Beste leden van HCC!Forth en in Forth geïnteresseerde computeraars,

Zaterdag 10 december 2022 van 11:00 tot 15:00 is er weer een fysieke bijeenkomst op de Boslaan 1a naast de Zuiderkapel in Bilthoven.

Zie ook de HCC!Forth agenda-pagina op https://forth.hcc.nl/agenda

Programma

11.00 – Kort verslag van de werkgroepen door
Willem Ouwerkerk & Albert van der Horst

11.15 – PostIt Fixup, een systeem voor assemblers door
Albert van der Horst

12.30 – Pauze met Sinterklaas en Kerstliedjes

13.00 - Onderlinge Kennis-Uitwisseling (OKU)

15.00 – Sluiting

PostIt FixUp, een systeem voor assemblers

Wat is an assembler?
Een assembler is een hulpmiddel voor het maken van code woorden.
Wij voegen een woord toe dat twee getallen kan optellen.

```
CODE JAN HEX 58 C, 5B C, 01 C, D8 C, 50 C, NEXT, END-CODE
```

Dit is een Intel 86 code dat hetzelfde werkt als +
De hex codes zijn onhandig. We gebruiken liever geheugensteuntjes, met een latijns woord mnemoniek.
Dan wordt het

```
CODE +' popa, popb, addb, psha,  END-CODE
```

Het eerste hulp woord van de assembler is `RPI1` en we gebruiken het als volgt:

```
  58 1PI popa,
```

`1PI` is een definierend woord, te vergelijken met `CONSTANT`.
Na het uitvoeren van de bovenstaande code kunnen we zien dat `popa,` in de woordenlijst staat, klaar voor gebruik.

De uitleg gaat verder aan de hand van een eenvoudiger processor dan de Pentium namelijk de 8080. Die gebruikt enkele byte instructies, zodat we ze makkelijk kunnen laten zien in een overzicht.
Zie het schema op de volgende pagina.
We kunnen aan de titels aan de bovenkant en links aflezen wat de code is. In het hokje staat de mnemoniek. Merk op dat dezelfde mnemoniek in meerdere hokjes kan staan.
Voor het moment concentreren we ons op de unieke mnemonieken, met name RST0 RST1 links onder. Deze zijn in blauw afgedrukt.
Ik ga niet uitleggen wat de processor doet voor deze instructies. Daar hebben we het niet over!

We krijgen dus

```
HEX
C7 RP1 RST0,
CF RP1 RST1,
D7 RP1 RST2,
..
FF RP1 RST7,
```

Handiger is `1FAMILY,` dat het in een keer doet. Hex codes intikken leidt tot fouten, vooral als het ingewikkelder wordt. Met deze code definieren we deze 8 mnemonieken tegelijkertijd.

```
8 C7 8 1FAMILY, RST0, RST1, RST2, RST3, RST4, RST5, RST6, RST7,
```

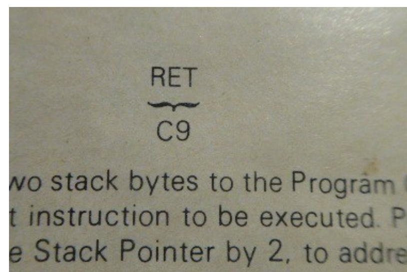
De eerste 8 is het increment, C7 is de eerste opcode, en de tweede 8
is het aantal namen dat volgt.

QUICK REFERENCE PAGE FOR 8080 ASSEMBLER

	0 / 8	1 / 9	2 / A	3 / B	4 / C	5 / D	6 / E	7 / F
0	NOP	LXI	STAX	INX	INR	DCR	MVI	RLC
8		DAD	LDAX	DCX	INR	DCR	MVI	RRC
10		LXI	STAX	INX	INR	DCR	MVI	RAL
18		DAD	LDAX	DCX	INR	DCR	MVI	RAR
20		LXI	STAX/SHLD	INX	INR	DCR	MVI	DAA
28		DAD	LDAX/LHLD	DCX	INR	DCR	MVI	CMA
30		LXI	STAX	INX	INR	DCR	MVI	STC
38		DAD	LDAX	DCX	INR	DCR	MVI	CMC
40	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
48	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
50	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
58	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
60	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
68	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
70	MOV	MOV	MOV	MOV	MOV	MOV	MOV/HLT	MOV
78	MOV	MOV	MOV	MOV	MOV	MOV	MOV	MOV
80	ADD	ADD	ADD	ADD	ADD	ADD	ADD	ADD
88	ADC	ADC	ADC	ADC	ADC	ADC	ADC	ADC
90	SUB	SUB	SUB	SUB	SUB	SUB	SUB	SUB
98	SBB	SBB	SBB	SBB	SBB	SBB	SBB	SBB
A0	ANA	ANA	ANA	ANA	ANA	ANA	ANA	ANA
A8	XRA	XRA	XRA	XRA	XRA	XRA	XRA	XRA
B0	ORA	ORA	ORA	ORA	ORA	ORA	ORA	ORA
B8	CMP	CMP	CMP	CMP	CMP	CMP	CMP	CMP
C0	RC,	POP	JC,	JMP	CC,	PUSH	ADI	RST0
C8	RC,	RET	JC,	CALL	CC,		ACI	RST1
D0	RC,	POP	JC,	OUT	CC,	PUSH	SUI	RST2
D8	RC,		JC,	IN	CC,		SBI	RST3
E0	RC,	POP	JC,	XTHL	CC,	PUSH	ANI	RST4
E8	RC,	PCHL	JC,	XCHG	CC,		XRI	RST5
F0	RC,	POP	JC,	DI	CC,	PUSH	ORI	RST6
F8	RC,	SPHL	JC,	EI	CC,		CPI	RST7

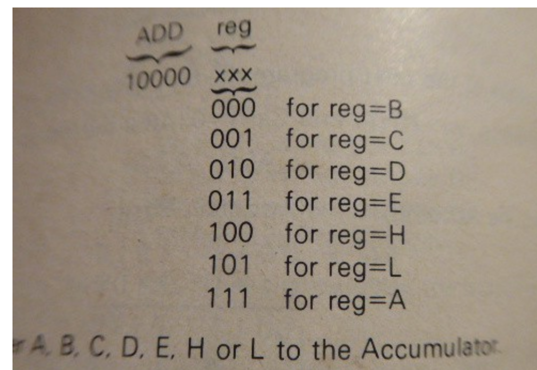
By Albert van der Horst DFW Holland

Een voorbeeld van een zo'n enkele instructie voor de 8080 zien we hier, benevens een uitleg wat de processor doet bij deze instructies (wat ons even niet interesseert.)


RET
C9
wo stack bytes to the Program
t instruction to be executed. P
e Stack Pointer by 2, to addre

Nu richten we onze aandacht op de rij die begint met `ADD`. (afgedrukt in rood)

Deze instructie telt een register op bij register A.
In het overzicht zien we dat `ADD,` de opcodes 80, 81, 82,...87 beslaan.
Deze beschrijving ziet er zo uit:


ADD reg
10000 xxx
000 for reg=B
001 for reg=C
010 for reg=D
011 for reg=E
100 for reg=H
101 for reg=L
111 for reg=A
A, B, C, D, E, H or L to the Accumulator

Op de plaatsen van de iksen, drie bits, moeten we nog een register invullen.
Dat is de taak van een Fix Up mnemoniek.
Hier werken we ook met het family principe.

01 00 8 xFAMILY| B| C| D| E| H| L| M| A|

Een complete instructie met ADD, is dus

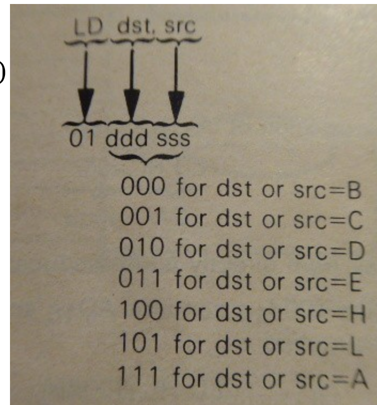
ADD, E|

Een Post It reserveert plaats voor een instructie.
De dictionary pointer hoogt op.
Een Fix Up lapt de voorgaande instructie op door gaten op te vullen.
De dictionary pointer hoogt niet op.

Om aan te geven dat er plaats in het dictionary gereserveerd wordt, eindigen we de Post It met een ',' .
Een Fix Up is in feite een bitsgewijs-OR operatie. De data van ADD, en van E| wordt ge-ORed en in het byte geplaatst. Voor mensen die bekend zijn met taal c : '|' is het symbool voor de bitwise OR operatie in c.

We gaan een stap verder met de MOV, instructie.
Die beslaat maar liefst een kwart van de instructie ruimte
en is groen afgedrukt.

De beschrijving van MOV ziet er zo uit:
(Merk op dat de schrijvers van het handboek niet met ons eens zijn over de naam van de mnemoniek.)



Copieer het source register naar het destination register.
Maar liefst 6 bits zijn nog niet ingevuld.
Voor het source register hebben we al wat.
Voor het destination register (de ddd bits) hebben we een andere Fix Up nodig.

```
08 00 8 xFAMILY| >B >C >D >E >H >L >M >A
```

Die vullen op de ddd respectievelijk 00 08 10 18 .. 38 in.
En een voorbeeld

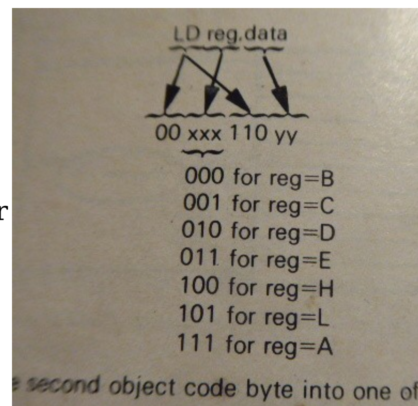
```
MOV, A| >B
```

Maar

```
MOV, >B A|
```

werkt even goed!
De assembler houdt bij wat de ingevulde bitjes zijn.
In het eerste voorbeeld gaat de turf (TALLY)
0011 1111 -> 0011 1000 -> 0000 0000
in het tweede voorbeeld
0011 1111 -> 0000 0111 -> 0000 0000
Als de turf niet op nul staat voor de volgende instructie is het een fout.

Dan daar is het derde aspect. Sommige instructie bevatten data dat onmiddellijk volgt op de instructie. De Engelse term is immediate.
Een voorbeeld is `MVI, .` (paars in het overzicht).
Deze instruktie vult een register met de data in het volgende byte.
Dus ook deze instructie behoeft een complementering.
In het schema is af te leiden waar de niet ingevulde bits zitten.



Copieer het onmiddellijk volgende byte naar het register aangegeven met ddd.
Dit byte kan eenvoudig geassembleerd worden met `C,` ,
maar dan ontbreekt enige vorm van controle.
MVI, werkt de turf bitjes bij, maar kondigt tevens aan dat er een immediate byte vereist wordt.
IC, doet hetzelfde als C, maar hij meldt dat er aan het immediate byte voldaan is. Als je C, zou gebruiken dan krijg je een foutmelding.
Merk op dat de schrijvers het niet met ons eens zijn over de naam van de instructie.
Erger is dat ze dezelfde naam gebruiken voor register naar register copieren en laden van onmiddellijke data in een register. In een PostIt kan dat niet! Dan zouden dezelfde bitjes voor ingevuld worden.

De ontwerpers van processoren houden de bit-velden op dezelfde plaats, gelukkig. De fixup's zoals >E (naar E-register) zijn te gebruiken.
Dan ziet die instructie er zo uit.

```
MVI,  >D  123 IC,
```

Met dit principe heb ik 8080 8086 80386 Pentium en DEC-Alpha en diverse andere assemblers gebouwd.

De assembler elementen -- postit fixup commaer – zijn objecten, alle eigenschappen zijn bekend.
Dat geeft diverse mogelijkheden

- uit de objecten kan je een volledige testset genereren.
- de objecten kunnen worden gebruikt om te disassembleren
- het disassembleren van de volledige testset en dan opnieuw assembleren, is een soliede controle van het systeem
- halverwege een incomplete instructie kan je ?? intikken. Dat laat alle mogelijkheden zien om de instructie te completeren.
- disassembly gevolgd door assembly genereert een exact gelijk programma

In assembler 8086 (etc.) zien we

```
MOV, X| T| AX'| R| BX|        \ Move to register AX : register BX
MOV, X| F| BX'| R| AX|        \ Move from register BX : register AX
```

Deze instruktie doen hetzelfde, maar de programmeur moet de keuze maken.
Deze assembler maakt geen keuzes!
Een ander voorbeeld is

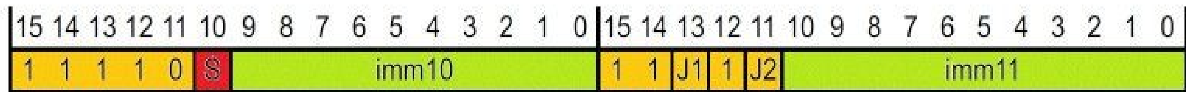
```
PUSH, R| BP|               \ Normal
PUSH|X, BP|                \ Short form, saves one byte.
```

Voorbeelden ontleend aan "Osborne Z80 programming for logic design". De mnemonieken die ik gebruik zijn van Intel.

AH

PIO (dis)assembler voor RP2040, Willem Ouwerkerk

Dit is de 24-bits spring en link opcode, de basis voor een subroutine call.




```
Geel  = De opcode velden,
Rood  = Het teken (de richting van de sprong)
IMM11 = Bit 0 t/m 10
IMM10 = Bit 11 t/m 20
J2    = Bit 21
J1    = Bit 22
```

Dit stukje code uit de assembler stelt de BL opcode samen:

```
\ .....7FF - 11 bits, bit 0 to 10    ..1FF800 - 10 bits, bit 11 to 20
\ ..600000 -  2 bits, bit 21 & 22    ..800000 - Sign bit, bit 23
: BL)       ( a1 a2 -- opc )    \ Build 32-bits branch & link opcode, range is
+-24-bits
    >jump  dup FFFFFF ?range-s       \ Calc. offset & check range
    F000D000  over 7FF and or        \ Add first 11 bits to basic opcode
    over 1FF800 and  5 lshift or     \ Add next 10 bits
    over 0< >r  swap 0A rshift       \ Save sign & get bit 21&22 to bit 11&12
    invert  r@ xor  dup 800 and      \ Invert & add sign to J1 & J2, J2 is ok
    swap 1000 and  2* or  or         \ J1 to bit 13 & add to J2 and to opcode
    r> 4000000 and or ;              \ Add J1, J2 and sign, generate opcode
```

En dit stukje uit de disaasembler decodeert hem weer:

```
: .BL       ( a opc -- a )      \ Decode branch & link opcode
    dup 4000000 and 0= 0= >r        \ Make & save sign
    ." bl    "  r@ FF800000 and     \ Extend sign   opc sign
    over 7FF and  or                \ Bit 0 to 10   opc bl..
    over 3FF0000 and 5 rshift  or   \ Bit 11 to 21  opc bl..
    swap -1 xor r> xor  2800 and    \ Isolate bit 21 & 22
    dup >r 800 and A lshift  or     \ Add bit 21
     r> 2000 and 9 lshift or .jump ; \ & bit 22
\ .....7FF - 11 bits, bit 0 to 10   ..1FF800 - 10 bits, bit 11 to 20
\ ..600000 -  2 bits, bit 21 & 22   ..800000 - Sign bit, bit 23
: BL)       ( a1 a2 -- opc )    \ Build 32-bits branch & link opcode, range is
+-24-bits
    >jump  dup FFFFFF ?range-s      \ Calc. offset & check range
    F000D000  over 7FF and or       \ Add first 11 bits to basic opcode
    over 1FF800 and  5 lshift or    \ Add next 10 bits
    over 0< >r  swap 0A rshift      \ Save sign & get bit 21&22 to bit 11&12
    invert  r@ xor  dup 800 and     \ Invert & add sign to J1 & J2, J2 is ok
    swap 1000 and  2* or  or        \ J1 to bit 13 & add to J2 and to opcode
    r> 4000000 and or ;             \ Add J1, J2 and sign, generate opcode
```


En dit stukje uit de disaasembler decodeert hem weer:

```
: .BL       ( a opc -- a )      \ Decode branch & link opcode
    dup 4000000 and 0= 0= >r        \ Make & save sign
    ." bl    "  r@ FF800000 and     \ Extend sign   opc sign
    over 7FF and  or                \ Bit 0 to 10   opc bl..
    over 3FF0000 and 5 rshift  or   \ Bit 11 to 21  opc bl..
    swap -1 xor r> xor  2800 and    \ Isolate bit 21 & 22
    dup >r 800 and A lshift  or     \ Add bit 21
    r> 2000 and 9 lshift or .jump ; \ & bit 22
```