Opdrachten week 2 lab 2 – MSP430 Assembler

In de vorige lessen heb je C-code geschreven voor, en getest op, de SimpleLink™ Wi-Fi® CC3220S LaunchPad™. In de theorielessen heb je geleerd hoe een eenvoudige 8-bit CPU werkt[1]. In deze les gaan we dieper in op de werking van de MSP430G2553, de 16-bit µC die je bij EMS10 hebt leren kennen. Daarbij maken we gebruik van de EXP430G2 LaunchPad.

Je gaat deze les leren:

- wat de verschillen en overeenkomsten zijn met de in het theorieboek gepresenteerde computerarchitectuur en de MSP430-architectuur;
- hoe je een eenvoudige assemblertaalprogramma kunt schrijven voor de MSP430G2553;
- hoe je een functie in assembler kunt implementeren zodanig dat je deze functie vanuit C kunt aanroepen;
- hoe je vanuit C-code een in assembler geschreven functie kunt aanroepen;
- hoe het functie call en return mechanisme gerealiseerd wordt door het gebruik van een stack en een stack pointer.

De in paragraaf 7.4 van het theorieboek besproken CPU heeft aantal general purpose registers en de specifieke registers: program counter, statusregister en stack pointer. De CPU van de MSP430G2553 heeft dezelfde registers. In de MSP430 CPU[2] kunnen alle registers gebruik maken van de ALU. De MSP430 heeft 16 registers van 16 bits breed. Register R0 is de Program Counter, R1 is de Stack Pointer, R2 is het statusregister, R3 is een register waarmee constante getallen gegenereerd kunnen worden (dit register wordt verder niet besproken), R4 t/m R15 kunnen als general purpose registers gebruikt worden.

2.2.1 Vergelijk figuur 7.30 en figuur 7.35 uit het theorieboek met figuur 3.1 uit de *MSP430x2xx Family User's Guide*. Beantwoord de volgende vragen:

A Uit figuur 7.35 van het boek blijkt dat de ALU alleen resultaten kan wegschrijven in de general purpose registers. De MSP430 kan het resultaat van een ALU-bewerking ook wegschrijven naar een general purpose register maar heeft daarnaast nog andere mogelijke bestemmingen. Welke zijn dit?

---

[1] Zie Leo van Moergestel. *Computersystemen en embedded systemen*. 4de ed. Boom, 2016. ISBN: 9-789058-754233, paragraaf 7.4.

[2] Zie *MSP430x2xx Family User's Guide*. Texas Instruments Incorporated. 2013. URL: https://www.ti.com/lit/pdf/slau144, hoofdstuk 3.

B Wat gebeurt er als we tijdens het uitvoeren van een programma een waarde van de program counter aftrekken?

C In het control word dat weergegeven is in figuur 7.35 van het boek worden vier bits gebruikt om te selecteren welke registers op de A en B ingangen van de ALU worden geplaatst. Hoeveel bits zijn er in het control word van de MSP430 nodig om te selecteren welke registers op de src en dst ingangen van de ALU worden geplaatst?

De MSP430 CPU kent 51 verschillende instructies. Per instructie kunnen er nog maximaal twee zogenoemde operands worden meegegeven. De machinecode-instructies van de MSP430 bestaan uit 16, 32 of 48 bits. Instructies in machinecode bestaan uit een aantal nullen en enen en zijn dus lastig leesbaar. Wij maken daarom gebruik van zogenoemde assemblercode. Een instructie in assemblercode kan door een zogenoemde assembler, één op één, vertaald worden in machinecode. Elke processorfamilie heeft zijn eigen machinetaal en dus ook zijn eigen assemblertaal. Een regel assemblercode uit een assemblertaalprogramma voor de MSP430 is als volgt opgebouwd:

```
label mnemonic src,dst ;commentaar
```

De verschillende delen worden hieronder verklaard:

- `label`. Een label is optioneel en kan gebruikt worden om naar een bepaalde instructie te refereren.
- `mnemonic`. De mnemonic[3] is een afkorting waarmee wordt aangegeven wat de instructie doet. Bijvoorbeeld de instructie `add` voert een optelling (addition) uit en de instructie `dec` verlaagt iets met één (decrement).
- `src,dst`. De source (bron) en destination (bestemming) operands geven aan waarop de instructie wordt uitgevoerd. De verschillende manieren waarop de source en destination operands kunnen worden gedefinieerd worden addressing modes genoemd. De MSP430 heeft zeven verschillende addressing modes waar wij er in slechts drie van gebruiken[4]:

[3] Het Engelse woord mnemonic betekent geheugensteuntje.

[4] Zie voor een compleet overzicht *MSP430x2xx Family User's Guide*. Texas Instruments Incorporated. 2013. URL: https://www.ti.com/lit/pdf/slau144, paragraaf 3.3.

- Register mode. In dit geval is de operand een register van de MSP430. Bijvoorbeeld: de instructie `add r4,r7` telt de inhoud van register R4 op bij de inhoud van register R7.
- Absolute mode. In dit geval is de operand een geheugenadres in de memorymap van de MSP430.
  Bijvoorbeeld: de instructie `add &0x0300,&0x0200` telt de inhoud van geheugenadres 0x0300 op bij de inhoud van geheugenadres 0x0200.
- Immediate mode. In dit geval is de operand een constant getal. Deze addressing mode kan (uiteraard) alleen als source operand gebruikt worden. Bijvoorbeeld: de instructie `add #0x0123,r7` telt het getal 0x0123 op bij de inhoud van register R7.

Sommige instructies hebben alleen een destination operand. Bijvoorbeeld: de instructie `dec r7` verlaagd de inhoud van register R7 met één. Er zijn ook enkele instructies die helemaal geen operand hebben. Bijvoorbeeld: de instructie `eint` kan gebruikt worden om de interrupts vrij te geven.

- `;commentaar`. Alle tekst na een puntkomma wordt gezien als commentaar.

Assemblercode is overigens niet hoofdlettergevoelig. De instructie `add r4,r7` kun je dus ook schrijven als `ADD R4,R7`.

De MSP430 CPU kent 51 verschillende instructies waar wij er in slechts twintig van gebruiken[5]. De instructies die we deze les gebruiken zijn in te delen in drie groepen:

- De rekenkundige en logische instructies. Met behulp van deze instructies kunnen rekenkundige en logische bewerkingen op operands worden uitgevoerd. Zie tabel 1. Deze instructies werken per default op alle 16 bits van de operands. Dit kun je benadrukken door de suffix `.W` achter de instructie te plaatsen. De instructie `add r4,r7` kun je dus ook schrijven als `add.w r4,r7`. Het is echter ook mogelijk om deze instructies alleen het least significant byte (lsb) van de operand te laten gebruiken. Dit kun je bewerkstelligen door de suffix `.B` achter de instructie te plaatsen. De instructie `add.b r4,r7` telt dus het lsB (bit 0 t/m bit 7) van register R4 op bij het lsB van register R7. Het most significant byte (bit 8 t/m 15) van register R7 wordt nul gemaakt. De meeste instructies uit tabel 1 werken de statusbits V, N, Z en C bij[6]. De betekenis van deze bits wordt beschreven op pagina 134 van je boek. Deze bits worden opgeslagen in

[5] Zie voor een compleet overzicht *MSP430x2xx Family User's Guide*. Texas Instruments Incorporated. 2013. URL: https://www.ti.com/lit/pdf/slau144, paragraaf 3.4.6.

[6] Dit geldt niet voor de instructies: `clr`, `mov`, `bic` en `bis`. Deze instructies beïnvloeden de statusbits niet.

het statusregister R2. De laatste instructie uit tabel 1 produceert geen resultaat maar werkt alleen de statusbits bij. Deze instructie is handig om te gebruiken voorafgaande aan een spronginstructie.

- De spronginstructies. Met behulp van deze instructies kun je beslissingen nemen en herhalingen toepassen in een assemblertaalprogramma. Zie tabel 2. Deze instructies gebruiken een label als operand. Als er wordt gesprongen, dan wordt er gesprongen naar de instructie die voorafgegaan wordt door dit specifieke label. In listing 1 wordt getoond hoe je in assemblertaal de uitvoering van een stukje code tien keer kunt laten herhalen. In listing 2 wordt getoond hoe je in assemblertaal een stukje code kunt laten uitvoeren als een bepaalde waarde nul is en een ander stukje code kunt laten uitvoeren als dit niet zo is.
- De instructies waarmee functies geïmplementeerd kunnen worden. Wat we in Python en C een functie noemen wordt in assemblertaal een subroutine genoemd. De MSP430 heeft twee instructies die speciaal bedoeld zijn voor het implementeren van subroutines. Zie tabel 3. We komen hier later in deze les nog uitgebreid op terug.

Tabel 1: Enkele rekenkundige en logische instructies van de MSP430.

mnemonic	operand(s)	betekenis	operatie
`add`	`src,dst`	addition	`dst = dst + src`
`sub`	`src,dst`	substraction	`dst = dst - src`
`inc`	`dst`	increment	`dst = dst + 1`
`dec`	`dst`	decrement	`dst = dst - 1`
`clr`	`dst`	clear	`dst = 0`
`mov`	`src,dst`	move	`dst = src`
`bic`	`src,dst`	bit clear	`dst = dst AND NOT src`
`bis`	`src,dst`	bit set	`dst = dst OR src`
`xor`	`src,dst`	exor	`dst = dst EXOR src`
`tst`	`dst`	test	compare `dst` with `0`

2.2.2 Je hebt nu voldoende kennis om een eenvoudig assemblertaalprogramma op de MSP430 te implementeren. Start Code Composer Studio en kies de menuoptie Project › New CCS Project... . Selecteer vervolgens de juiste Target: MSP430G2553 en geef het project de naam `asm01`. Kies nu onder Project templates and examples voor Empty

Tabel 2: Enkele sprong instructies van de MSP430.

mnemonic	operand	betekenis
`jmp`	`label`	spring naar `label`
`jz`	`label`	spring naar `label` als statusbit Z één is
`jnz`	`label`	spring naar `label` als statusbit Z nul is
`jc`	`label`	spring naar `label` als statusbit C één is
`jnc`	`label`	spring naar `label` als statusbit C nul is

Tabel 3: Instructies van de MSP430 om subroutines mee te implementeren.

mnemonic	operand	betekenis
`call`	`dst`	roep de subroutine op geheugenadres `dst` aan
`ret`		return vanuit deze subroutine

```
        mov.w   #10,r15  ; laad teller
herhaal ; plaats hier de code die 10x herhaald moet worden
        dec.w   r15      ; verlaag teller
        jnz     herhaal  ; herhaal zolang teller niet nul is
```

Listing 1: De implementatie van een herhaling in MSP430 assemblercode.

```
        tst.w   r4       ; vergelijk R4 met 0
        jnz     n_nul
        ; code die uitgevoerd wordt als R4 nul is
        jmp     einde
n_nul   ; code die uitgevoerd wordt als R4 niet nul is
einde
```

Listing 2: De implementatie van een beslissing in MSP430 assemblercode.

Assembly-only Project en klik op Finish. Er wordt nu een leeg assemblertaalprogramma aangemaakt en geopend waarin:

- de reset vector wordt geïnitialiseerd zodat het programma start als op de resetknop wordt gedrukt;
- de stack pointer wordt geïnitialiseerd zodat het programma indien nodig de stack kan gebruiken;

- de watchdog timer wordt gestopt zodat het programma niet periodiek wordt gereset door deze timer.

Het programma bevat een aantal regels waarbij de mnemonic met een punt begint. Dit zijn geen instructies voor de MSP430 maar zogenoemde directives voor de assembler[7]. Wij gaan hier verder niet op in. Kopieer nu het programmagedeelte gegeven in listing 3 in de main loop van het gegenereerde assemblertaalprogramma.

A Voer het programma nu uit op een MSP430G2553 microcontroller die geplaatst is op je EXP430G2 LaunchPad. Zorg ervoor dat alle jumpers correct geplaatst zijn. Als het goed is, knippert de groene led als je het programma uitvoert.

B De eerste drie regels uit listing 3 zijn gekopieerd uit paragraaf 5.2.5.2 van de *MSP430x2xx Family User's Guide*[8]. Zoek deze code op en voeg het daar gegeven commentaar toe aan je eigen code.

C Wat was ook alweer de betekenis van het gebruik van de tekens # en & bij een operand?

D Zorg ervoor dat je het programma uit listing 3 volledig begrijpt en voeg na elke instructie commentaar toe.

E Als gegeven is dat de instructie `dec.w` één klokperiode duurt en dat de instructie `jnz` twee klokperioden duurt, hoe vaak knippert de led dan per minuut? Verifieer je berekening door gedurende een halve minuut te tellen hoe vaak de led knippert.

2.2.3 Kopieer het project `asm01` en noem dit project `asm02`. Pas het programma nu aan zodat de rode led op je EXP430G2 LaunchPad knippert in plaats van de groene led. Zorg er ook voor dat de led twee keer zo snel knippert.

2.2.4 Kopieer het project `asm01` en noem dit project `asm03`. Pas het programma nu aan zodat de groene led twee keer zo traag knippert. Dit is minder gemakkelijk dan het lijkt. De constante 100000 past namelijk niet in een 16-bits register[9]. Bedenk een generieke

[7] Zie *MSP430 Assembly Language Tools User's Guide*. Texas Instruments Incorporated. 2018. URL: https://www.ti.com/lit/pdf/slau131, hoofdstuk 5.

[8] Zie https://www.ti.com/lit/pdf/slau144.

[9] Het is wel vreemd dat de MSP430-assembler geen foutmelding of waarschuwing geeft als je probeert om de constante 100000 in een register te laden. De constante 100000 is uitgedrukt in het hexadecimale stelsel

```
            clr.b   &DCOCTL
            mov.b   &CALBC1_1MHZ,&BCSCTL1
            mov.b   &CALDCO_1MHZ,&DCOCTL

            bis.b   #0x01,&P1DIR
loop        xor.b   #0x01,&P1OUT
            mov.w   #50000,r15
telaf       dec.w   r15
            jnz     telaf
            jmp     loop
```

Listing 3: Main loop van een assemblertaalprogramma voor de MSP430 dat een ledje laat knipperen, zie asm01.asm.

oplossing waarmee je de led naar believen 2, 3, 4, ..., N keer zo traag kan laten knipperen.

2.2.5 Kopieer het project asm01 en noem dit project asm04. Pas het programma nu aan zodat de groene led één keer per seconde knippert.

Het is in de praktijk niet handig om een heel programma in assemblertaal te schrijven. Elke processorfamilie heeft namelijk zijn eigen assemblertaal. Daarom gebruiken we in de praktijk meestal de programmeertaal C en laten we het vertalen van deze code naar assemblertaal over aan de C-compiler. Toch is het soms wel handig om delen van een programma in assembler te schrijven. Bijvoorbeeld als we de code handmatig willen optimaliseren voor snelheid of geheugengebruik of als we instructies willen uitvoeren die niet in de programmeertaal C zijn uit te drukken.

In het vervolg van deze les ga je leren hoe je vanuit een C-programma een functie die geschreven is in assembler (als een subroutine) kunt aanroepen. Ook ga je leren hoe het functie call en return mechanisme in een processor gerealiseerd wordt door het gebruik van een stack en een stack pointer.

2.2.6 Kies de menuoptie Project > New CCS Project... in Code Composer Studio. Selecteer vervolgens de juiste Target: MSP430G2553 en geef het project de naam mul01. Kies nu onder Project templates and examples voor Empty Project (with main.c) en klik op

gelijk aan 0x186A0. Als de waarde in een register wordt geladen, dan wordt het hoogste bit weggegooid en wordt het getal 0x86A0, oftewel decimaal 34464,in het register geladen.

Finish. Kopieer de code die gegeven is in listing 4 in de file `main.c`. De variabele `a` is **`volatile`** gedefinieerd om te voorkomen dat deze variabele door de compiler weggeoptimaliseerd wordt. Het keyword **`extern`** voor de declaratie van de functie `mul` geeft aan dat deze functie in een andere file is gedefinieerd. De functie `mul` kan nu in C maar ook in assembler gedefinieerd worden. In dit geval kiezen we ervoor om deze functie in assembler te definiëren. Kies de menuoptie File › New › Other..., selecteer vervolgens General › File en klik op Next. Vul als File name `mul.asm` in. Kopieer de code die gegeven is in listing 5 in de file `mul.asm`. De MSP430 heeft geen machinecode instructie waarmee een vermenigvuldiging kan worden uitgevoerd. Daarom voert de functie `mul` de vermenigvuldiging uit door schuifbewerkingen en optellingen uit te voeren. Daarbij worden twee instructies (`clrc` en `rrc`) gebruikt die hierboven niet zijn behandeld. Het is op dit moment niet nodig dat je de werking van deze functie begrijpt. De argumenten 130 en 140 die bij de aanroep van de functie `mul` in de C-code worden doorgegeven worden door de C-compiler in de registers R12 en respectievelijk R13 geplaatst. Dit is vastgelegd in een zogenoemde Application Binary Interface (ABI) definitie[10]. De returnwaarde wordt vanuit de `mul` functie teruggegeven via register R12. Ook dit is vastgelegd in de ABI. We gaan in het vervolg van deze opdracht onderzoeken hoe het functie call en return mechanisme exact werkt.

```
#include <msp430.h>

extern int mul(int x, int n);

int main(void)
{
    WDTCTL = WDTPW | WDTHOLD;

    volatile int a = 0;

    a = mul(130, 140);

    while (1);
}
```

Listing 4: Een C-programma waarin de externe functie `mul` wordt aangeroepen, zie `mul.c`.

[10] Zie *MSP430 Embedded Application Binary Interface Application Report*. Texas Instruments Incorporated. 2013. URL: http://www.ti.com/lit/pdf/slaa534, hoofdstuk 3.

```
        .def    mul
        .text
mul     clr.w   r14
next    clrc
        rrc.w   r12
        jnc     no_add
        add.w   r13,r14
no_add  add.w   r13,r13
        tst.w   r12
        jne     next
        mov.w   r14,r12
        ret
```

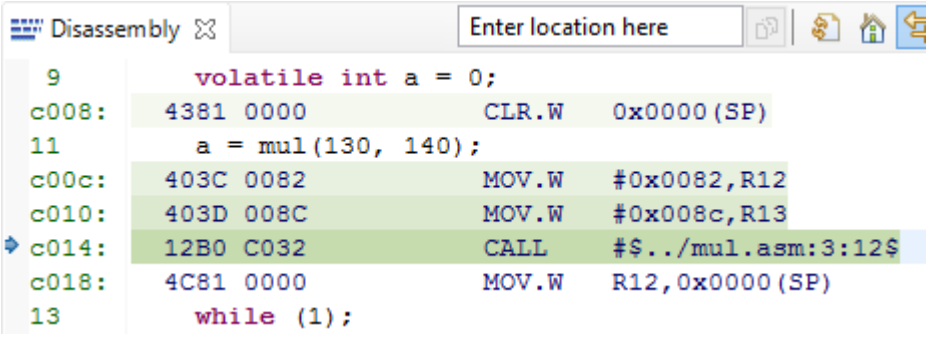
Listing 5: De assemblertaal implementatie van de functie `mul`, zie `mul.asm`.

A Voer het programma uit door achtereenvolgens op de knoppen en te drukken. Druk op de knop en controleer in het Variables window of de variabele a de waarde $130 \times 140 = 18200$ heeft gekregen.

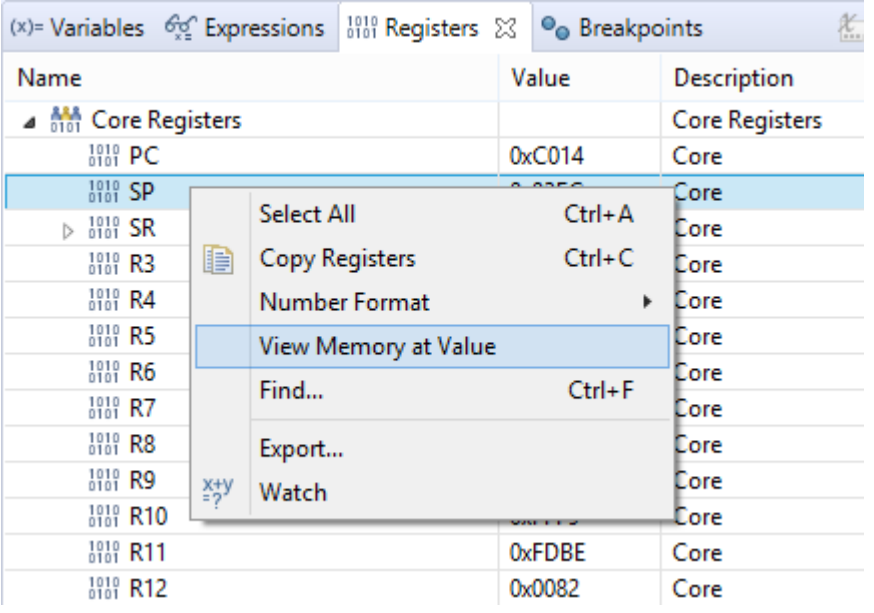
B Start het programma opnieuw door op de knop te drukken. Kies nu voor de menuoptie View > Disassmbly. In dit window kun de assemblercode zien die de C-compiler heeft aangemaakt. Op welk adres bevindt zich de `call`-instructie waarmee de functie (subroutine) `mul` wordt aangeroepen? Open de tab Registers en open de Core Registers. Wat is de waarde van de PC en wat is de waarde van de SP op dit moment?

C Je kunt stap voor stap door de assemblercode heenstappen door herhaaldelijk op de knop te drukken. Stap door het gedisassembleerde programma tot de regel met de `call`-instructie, zie figuur 1. Wat is de waarde van de PC op dit moment?

D Selecteer nu de regel in de tab Registers waarop de SP wordt weergegeven en klik op je rechter muisknop. Kies vervolgens voor View Memory At Value, zie figuur 2. In het Memory Browser window dat nu wordt geopend kunnen we zien welke twee getallen er op de stack staan. Teken de stack en de stackpointer op dit moment op een vergelijkbare manier als in paragraaf 7.4.4 van je boek. Als het goed is ziet je tekening er ongeveer uit zoals figuur 3.



Figuur 1: Het Disassembly window net voordat de subroutine `mul` wordt aangeroepen.

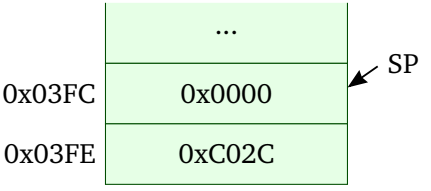


Figuur 2: Met dit contextmenu kun je de inhoud van de stack bekijken.

E Druk nu weer op de knop zodat de `call`-instructie uitgevoerd wordt. Teken nu opniew de stack en de stackpointer op dit moment. Welke waarde staat er nu bovenaan de stack? Begrijp je waarom deze waarde daar staat?

F Zet een breakpoint op de `ret`-instructie in `mul.asm` en run het programma tot aan dit breakpoint. Druk op de knop zodat de `ret`-instructie uitgevoerd wordt. Teken nu opniew de stack en de stackpointer op dit moment. Welke waarde staat er nu in de PC? Begrijp je waar deze waarde vandaan komt?

G Druk nogmaals op de knop zodat de returnwaarde van de functie `mul` wordt weggeschreven in de variabele a. Teken nu opnieuw de stack en de stackpointer op dit moment. Waar slaat de C-compiler de lokale variabele a op?



Figuur 3: De stack net voordat de subroutine `mul` wordt aangeroepen.

2.2.7

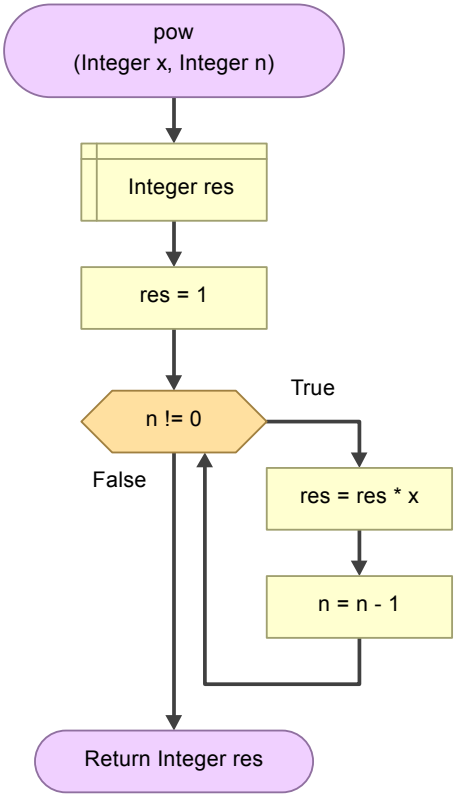
A Schrijf nu een subroutine in assemblertaal genaamd `pow` om x^n uit te rekenen. De parameters x en n zijn 16-bits integers en je mag ervan uitgaan dat het resultaat ook past in 16 bits. Een eenvoudig algoritme waarmee je x^n uit kunt rekenen is gegeven in figuur 4. Zorg ervoor dat je deze subroutine vanuit C-code als een functie kunt aanroepen. Schrijf ook een testprogramma in C om 3^9 uit te rekenen: `b = pow(3, 9);`. Het juiste antwoord is 19683. Maak bij het implementeren van de subroutine `pow` gebruik van de al eerder in listing 5 gegeven subroutine `mul`. Als je de subroutine `mul` vanuit assemblercode wilt aanroepen, dan kan dat met de code `call #mul`. Let daarbij op het gebruik van het teken #.

B Zet een breakpunt in de subroutine `mul` en teken de stack en de stackpointer op dit moment. Verklaar de inhoud van de stack met behulp van het Disassembly window.

De volgende opgave behoort niet tot de verplichte stof voor EMS20 en is bedoeld voor studenten die zich verder in het programmeren in assemblertaal willen bekwamen.

2.2.8

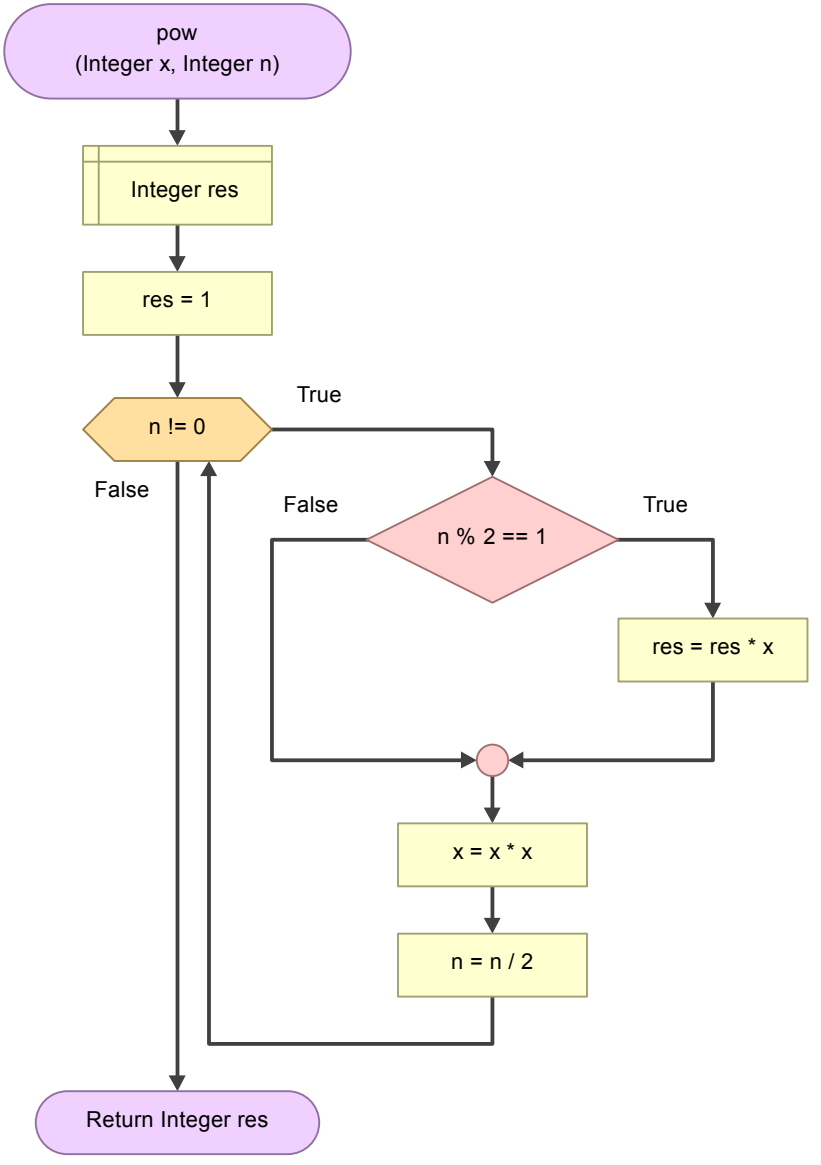
Het algoritme dat gegeven is in figuur 4 is niet het snelste algoritme om x^n te berekenen. Een sneller algoritme is gegeven in figuur 5. Schrijf nu een subroutine in assemblertaal genaamd `pow2` om x^n uit te rekenen volgens het algoritme dat gegeven is in figuur 5. De parameters x en n zijn 16-bits integers en je mag ervan uitgaan dat het resultaat ook past in 16 bits. Zorg ervoor dat je deze subroutine vanuit C-code als een functie kunt aanroepen. Schrijf ook een testprogramma in C om 3^9 uit te rekenen: `c = pow2(3, 9);`. Het juiste antwoord is 19683. Maak bij het implementeren van de



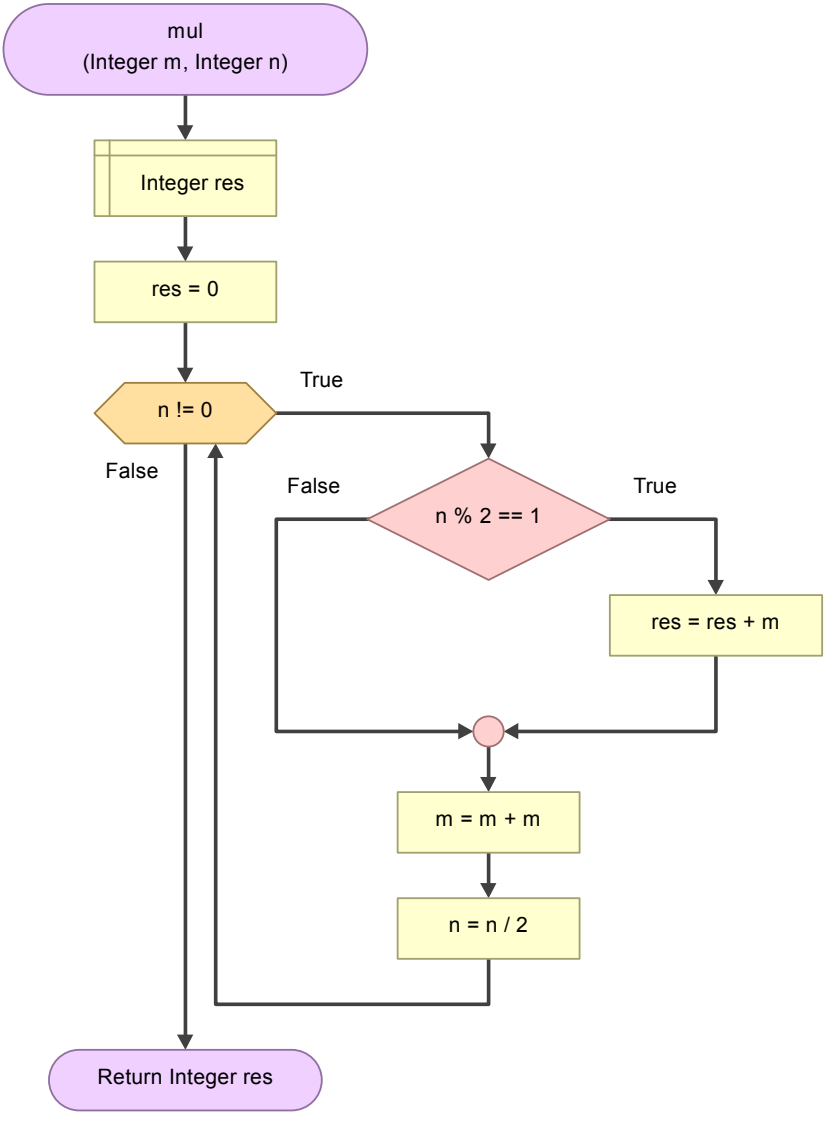
Figuur 4: Een eenvoudig algoritme om x^n te berekenen.

subroutine `pow2` gebruik van de al eerder in listing 5 gegeven subroutine `mul`. De vergelijking `n % 2 == 1` kan in MSP430 assemblercode worden uitgevoerd door eerst de `rrc`-instructie uit te voeren en daarna het C statusbit te testen. Als voordat de `rrc`-instructie het C statusbit gereset wordt (met de `clrc`-instructie) dan wordt meteen de operatie `n = n / 2` uitgevoerd. Zie *MSP430x2xx Family User's Guide* pagina 104 voor gedetailleerde informatie over de `rrc`-instructie.

Het algoritme gegeven in figuur 5 lijkt veel op het algoritme dat gebruikt is om de vermenigvuldiging mee te implementeren. Het algoritme waarvan de assemblercode is gegeven in listing 5 is gegeven in figuur 6. Je kunt dus eventueel deze assemblercode als voorbeeld nemen bij het implementeren van de subroutine `pow2`.



Figuur 5: Een sneller algoritme om x^n te berekenen.



Figuur 6: Een algoritme om $m \times n$ te berekenen.