**Project Assembly Rekenmachine**

Onderzoeks- en ontwerpdocument

|  |  |
| --- | --- |
| **Versie** | 1.0 |
| **Datum** | 23/04/2020 |

|  |  |
| --- | --- |
| **Auteur** | **Studentnummer** |
| Renée van Liempt | 2110780 |

**Begeleiders**: Joan Schrasser

Inhoudsopgave

[Inleiding 3](#_Toc43475324)

[Deelvraag 1 4](#_Toc43475325)

[8-bit AVR core 4](#_Toc43475326)

[Registers 4](#_Toc43475327)

[Stack Pointer 5](#_Toc43475328)

[Status Register 5](#_Toc43475329)

[Arithmetic Logic Unit (ALU) 6](#_Toc43475330)

[Memory 8](#_Toc43475331)

[Program Memory 8](#_Toc43475332)

[EEPROM Data Memory 9](#_Toc43475333)

[SRAM Data Memory 9](#_Toc43475334)

[Deelvraag 2 10](#_Toc43475335)

[Standaarden 10](#_Toc43475336)

[General Purpose Working Registers 10](#_Toc43475337)

[SRAM 11](#_Toc43475338)

[Programma 13](#_Toc43475339)

[Knoppen 14](#_Toc43475340)

[Getal knop invoer 15](#_Toc43475341)

[Operator knop invoer 16](#_Toc43475342)

[Som berekenen 17](#_Toc43475343)

[Som loop 18](#_Toc43475344)

[Operator 1 18](#_Toc43475345)

[Operator 2 19](#_Toc43475346)

[Antwoord opslaan 20](#_Toc43475347)

# Inleiding

Er zal een rekenmachine gerealiseerd worden in Assembly. De architectuur die gebruikt wordt is 8-bit AVR. Dit document bevat alle voorgaande onderzoek. Hierin worden de volgende vragen beantwoord:

* Welke functionaliteiten worden belangrijk voor dit project?
* Hoe wordt het programma ingedeeld?

De eerste vraag verdiept zich wat 8-bit AVR allemaal aanbied qua functionaliteiten, en hoe dit het beste gebruikt kan worden.

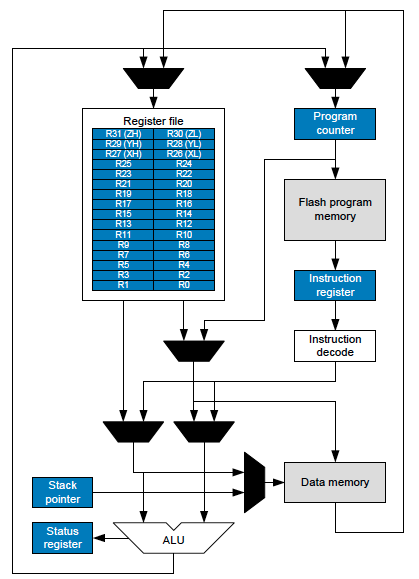
De tweede vraag verdiept zich in het ontwerp van dit project. Assembly is moeilijk om leesbaar en gestructureerd te houden. Een duidelijk ontwerp is van belang bij het schrijven van deze code

# Deelvraag 1

Welke functionaliteiten worden belangrijk voor dit project?

## 8-bit AVR core

De core van een 8-bit AVR systeem kan als volgt uit worden gebeeld.



Figuur 1 Block Diagram van de AVR Architectuur

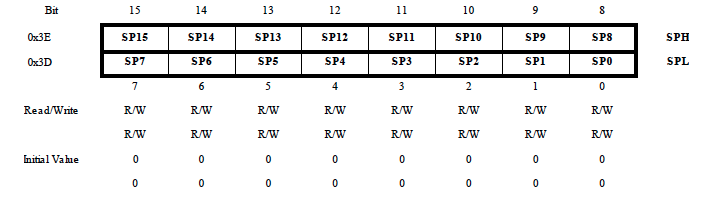
Het systeem gebruikt een Harvard architectuur en kan met behulp van pipelining in iedere klokcyclus een instructie uitvoeren als de instructies efficient geschreven zijn

### Registers

Er zijn 32 8-bit General Purpose Working registers aanwezig in de Register file. Dit zijn register die snel beschikbaar zijn, zonder in RAM te hoeven zoeken. Zes van deze registers kunnen gebruikt worden als drie 16-bit registers. Deze X-, Y-, en Z-registers kunnen ondermeer gebruikt worden voor Data Space adressering.

### Stack Pointer

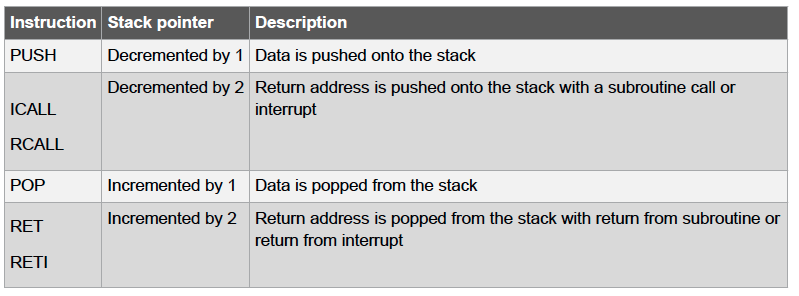
De stack wordt voornamelijk gebruikt voor het opslaan van tijdelijke data zoals lokale variabelen en return adressen. Het groeit van hogere adressen naar lagere adressen. De Stack Pointer wijst altijd naar het onderste adres van de stack.



Figuur 2 Stack Pointer

Dit adres zit in I/O space en heeft een eigen instructieset.

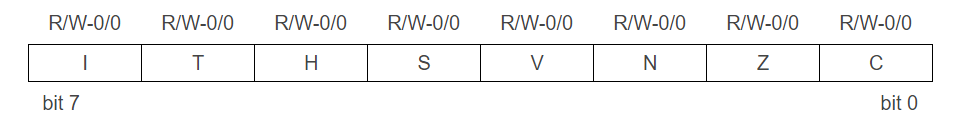
Tabel 1 Stack Instructies



Met push wordt de de Stack Pointer verlaagd en is er meer ruimte op de stack. Het defineren van de stack in de data SRAM moet voor het aan zetten van de interrupts. Deze staat namelijk in het begin altijd op het laatste adres van de SRAM en moet naar het begin van de SRAM gezet worden.

### Status Register

Het Status Register bewaart informatie over de laatst uitgevoerde instructie door het Arithmetic Logic Unit. Deze wordt dus bij iedere arithmatische instructie geüpdatet.



Figuur 3 Status Register

**Bit 7 I : Global Interrupt Enable**

Voordat er gebruik gemaakt kan worden van interrupts, moet deze bit gezet worden. Vervolgens kunnen interrupts indiviueel gezet worden in een apart register.   
Deze bit wordt door hardware gecleared door hardware vlak nadat er een interrupt plaatsvindt. Als de Return from Interrupt (RETI) instructie wordt aangeroepen aan het einde van een interrupt, wordt deze bit weer gezet. De gebruiker kan deze bit ook zetten en clearen met de Set Global Interrupt Flag (SEI) en Clear Global Interrupt Flag (CLI) instructies.

**Bit 6 T : Copy Storage**

Deze plek wordt gebruikt voor de Bit Load (BLD) en de Bit Store (BST) instructies. Deze instructies kunnen een specifieke bit uit een register uit de Register file opslaan of uitlezen.

**Bit 5 H : Half Carry Flag**

De Half Carry Flag geeft aan of er een half carry was in sommige arithmatische operaties.. Dit kan handig zijn in binary code decimal arithmatiek.

**Bit 4 S : Sign Flag**

Deze flag is een xor van de N en V flag. Het is dus gezet als allen de V of N flag gezet is.

**Bit 3 V : Two’s Compliment Overflow Flag**

Deze flag wordt gezet als two’s compliment wordt ondersteund.

**Bit 2 N : Negative Flag**

Deze flag wordt gezet als de uitkomst van een arithmatische of logische operatie negatief is.

**Bit 1 Z : Zero Flag**

Deze flag wordt gezet als de uitkomst van een arithmatische of logische operatie 0 is.

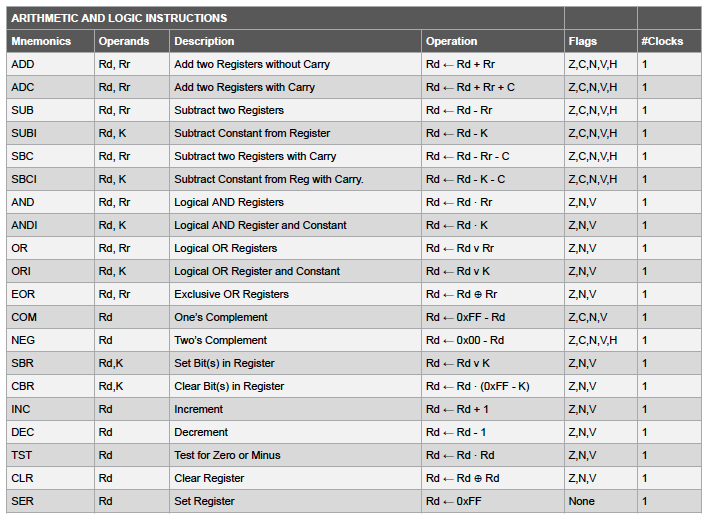
**Bit 0 C : Carry Flag**

Deze flag wordt gezet als een carry uit een arithmatische of logische operatie ontstaat.

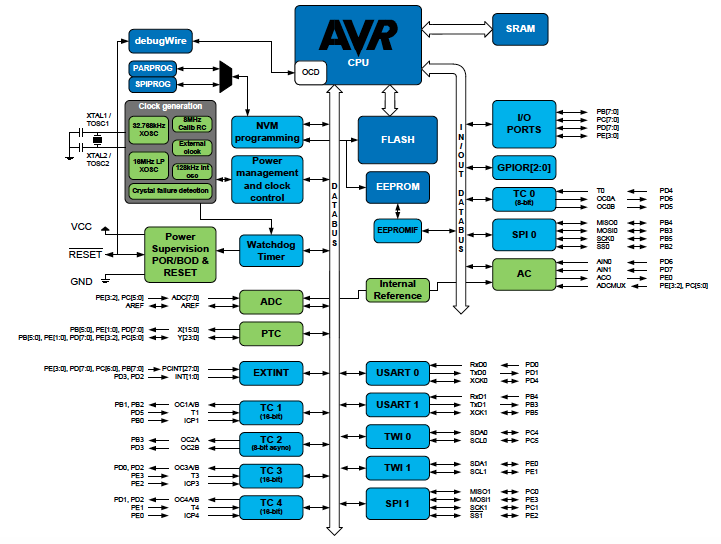
### Arithmetic Logic Unit (ALU)

In de ALU wordt er gerekend met data uit de Register file en met constanten. Er worden altijd één of twee input gebruikt, en er is één output die opgeslagen kan worden in de Register file. Verder wordt er na iedere operatie de Status register geüpdatet om extra informatie te geven over de operatie en de uitkomst. De volgende operaties kunnen uitgevoerd worden door de ALU:

Tabel 2 ALU Instructies



## Memory

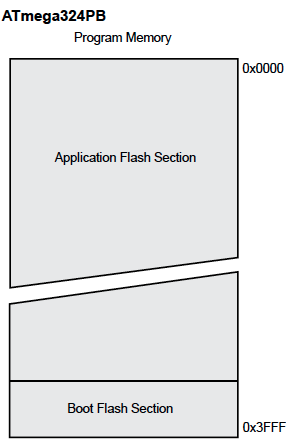


Figuur 4 AVR Memory

AVR microcontroller zijn gebouwd met een gemodificeerd Harvard architectuur. De Flash Program space heeft een andere bus dan de Static Random Acces Memory (SRAM). Er zijn twee verschillende databussen, eentje voor het grootste gedeelte van memory en een andere bus voor de In/Out memory.

### Program Memory

De Program Memory zit op een On-chip reprogrammeerbaar Flash memory. Hier wordt dus 16 of 32 bits lange instructies opgeslagen. Dit is een 32K x 16 structuur. Flash Program memory space is verdeelt over twee stukken, Boot Loader Section en Application Program Section.



Figuur 5 Program Memory

### EEPROM Data Memory

Electrically Erasable Programmable Read-Only Memory (EEPROM) is een apart stuk data space. Hiernaar kan geschreven of gelezen worden met een enkele byte per keer. De EEPROM kan vanaf de CPU aangestuurd worden via het EEPROM Address Register, het EEPROM Data Register en het EEPROM Control Register.

### SRAM Data Memory

Data Memory bestaat uit verschillende onderdelen:

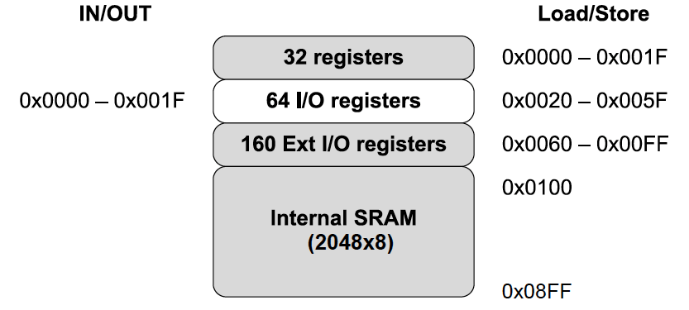
* Registers
* I/O Memory
* Extenden I/O Memory (verschilt per chip)
* Internal SRAM

De Register Space bestaat uit 32 general purpose working 8-bit registers (R0-R31). Deze zijn echter niet fysiek gemapt op deze locatie. Deze registers komen uit het Register file.

I/O Memory bevat de adres ruimte voor peripheral functies zoals control registers en andere I/O functies. Deze registers zijn extra snel berijkbaar door de In/Out memory data bus.

Extended I/O Memory is geheugen dat voor iedere chip anders kan zijn, afhankelijk van welke extra peripherals de chip bezit.

Internal SRAM (Data Memory) is gebruikt voor het tijdelijk opslaan van data en variabelen binnen een applicatie. De stack begint vanuit het einde van de Internal SRAM en groeit naar boven naarmate er meer op gepusht wordt.



Figuur 6 Indeling SRAM

Met de load of store kan ieder gedeelte van het SRAM bereikt worden. Met de in en out kunnen alleen de I/O registers aangesproken worden en wordt het adres anders genoteerd. Deze bus maakt het bereiken van de I/O registers een stuk sneller.

# Deelvraag 2

Hoe wordt het programma ingedeeld?

## Standaarden

Assembly is een programmeertaal die erg moeilijk te lezen is. Daarom is het handig om in het begin vast te stellen hoe er met data wordt omgegaan. Als er consistent wordt geprogrammeerd is het makkelijker om code terug te lezen.

### General Purpose Working Registers

Er zijn 32 8-bit General Purpose Working Registers, waarvan zes regesters drie 16-bit registers kunnen vormen voor adressering. De indeling van de registers ziet er als volgt uit:

Tabel 3 Register file indeling

|  |  |
| --- | --- |
| **Register** | **Gebruik** |
| R0 | Uitkomst van keer som laag |
| R1 | Uitkomst van keer som hoog |
| R2 | Invoer knop rij 1 2 3 4 nieuw |
| R3 | Invoer knop rij 5 6 7 8 nieuw |
| R4 | Invoer knop rij 9 0 + - nieuw |
| R5 | Invoer knop rij \* / Erase Backspace = nieuw |
| R6 | Invoer knop rij 1 2 3 4 oud |
| R7 | Invoer knop rij 5 6 7 8 oud |
| R8 | Invoer knop rij 9 0 + - oud |
| R9 | Invoer knop rij \* / Erase Backspace = oud |
| R10 | Return waarde laag |
| R11 | Return waarde hoog |
| R12 | Ingevoerde digit of operator |
| R13 | Tijdelijk register 6 |
| R14 | Tijdelijk register 7 |
| R15 | Tijdelijk register 8 |
| R16 | Parameter 0 |
| R17 | Parameter 1 |
| R18 | Parameter 2 |
| R19 | Parameter 3 |
| R20 | Tijdelijk register 0 |
| R21 | Tijdelijk register 1 |
| R22 | Tijdelijk register 2 |
| R23 | Tijdelijk register 3 |
| R24 | Tijdelijk register 4 |
| R25 | Tijdelijk register 5 |
| R26 | X-register, De positie van het einde van de som laag |
| R27 | X-register, de positie van het einde van de som hoog |
| R28 | Y-register, de somteller laag |
| R29 | Y-register, de somteller hoog |
| R30 | Z-register, de positie van een operator laag |
| R31 | Z-register, de positie van een operator hoog |

R0 en R1 is de plek waar het antwoord van een vermenigvuldig operatie in gaan. De 17 knoppen worden uitgelezen via multiplexing, welke invoer gegeven is wordt opgeslagen in de registers R2 t/m R5. De vorige waarde van de invoer wordt onthouden in de registers R6 t/m R9. Eén 16-bit getal kan gereturnd worden, deze wordt opgeslagen in R10 en R11. Welk getal/operator er wordt ingevoerd staat in R12. Alle registers vóór R16 hebben een gelimiteerde instructionset. Zo kunnen operaties met een immediate bijvoorbeeld niet uitgevoerd worden me deze registers. Er zijn 3 tijdelijke registers hier. De registers met een volledige instructionset worden gebruikt door de parameters en door nog 6 tijdelijke registers. Het 16 bits X register wordt gebruikt voor het bijhouden van het einde van de som, het Y register voor de navigatie door de som en het Z register om de positie van de operator te kunnen onthouden.

### SRAM

Sommen worden opgeslagen in SRAM. Dit wordt gedaan door iedere digit of operator op te slaan in een adres.

Tabel 4 Digit en operator waardes

|  |  |  |
| --- | --- | --- |
| **Waarde** | **Binaire waarde** | **Hex waarde** |
| 0 | 0000 0000 | 00 |
| 1 | 0000 0001 | 01 |
| 2 | 0000 0010 | 02 |
| 3 | 0000 0011 | 03 |
| 4 | 0000 0100 | 04 |
| 5 | 0000 0101 | 05 |
| 6 | 0000 0110 | 06 |
| 7 | 0000 0111 | 07 |
| 8 | 0000 1000 | 08 |
| 9 | 0000 1001 | 09 |
| + | 0001 XXXX | 1X |
| - | 0010 XXXX | 2X |
| \* | 0100 XXXX | 4X |
| / | 1000 XXXX | 8X |
| Leeg | 0000 1111 | 0F |

In het geheugen kan de som er als volgt uit zien:

Tabel 5 Som in geheugen voorbeeld

|  |  |  |  |
| --- | --- | --- | --- |
| **Adres** | **Waarde** | **Binaire waarde** | **Hex waarde** |
| 0x100 | 1 | 0000 0001 | 01 |
| 0x101 | 2 | 0000 0010 | 02 |
| 0x102 | + | 0001 XXXX | 1X |
| 0x103 | 3 | 0000 0011 | 03 |
| 0x104 | \* | 0100 XXXX | 4X |
| 0x105 | 6 | 0000 0110 | 06 |
| 0x106 | - | 0010 XXXX | 2X |
| 0x107 | 9 | 0000 1001 | 09 |

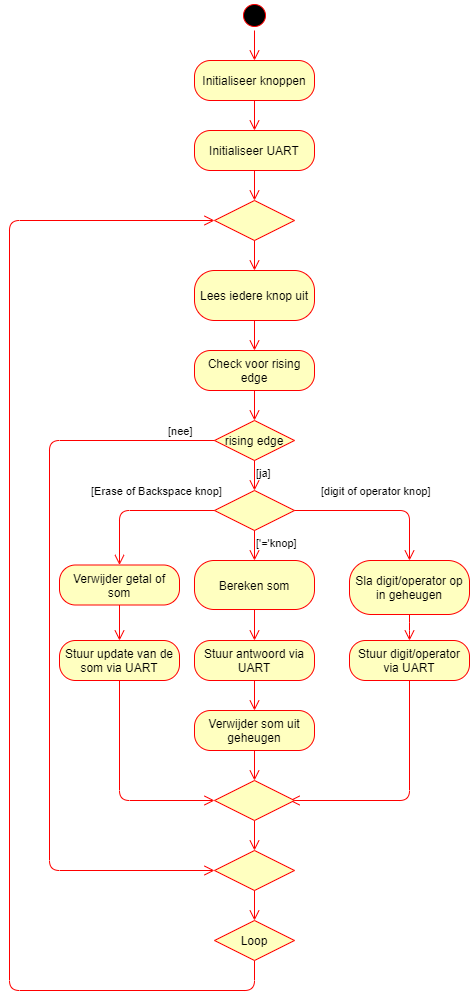
Bij een tussenstap tijdens het berekenen van de som kan de ‘Leeg’ waarde gebruikt worden.

Tabel 6 Som in geheugen voorbeeld stap 2

|  |  |  |  |
| --- | --- | --- | --- |
| **Adres** | **Waarde** | **Binaire waarde** | **Hex waarde** |
| 0x100 | 1 | 0000 0001 | 01 |
| 0x101 | 2 | 0000 0010 | 02 |
| 0x102 | + | 0001 XXXX | 1X |
| 0x103 | 1 | 0000 0001 | 01 |
| 0x104 | 8 | 0000 1000 | 08 |
| 0x105 | Leeg | 0000 1111 | 0F |
| 0x106 | - | 0010 XXXX | 2X |
| 0x107 | 9 | 0000 1001 | 09 |

Bij het opslaan van de som mag een groot deel van de RAM gebruikt worden. Dit vanaf begin internal SRAM tot het einde - de maximale stack grootte.

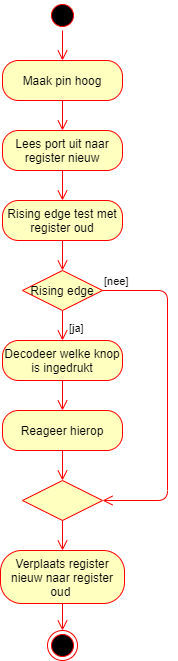
## Programma



Figuur 7 Hoogst Niveau Software

Het programma begint het met een initialisatie dat wordt vervolgd met een oneindige loop. In de loop wordt eerst de knoppen gecontrolleerd. Als er een knop laag is die eerst hoog was, wordt dit gebruikt als invoer. Dit wordt gedaan ter vervanging van een interrupt, aangezien het programma nog even zo makkelijk mogelijk wordt gehouden. Als later tijd is om een interrupt te implementeren, dan zal het ontwerp veranderen.

### Knoppen



Figuur 8 Software Knoppen uitlezen

Het ontwerp in Figuur 8 beschrijft het proces van het uitlezen van een knoppen. Deze stap wordt vier keer gedaan met een andere pin in verband met multiplexing.

De rising edge test wordt gedaan door eerst te kijken of de waarde in Nieuw register hoger is dan in Oud register.

Actie Nieuw register Oud register Uitkomst

Bij indrukken 0000 0100 > 0000 0000 klopt

Bij ingedrukt houden 0000 0100 > 0000 0100 klopt niet

Bij loslaten 0000 0000 > 0000 0100 klopt niet

Bij extra knop indrukken 0000 1100 > 0000 0100 klopt

Als dit klopt wordt de berekening “Nieuw register – Register oud = verschil” gebruikt

Hiermee kan gekeken worden welke knop er wordt er is ingedrukt.

Actie Nieuw register Oud register Flank

Bij een ingedrukte knop 0000 0100 - 0000 0000 = 0000 0100

Bij extra ingedrukte knop 0000 1100 - 0000 0100 = 0000 1000

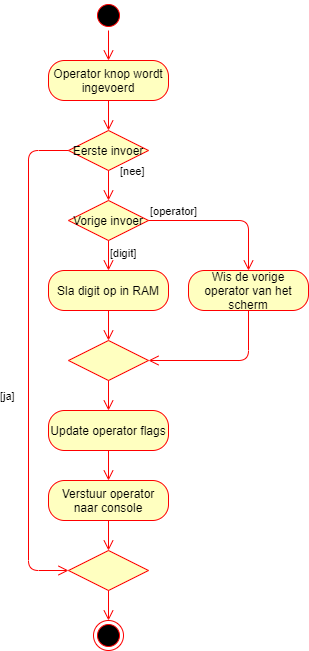
#### Getal knop invoer



Figuur 9 Software Getalknop Invoer

Figuur 9 beeld uit wat er gebeurt als een getal knop, 0 tot en met 9, wordt ingevoerd.

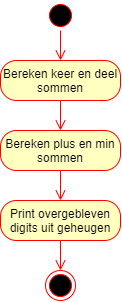
#### Operator knop invoer



Figuur 10 Software Operatorknop Invoer

Figuur 10 beeldt uit wat er gebeurt als een operatorknop zoal +, -, \* of / ingevoerd wordt. Een som kan niet beginnen met een operator. Als er twee operators achter elkaar verstuurd worden, zal de eerste vervangen worden. De operator flags staan in R14.

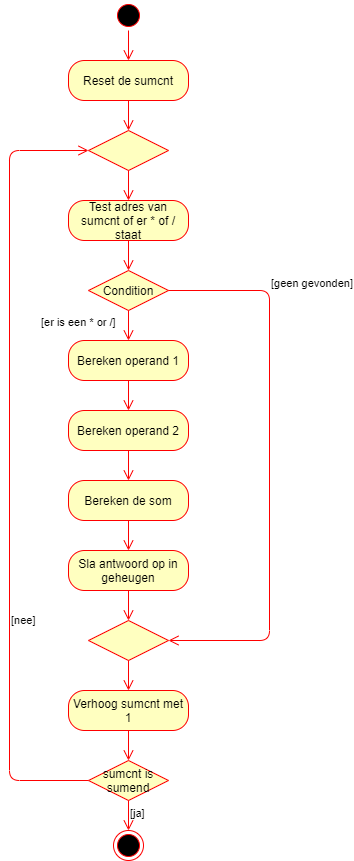
### Som berekenen



Figuur 11 Software ontwerp som berekenen

Om de rekenregels te kunnen volgen, worden eerst de vermenigvuldigingen en delingen opgelost. Vervolgens wordt er door dezelfde som alle optellingen en aftrekkingen opgelost. Het getal dat overblijft in geheugen is het antwoord van de laatste som, dus ook het antwoord van de hele som.

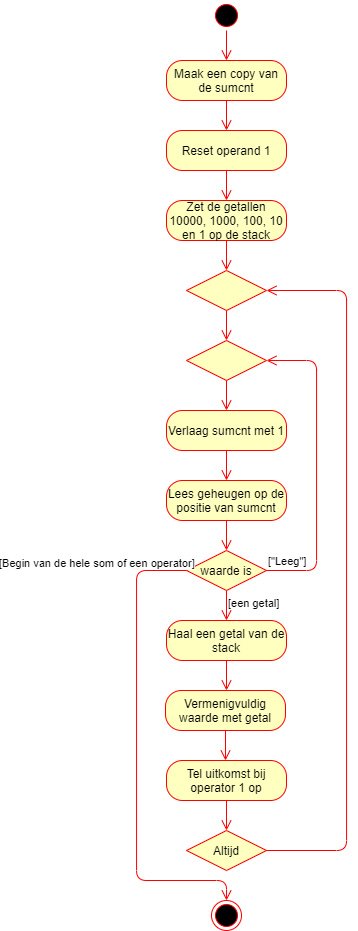
#### Som loop



Figuur 12 Software ontwerp vinden van een som

Voor vermenigvuldigen en delen geldt dezelfde procedure als voor optellen en afrekken. Met de sumcnt wordt door memory heen gekeken. Bij iedere lezing wordt getest of dit een operator is. Als dit zo is, behoren de getallen ervoor tot operand 1 en de getallen erna tot operand 2. Deze moeten berekend worden voordat de som berekend kan worden. Het antwoord wordt vervolgens opgeslagen in geheugen. De sumcnt wordt verhoogt om de volgende waarde uit geheugen te lezen. Dit eindigt als het einde van de volledige som bereikt is.

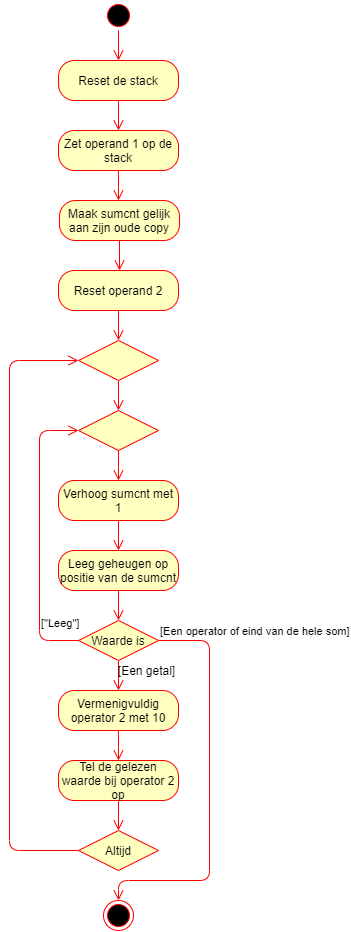
##### Operator 1



Figuur 13 Software ontwerp operator 1

Er moet een kopie gemaakt worden van sumcnt om te onthouden waar de operator zat. Operand 1 staat voor de operator. Door de manier dat het opgeslagen is, is het bekend dat het eerste getal voor de operator een eenheid is, het tweede een tiental, derde een honderdtal etc. . Met 16 bits kan maximaal kan niks hoger dan een tienduizendtal opgeslagen worden. Het einde van een getal is te herkennen aan dat er een operator gelezen wordt, of het begin van de som bereikt wordt. Als dit een getal is, wordt dit getal vermenigvuldigd met wat er op de stack staat (bijvoorbeeld 1, 10, 100 … afhankelijk van hoe vaak er al eerder een getal is uitgelezen in deze loop). Vervolgens wordt dit bij de operator 1 opgeslagen.

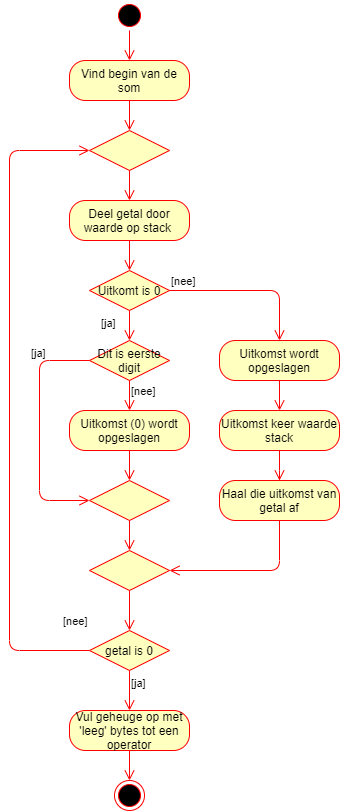
##### Operator 2



Figuur 14 Software ontwerp operator 2

Eerst worden de stappen van het berekenen van operator 1 ongedaan gemaakt. De overgebleven vermenigvuldigers (1, 10, 100, 1000 en 10000) worden van de stack af gehaald. De operand 1 is tijdelijk niet nodig en wordt op de stack gezet. De positie van de operator wordt terug gezet op de sumcnt. Dit keer is niet bekend of een gelezen getal een eenheid, tiental, honderdtal etc. is. Wel is bekend dat iedere keer dat er een nieuw getal gelezen wordt, de oude waarde van operator 2 dus tien keer groter is dan voorheen. Hierbij kan de nieuwe gelezen waarde opgeteld worden. Hier geldt dat het einde van operator 2 herkent kan worden aan een nieuwe operator of het einde van de som.

##### Antwoord opslaan



Figuur 15 Software ontwerp antwoord opslaan

De som met de voorgaande operands en de operator is berekend, hieruit komt een getal. Een getal heeft maximaal 5 digits, dus er moet maximaal tienduizenden, duizenden, honderden, tientallen en eenheden opgeslagen worden. Op de stack staat daarom van onder naar boven 1, 10, 100, 1000 en 10000. Het getal, de uitkomst van de voorgaande som, wordt steeds gedeeld door bovenste waarde van de stack. Wat er op de stack stond wordt overigens wel even opgeslagen voor later. De eerste keer zal dit 10000 zijn. Als de uitkomst hiervan 0 is, en er is geen vorige digit opgeslagen, dan wil dit zeggen dat er niks opgeslagen hoeft te worden. Er zijn bijvoorbeeld geen tienduizentallen in het getal. Als de uitkomst geen 0 is, dan kan dit opgeslagen worden. De uitkomst keer de stack wordt van het getal af gehaald. Als het getal 0 is, is alles opgeslagen. Derest van het geheugen tot het eind van som of de volgende operator wordt opgevuld met de ‘leeg’ byte.

Hieronder staan twee voorbeelden gegeven. De eerste, Tabel 5, demonstreert een volledig getal, dus met 5 digits. De tweede, Tabel 6, demonstreert een getal dat korter is. De tienduizendtallen en duizendtallen worden niet opgeslagen. Dit voorbeeld demonstreert ook dat er soms wel een 0 opgeslagen moet worden.

Tabel 7 Opslaan van een getal met 5 digits

|  |  |  |  |
| --- | --- | --- | --- |
| Getal | Uitkomst | Stack (wat er het laatst gelezen is) | Actie |
| 12987 | - | 10000 | Deel Getal door stack |
| 12987 | 1 | 10000 | Sla Uitkomst op |
| 12987 | 1 | 10000 | Getal – Uitkomst \* Stack |
| 2987 | 1 | 10000 | Loop opnieuw |
| 2987 | X | 1000 | Deel Getal door stack |
| 2987 | 2 | 1000 | Sla Uitkomst op |
| 2987 | 2 | 1000 | Getal – Uitkomst \* Stack |
| 987 | 2 | 1000 | Loop opnieuw |
| 987 | X | 100 | Deel Getal door Stack |
| 987 | 9 | 100 | Sla Uitkomst op |
| 987 | 9 | 100 | Getal – Uitkomst \* Stack |
| 87 | 9 | 100 | Loop opnieuw |
| 87 | X | 10 | Deel Getal door Stack |
| 87 | 8 | 10 | Sla Uitkomst op |
| 87 | 8 | 10 | Getal – Uitkomst \* Stack |
| 7 | 8 | 10 | Loop opnieuw |
| 7 | X | 1 | Deel Getal door Stack |
| 7 | 7 | 1 | Sla Getal op |
| 7 | 7 | 1 | Getal – Uitkomst \* Stack |
| 0 | 7 | 1 | Eind |

Tabel 8 Opslaan van getal met minder dan 5 digits en een 0 in het midden

|  |  |  |  |
| --- | --- | --- | --- |
| Getal | Uitkomst | Stack (wat er het laatst gelezen is) | Actie |
| 102 | - | 10000 | Deel Getal door Stack |
| 102 | 0 | 10000 | Loop opnieuw |
| 102 | X | 1000 | Deel Getal door Stack |
| 102 | 0 | 1000 | Loop opnieuw |
| 102 | X | 100 | Deel Getal door Stack |
| 102 | 1 | 100 | Sla Uitkomst op |
| 102 | 1 | 100 | Getal – Uitkomst \* Stack |
| 2 | 1 | 100 | Loop opnieuw |
| 2 | X | 10 | Deel Getal door Stack |
| 2 | 0 | 10 | Sla 0 op (Dit is namelijk niet de eerste digit) |
| 2 | 0 | 10 | Loop opnieuw |
| 2 | X | 1 | Deel Getal door Stack |
| 2 | 2 | 1 | Sla Uitkomst op |
| 2 | 2 | 1 | Getal – Uitkomst \* Stack |
| 0 | 2 | 1 | Eind |