LBA (Long Binary Architecture)
door Theo Kortekaas

Doel

Doel van de Lba architectuur is het mogelijk maken op een eenvoudige manier vanuit een Assembler-programma en vanuit een C^{++} programma rekenkundige bewerkingen uit te voeren op zeer grote getallen. Een dergelijke voorziening wordt ook wel aangeduid als arbitrary precision arithmetic of multiple precision arithmetic. Een andere veel gebruikte term voor grote getallen bij computer-rekenen is BigNum.
Eenvoudige bewerkingen (optellen, aftrekken, vermenigvuldigen, shift enz.) vinden meestal plaats in integer registers die een beperkte capaciteit hebben. Lba biedt de mogelijkheid om dezelfde bewerkingen uit te voeren op integer variabelen met een grote capaciteit. Daartoe is een Lba variabele gedefinieerd, die we Lba-veld noemen, met een nagenoeg onbeperkte capaciteit. Verder is er een aantal Lba functies gedefinieerd die op deze Lba-velden werken. Het streven is om de uitvoering van de Lba functies zo snel mogelijk te maken. Daarom is er voor gekozen om alleen met natuurlijke getallen en nul te werken, en niet met negatieve getallen.
De Lba architectuur is bedoeld voor een computer-omgeving gebaseerd op een X86 processor werkend onder een 32-bits Windows-systeem. De gebruikte programma ontwikkelomgeving is een Windows XP systeem met Borland Cbuilder 4.0 met daarin de tools TASM32, BCC32, ILINK32 en BRC32.
Deze ontwikkelomgeving is nogal gedateerd; de daarmee geproduceerde programma's draaien echter zonder problemen op Windows 7, Windows 8 en Windows 10. Een modernisering zou wenselijk zijn; evenals een overgang naar 64-bits. Het vooruitzicht, echter, om duizenden *lines of code* te moeten herzien, vormt nog een flinke barrière.
De capaciteit van een Lba-veld is op dit moment alleen beperkt door een paar praktische limieten, zoals de maximale grootte van een adres-ruimte onder een Win32 operating systeem en het maximale getal dat wordt gebruikt voor bitnummer in een 32-bits integer.
Een praktische limiet is daarom voor de grootte van een Lba-veld 100.000.000 integers van 32 bit. Er wordt geen controle uitgevoerd op deze limiet, maar overschrijding ervan kan leiden tot onvoorspelbare resultaten.

Historie

Het begin van de ontwikkeling van Lba stamt uit de tijd dat de 80386 processor werd ingevoerd en OS/2 nog een veel belovend operating systeem was. Op basis van dit platform (80386 en OS/2) is toen een library opgezet met de belangrijkste rekenkundige functies voor Lba-velden. Deze library is uitgevoerd in de vorm van assembler macro's.
De verdere ontwikkeling in de computer-wereld maakte het nodig om voor deze functie-library verschillende malen een migratie uit te voeren: van een OS/2 omgeving naar Windows 3.1; en van Windows 3.1 naar de Win32 application program interface. Een overgang naar een 64-bit omgeving is nog niet voorzien.

Verdere ontwikkelingen in Lba bieden speciale functies zoals berekening van grootste gemene deler; het Uitgebreide Euclidische Algoritme en de modulaire Montgomery vermenigvuldiging. Ook is er een interface ontwikkeld tussen Lba en C^{++}.
De documentatie die met deze publicatie beschikbaar komt beperkt zich voornamelijk tot de met X86 equivalente operaties.

X86 terminologie

In deze beschrijving wordt de X86 terminologie gehanteerd. In deze terminologie is het computergeheugen verdeeld in bytes. Elke byte heeft 8 bits en is apart adresseerbaar. De bits worden aangeduid met *bit 0* tot en met *bit 7*. Een *word* heeft 16 bits en kan zich bevinden in een 16-bits register of in twee aaneensluitende bytes in het computergeheugen. De bitnummering in een word is *bit 0* tot en met *bit 15*. Bit 0 heeft de waarde 1 als de bit “on” staat en bit 15 heeft de waarde 32.768 als de bit “on” staat.
Een doubleword heeft 32 bits en kan een *32-bits integer* bevatten. Gewoonlijk wordt een doubleword aangeduid met *dword*. Een *dword* kan zich bevinden in een *32-bits register* of in vier aaneensluitende bytes in het computergeheugen. De bitnummering in een *dword* is *bit 0* tot en met *bit 31*.
Indien een *dword* zich in vier aaneensluitende bytes in het computergeheugen bevindt dan staan *bit 0* tot en met *bit 7* in de eerste byte; *bit 8* tot en met *bit 15* in de tweede byte; *bit 16* tot en met *bit 23* in de derde byte en *bit 24* tot en met *bit 31* in de vierde byte. Deze wijze van opbergen wordt *little-endian* genoemd.
X86 kent acht 32-bits registers (zeven hiervan zijn voor algemeen gebruik). Dit worden *general registers* genoemd. Deze registers heten: EAX, EBX, ECX, EDX, EDI, ESI, EBP en ESP (deze laatste dient als stack-pointer). Voorts is er een Flag-register met één-bits indicatoren (flags) zoals zero-flag; carry-flag enz.

Lba-veld

Binnen de Lba architectuur worden meerdere formaten gehanteerd. Het belangrijkste formaat is het Lba-veld. Deze beschrijving beperkt zich tot het Lba-veld. Het Lba-veld is een veld in het computergeheugen waarin een getal kan worden opgeborgen. (Een Lba-veld kan naruurlijk ook in een extern bestand worden weggeschreven). Het Lba-veld bestaat uit twee delen: het Lba-lengte veld en de Lba-body. Het Lba-lengte veld bestaat uit één dword waarin de lengte *n* van de Lba-body wordt vastgelegd. De Lba-body bestaat uit *n* dwords die direct aansluitend op het Lba-lengte veld in het computergeheugen wordt gepositio-neerd. Een praktische limiet voor *n* is 100.000.000. Hierop wordt echter niet gecontroleerd. Alle (rekenkundige) bewerkingen worden in principe op Lba-velden uitgevoerd en gebeuren in eenheden van dwords. In de literatuur worden deze eenheden (van BigNums) ook wel limbs genoemd (limbs betekent ledematen). De bitnummering loopt voor de eerste limb (na het Lba-lengte veld) gewoon van bit-nummer nul tot en met bit-nummer 31. Voor de volgende limbs loopt de bitnummmering door tot en met resp. bit-nummer 63 voor een Lba-veld van twee dwords, bit-nummer 95voor een Lba-veld van drie dwords enzovoorts. Een praktische limiet voor bitnummer is 3.199.999.999. Ook hierop wordt echter niet gecontroleerd.
Bij de definitie van een Lba-veld wordt de lengte vastgesteld; deze kan gedurende de looptijd van het programma niet meer worden gewijzigd. (Er zijn mogelijkheden om dynamisch Lba-velden te creëren en te verwijderen; daar wordt hier niet op ingegaan). Bij de definitie van een Lba-veld is de inhoud van de lba-body ongedefinieerd. Lba-velden mogen elkaar niet overlappen.

LBA componenten

Lba krijgt gestalte in de vorm van deze beschrijving. Verdere componenten zijn een bestand met assembler sources (lba.asm); een bestand met assembler macro's (lba.mac) en een bestand met C++ definities (lba.h). Deze bestanden kunnen hier worden ingezien en worden gedownload.

- tonjanee.home.xs4all.nl/lba.asm
- tonjanee.home.xs4all.nl/lba.mac
- tonjanee.home.xs4all.nl/lba.h

Voorts is er een gecompileerde versie van de lba library in de vorm van een object file: deze kan meegelinked worden met een ander programma.

- tonjanee.home.xs4all.nl/lba.obj

Deze object file bevat *niet* de reporting functies “Report” en “Mes”.
Deze functies kunnen alleen gebruikt worden in de “lba operating environment”.
Dat houdt in een gecombineerd Assembler en C++ programma met menu-structuur en voorziening voor het creëeren en displayen van een log-bestand.

Beschrijving van LBA definities en operaties

De hier beschreven operaties zijn overwegend eenvoudige operaties waarvoor geen speciale services nodig zijn zoals: memory management, interrupties of display services. Dit zijn operaties waarvoor in het algemeen een X86 equivalent beschikbaar is. In dat geval zijn de parameters van de operaties zoveel mogelijk vergelijkbaar met het X86 equivalent.
Indien een ongeldige conditie wordt geconstateerd, dan wordt het resultaatveld, meestal, op nul gesteld. De Lba velden mogen elkaar niet overlappen.
In de beschrijving per operatie wordt de volgende informatie achtereenvolgens gegeven:

De eerste regel(s): de naam van de operatie en de parameters.
De parameters vermelden een type:
lba; dit betreft het adres van een Lba-veld;
int; dit betreft de inhoud van een 32-bits integer;
intAd; dit betreft het adres van een 32-bits integer.

De tweede regel: de aanroep-informatie vanuit een C^{++} omgeving.
Deze informatie is gebaseerd op de Borland bcc32 compiler en verwijst naar een functie in de lba.h file

De derde regel: de aanroep-informatie vanuit een Assembler omgeving.
Deze informatie is gebaseerd op de Borland tasm32 Macro-assembler, en verwijst naar een macro in de lba.mac file.
De naam van de macro is, indien mogelijk, de naam van het X86 equivalent aangevuld met een M.

De vierde regel: de mnemonic van het X86 equivalent.
Dit is de mnemonic zoals gehanteerd in de Intel documentatie[1].

De overige regels beschrijven de operatie.

LBA definities

Define LBA variabele Parameters: LBA-name,Length
C++: Ldefb (Name,n) ;
Asm: LDEFB Name,n
X86: geen

Met Ldefb wordt een LBA variabele (Lba-veld) gedefinieerd met als naam Name en met als grootte *n* 32-bits integers (dwords). *n* is de grootte van de Lba-body in aantal dwords.
De eerste 32-bits integer (dword) is het Lba-lengte veld en krijgt als inhoud *n*; de inhoud van de volgende *n* 32-bits integers (dwords) is ongedefinieerd.

LBA Operaties

Move **Parameters: Naar(lba),Van(lba)**
C++: **Mov (Naar,Van) ;**
Asm: **MOVM Naar,Van**
X86: **MOV**

De inhoud van het van-veld wordt gekopiëerd naar het naar-veld. Het eerste element van de Lba-body van het van-veld gaat naar het eerste element van de Lba-body van het naar-veld enzovoorts. Indien de lengte van het naar-veld groter is dan de lengte van het van-veld, dan wordt het deel van het naar-veld dat overblijft opgevuld met elementen met de inhoud nul. Indien de lengte van het van-veld groter is dan de lengte van het naar-veld dan wordt slechts het gedeelte van het van-veld dat past in het naar-veld overgebracht. Van het gedeelte van het van-veld dat niet kan worden overgebracht wordt de inhoud van de elementen gecontroleerd op nul. Indien een of meerdere elementen van dat deel een inhoud hebben ongelijk aan nul, dan wordt de gehele operatie beschouwd als ongeldig en wordt het naar-veld in zijn geheel op nul gesteld.
Het van-veld mag niet hetzelfde zijn als het naar-veld.
De inhoud van alle general registers blijft onaangetast, evenals de status van de flags.

Copy **Parameters: Naar(lba),Van(lba)**
C++: **Copy (Naar,Van)**
Asm: **Copy Naar,Van**
X86: **Geen**

De inhoud van het van-veld wordt gekopieerd naar het naar-veld. Er wordt vanuit gegaan dat beide velden even groot zijn. Daar wordt niet op gecontroleerd. Deze operatie is de snelste manier om een Lba-veld te verplaatsen. Er wordt echter geen controle uitgevoerd op de lengte van beide velden. De inhoud van alle general registers blijft onaangetast, evenals de status van de flags.

Load **Parameters: To-field(lba),From-field(int)**
C++: **Load (To,From)**
Asm: **LOAD To,From**
X86: **Geen**

Het Lba-veld “To” wordt op nul gesteld; daarna wordt de inhoud van de 32 bits integer “From” in de eerste 32 bits van het Lba-veld geplaatst. De aanduiding “int” kan een van de general registers bevatten, of een waarde die maximaal 2^{32} - 1 is. De inhoud van alle general registers blijft onaangetast. Ook de inhoud van het integer veld blijft ongewijzigd.
De status van de flags is onzeker.

Store **Parameters: To-field(intAd),From-field(lba)**
C++: **Store (To,From)**
Asm: **STORE To,From**
X86 **Geen**

De eerste 32 bits van het Lba-veld “From” worden in het 32 bits integer-veld “To” geplaatst. “To” bevat het adres van een 32-bits geheugen-veld. De inhoud van het Lba-veld blijft ongewijzigd. De inhoud van alle general registers blijft onaangetast, evenals de status van de flags.

Add **Parameters: Som(lba),Num(lba)**
C++: **Add (Som,Num)**
Asm: **ADDM Som,Num**
X86: **ADD**

De inhoud van het Num-veld wordt geteld bij het Som-veld. Indien het Num-veld kleiner is dan het Som-veld wordt het Num-veld tot de grootte van het Som-veld geacht te zijn aangevuld met elementen met de waarde nul. Indien de twee velden even groot zijn, dan is het resultaat betrouwbaar tenzij er bij het laatste element een carry ontstaat. In dat geval (en ook in het geval dat bij het laatste element een carry ontstaat wanneer het Num-veld kleiner is dan het Som-veld) is het resultaat van de optelling ongeldig en wordt het gehele Som-veld op nul gesteld.
Wanneer het Som-veld kleiner is dan het Num-veld moet de rest van het Num-veld, dat de grootte van het Som-veld overstijgt slechts elementen met de waarde nul bevatten. Hierop wordt gecontroleerd en indien dit niet zo is, is het resultaat van de optelling ongeldig en wordt het gehele Som-veld op nul gesteld.
Het Num-veld blijft ongewijzigd.
De inhoud van alle general registers blijft onaangetast, evenals de status van de flags.

Subtract **Parameters: Som(lba),Num(lba)**
C++: **Sub (Som,Num)**
Asm: **SUBM Som,Num**
X86: **SUB**

De inhoud van het Num-veld wordt afgetrokken van het Som-veld. Indien het Num-veld kleiner is dan het Som-veld wordt het Num-veld tot de grootte van het Som-veld geacht te zijn aangevuld met elementen met de waarde nul. Indien de twee velden even groot zijn, dan is het resultaat betrouwbaar tenzij er bij het laatste element een borrow ontstaat. In dat geval (en ook in het geval dat bij het laatste element een borrow ontstaat wanneer het Num-veld kleiner is dan het Som-veld) is het resultaat van de aftrekking ongeldig en wordt het gehele Som-veld op nul gesteld. Wanneer het Som-veld kleiner is dan het Num-veld moet de rest van het Num-veld, dat de grootte van het Som-veld overstijgt slechts elementen met de waarde nul bevatten. Hierop wordt gecontroleerd en indien dit niet zo is, is het resultaat van de aftrekking ongeldig en wordt het gehele Som-veld op nul gesteld. Het Num-veld blijft ongewijzigd. De inhoud van de general registers blijft onaangetast.

Multiply **Parameters: Product(lba),Mc(lba),Mr(lba)**
C++: **Mul (Product,Mc,Mr)**
Asm: **MULM Product,Mc,Mr**
X86: **MUL**

De inhoud van Multiplicand (Mc) en Multiplier(Mr) worden met elkaar vermenigvuldigd volgens de klassieke methode[2] en het resultaat wordt in Product geplaatst. Indien de som van de lengte van Mc en de lengte van Mr (in dwords) groter is dan de lengte van Product, dan wordt aangenomen dat Product te klein is om het resultaat te bevatten. In dat geval wordt de operatie als ongeldig beschouwd en wordt Product op nul gesteld. De Lba-velden Mc en Mr blijven ongewijzigd. De inhoud van de general registers blijft onaangetast. De status van de Flags is gewijzigd.

Divide **Parameters: Quotiënt(lba),Dividend(lba),Divisor(lba)**
C++: **Div (Quotiënt,Dividend,Divisor)**
Asm: **DIVM Quotiënt,Dividend,Divisor**
X86: **DIV**

De inhoud van Dividend wordt gedeeld door de inhoud van Divisor en het resultaat wordt geplaatst in Quotiënt. Een eventuele rest wordt in Dividend geplaatst. De inhoud van Divisor mag niet nul zijn. Indien Divisor toch nul is wordt de operatie als ongeldig beschouwd en wordt Quotiënt op nul gesteld. De inhoud van Divisor blijft ongewijzigd. De inhoud van de general registers blijft onaangetast. De status van de Flags is gewijzigd.

Increase **Parameters: Lba-veld(lba)**
C++: **Inc (Lba-veld) ;**
Asm: **INCM Lba-veld**
X86: **INC**

Lba-veld bevat het adres van het Lba-veld. De inhoud van het Lba-veld wordt met één verhoogd. Indien het Lba-veld vóór de operatie allemaal binair 1 bevat dan wordt de inhoud geheel op nul gezet en wordt de carry-flag op 1 gezet. In elke andere situatie wordt de carry-flag op 0 gereset.
De inhoud van alle general registers blijft onaangetast.

Decrease **Parameters: Lbaveld(lba)**
C++: **Dec (Lbaveld)**
Asm: **DECM Lbaveld**
X86: **DEC**

Lba-veld bevat het adres van het Lba-veld. De inhoud van het Lba-veld wordt met één verminderd. Indien het Lba-veld vóór de operatie geheel 0 was, dan blijft de inhoud geheel nul en wordt de carry-flag op 1 gezet. In elke andere situatie wordt de carry-flag op 0 gereset.
De inhoud van alle general registers blijft onaangetast.

Shift Left Parameters: Lba-veld(lba),Nbits(int)
C++: Shl (Lba-veld,Nbits)
Asm: SHLM Lba-veld,Nbits
X86: SHL

Lba-veld bevat het adres van het Lba-veld en Nbits bevat, als 32-bit integer, de waarde van het aantal bits dat de inhoud van het Lba-veld naar links moet worden verschoven. Een verschuiving van *n* bit naar links staat gelijk aan een vermenigvuldiging van de inhoud van het Lba-veld met een factor 2^n. Indien het aantal te schuiven bits in Nbits groter is dan het aantal bits waaruit het Lba-veld bestaat, dan is de operatie ongeldig en wordt deze niet uitgevoerd. Dit wordt niet gesignaleerd.
De inhoud van de bits die aan de linkerkant worden uitgeschoven gaat verloren. De bits die aan de rechterkant "vrij" komen krijgen de inhoud 0.
De inhoud van alle general registers blijft onaangetast, evenals de status van de flags.

Shift Right Parameters: Lba-veld(lba),Nbits(int)
C++: Shr (Lba-veld,Nbits)
Asm: SHRM Lba-veld,Nbits
X86: SHR

Lba-veld bevat het adres van het Lba-veld en Nbits bevat, als 32-bit integer, de waarde van het aantal bits dat de inhoud van het Lbaveld naar rechts moet worden verschoven. Een verschuiving van *n* bit naar rechts staat gelijk aan een deling van de inhoud van het Lba-veld met een factor 2^n. Indien het aantal te schuiven bits in Nbits groter is dan het aantal bits waaruit het Lba-veld bestaat, dan is de instructie ongeldig en wordt deze niet uitgevoerd. Dit wordt niet gesignaleerd.
De inhoud van de bits die aan de rechterkant worden uitgeschoven gaat verloren. De bits die aan de linkerkant "vrij" komen krijgen de inhoud 0.
De inhoud van alle general registers blijft onaangetast, evenals de status van de flags.

Set Bit **Parameters: Lba-veld(lba), Bitnr(int)**
C++: **Bts** **(Lba-veld,Bitnr) ;**
Asm: **BTSM** **Lba-veld,Bitnr**
X86: **BTS**

Lba-veld bevat het adres van het Lba-veld. Bitnr bevat, als 32-bit integer, het nummer van de bit in het Lba-veld, die op 1 gezet moet worden. Bitnr kan geen grotere waarde bevatten dan het aantal bits dat het Lba-veld groot is (aantal dwords * 32 -1). Indien Bitnr groter is, dan is de operatie ongeldig en wordt deze niet uitgevoerd. Dit wordt echter niet gesignaleerd.
De inhoud van alle general registers blijft ongewijzigd.

Complement Bit **Parameters: Lba-veld(lba), Bitnr(int)**
C++: **Btc** **(Lba-veld,Bitnr) ;**
Asm: **BTCM** **Lba-veld,Bitnr**
X86: **BTC**

Lba-veld bevat het adres van het Lba-veld. Bitnr bevat, als 32-bit integer, het nummer van de bit in het Lba-veld, die "geflipt" moet worden; een 0 wordt 1; en een 1 wordt 0. Bitnr kan geen grotere waarde bevatten dan het aantal bits dat het Lba-veld groot is (aantal dwords * 32 -1). Indien Bitnr groter is, dan is de operatie ongeldig en wordt deze niet uitgevoerd. Dit wordt echter niet gesignaleerd.
De inhoud van de general registers blijft ongewijzigd.

Reset Bit **Parameters: Lba-veld(lba), Bitnr(int)**
C++: **Btr** **(Lba-veld,Bitnr) ;**
Asm: **BTRM** **Lba-veld,Bitnr**
X86: **BTR**

Lba-veld bevat het adres van het Lba-veld. Bitnr bevat, als 32-bit integer, het nummer van de bit in het Lba-veld, die op 0 gezet moet worden. Bitnr kan geen grotere waarde bevatten dan het aantal bits dat het Lba-veld groot is (aantal dwords * 32 -1). Indien Bitnr groter is dan is de operatie ongeldig en wordt deze niet uitgevoerd. Dit wordt echter niet gesignaleerd.
De inhoud van alle general registers blijft ongewijzigd.

Bit Scan Forward **Parameters: Bitnr(intAd),Lba-veld(lba)**
C++: **Bsf** **(Bitnr,Lba-veld)**
Asm: **BSFM** **Bitnr,Lba-veld**
X86: **Bsf**

Lba-veld bevat het adres van het Lba-veld. Bitnr bevat het adres van een 32-bit integer. Het Lba-veld wordt gescand vanaf bit nummer nul (het bit met de kleinste waarde) tot de eerste bit die op een staat. Het nummer van deze bit wordt in Bitnr geplaatst. Indien het Lba-veld geheel nul is dan wordt Bitnr op -1 gesteld om deze situatie te kunnen onderscheiden van de situatie dat de nulde bit op een staat.
De inhoud van alle general registers blijft ongewijzigd.

Bit Scan Reverse **Parameters: Bitnr(intAd),Lba-veld(lba)**
C++: **Bsr** **(Bitnr,Lba-veld) ;**
Asm: **BSRM** **Bitnr,Lba-veld**
X86: **BSR**

Lba-veld bevat het adres van het Lba-veld. Bitnr bevat het adres van een 32-bit integer. Het Lba-veld wordt gescand vanaf de bit met de hoogste waarde tot de eerste bit die op 1 staat. Het nummer van deze bit wordt in Bitnr geplaatst. Indien het Lba-veld geheel nul is dan wordt Bitnr op -1 gesteld om deze situatie te kunnen onderscheiden van de situatie dat de nulde bit op 1 staat en dit de enige bit van het Lba-veld is die op 1 staat.
De inhoud van alle general registers blijft ongewijzigd.

Modulo **Parameters: Base(lba),Modulus(lba)**
C++: **Mod (Base,Modulus) ;**
Asm: **MOD Base,Modulus**
X86: **Geen**

Het getal in Lba-veld Base wordt gedeeld door het getal in Lba-veld Modulus. Het quotiënt speelt geen rol en wordt niet bewaard. De rest van de deling wordt in Lba-veld Base geplaatst en vervangt de oorspronkelijke inhoud. (Deze rest kan nul zijn.)
De inhoud van Modulus blijft ongewijzigd. Indien het Lba-veld Modulus nul bevat dan is de operatie ongeldig en wordt het Lba-v eld Base op nul gesteld.
De inhoud van de general registers blijft onaangetast. De status van de Flags is gewijzigd.

Compare Zero Parameters: Lba-veld(lba)
C++: EqualZero (Lba-veld) ;
Asm: EqualZero Lba-veld
X86: Geen

De inhoud van het Lba-veld wordt vergeleken met de waarde 0.
In de assembler omgeving functioneert het AL register als result-code.
AL = 0 indien de inhoud ≠ 0.
AL = 1 indien de inhoud de waarde 0 heeft.
In een C++ omgeving kan de instructie als volgt worden gebruikt:
if (EqualZero(Lba-veld)) {wordt uitgevoerd indien inhoud = 0}
else {dit wordt uitgevoerd indien inhoud ≠ 0} ;
In een assembler omgeving wordt de instructie als volgt gebruikt:
EqualZero Lba-veld
JE label ;ga naar label indien inhoud = 0
De inhoud van alle general registers, behalve het EAX register, blijft onaangetast. De status van de flags is gewijzigd.

Compare One Parameters: Lba-veld(lba)
C++: EqualOne (Lba-veld) ;
Asm: EqualOne Lba-veld
X86: Geen

De inhoud van het Lba-veld wordt vergeleken met de waarde 1.
In de assembler omgeving functioneert het AL register als result-code.
AL = 0 indien de inhoud ≠ 1.
AL = 1 indien de inhoud de waarde 1 heeft.
In een C++ omgeving kan de instructie als volgt worden gebruikt:
if (EqualOne(Lba-veld)) {wordt uitgevoerd indien inhoud = 1}
else {dit wordt uitgevoerd indien inhoud ≠ 1} ;
In een assembler omgeving wordt de instructie als volgt gebruikt:
EqualOne Lba-veld
JE label ;ga naar label indien inhoud = 1
De inhoud van alle general registers, behalve het EAX register, blijft onaangetast. De status van de flags is gewijzigd.

Compare Equal **Parameters: Lba-veld1(lba), Lba-veld2(lba)**
C++: EqualTo (Lbaveld1,Lba-veld2)
Asm: EqualTo Lbaveld1,Lba-veld2
X86: geen

Indien de inhoud van het eerste Lba-veld gelijk is aan de inhoud van het tweede Lba-veld, dan is de conditie true en zal als zodanig worden behandeld in een C++ programma. In een assembler programma wordt de zeroflag gezet indien de inhoud van het eerste Lba-veld gelijk is aan de inhoud van het tweede Lba-veld.
Indien de Lba-velden ongelijk zijn van lengte dan wordt het kortste veld geacht te zijn aangevuld met nullen.
In een C++ omgeving kan de instructie als volgt worden gebruikt:

```
if ( EqualTo(Lba-veld1,Lba-veld2)) {dit wordt uitgevoerd indien
                                    de inhoud gelijk is}
     else {dit wordt uitgevoerd indien de inhoud ongelijk is} ;
```

In een assembler omgeving wordt de instructie als volgt gebruikt:

```
EqualTo   Lba-veld1,Lba-veld2  ;ga naar label indien inhoud
JE        label                ;gelijk is
```

De inhoud van alle general registers, behalve het EAX register, blijft onaangetast.

Compare Less Than **Parameters: Lba-veld1(lba), Lba-veld2(lba)**
C++: LessThan (Lba-veld1,Lba-veld2)
Asm: LessThan Lba-veld1,Lba-veld2
X86: geen

Indien de inhoud van het eerste Lba-veld kleiner is dan de inhoud van het tweede Lba-veld, dan is de conditie true en zal als zodanig worden behandeld in een C++ programma. In een assembler programma wordt de zeroflag gezet indien de inhoud van het eerste Lba-veld kleiner is dan de inhoud van het tweede Lba-veld.
Indien de Lba-velden ongelijk zijn van lengte dan wordt het kortste veld geacht te zijn aangevuld met nullen.
In een C++ omgeving kan de instructie als volgt worden gebruikt:

```
if (LessThan(Lba-veld1,Lba-veld2)) {wordt uitgevoerd indien
      inhoud 1e Lba-veld kleiner dan inhoud 2e Lba-veld}
else  {dit wordt uitgevoerd indien inhoud eerste Lba-veld
      groter is dan of gelijk is aan tweede Lba-veld} ;
```

In een assembler omgeving wordt de instructie als volgt gebruikt:

```
LessThan  Lba-veld1,Lba-veld2
JE        label     ;ga naar label indien inhoud 1e  lba-veld
                    ;kleiner is daninhoud van het 2e lba-veld
```

De inhoud van de general registers blijft ongewijzigd, behalve EAX

Bit test **Parameters: Lba-veld(lba),Bitnr(int)**
C++: **Bt (Lba-veld,Bitnr) ;**
Asm: **BTM Lba-veld,Bitnr**
X86: **BT**

Lba-veld bevat het adres van het Lba-veld en Bitnr bevat, als 32-bit integer, het bitnummer van de bit waarvan de waarde moet worden getest. Indien het bitnummer de lengte van het Lba-veld (in aantal bits) te boven gaat, dan is de instructie ongeldig en wordt deze niet uitgevoerd. Dit wordt (in een C++ omgeving) niet gesignaleerd. In de assembler omgeving functioneert het AL register als result-code.
AL = 0 indien de geselecteerde bit de waarde 0 heeft.
AL = 1 indien de geselecteerde bit de waarde 1 heeft.
AL = 9 indien de instructie ongeldig is.
In een C++ omgeving kan het resultaat van de instructie als volgt worden verwerkt:

if (Bt(Lba-veld,Bitnr)) { dit wordt uitgevoerd indien de bit = 1}
else {dit wordt uitgevoerd indien de bit = 0} ;

In een assembler omgeving wordt de instructie als volgt gebruikt:

```
BTM Lba-veld,Bitnr
JE  label  ;ga naar label indien bit = 1
```

De inhoud van alle general registers, behalve het EAX register, blijft onaangetast. De status van de flags is gewijzigd.

Number Of Significant Bits **Parameters: Nbits(intAd),Lba-veld(lba)**
C++: **Nosb (Nbits,Lba-veld) ;**
Asm: **NOSB Nbits,Lba-veld**
X86: **geen**

Het aantal betekenisvolle bits van het Lba-veld wordt bepaald en dit aantal wordt in de 32-bits integer geplaatst waarvan het adres in Nbits staat.
Het aantal betekenisvolle bits is het totale aantal bits van het Lba veld verminderd met het aantal voorlopende bits dat op nul staat.
Het Lba-veld blijft ongewijzigd. De inhoud van alle general registers blijft onaangetast evenals de status van de flags.

Special LBA Operations

Greatest Common Divisor Parameters: Gcd(lba),Factor1(lba),Factor2(lba)

C++:	**Gcd**	**(Gcd,Factor1,Factor2)**
Asm	**GCD**	**Gcd,Factor1,Factor2**
X86:	**geen**	

Deze functie berekent de grootste gemene deler (ggd)[3] van Factor1 en Factor2. De drie parameters zijn adressen van Lba-velden. De beide factoren mogen niet nul zijn. Indien een van de factoren of beide nul is, dan is de operatie ongeldig en wordt Gcd op nul gezet. De inhoud van alle general registers blijft onaangetast.

Uitgebreid Euclidisch algoritme[4]

Parameters: Pos(lba),Neg(lba),Factor1(lba),Factor2(lba)

C++:	**Eucal**	**(pos,neg,f1,f2) ;**
Asm:	**Eucal**	**Pos,Neg,F1,F2**
X86	**geen**	

De operatie Eucal berekent volgens het Uitgebreid Euclidisch Algoritme: xF1 - yF2 = Gcd.
Factor1 en Factor2 (F1 en F2) zijn input parameters. Pos en Neg zijn output parameters. Alle parameters zijn adressen van Lba-velden.
Eerst wordt de grootste gemene deler van F1 en F2 berekend. Daarna worden x en y berekend. Van deze output variabelen is x positief en y is negtief. Omdat LBA geen negative getallen kent, wordt de positieve waarde x in het Lba-veld *Pos* geplaatst en de negatieve waarde y (als positief getal) in het Lba-veld *Neg* geplaatst.
Zonodig worden de waarden van F1 en F2 verwisseld om de berekening xF1 - yF2 = Gcd te laten kloppen.
De LBA werkvelden die nodig zijn bij de berekening worden dynamisch gecreëerd.
De inhoud van alle general registers blijft bewaard.
De inhoud van F1 en F2 kan eventueel verwisseld worden maar blijft verder ongewijzigd.

Square Multiplication Parameters: Result(lba),Multiplier(lba)
C++: Smul (Result,Multiplier)
Asm SMUL Result,Multiplier
X86: geen

Smul berekent het kwadraat van Multiplier en plaatst het resultaat in Result. Result en Multiplier bevatten beiden de adressen van Lba-velden. Indien het Lba-veld Result te klein is om het kwadraat te kunnen bevatten, dan wordt Result geheel op nul gezet. De inhoud van alle general registers blijft onaangetast.

Montgomery Square Multiplication Parameters: Result(lba),Mr(lba),Modulus(lba),ModInv(int),NS(int)
C++: Mgs (Result,Mr,Mod,ModInv,NS)
Asm MGS Result,Mr,Mod,ModInv,NS
X86: geen

Deze functie berekent het kwadraat van Mr (mod Modulus) volgens de Montgomery methode[5]. Result is de output, en Mr is de input. Modulus is de modulus. Deze drie parameters zijn adressen van Lba-velden. De parameters ModInv en NS zijn 32-bits integer velden. ModInv is de modulaire multiplicatieve inverse (van de laatste 32 bits van het tussenresultaat) en NS bevat het aantal shifts dat moet worden uitgevoerd om de gehele Montgomery functie uit te voeren. Deze functie (MGS) wordt aangeroepen vanuit de functie MGM. De parameter NS moet telkens vernieuwd worden omdat NS functioneert als een count-down en deze wordt gedurende de uitvoering van MGS gemodificeerd.
De inhoud van alle general registers blijft onaangetast

Montgomery Multiplication Parameters: Result(lba), Multiplier(lba), Multiplicand(lba),Modulus(lba),ModInverse(int),NS(int)

C++: Mgm (Result,Mr,Mc,Mod,ModInv,NS)

Asm MGM Result,Mr,Mc,Mod,ModInv,NS

X86: geen

Deze functie berekent het product van Mr en Mc (mod Modulus) volgens de Montgomery methode[5]. Result is de output, en Mr en Mc is de input. Modulus is de modulus. Deze vier parameters zijn adressen van Lba-velden. De parameters ModInv en NS zijn 32-bits integer velden. ModInv is de modulaire multiplicatieve inverse (van de laatste 32 bits van het tussenresultaat) en NS bevat het aantal shifts dat moet worden uitgevoerd om de gehele Montgomery functie uit te voeren.

Vanuit de functie MGM wordt herhaaldelijk de functie MGS aangeroepen. NS functioneert daarbij als een count-down teller en wordt gedurende de uitvoering van MGS gemodificeerd. De parameter NS moet daarom bij elke call van MGS worden ververst. De inhoud van alle general registers blijft onaangetast

Modulair Machtsverheffen Parameters:
RC(intAd),Result(lba),Base(lba),Exponent(lba),Modulus(lba)

C++: Modex (RC,Result,Base,Exponent,Modulus)

Asm MODEX RC,Result,Base,Exponent,Modulus

X86: geen

Deze functie berekent Base tot de macht Exponent (mod Modulus) volgens de Montgomery methode[6]. RC is het adres van een integer die de result-code bevat:
RC = 0 operatie is succesvol afgelopen
RC = 8 verkeerde input
RC = 9 berekening afgebroken
Result, Base, Exponent en Modulus zijn alle Lba-velden.
Het Lba-veld Modulus mag niet nul zijn. Verder moet de inhoud van het Lba-veld Base kleiner zijn dan het Lba-veld Modulus.
Indien aan deze voorwaarden niet wordt voldaan, dan is de operatie ongeldig en wordt het Lba-veld Result op nul gezet en wordt RC=8.
Deze functie maakt gebruik van de functies Mgs en Mgm.
Het Lba-veld Exponent wordt aangetast. Het Lba-veld Modulus blijjft ongewijzigd. De inhoud van alle general registers blijft onaangetast

Pseudo Random Number Generator Parameter: Result(lba)

C++: Prng (Result)

Asm PRNG Result

X86: geen

Deze functie genereert een string pseudo random bits en plaatst deze in Result. Pseudo random houdt in dat de string bits gedetermineerd is en niet volledig willekeurig is. De output voldoet evenwel aan de (verouderde) norm volgens FIPS 140-1[7].
Alle dwords van Result worden gevuld met bits, behalve het laatste dword (met de hoogste waarde).
De inhoud van alle general registers blijft onaangetast.

LBA rapportages

Report LBA variabele Parameter: LBA-name
C++: Report (LBA-name) ;
Asm: REPORT LBA-name
Met het Report statement wordt de inhoud van een LBA variabele getoond in de program-log.
De vorm waarin gerapporteerd wordt is:
"LBA-name" = xx.xxx.xxx.xxx.xxx....
De numerieke inhoud van de LBA-variabele wordt in decimale vorm getoond, in groepen van drie cijfers, telkens gescheiden door een scheidingsteken. Dit scheidingsteken kan een komma of een punt zijn, afhankelijk van de taal die op dat moment aktief is.

Toon messages Parameters: message-nr,integer
C++: Mes (msg-nr,integer) ;
Asm: Msg msg-nr,integer
Met het message statement wordt een vooraf gedefinieerde message met msg-nr. getoond in de program-log. Gelijk met de message wordt de inhoud van integer (optioneel) getoond. Dit mag een getal zijn met een maximale waarde van 2^{32} - 1.
Bij de assembler-vorm mag integer de specificatie van een general register zijn.

Referenties:

1. *Intel documentation*
http://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf

2. Theo Kortekaas (http://tonjanee.home.xs4all.nl/SSAbeschrijving.pdf) blz. 3 *Klassieke methode*

3. *Grootste Gemene Deler* (https://nl.wikipedia.org/wiki/Grootste_gemene_deler)

4. *Uitgebreid algoritme van Euclides* (https://nl.wikipedia.org/wiki/Uitgebreid_algoritme_van_Euclides)

6. Theo Kortekaas *Montgomery vermenigvuldiging* (http://tonjanee.home.xs4all.nl/Montgomerybeschrijving.pdf) blz. 6

7. Theo Kortekaas *Modulair Machtsverheffen* (http://tonjanee.home.xs4all.nl/Montgomerybeschrijving.pdf) blz.11

8. *FIPS 140-1 Federal Information Processing Standards* (https://en.wikipedia.org/wiki/FIPS_140)

Auteur: Theo Kortekaas
Email: t.kortekaas@xs4all.nl
Web-pagina's: http://tonjanee.home.xs4all.nl