<u>Oefeningen Computersystemen 7.11.1996</u>

1. <u>Machine code voorstelling van een programma</u>

Beschouw de “SM1” machine. Voor de “opcode” gebruiken we 8 bits. Voor de specificatie van de operatie bij de “OPR” instructie gebruiken we 4 bits. Addressen (indirect en direct) en constanten (“Immediate” addressering) stellen we voor met 32 bit. Het geheugen wordt *per byte* geaddresseerd. Voor de duidelijkheid is bij deze oefening het “level” veld weggelaten. Bijgevolg is de machine code (voorgesteld in hexadecimale notatie) voor de respectievelijke instructies :

NULL	00	*doet niets, behalve P := P + 1 ;*
OPR x	1x	*waarbij x = 0..B (cf. Pagina 4-5). Als instructie C wordt “DUP” gedefinieerd. DUP dupliceert de “top of stack”.*
LIT xxxxxxxx	02 xxxxxxxx	*waarbij xxxxxxxx de two’s complement voorstelling van de constante is(“immediate” operand).*
LOD xxxxxxxx	03 xxxxxxxx	*waarbij xxxxxxxx het adres van de operand is (“ direct” addressing).*
STO xxxxxxxx	04 xxxxxxxx	*idem als bij “LOD”.*
LDI xxxxxxxx	05 xxxxxxxx	*waarbij xxxxxxxx het adres van het adres van de operand is (“indirect” addressing).*
STI xxxxxxxx	06 xxxxxxxx	*idem als bij “LDI”.*
JMP xxxxxxxx	07 xxxxxxxx	*waarbij xxxxxxxx het bestemmingsadres is*
JPC xxxxxxxx	08 xxxxxxxx	*idem als JMP*
CAL xxxxxxxx	09 xxxxxxxx	*idem als JMP*
RET	0A	
INT xxxxxxxx	0B xxxxxxxx	met xxxxxxxx de “two’s complement” voorstelling van het increment.

1.a. Waarom stellen we de instrukties voor door een volledige byte terwijl we principieel genoeg zouden hebben aan 4 bits aangezien er slechts 12 instrukties zijn (het x veld van OPR zouden we dan beschouwen als een immediate operand van 4 bits) ?

1.b. Er is gegeven (bv. door de linker) dat het programma begint vanaf adres 0_{16} en de gegevens beginnen vanaf adres 40_{16}.
Wat doet het volgende programma ?
Wat is de stopvoorwaarde (door naar adres 0 te springen) ?
Wat is het eindresultaat van de uitvoering van het voorbeeld, i.e. welke gegevens zullen er in het geheugen staan vanaf adres 40_{16} na de uitvoering van het programma ?

adres	code
0000	0005000000401C02000000001B080000
0010	0013111C020000000016080000003906
0020	00000040030000004002000000041204
0030	00000040070000000107000000000000
0040	00000050000000000000000000000000
0050	00000001FFFFFFFE00000003FFFFFFFC
0060	00000005FFFFFFFA00000007FFFFFFF8
0070	00000009FFFFFFE60000000BFFFFFFE4
0080	00000000000000000000000000000000

2.Foutconditie

Door een externe oorzaak wordt er 1 bit in het geheugen gewijzigd. Mogelijke oorzaken hiervan kunnen zowel physisch (bv. cosmische straling, een geheugencel die slecht werkt, …) als logisch zijn (een ander programma in het geheugen van de processor wijzigt ongewild (softwarefout) dan wel gewild (virus) een gebied in het geheugen waar het eigenlijk geen gebruik van mag maken).
Wat gebeurt er met elk van de volgende programma's : werkt het nog ? Stopt het nog ? Produceert het nog een (gedeeltelijk) juist resultaat ? Wijzigt het het geheugen van een ander programma ?

2a. Geheugenpositie $2D_{16}$ verandert van de waarde 00000100_2 in 00000000_2.

```
0000 0005000000401C02000000001B080000
0010 0013111C020000000016080000003906
0020 00000040030000004002000000001204
0030 00000040070000000107000000000000
0040 00000050000000000000000000000000
0050 00000001FFFFFFFE00000003FFFFFFFC
0060 00000005FFFFFFFA00000007FFFFFFF8
0070 00000009FFFFFFE60000000BFFFFFFE4
0080 00000000000000000000000000000000
```

2b. Geheugenpositie 43_{16} wijzigt van de waarde 01010000_2 in 00010000_2

```
0000 0005000000401C02000000001B080000
0010 0013111C020000000016080000003906
0020 00000040030000004002000000041204
0030 00000040070000000107000000000000
0040 00000010000000000000000000000000
0050 00000001FFFFFFFE00000003FFFFFFFC
0060 00000005FFFFFFFA00000007FFFFFFF8
0070 00000009FFFFFFE60000000BFFFFFFE4
0080 00000000000000000000000000000000
```

2c. Geheugenpositie 74_{16} wijzigt van de waarde 11111111_2 in 01111111_2.

```
0000 0005000000401C02000000001B080000
0010 0013111C020000000016080000003906
0020 00000040030000004002000000041204
0030 00000040070000000107000000000000
0040 00000050000000000000000000000000
0050 00000001FFFFFFFE00000003FFFFFFFC
0060 00000005FFFFFFFA00000007FFFFFFF8
0070 000000097FFFFFE60000000BFFFFFFE4
0080 00000000000000000000000000000000
```

3. Vectorinstructies

We maken een "high performance" versie van SM1. Deze is binair compatibel met de "standaardversie", i.e. ze voert dezelfde instructieset uit. Daarnaast kan ze echter ook "vectorinstructies" uitvoeren. Deze hebben dezelfde structuur als de "OPR" opcode, doch op de stack staan niet meer de operanden, maar de startaddressen van de sourcevector(en) en destination vector, evenals de lengte van de vectoren. De vectorinstructies worden als volgt in het geheugen voorgesteld :

VOPR x	2x *waarbij x = 0..B (cf. Pagina 4-5). Als instructie C wordt ook "DUP" gedefinieerd. DUP dupliceert de sourcevector in de destination vector.*

De interpretatie van deze instructies is als volgt (cf. Tabel 4.9) : de top of stack, i.e. Data[T] bevat het startadres van de *destination* vector : het stuk geheugen waar de resultaten weggeschreven worden. Data[T-1] bevat het sourceadres : het stuk geheugen waar de array met operanden staan. Indien er twee sourceadressen nodig zijn (als operatie "x" twee operanden nodig heeft) bevatten Data[T-1] en Data[T-2] het startadres van respectievelijk de tweede en de eerste source vector. De "count" waarde, i.e. de *lengte* der vectoren, zit in het geval er slechts 1 source is in Data[T-2], in het geval er 2 sources zijn in Data[T-3]. De elementen der vectoren, evenals addressen en constanten zijn 32 bit, i.e. 4 bytes, lang. We nemen aan dat de "count" waarde > 0 is (bv. omdat de compiler hiervoor zorgt).

```
CASE opc OF
        VOPR : CASE m OF
                0;      BEGIN (* Exchange vectors *)
                                FOR i := 1 TO Data[T-2] DO
                                        Data[T+1] := Data[Data[T]];
                                        Data[Data[T]] := Data[Data[T-1]];
                                        Data[Data[T-1]] := Data[T+1];
                                        Data[T] := Data[T] + 1; Data[T-1] := Data[T-1] + 1;
                                END (* FOR *);
                                T := T - 3;
                        END; (* 0 *)
                1;      BEGIN (* Change Sign *)
                                FOR i := 1 TO Data[T-1] DO
                                        Data[Data[T]] := - Data[Data[T]];
                                END (*FOR*);
                                T := T -2;
                        END; (*1*)
                2;      BEGIN (* ADD 2 source vectors in a destination vector *)
                                FOR i := 1 TO Data[T-3] DO
                                   Data[Data[T]] := Data[Data[T-1]] + Data[Data[T-2]];
                                        Data[T] := Data[T]+1; Data[T-1] := Data[T-1]+1;
```

```
                    Data[T-2] := Data[T-2]+1;
                END (*FOR*);
                T := T - 4;
            END; (* 2 *)
        …
        6;      BEGIN(*Vector logical operation: destination is a boolean vector *)
                FOR i := 1 TO Data[T-3] DO
                    Data[Data[T]]:=(Data[Data[T-1]]=Data[Data[T-2]]);
                    Data[T] := Data[T] + 1; Data[T-1] := Data[T-1] + 1;
                        Data[T-2]:=Data[T-2]+1;
                END (* FOR *);
                T := T - 4;
        …
    END;
END;
```

3a. Kunnen we ons oorspronkelijk programma "verbeteren" (i.e. spatiaal door minder instructiegeheugenruimte in te nemen en/of temporeel door minder instructies uit te voeren om hetzelfde resultaat te bekomen) met de nieuwe "vectorinstructies" ? Zo ja : herschrijf het programma (in assembler). Zo neen : waarom niet ?

3b. Beschouw het volgende programma : de kode verschilt slechts op 1 positie met de kode van het programma uit vraag 1. Het resultaat is uiteraard verschillend. Welke functie van de sourcevector berekent het programma ?

```
adres   code

0000    0005000000401C02000000001B080000
0010    0012111C020000000016080000003906
0020    00000040030000004002000000041204
0030    00000040070000000107000000000000
0040    00000050000000000000000000000000
0050    00000001FFFFFFFE00000003FFFFFFFC
0060    00000005FFFFFFFA00000007FFFFFFF8
0070    00000009FFFFFFE60000000BFFFFFFE4
0080    00000000000000000000000000000000
```

3.c. Herschrijf dit programma (in assembler) met een vectorinstructie. Vergelijk Uw resultaat met bovenstaande implementatie op een "standaard" SM1.