z týchto periférií bude nevyužitá, pretože je treba len dostatočný výkon, pamäť a IIC zbernicu.
Hardware je tak jednoduchý, že z neho prakticky niet čo uberať. Obsahuje MCU, stavové LED, EEPROM 24C08 na bootovanie a „bižutériu“ k nim prislúchajúcu.
Dolná polovica portu B je využitá pre potreby NP1. RB0 a RB1 sú piny ICSP, piny RB2 až 4 sú vyhradené pre LED diódy indikujúce stav NP1 (viď errata), RB5 a RB6 sú piny rozhrania IIC pre bootovaciu EEPROM, RB7 je pin rozhodujúci o tom, či po resete začne vykonávanie programu z FLASH alebo sa nabootuje z EEPROM.

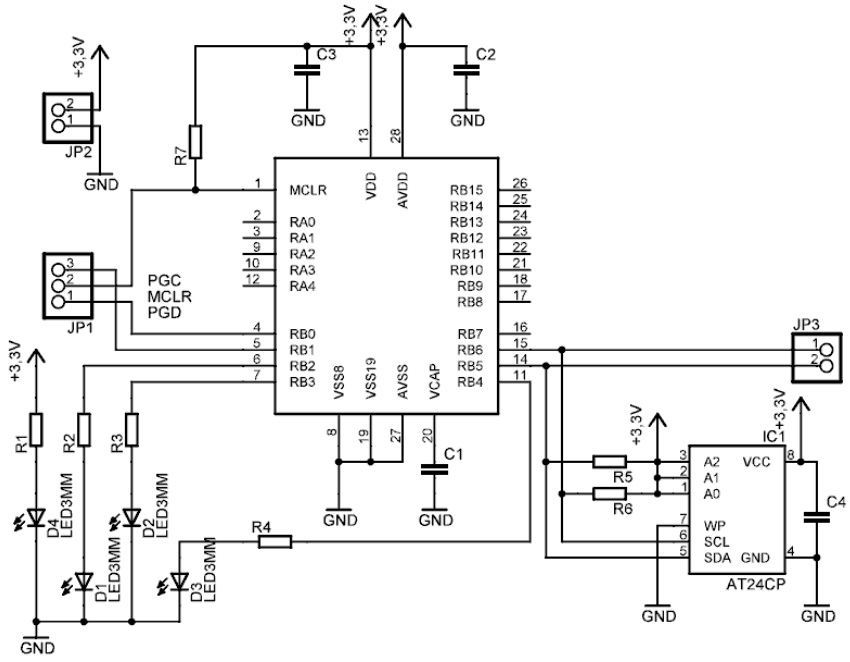
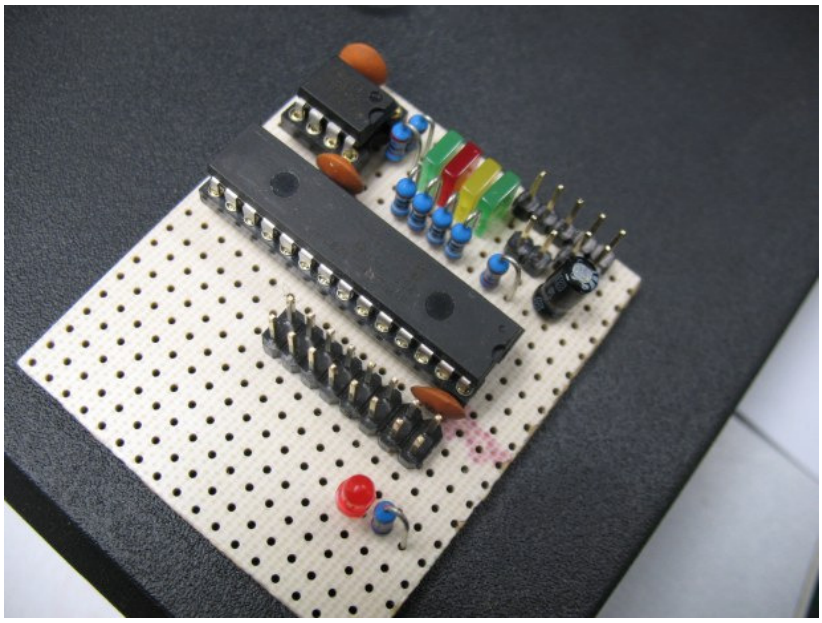


Schéma zapojenia NP1



Prototyp NP1

Realizácia, časť štvrtá – programovanie pre programovanie

Firmware emulujúceho MCU je (a musí byť) pomerne jednoduchý. Po resete sa inicializujú systémové hodiny, IIC a začne sa bootovanie z externej EEPROM – kopírovanie obsahu EEPROM do RAM od adresy 0x1100. Po tomto kroku nastane spustenie emulácie NP1 s PC nastaveným na adresu 0x1100.
Vlastná emulácia pozostáva z rozskokovej tabuľky, ktorá podľa čísla operačného kódu odovzdá riadenie príslušnej časti programu, na jej konci je opäť skok na začiatok emulačného cyklu. Pretože jednotlivé emulované inštrukcie vyžadujú nerovnaké časy na vykonanie, sú „dotiahnuté“ nop-mi na jednotný čas (podľa najdlhšej inštrukcie), aby sa zjednodušilo počítanie časovania programov NP1.
Vzhľadom k tomu, že použitý MCU umožňuje namapovať časť FLASH do dátovej RAM, je v hornej polovici pamäte FLASH – 32kB (viď obrázok rozloženia pamäte). Toto by umožnilo vytvorenie knižníc pre NP1, aby sa zjednodušil a skrátil program v RAM.

Hrubá mapa pamäte NP1:

memory map	
0x000 … 0x7FF	here are dragons (SFR)
0x800 … 0x47FF	RAM (16kB)
0x4800 … 0x7FFF	unimplemented
0x8000 … 0xFFFF	user FLASH

RAM je celá voľná, ale po resete sa obraz EEPROM ukladá od adresy 0x1100 (nič nebráni ho niekam „odpratať“), SP je po resete nastavený na 0x1000 (podobne sa dá presunúť). Poznamenal by som, že všetky adresy musia byť párne čísla. Registre A, B, C a F sú po resete vynulované. Jediný rozdiel medzi registrami A, B, C a akýmkoľvek iným registrom je to, že po resete sú vynulované. To sa dá s výhodou použiť pre nastavenie nulového počiatočného offsetu - ušetria sa dve inštrukcie nulovanie registrov. Inak nič nebráni tieto inštrukcie vykonať na ľubovoľnom miesta pamäte a na to miesto aj odkázať offset.

Registre NP1:

address	loc	default value
0x0800	A	0x0000
0x0802	B	0x0000
0x0804	C	0x0000
0x0806	F	0x0000
0x0808	PC	0x1100
0x080A	SP	8x1000

Register F obsahuje príznaky stavu emulovaného CPU. Jeho nultý bit je bitom C, indikujúcim prenos s aritmetických, resp. logických inštrukcií. Z bit označuje nulový výsledok inštrukcie u tej istej skupiny inštrukcií, bit I je maska prerušenia. Pokiaľ je v stave 1, tak je prerušenie akceptované, v stave 0 je ignorované.
V rozsahu adries 0x0000 až 0x07FF je možné nájsť SFR emulujúceho MCU, a pristupovať cez ne napríklad k portom, USART-u atď.

Prvé kroky

Pri ladení a overovaní činnosti NP1 som program ukladal priamo do FLASH, bootovanie z externej EEPROM som doplnil až neskôr.
Na jednoduché overenie funkcie NP1 (samozrejme po dôkladnej simulácii) som použil jednoduchý program – typické blikanie LED:

```
        off     850,852
        mvi     FFFF,854
        mvi     0,2C0
loop
        xor     854,2C4
        jmp     loop
```

V prvej inštrukcii sa nastaví offset na nulové hodnoty (na adresách 0x0850 a 0x0852 sú registre A a B, ktoré sú po resete nulované), potom sa do 0x0854 vloží hodnota 0xFFFF. Následne sa vynuluje register TRISB, čím sa nastavia piny portu B do funkcie výstupov. V štvrtom riadku je návestie, na ktoré odkazuje inštrukcia jmp hneď po invertovaní stavu portu B inštrukciou xor (xor 854,2C4 – to znamená, že hodnotou z 0x0854 sa XOR-ne hodnota na 0x02C4). Po ručnom preklade (môj prvý listing písaný ceruzkou na papier) som tieto hodnoty uložil na správne miesta vo FLASH, napálil program do dsPIC, resetol, LED svietila polovičným jasom a osciloskop ukázal podľa očakávania rýchly obdĺžnikový signál na pinoch portu B...

Cross-assembler a EEPROM

Ručné prekladanie programu a prepisovanie FLASH pri každej zmene programu NP1 je síce osviežením denného programu, ale po istej dobe začne unavovať, zvlášť ak si človek uvedomí, že je v jeho silách to urobiť jednoduchšie.
V prvom rade som pridal rozhranie s externou EEPROM. Do nej sa uloží program, ktorý si MCU potom len pri resete „vyzdvihne“. Na programovanie tejto EEPROM priamo v aplikácii som použil

PicKit2, ale je možné pravdepodobne použiť niektorý z plejády lacných samo-domo programátorov, ktoré zhusta podporujú aj IIC EEPROM.
Týmto sa veci zjednodušili, ale stále som musel „assemblovať“ ručne.
Preto som jedného pekného popoludnia na jedno sedenie naprogramoval veľmi primitívny NP1 cross-assembler pre PC. Je veľmi prostý, pretože aj moje znalosti PC programovania sú nepríliš rozsiahle. V zásade ma iba oslobodzuje od ručného počítania adries, sú nahradené návestiami, pričom pri volaní musí byť každé návestie označené podčiarovníkom... lebo som to lepšie nevedel urobiť.
Napríklad:

```
        ...
        jsr     _delay
        jmp     _loop

delay
        mvi     0,860
        ...
```

Je zrejmé, ako musí vyzerať volanie adresy z nejakého návestia.
Assembler pracuje iba s hexadecimlnymi číslami.
Assembler nepozná príkazový riadok (lebo sa mi ho nechcelo dotvárať) a po spustení hľadá súbor „source.asm“. Ak ho nájde, začne prekladať. Jeho výsledkom je súbor „binary.bin“. Vytvoril som si jednoduchý dávkový súbor (prípona .bat), ktorý spustí assembler, potom utilitu bin2hex.exe, ktorá prevedie binárny výsledok do intel-hex formátu. Tento je možné použiť pre program, ktorý ho dopraví na určené miesto – EEPROM NP1.
Program pre skutočné blikanie LED vyzerá nasledovne:

```
        org     1100
        mem     0

        off     850,852
        mvi     FFFF,854
        mvi     0,2C0
loop
        xor     854,2C4
        jsr     _delay
        jmp     _loop

delay
        mvi     0,860
        mvi     1,862
deloop
        jsr     _delay_in
        add     862,860
        bts     856,1,0,_deloop
        rts

delay_in
        mvi     FFF9,864
        mvi     1,866
deloop2
        add     866,864
        bts     856,1,0,_deloop2
        rts
```

A to isté s komentármi:

```
        org     1100
        mem     0
```

Dve pseudoinštrukcie určujúce, že program začína na adrese 0x1100, ale bude uložený v EEPROM od adresy 0.

```
        off     850,852
        mvi     FFFF,854
        mvi     0,2C0
loop
        xor     854,2C4
        jsr     _delay
        jmp     _loop
```

Tento kúsok som už vlastne komentoval v predošlej kapitole. Aby bolo blikanie LED viditeľné pre priemerný ľudský zrak, je treba ho o niekoľko rádov spomaliť - na to je pridaný odskok na podprogram *delay*.

```
delay
        mvi     0,860
        mvi     1,862
deloop
        jsr     _delay_in
        add     862,860
        bts     856,1,0,_deloop
        rts
```

To je spomínaný podprogram na vykonanie čakacej slučky. Využíva lokáciu 0x0860 ako počítadlo a 0x0862 je konštanta. Táto konštanta sa v kažkom kroku pripočítava k hodnote z 0x0860 (*add 862,860*) a potom sa zisťuje, či v registri 0x0856 (F) sa bit číslo 1 (Zero flag) rovná nule (táto podmienka označuje, že výsledok operácie bol nenulový). Ak áno, skočí na dané návestie.Vzhľadom k tomu, že sa počítanie začalo od nuly, je treba, aby sa jednotka pripočítala 65536-krát, kým bude nula znovu nulou. Dovtedy bude výsledok súčtu nenulový, bit 1 v F registri bude nulový a nastane skok. Po vykonaní daného počtu cyklov skok nenastane a vykonávanie bude pokračovať ďalej, na inštrukciu rts, čím sa ukončí podprogram čakania. V slučke je odskok na ďalší, vnorený podprogram, podobný ako ten predošlý:

```
delay_in
        mvi     FFF9,864
        mvi     1,866
deloop2
        add     866,864
        bts     856,1,0,_deloop2
        rts
```

Je to podobný program, iba jeho cyklus je skrátený – počítanie po jednotke začína od 0xFFF9 a pretečenie do hodnoty 0x0000 tým nastane rýchlejšie.

A čo ďalej...

No okrem povinnej jazdy každého MCU v podobe blikania LED-ky sa s tým dá robiť prakticky čokoľvek, pokiaľ je chuť na programovanie pomocou tých sparťanských prostriedkov, ktoré sú k dispozícii. Okrem toho, čo som zverejnil, mám rozpracovaný monitor, umožňujúci čítanie, modifikovanie RAM, ako aj spúštanie programu z pamäte NP1 pomocou RS232 rozhrania.

Všetky zdrojové súbory k zverejneným častiam projektu sú voľne k dispozícii, takže každý, komu sa veci nepáčia, ich môže zmieniť a zlepšiť.

FAQ

1. **Načo je to dobré?**
   Na nič. Je to jednoducho projekt na skrátenie voľnej chvíle, človek si môže skúsiť, ako vyzerá vývoj bez poriadnych vývojových nástrojov, preklad a ladenie programu na papieri s ceruzkou, tak ako to vyzeralo kedysi – pre niekoho pripomenutie starých časov, pre mňa exkurz do histórie.

2. **Nie je ten inštrukčný súbor trochu divný?**
   Je.

3. **Prečo sú niektoré popisy v angličtine?**
   Pretože pochádzajú z anglickej verzie článku o NP1, ktorú (neúspešne) plánujem už dlho zavesiť na web...

Zdroje:

[1] http://www.efton.sk/qr6/
[2] http://en.wikipedia.org/wiki/PicoBlaze
[3] http://en.wikipedia.org/wiki/CDP1802

Poznámky pod čiarou:
* Nič v zlom, ale je to naozaj tak.
** Existuje aj dvojka, pracujem na trojke, každý z týchto NP je kompletne iný.

Sumár inštrukcií

Opcode	Mnem.	Op1	Op2	Op3	Words	Flags
0	mvi	imm	dst16		3	
1	mov	*src16*	*dst16*		3	
2	rol	*mem16*			2	c,z
3	ror	*mem16*			2	c,z
4	add	*opr16*	*mem16*		3	c,z
5	and	*opr16*	*mem16*		3	c,z
6	ior	*opr16*	*mem16*		3	c,z
7	xor	*opr16*	*mem16*		3	c,z
8	bts	*mem16*	bit	*adr16*	4	
9	jmp	*adr16*			2	
10	jsr	*adr16*			2	
11	rts				1	
12	rti				1	
13	psh	mem16			2	
14	pop	mem16			2	
15	off	mem16	mem16		3	

Popis inštrukcií (Instruction set description)

mvi imm16, dst16
move 16-bit immediate value into location dst16

```
0000    0000    0000    0000
iiii    iiii    iiii    iiii
dddd    dddd    dddd    dddd
```

mov src16, dst16
move value from source location into location dst16

```
0000    0000    0000    0001
ssss    ssss    ssss    ssss
dddd    dddd    dddd    dddd
```

rol mem16
rotate left value from location mem16

```
0000    0000    0000    0010
mmmm    mmmm    mmmm    mmmm
```

ror mem16
rotate right value from location mem16

```
0000    0000    0000    0011
mmmm    mmmm    mmmm    mmmm
```

add opr16, mem16
add operand to location mem16, store into location mem16, operand remains unchanged

```
0000    0000    0000    0100
oooo    oooo    oooo    oooo
mmmm    mmmm    mmmm    mmmm
```

and opr16, mem16
logical and operand with location mem16, store into location mem16, operand remains unchanged

```
0000    0000    0000    0101
oooo    oooo    oooo    oooo
mmmm    mmmm    mmmm    mmmm
```

ior opr16, mem16
inclusive or operand with location mem16, store into location mem16, operand remains unchanged

```
0000    0000    0000    0110
oooo    oooo    oooo    oooo
mmmm    mmmm    mmmm    mmmm
```

xor opr16, mem16
exclusive or operand with location mem16, store into location mem16, operand remains unchanged

```
0000    0000    0000    0111
oooo    oooo    oooo    oooo
mmmm    mmmm    mmmm    mmmm
```

bts mem16, B, P, adr16
compare bit number B with single bit value P, jump to adr16 when equal

```
0000    0000    0000    1000
mmmm    mmmm    mmmm    mmmm
0000    000p    0000    bbbb
aaaa    aaaa    aaaa    aaaa
```

jmp adr16
jump to adr16 immediately

```
0000    0000    0000    1001
aaaa    aaaa    aaaa    aaaa
```

jsr adr16
store PC onto stack, increment SP, jump to adr16 immediately

```
0000    0000    0000    1010
aaaa    aaaa    aaaa    aaaa
```

rts
restore PC from stack, decrement PC

```
0000    0000    0000    1011
```

rti
restore PC from stack, decrement PC, set I

```
0000    0000    0000    1100
```

psh mem16
push value from location mem16 onto stack, increment SP

```
0000    0000    0000    1101
mmmm    mmmm    mmmm    mmmm
```

pop mem16
pop value from stack into location mem16, decrement SP

```
0000    0000    0000    1110
mmmm    mmmm    mmmm    mmmm
```

off memA16, memB16
set locations mem16A and mem!6B as offset locations

```
0000    0000    0000    1111
mmmm    mmmm    mmmm    mmmm    (memA)
mmmm    mmmm    mmmm    mmmm    (memB)
```

Errata

Nie sú implementované prerušenia (ale je zabudovaná štruktúra na ich doplnenie).
Časom pribudne aj toto.

Nie sú implementované stavové výstupy.
A asi ani nebudú. Vo fáze budovania HW som predpokladal, že by to mohlo byť obohatenie, ale nakoniec sa to ukázalo ako zbytočnosť. IO piny vyhranené na tento účel sú voľne dostupné.

MODE pin nie je implementovaný.
Ale bude, podobne ako prerušenia.