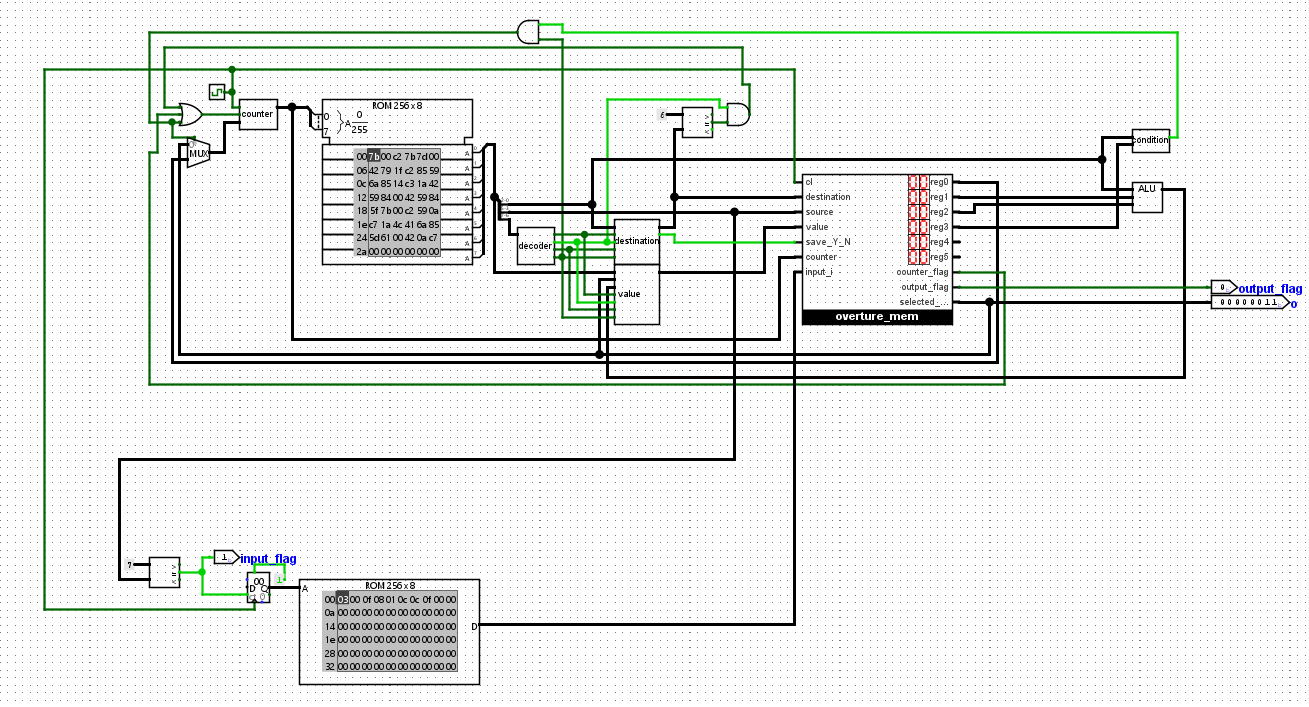
Computerdesign



Diagram, schematic

Description automatically generated

Contents

[Inleiding 3](#_Toc120455737)

[Indeling 3](#_Toc120455738)

[Doelen 3](#_Toc120455739)

[Onderdelen 6](#_Toc120455740)

[Binair tellen 6](#_Toc120455741)

[Logic gates 7](#_Toc120455742)

[Multiplexer en demultiplexer 9](#_Toc120455743)

[Optellen en aftrekken 9](#_Toc120455744)

[Vermenigvuldigen en delen 11](#_Toc120455745)

[Vergelijken 16](#_Toc120455746)

[Memory 17](#_Toc120455747)

[Teller 19](#_Toc120455748)

[Floats 20](#_Toc120455749)

[Stack 20](#_Toc120455750)

[Architectuur 21](#_Toc120455751)

[Overture 21](#_Toc120455752)

[Memory 22](#_Toc120455753)

[ALU 23](#_Toc120455754)

[Comparisons 23](#_Toc120455755)

[Decoder 23](#_Toc120455756)

[Totaal 23](#_Toc120455757)

[Problemen van Overture 23](#_Toc120455758)

[Regi V1 25](#_Toc120455759)

[Regi V2 27](#_Toc120455760)

[Programmeren 27](#_Toc120455761)

[Overture 28](#_Toc120455762)

[Regi 28](#_Toc120455763)

[RASM 29](#_Toc120455764)

[High 30](#_Toc120455765)

[Conclusie 31](#_Toc120455766)

[Evaluatie 32](#_Toc120455767)

[Bronnen 32](#_Toc120455768)

[Uren 32](#_Toc120455769)

# Inleiding

## Indeling

Dit document is opgedeeld in vier delen: Doelen, onderdelen, Architectuur en Assembly. Om ervoor te zorgen dat dit document overzichtelijk blijft zal ik soms verwijzen naar de bijlagen waar alle schema's in staan en, specificaties en voorbeelden van mijn programmeertaal.

In eerste deel 'doelen' zal ik het hebben over waar ik naar toe werk en verklaren waardoor ik tot bepaalde keuzes kwam.

Het deel daarna 'onderdelen' gaat over de kleinere circuits die nodig zijn om verder te werken naar een uiteindelijke CPU. Het zal ook uitleggen hoe tellen en andere operaties werken in binair.

'Architectuur' is een beetje te vergelijken met het vorige deel zoals de architectuur van een huis te vergelijken is met losse gangen en kamers. In dit deel worden de onderdelen van het vorige deel verbonden en worden er ook een paar nieuwe onderdelen gemaakt die specifieker zijn gemaakt voor mijn computer. Het zal het ook hebben over hoe een instructie wordt uitgevoerd in een computer.

Het laatste deel 'programmeren’ zal dieper in gaan op hoe je programmeert voor mijn computer en hoe ik assembly omzet naar bitcode die uitgevoerd kan worden door mijn computer. Ook gaat het over hoe ik een higher-level taal heb gemaakt. Verder staan in dit deel een paar voorbeeld programma’s om te laten zien wat de uiteindelijke computer kan en wordt er uitgelegd hoe de programma’s werken.

Alle circuits waarvan ik plaatjes gebruikt lever ik samen in met dit document, dus als iets niet helemaal duidelijk is dan is het mogelijk om de simulaties te gebruiken om het te visualiseren.

## Doelen

Mijn doel is om een computer te maken die zo gebruiksvriendelijk mogelijk is om in te programmeren in assembly. Dat betekent dat je in een programeer taal die zo dicht mogelijk bij de CPU staat moet kunnen programmeren zonder dat je gek wordt van alle ingewikkelde bochten waar je je in moet wringen. Een ding wat ik bijvoorbeeld niet wil hebben is 'memory juggling' waarbij je herhaaldelijk informatie heen en weer moet kopiëren omdat er maar twee registers die je kan gebruiken om operaties op uit te voeren waardoor je voor elke nieuwe som opnieuw waardes moet kopiëren naar die twee specifieke registers. Voor een deel van de inspiratie heb ik gekeken naar de x86 instructieset. (1)

De specificaties waar ik aan het begin op kwam die het per se moet hebben/kunnen zijn:

1. Minimaal 8bit words
2. Operaties kunnen worden uitgevoerd op twee willekeurige registers en je hoeft dus niet de waardes in twee specifieke registers te hebben staan.
3. Kan twee waardes vergelijken en dat gebruiken om iets conditioneel uit te voeren. (COMP en JUMP)
4. Informatie kunnen kopiëren van een plek naar een andere. (COPY)
5. Input kunnen krijgen van buiten en informatie kunnen doorgeven naar buiten de computer.
6. Immediates kunnen gebruiken direct vanuit de instructies.
7. Logische operaties (AND, NAND, OR, NOR, XOR, XNOR, NOT, TEST, HIGH)
8. Simpele wiskundige operaties (ADD, SUB)
9. Een instructie kunnen uitvoeren per cyclus

1: 8bit words

Dit is handig om te hebben, omdat de grootte van je words uitmaakt met hoe hoge getallen je kan werken. Zo heb je bij 8bits heb je maar 2^8 mogelijke waardes en daarmee het bereik [0,255] of als je een bit gebruikt als minteken [-128,127] dit is niet geweldig, maar er is nog redelijk mee te werken. Het was wel het doel om de word size hoger te maken dan dit want hier heb je dat je al moeilijk moet gaan doen als iemand je vraagt wat ‘-1 + -128’ is. Ik wist alleen nog niet hoe moeilijk hogere word sizes zouden zijn, dus ik begon conservatief met 8bit. Nog een reden dat ik minimaal 8 bits wou is, omdat dat het minimum is om een karakter op te kunnen slaan.

Ergens uitleggen hoe binair telllen werkt (signed en unsigned) en hoe ASCII werkt

2: Register-memory architectuur

Register-memory architectuur is een van de twee veel gebruikte architectuur vormen in computers. De naam komt van dat je beide registers en memory kan gebruiken in operaties in tegenstelling tot load-store waarbij je eerst de waardes laden van memory (load) en moet opslaan in een register speciaal voor operaties (store). Ik heb gekozen voor register-memory, omdat het ervoor zorgt dat je niet het grootste deel van de tijd bezig bent met het verplaatsen van informatie. Het enige nadeel van register-memory is dat de hele memory moet zijn verbonden met de ALU dat zorgt ervoor dat de chip complexer wordt en daardoor duurder en langzamer. Het is een trade-off tussen eenvoudigheid van programmeren en de eenvoudigheid van de chip. Ik ga voor gebruiksvriendelijkheid dus eenvoudigheid van programmeren gaat voor.

Moet ik uitleggen waarom de keus voor load-store bij normaal programmeren op je laptop niet uitmaakt? Het verschil tussen memory en registers uitleggen.

3: conditioneel uitvoeren

Het conditioneel uitvoeren van code is heel belangrijk. Als je bijvoorbeeld de getallen 1 t/m 100 wilt geven dan moet je de waarde steeds en verhogen en opgeven, maar je moet stoppen na 100. Dit is conditioneel uitvoeren totdat ‘x > 100’. Zonder conditionele instructies kun je geen loops hebben en is bijna elke instructie onmogelijk.

Ergens hebben over Turing-compleetheid

4: kopiëren

Kopiëren wordt veel gebruikt, als je bijvoorbeeld een output wil geven dan kopieer je een waarde van memory naar de output en als je een waarde in een lijst wil toevoegen dan moet je alles na die waarde opzij kopiëren om plek te maken voor je nieuwe waarde.

Voorbeeld schrijven naar beeld toevoegen?

5: Input en output

Deze is denk ik de meest voor de hand liggende. Als je een toetsenbord gebruikt krijgt je laptop input van het toetsenbord en als je dan tekst ziet verschijnen op je scherm dan is dat een output van je laptop. Zonder I/O heb je weinig aan een computer, omdat het niet kan laten zien wat je het hebt laten berekenen en dan heb je tijd verspilt (van de computer, dat is gemeen :(, heb medelijden).

6: Immediates

Immediates zijn waardes die niet uit memory komen, maar uit de instructie zelf. Als de instructie de som ‘x\*2’ was dan is **2** een immediate, omdat het niet uit memory komt, maar in de som zelf staat. Dit is nodig, omdat alle waardes moeten ooit een keer opgeslagen zijn en dat kan alleen via inputs en immediates. Je kan niet altijd inputs vragen, want dat wordt heel snel vervelend voor gebruikers.

Voorbeeld nodig?

7: logische operaties

Dit zijn operaties die de losse bits van twee getallen met elkaar vergelijkt, XOR wordt bijvoorbeeld veel gebruikt bij simpele cryptografie en AND kan worden gebruikt om te kijken of een getal deelbaar is door een macht van twee. Dit soort zijn niet strikt noodzakelijk, maar wel heel nuttig.

Uitleggen hoe XOR en AND worden gebruikt in meer detail (misschien in bijlage)?

8: (simpele) wiskundige operaties

Ik had origineel alleen gekozen voor optellen (ADD) en aftrekken (SUB), omdat je met die operaties veel andere kan maken door ze herhaaldelijk toe te passen. Vermenigvuldigen is herhaald optellen en machtsverheffen is herhaald keer, delen is herhaald aftrekken en wortel kun je oplossen door herhaald twee getallen te vermenigvuldigen totdat je genoeg in de buurt komt. Het is niet heel efficiënt, maar het kan wel. Het liefst zou ik vermenigvuldigen (MULT) en delen (DIV) ook hebben, maar die zijn moeilijker te maken en dus was dat een mogelijk doel voor later.

Algoritme voor delen, vermenigvuldigen, wortel en macht in bijlage?

9: 1 instructie per cyclus

Een instructie per cyclus wordt normaal niet gedaan doordat je hierdoor ervoor zorgt dat je langzaamste operatie je forceert je cyclus snelheid langzaam genoeg maakt om het eens per cyclus uit te kunnen voeren, maar het zorgt er ook voor dat je niet hoeft op te letten dat een instructie nog niet compleet klaar is zodra je de volgende uitvoert. Dit is dus weer een trade-off tussen gebruiksvriendelijkheid en snelheid, en gebruiksvriendelijkheid is voor mij het belangrijkst.

Uitleggen variabele cyclus lengte en de complexiteit daarvan?

# Onderdelen

De onderdelen die ik heb gemaakt voordat ik begon met de CPU designen zijn heb ik gekozen, omdat het degene zijn die je eerst moet maken voordat je alles in elkaar zet in ‘Nandgame’ en ‘Turing Complete’, dat zijn twee spellen die ik heb gebruikt om te leren hoe een computer werkt. (2)

## Binair tellen

Voordat ik kan hebben over tellen in binair moet het eerst hebben over normaal decimaal tellen. Als je **243** schrijft dan heb je het niet over **2+4+3**, maar over **2\*10\*10 + 4\*10 + 3 = 200 + 40 + 3.**

|  |  |  |
| --- | --- | --- |
| 10**2** | 10**1** | 10**0** |
| **2** | **4** | **3** |
|  |  |  |
| 2\*102 = 200 | 4\*101 = 40 | 3\*100 = 3 |
| 200+40+30 = 243 |  |  |

Elke plek die je naar links gaat wordt 10 keer zo veel waard. Het werkt met machten van 10 doordat we 10 getallen hebben 0 t/m 9. Je telt eenheden tot 9 en de 10de onthoud je en schrijf je bij de tientallen (ofwel overflow). Door deze 10 getallen heet het ook Base 10 of decimaal.

Binair werkt op dezelfde manier, maar binair heeft alleen de getallen **0** en **1**. Doordat het maar twee getallen heeft kun je maar tot 1 tellen en dan onthoud je de tweede. Elke plek wordt dus niet 10 keer, maar 2 keer zo veel waard.

|  |  |  |
| --- | --- | --- |
| 2**2** | 2**1** | 2**0** |
| **1** | **0** | **1** |
|  |  |  |
| 1\*22 = 4 | 0\*21 = 0 | 1\*20 = 1 |
| 4+0+1 = 5 |  |  |

|  |  |
| --- | --- |
| 0 | 0 |
| 1 | 1 |
| 2 | 10 |
| 3 | 11 |
| 4 | 100 |
| 5 | 101 |
| 6 | 110 |
| 7 | 111 |

Er is nog een laatste getallenstellen waarin vaak wordt gerekend bij computers en dat is hexadecimaal. In hexadecimaal zijn er 16 cijfers, 1 t/m 9, A, B, C, D, E en F. Doordat je nu tot vijftien kan tellen (F) heb je dat je pas een hoeft te onthouden bij 16. Dit betekent dat elke plaats naar links 16 keer zo veel waard is.

|  |  |  |
| --- | --- | --- |
| 16**2** | 16**1** | 16**0** |
| **5** | **D** | **A** |
|  |  |  |
| 5\*162 = 5 \* 256 = 1280 | 13\*161 = 13 \* 16 = 208 | 10\*160 = 10 \* 1 = 10 |
| 1280 + 208 + 10 = 1498 |  |  |

|  |  |  |
| --- | --- | --- |
| Decimaal | Hexadecimaal | Binair |
| 0 | 0 | 0 |
| 1 | 1 | 1 |
| 2 | 2 | 10 |
| 3 | 3 | 11 |
| 4 | 4 | 100 |
| 5 | 5 | 101 |
| 6 | 6 | 110 |
| 7 | 7 | 111 |
| 8 | 8 | 1000 |
| 9 | 9 | 1001 |
| 10 | A | 1010 |
| 11 | B | 1011 |
| 12 | C | 1100 |
| 13 | D | 1101 |
| 14 | E | 1110 |
| 15 | F | 1111 |

Hexadecimaal wordt vaak gebruikt, omdat 16 gelijk is aan 2^4, dat betekent dat een hexadecimaal cijfer gelijk is aan vier binaire. Je kan van hexadecimaal naar binair gaan door elk hexadecimaal los om te zetten naar binair en dan aan elkaar te plakken.

|  |  |  |
| --- | --- | --- |
| **5** | **D** | **A** |
| **0101** | **1101** | **1010** |
| 5DA = 0101 1101 1010 |  |  |

Om dingen overzichtelijker te maken wordt hexadecimaal, hex genoemd, decimaal wordt dec en binair wordt bin. Om het duidelijker te maken over welk getallenstelsel je het hebt gebruik je prefixen. Hex is 0x, Dec is 0d en bin is 0b, dus **0xF = 0d15 = 0b1111**. (3)

Uitleggen 2’s complement (min getallen)

## Logic gates

Alles in een computer is opgebouwd vanaf logic gates. Logic gates hebben een of twee inputs en geven een output. Alle inputs en outputs zijn een bit een **0** (False of geen stroom) en **1** (True of wel stroom).

Shape

Description automatically generated

Figuur 1: AND-gate ‘&’

|  |  |  |
| --- | --- | --- |
| In0 | In1 | Out |
| 0 | 0 | 0 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 1 |

De AND-gate heeft de naam de het heeft, omdat pas als input0 **en** input1 aan staan de output aan gaat.

Het kost te veel ruimte om elke gate los langs te gaan, dus voor als de korte uitleg niet duidelijk genoeg is, is hier een link. (4)

Shape

Description automatically generated with medium confidenceShape

Description automatically generatedA picture containing clipart

Description automatically generatedShape, circle

Description automatically generatedShape

Description automatically generated

XOR gate ‘^’

NOR gate

OR gate ‘|’

NAND-gate

XNOR gate

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| In0 | In1 | Out |  | In0 | In1 | Out |  | In0 | In1 | Out |  | In0 | In1 | Out |  | In0 | In1 | Out |
| 0 | 0 | 1 |  | 0 | 0 | 0 |  | 0 | 0 | 1 |  | 0 | 0 | 0 |  | 0 | 0 | 1 |
| 0 | 1 | 1 |  | 0 | 1 | 1 |  | 0 | 1 | 0 |  | 0 | 1 | 1 |  | 0 | 1 | 0 |
| 1 | 0 | 1 |  | 1 | 0 | 1 |  | 1 | 0 | 0 |  | 1 | 0 | 1 |  | 1 | 0 | 0 |
| 1 | 1 | 0 |  | 1 | 1 | 1 |  | 1 | 1 | 0 |  | 1 | 1 | 0 |  | 1 | 1 | 1 |

De NAND is de AND met de output omgekeerd (**N**ot **AND**), i.p.v alleen aan bij beide inputs aan wordt het alleen uit als beide inputs aan staan. De OR-gate staat aan wanneer minstens een input aan staat en de NOR is het omgekeerde. Als laatste heb je de XOR en XNOR. De XOR is de e**x**clusive **or**, dus alleen wanneer als precies een input aan staat en de XNOR is daar dan weer het tegenovergestelde van.

A picture containing text

Description automatically generatedIcon

Description automatically generated

NOT gate

Controlled buffer

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| In0 | Out |  | In0 | Control | Out |
| 0 | 1 |  | 0 | 0 | Z |
| 1 | 0 |  | 1 | 0 | Z |
|  |  |  | 0 | 1 | 0 |
|  |  |  | 1 | 1 | 1 |

‘Z’ betekent dat de output niet naar 0v (0) of 5v (1) wordt getrokken, dat betekent dat een andere gate dat kan doen. Normaal kan dit niet, want een 0 en een 1 op een draad betekent dat de batterij direct met grond is verbonden en dan heb je kortsluiting.

## Multiplexer

Een multiplexer is een onderdeel dat een van de inputs kiest om door te laten. De output staat dus aan als de input aanstaat **en** is geselecteerd.

Diagram

Description automatically generated

Dit zijn twee versies van hetzelfde circuit, maar de onderste gebruikt het symbool voor een multiplexer.

## Optellen en aftrekken

Het eerste onderdeel is om twee binaire getallen op te tellen, maar dan moet je eerst beginnen met het optellen van twee bits. Het optellen van bits werkt op dezelfde manier als decimaal optellen: Je telt twee cijfers bij elkaar op en als het meer is dan 9 dan onthoud je een tiental. Alleen is het bij binair al bij meer dan 1 in plaats van 9, dus **1 + 1 = 10**.

Voor het optellen van twee bits (a en b) kun je de volgende truth table maken, waarbij je het getal dat je onthoudt de carry genoemd wordt:

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | Carry | Som |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Als je kijkt naar de laatste twee komen die respectievelijk overeen met AND en OR wat leidt tot het volgende circuit:

Diagram

Description automatically generated

Half adder

Op deze manier kun je twee eencijferige getallen bij elkaar optellen, maar als je meercijferige getallen bij elkaar wil optellen krijg je een probleem.

Stel je telt 19 en 81 bij elkaar op dan onthoud je 1 van het optellen van 9 en 2, daarna tel je 1, 8 en 1 bij elkaar op en dat betekent dus dat je drie cijfers bij elkaar op moet tellen.

|  |  |  |  |
| --- | --- | --- | --- |
| Decimaal | | Binair | |
| 1 |  | 1 |  |
| 1 | 9 | 1 | 1 |
| 8 | 1 | 0 | 1 |
| 10 | 0 | 10 | 0 |

Je moet dus drie bits tegelijk bij elkaar kunnen optellen. Dan pas kun je pas twee getallen volledig bij elkaar optellen, daarom heet dit een full adder terwijl het vorige circuit een half adder heet.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C\_in | a | b | Carry | Som |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 1 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 |
| 1 | 0 | 1 | 1 | 0 |
| 1 | 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 | 1 |

Dit is moeilijker in een keer te zien als logic gates, maar dit circuit hoort erbij:

Diagram

Description automatically generated

Full adder

Het is nu mogelijk om zo grootte getallen als je maar wilt bij elkaar op te tellen door de carry steeds te verbinden met de volgende adder.

Diagram, schematic

Description automatically generated

Byte adder

Het is redelijk makkelijk om van een adder een subtract circuit te maken door elke bit van een van de inputs door een NOT heen te halen. Dit werkt wel, maar je kan dan niet meer optellen, dus gebruik je een XOR die eigenlijk werkt als een NOT-gate die je aan en uit kan zetten. Door een bit (genaamd subtract in het plaatje) kun je dan kiezen of je optelt of aftrekt.

Diagram, schematic

Description automatically generated

Adder/subtracter

## Vermenigvuldigen en delen

Het vermenigvuldigen van getallen werkt door enkele cijfers te vermenigvuldigen en daarna getallen bij elkaar op te tellen.

A picture containing application

Description automatically generated

Plaatje verbeteren?

In enkele bits is vermenigvuldigen makkelijk, want 1\*1=1 en alle andere permutaties hebben als uitkomst 0, dit is precies het patroon van een AND-gate.

Als voorbeeld heb ik 4bit vermenigvuldiger. Het eerste wat het doet is het eerste getal vermenigvuldigen met de 1ste, 2de , 3de en 4de bit van de tweede met de hulp van 4 keer 4 AND-gates, daarmee krijg je vier getallen die steeds 2 (10 in binair) keer zo veel waard zijn en daarna tel je de vier getallen een voor een bij elkaar op.

Diagram, schematic

Description automatically generated

4bit vermenigvuldiger

Beter uitleggen?

Net zoals vermenigvuldigen werkt met herhaaldelijk optellen werkt delen met herhaaldelijk aftrekken.

Calendar

Description automatically generated

In decimaal kun op deze manier delen 46 door 21 delen.

Stap 1: Eerst doe je 46 min 21\*10, maar je komt uit op een negatief getal dus 21 past niet 10 keer in 46. Het lijkt misschien overduidelijk, maar je weet pas zodra je het hebt geprobeerd dat 46 niet deelbaar is door 21\*10. We kiezen verder te gaan met 46 in de volgende stap.

Stap 2: Je doet 46 – 21 en de uitkomst is 25 en positief, dus 21 past minstens 1 keer in 46. We kiezen om verder te gaan met 25.

Stap 3: 25 – 21 = 4 en dat is positief, dus 21 past minstens 2 keer in 46. Verder met 4.

Stap 4: 4 – 21 = -17 en dat is negatief dus 21 past niet in 4, dus 46 / 21 = 2 rest 4.

We gaan niet door tot komma getallen, want we hebben nog geen handige manier om die in bits weer te geven.

Nu we hebben herhaald hoe delen in decimaal werkt kunnen we hetzelfde doen in binair met 1010/11.

A picture containing text, white

Description automatically generated

Stap 1: 1010 – 1100 = 1 110 (-2), De eerste 1 geeft hier aan dat het getal negatief is, 1100 past dus niet in 1010. 11 past dus niet 100 keer in 1010. We gaan verder met 1010.

Stap 2: 1010 – 110 = 100, dus 11 past 10 keer in 1010. We hoeven in de volgende stap niet nog een keer -110 te proberen, want in de vorige stap hebben we gekeken of 11 vier keer in 1010 past en nu of het er twee keer in past, dus we weten al dat het er niet twee keer twee keer in past want dat is vier.

Stap 3: 100 – 11 = 1, dus 11 past nog 1 keer in 1010. We weten nu dus dat 11, 10 + 1 = 11 keer in 1010 past.

We kunnen een stap als volgt als circuit maken voor een 4bit getal:

Diagram

Description automatically generated

In plaats van A – (B\*1000) te proberen, proberen we (A/1000) – B, want dat werkt hetzelfde, daarom gaat alleen de eerste bit van A in de aftrekker. Nadat de som is uitgevoerd kiest de multiplexer als de uitkomst positief is de uitkomst als output en anders kiest het A als output. Q4 is de uitkomst van of B 1000 keer in A past en de andere vier bits onderaan is onze rest die we gebruiken in de volgende stap.

Om het deel circuit af te maken hoeven we dit alleen maar vier keer te herhalen. (5)

Diagram, schematic

Description automatically generated

## Vergelijken

Het vergelijken van twee getallen is niet heel moeilijk. Je kijkt naar het eerste cijfer dat verschilt en het getal met daar het hoogste cijfer is het hoogst. Als alle cijfers gelijk zijn dan zijn de getallen gelijk.

|  |  |
| --- | --- |
| Decimaal | Binair |
| 2**4**95 | 101**0**0110 |
| 2**6**94 | 101**1**0101 |
| 6 is groter dan 4 dus het tweede getal is groter | 1 is groter dan 0 dus het tweede getal is groter |

Diagram

Description automatically generated

4bit comparator

De xor-gates vinden welke bits er verschillend zijn. Daarna output de bitfinder naar rechts de index van de hoogste bit die aan staat en naar beneden of er een is die aanstaat. De selector (sel) laat de bit die de bitfinder aangeeft door en wordt dan door een and-gate gehaald met de output of er wel een verschil is tussen bits. Als deze bit van a aanstaat dan moet ‘a’ hoger dan ‘b’ zijn, want als een bit 1 is en een ander niet 1 dan moet de tweede bit 0 zijn oftewel lager. De output ‘equal’ staat aan als er geen bits zijn die verschillen en ‘smaller’ is aan als een ‘a’ niet hoger of gelijk is aan ‘b’.

Uitleggen vergelijken signed

## Memory

Het simpelste circuit om een bit mee op te slaan is de SR-flipflop.

Diagram

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Set | Reset | Q |
| 0 | 0 | - (verboden) |
| 1 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 1 | Q (wat Q hiervoor was) |

Het probleem met deze manier van opslaan is dat je vaak alleen iets wil opslaan wanneer klok (clk) hoog is aan het einde van het uitvoeren van een instructie. Daarvoor is de gated SR- flipflop.

Diagram

Description automatically generated

Zodra beide clk en set aanstaan gaat de reset van de SR-flipflop uit waardoor er wordt opgeslagen. Hetzelfde gebeurt met de set van de SR-flipflop als clk en reset aan staan.

|  |  |  |  |
| --- | --- | --- | --- |
| Clk | Set | Reset | Q |
| 0 | x | x | Q |
| 1 | 0 | 0 | Q |
| 1 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | - |

Er zijn nog twee laatste verbeteringen en dat zijn dat je wilt kunnen kiezen of je wilt opslaan. En dat je set en reset met één input kan doen.

Diagram

Description automatically generated

|  |  |  |
| --- | --- | --- |
| Save | Clk | Q |
| 0 | 0 | Q |
| 1 | 0 | Q |
| 0 | 1 | Q |
| 1 | 1 | Value |

Nu kunnen we eindelijk een bit opslaan op een handige manier. Het is redelijk makkelijk om meerdere bits aan memory aan elkaar te plakken tot bijvoorbeeld een 4bit register.

Diagram, schematic

Description automatically generated

Een register is nog niet heel handig, dus de volgende stap is RAM, oftewel Random Access Memory.

Diagram, schematic

Description automatically generated

4x8bit RAM

Dit is een RAM-circuit dat 4 keer 8bits kan opslaan. Elke van de vier registers is verbonden met de waarde om op te slaan en de klok, maar slaat alleen de waarde op wanneer de middelste input van het register aanstaat. De enable input van het hele circuit kiest of er een register is die iets moet opslaan en de select2 input kiest welke. (6)

## Teller

Een teller of counter is een circuit dat een omhoog telt. Omhoog tellen kan gewoon door met een adder een op te tellen. Diagram

Description automatically generated

Incrementer

De counter die ik heb gemaakt en veel gebruik is eentje die elke keer dat de klok hoog wordt een optelt en als de ‘overwrite’ input aanstaat verder gaat met optellen vanaf de ‘value’ input.

Diagram

Description automatically generated

De select kiest om verder te gaan met tellen met het opgeslagen getal of een nieuwe waarde en zodra de klok hoog wordt de waarde opgeslagen. Als laatst zet de incrementer alvast de nieuwe waarde klaar voor als de klok weer hoog wordt.

## Floats

Een float is een binair getal met een komma in wetenschappelijke notatie. Float staat in het geheel voor een ‘Floating point number’. Mijn latere computer design werkt met 32bit of single-precision floats. De 32bit floats werken in drie delen: 1bit voor sign, 8bits exponent en 23bits aan fractie. De fractie wordt gezien alsof er een 1, Voor staat en de exponent – 128 is de werkelijke exponent.

|  |  |  |
| --- | --- | --- |
| 0 | 10111010 | 0011001010101001011001 |
|  | 186-128=58 | 1, + ,0011001010101101011001 |
| + | 2^58 | 1,19789719581604003906 |
| + | 10^17 | 3,45270359341137917387 |

Er zijn verder nog een paar waardes gereserveerd voor oneindig en NaN (not a number). NaN is een waarde die wordt gebruikt voor 0/0 en oneindig voor delen door 0. Verder is een bij effect van de sign bit dat er naast een 0 ook een -0 is.

Het uitleggen van hoe optellen, aftrekken, vermenigvuldigen en delen werkt bij floats is te ingewikkeld, maar de circuits zijn wel te vinden in de bijlage. (7)

Deze floats zijn alleen deel van de digital bijlage die iets minder overzichtelijk is.

Verder float uitleggen

## 

## Stack

Een stack is vorm van geheugen waar je een voor een nieuwe waardes op ‘gooit’ en elke keer dat je er een waarde afleest je de bovenste waarde verwijdert. Deze instructies heten ‘push’ en ‘pop’.

Diagram

Description automatically generated

De stack werkt met een RAM waarvan de (A)dres steeds telt als pop of push aanstaat, maar alleen omhoog als push aanstaat (input bovenkant counter) anders telt het omlaag. De tweede input van boven wordt via (D) geschreven wanneer de klok hoog is. De output is altijd de waarde op het huidige adres en de input in het midden van het circuit is de reset voor de counter die niet gebruikt wordt.

# Architectuur

Het probleem bij het maken van een computer is dat je niet weet welke losse functies nodig hebt voordat je klaar bent en dus heb ik twee designs gemaakt. De eerste is heel erg geïnspireerd door degene die je maakt in ‘Nandgame’, terwijl de tweede alle tekortkomingen oplost van de eerste en compleet alleen door mij is gemaakt.

## Overture

Overture is mijn eerste design. Net als de meeste computers volgt het de standaard instructiecylus: fetch -> decode -> execute.

Fetch: het haalt de instructie uit ROM

Decode: Hoe leg ik dit uit????

Execute: voert de instructie uit. Dit is in drie delen op te delen. Load: haalt informatie uit RAM, Execute: voert een operatie uit op de informatie, Store: slaat de uitkomst op in RAM.

Overture heeft vier soorten instructies (x en y in de bitcode zijn variabelen):

|  |  |  |
| --- | --- | --- |
| Bitcode |  |  |
| 00xx xxxx | Immediate | Sla x op in register 0, x is 6 bits lang dus maximaal 26-1 = 63 |
| 01xx xxxx | Copy | Kopieer register x naar register y. |
| 10-- -xxx | Operation | Voeg een instructie uit op register 1 en 2, sla de uitkomst op in register 3. (or, nand, nor, and, add, sub) |
| 11-- -xxx | Jump | Jump naar het adres in register 0 als register 3 voldoet aan de vergelijking, (>0,=0,>=0,<0,!=0,<=0,True) |

Overture werkt met acht adressen waar in elk 8bits van te lezen en naar te schrijven zijn. Deze bestaan uit 6 registers genummerd van 0-5, de counter die bijhoudt bij welke instructie je bent (adres 6) en I/O op adres 7.

Stap 1: decode

* Immediate: De instructie begint met 00 (1) en dus is het een immediate instructie (a1), een waarde moet worden opgeslagen (b), de waarde (value) die moet worden opgeslagen is de laatste 6 bits van de instructie (2)(c) en het wordt opgeslagen in register 0 (d).

### Memory

Diagram, schematic

Description automatically generated

Inputs:

* Clk: zorgt ervoor dat de nieuwe waarde pas wordt opgeslagen zodra de klok hoog wordt, dit is zodat waardes niet zichzelf beïnvloeden. Bijvoorbeeld: a = a + 2, als je de nieuwe a continu opslaat terwijl je bezig bent met berekenen kun je niet voorspellen wat het is.
* Destination: Zorgt dat je kan kiezen waar je een waarde opslaat.
* De rest van de uitleg

### ALU

### Comparisons

### Decoder

### Totaal

Diagram

Description automatically generated

Overleggen met Chris hoe ik ga uitleggen hoe Overture werkt

## Problemen van Overture

Laten we voor het vaststellen van de problemen een keer langs alle originele eisen gaan.

1: Minimaal 8bit words

Dit blijkt duidelijk te weinig om mee te werken. De acht bits van Overture zijn net genoeg om de eerste 12 getallen van de fibonacci reeks berekenen (bijlage fibonacci code), maar 120 + 140 lukt al niet zonder heel wat moeite, omdat het boven 255 komt (bijlage optellen boven 255). Ik had 8bit gebruikt voor Overture, omdat ik niet wist hoe moeilijk groter wordt lengtes zouden worden, maar dat lijkt redelijk makkelijk en daarom wordt Regi 32bit.

2: Operaties op twee willekeurige registers

Overture kan dit niet. Dit zorgt ervoor dat hoewel ADD maar één instructie is, het eerst twee instructies kost om de waardes die je optelt naar registers 1 en 2 te verplaatsen en daarna nog één om de uitkomst verplaatsen zodat de volgende uitkomst er niet overheen schrijft. Dit probleem wordt verholpen in Regi.

3: waarden vergelijken

Ik was origineel van plan om uiteindelijk COMP en JUMP te gebruiken, maar door Overture heb ik gemerkt dat alleen JUMP genoeg was. Dit komt, omdat je in plaats van iets conditioneel uit te voeren je ook het bij het tegenovergestelde kan overslaan (‘als a>2 doe a-2’ -> ‘als a <= 2 doe niet a-2’). Regi zal dus weer alleen JUMP gebruiken. Een probleem bij Overture is dat je waardes kan vergelijken met 0, dit is onhandig en Regi zal twee willekeurige waarden kunnen vergelijken.

4: Kopiëren

Overture heeft COPY en door punt 2 is dat heel erg nodig, nog meer dan normaal. Regi zal dus natuurlijk ook COPY als instructie hebben.

5: input & output

Bij Overture merk je door hoe erg punt 2 tegenwerkt niet goed hoe onhandig het hebben van maar één input en één output register is. Het is wel iets waar ik al voorzag dat er last van zou krijgen bij Regi. Je hebt veel aan twee outputs bij het schrijven van een scherm zodat je een voor x en een voor de y coördinaat hebt.

6: immediates

Overture kan immediates gebruiken, maar alleen bij het schrijven naar register 0 en niet naar andere of bij operaties. Dit zorgt ervoor dat je voordat je een immediate kan gebruiken in een som, je het eerst moet kopiëren naar register 1 of 2 waardoor het minder een ‘immediate’ wordt. Regi zal op dezelfde manier als punt 2 ook meteen dit punt oplossen.

7: logische operaties

Overture heeft een paar logische operaties, maar geen XOR, XNOR, NOT, TEST en HIGH. Ik heb hier nog geen problemen mee gehad, maar toch zal Regi die wel voor de zekerheid hebben.

8: ADD en SUB

Overture had optellen en aftrekken, maar dit waren niet genoeg wiskundige operaties om mee te goed mee te werken. Als je met Overture twee getallen, a en b, wil vermenigvuldigen dan kost dat 11\*(min(a,b)) dat staat gelijk aan O(n), hoewel dat niet heel langzaam is kost dat wel een hoop extra moeite met programmeren. Regi zal instructies hebben voor vermenigvuldigen en delen. Dat brengt het terug naar O(1) en maakt programmeren een stuk simpeler. (Bijlage overture vermenigvuldigen)

9: één instructie per cyclus

Overture heeft dit en Regi zal het ook hebben.

10: shift

Dit is een nieuw punt waar de computer aan moet voldoen. Shift helpt erg bij het vermenigvuldigen met machten van twee en dingen die werken via hetzelfde soort algoritme als mod-pow.

11: geheugen van minstens 16 registers

Dit is een tweede nieuwe eis. Overture heeft slechts 6 registers en dat is duidelijk te weinig om ingewikkeldere data in op te slaan. Je kan er bijvoorbeeld alleen ‘Hello ’ van de standaard ‘Hello world!’ van programmeren in opslaan.

## Regi V1

Er zijn veel verbeteringen bij Regi V1 ten opzichte van Overture. Zo is bijvoorbeeld het programma om twee getallen te vermenigvuldigen gegaan van 29 regels naar 1 regel.

Het uitleggen van Regi is te ingewikkeld om binnen dit document te doen. Het werkt volgens dezelfde principes als Overture, maar dan uitgebreider. De simulatie bestanden en de documentatie van Regi zullen bij dit document worden gevoegd voor als je wilt kijken hoe alles precies werkt. Ook staan er in de documentatie minder belangrijke specificaties.

De lijst van eisen na Overture: (geen opmerking betekent dat het eraan voldoet)

1: Minimaal ~~8bit~~ **32bit** words

2: Operaties op twee willekeurige registers

3: waarden vergelijken**, zonder dat een van de twee 0 is**

Een nieuw verbeterpunt dat ik heb gevonden met Regi V1 is dat het ook handig is om getallen met elkaar te kunnen vergelijken zonder minteken. Regi V1 ziet namelijk bij vergelijken getallen altijd als signed en dat wil je niet altijd en dat is dus een verbetering voor V2.

4: Kopiëren

Regi voldoet net als Overture aan dit punt, maar er is een nieuwe uitbreiding aan deze eis. Het is belangrijk om ook te kunnen kopiëren waar een pointer naar wijst. Als je bijvoorbeeld een waarde die in een lijst wil kopiëren dan wil je kopiëren waar de lijst-pointer naar wijst en niet die pointer zelf. Dit noem ik COPYPTR (copy pointer) en V2 zal dit hebben.

5: input & output,

Regi V1 heeft in plaats van een input en een output, drie inputs en vier outputs.

6: immediates**, in alle instructies**

Regi V1 heeft dat het als alle variabelen in instructies ook variabelen kan gebruiken. Het enige probleem is dat immediates maximaal 16bit kunnen zijn waardoor je geen min-getallen kan gebruiken als immediates. Dit is een te groot probleem om op te lossen zonder een groot redesign van de computer, dus dit wordt niet opgelost in V2.

7: logische operaties

8: ADD en SUB**, MULT en DIV**

Regi V1 heeft MULT en DIV nieuw, maar ook een hoop andere features. ADDC en SUBB zijn nieuw en gebruiken een eerder opgeslagen carry bij de som. Om dit te laten werken wordt nu bij elke wiskundige en shift operatie de carry opgeslagen.

9: één instructie per cyclus

10: **shift**

Regi V1 heeft beide shift left, shift right en rotate. Verder is er ook een carry versie van shift die de lege plekken opvult met de opgeslagen carry.

11: geheugen van meer dan ~~16~~ **32** registers

Regi V1 heeft 32 registers wat hier goed aan voldoet, maar het is nog steeds niet genoeg voor langere programma’s. Een voorbeeld is een plaatje van 8\*8 pixels, daar heb je al 64 registers aan geheugen voor nodig en dus is 32 registers niet genoeg om een klein plaatje in op te slaan.

12: **floats en flops**

Dit is een nieuwe eis. Regi V1 kan al veel wiskundige operaties, maar is moeilijk mee te werken als je fracties hebt. Als je 3 deelt door 2 dan krijg je 1 rest 1, maar je kan geen komma getal als uitkomst hebben. Floats lossen dit op door 23 bits te gebruiken voor precisie en 8 bits voor een exponent. Dit werkt precies zoals wetenschappelijke notatie maar dan met een macht van 2 (bijv. 1,493 \* 2^11).

## Regi V2

Regi V2 is de opvolger van V1. Het is niet een compleet nieuw design, maar een aanpassing op de vorige. Dit betekent ook dat er maar een paar punten zijn aangepast: vergelijken (3), geheugen (11) en floats (12).

3: vergelijken

Naast de manier van vergelijken van Regi V1 kan V2 ook twee unsigned getallen en floats met elkaar vergelijken.

11: geheugen

Naast de 32 registers van V1 heeft V2 ook RAM. Om precies te zijn 2^24 registers of 64MB aan memory. Het enige nadeel aan deze extra RAM is dat copyptr niet werkt, omdat dat te complex wordt met het aantal verbindingen wat daarvoor nodig is.

12: floats

Regi V2 kan binair omzetten naar float en omgekeerd. Verder kan het floats optellen, aftrekken, vermenigvuldigen en delen.

# Programmeren

Het maken van een programmeertaal is niet een heel moeilijk concept. Je vervangt keywords met bitcode die de computer kan begrijpen. Dit is bij assembly heel makkelijk, omdat een keyword altijd wordt vervangen door hetzelfde bits. Higher level talen zijn moeilijker, omdat het een beetje werkt als een gesproken taal, net zoals bijvoeglijke naamwoorden kan een keyword van actie veranderen door een andere.

## Overture

Voor Overture heb ik alleen een assembly taal gemaakt.

Het makkelijkste deel is het vervangen van keywords. Dit is een voorbeeld:

|  |  |  |
| --- | --- | --- |
| Add | 10 000101 | 10000101 |
| Copy 2 5 | 01 010 101 | 01010101 |

‘Copy’ wordt vervangen door ‘01’ en daarna worden 2 en 5 omgezet naar binair en erachter geplakt.

Het moeilijker deel was het omzetten van labels en const’s. Labels verwijzen naar de regel waar ze voor staan, waardoor je de waarde van de label kan gebruiken om naar toe te loopen in plaats van steeds bij te moeten houden naar de hoeveelste instructie je toe wilt. Const’s helpen op ongeveer dezelfde manier. Je kan een const een constante waarde geven aan het begin van je programma waardoor je kan verwijzen naar een naam in plaats van een waarde.

|  |  |  |  |
| --- | --- | --- | --- |
| Geef 7 de naam output\* | const output 7  imme 9  copy 0 1  imme 1  copy 0 2  imme loop  label loop  copy 1 output  sub  copy 3 1  jump !=0 | imme 9  copy 0 1  imme 1  copy 0 2  imme 6  copy 1 7  sub  copy 3 1  jump !=0 |  |
|  |  |
| Laad 9 in reg0 | 00 001001 |
| Reg1 ← reg0 | 01 000 001 |
| Laad 1 in reg0 | 00 000001 |
| Reg2 ← reg0 | 01 000 010 |
| Laad loop in reg0 | 00 000110 |
| \*\* |  |
| Loop = (regel) 6\*\*\* |  |
| Geef reg1 als output | 01 001 111 |
| Reg3 ← reg0 – reg1 | 10 000101 |
| Reg1 ← reg3 | 01 011 001 |
| Ga naar regel reg0 als reg3 niet 0 is | 11 000101 |

\*register 7 is het register waar je naar schrijft om een output te geven

\*\* memory op dit punt voor de registers 0-5 is: (6,9,1,0,0,0)

\*\*\* label en const zijn geen instructies, dus telt ook niet mee voor regelnummer

Dit programma geeft eerst 9 als output, trekt er dan een van af en als het gelijk is 0 doet het dan opnieuw met nu een getal dat een lager is. Dus de output is (9,8,7,6,5,4,3,2,1).

Iets wat niet voorkomt in dit voorbeeld zijn comments. Je kan comments gebruiken in je code door die na een ‘;’ te typen, want de assembler negeert alles op een regel na een ‘;’.

## Regi

Ik heb voor Regi twee programmeertalen gemaakt. High en RASM, High compileert naar RASM en RASM assembled naar bitcode. Het werkt dus in lagen.

### RASM

RASM is te vergelijken met de assembly taal van Overture, maar dan ingewikkelder doordat de computer ook ingewikkelder is.

De bitcode is nu als volgt ingedeeld:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 0b xxxx xxxx klmx xxxx aaaa aaaa aaaa aaaa bbbb bbbb bbbb bbbb cccc cccc cccc cccc | | | | |
| x | klm | a | b | c |
| Dit deel van de instructie geeft aan welke operatie moet worden uitgevoerd  (0b 0000 0000 yyy1 0000 = add) | k: als deze 0 is dan is de eerste operand (a) een pointer en geen immediate  i: hetzelfde als k, maar met de tweede operand (b)\*  m: kiest of opgeslagen wordt in (c) of opgeslagen waar de waarde in (c) naartoe wijst | Operand 1 | Operand 2 | destination |

\*’l’ wordt ‘x’ als de operatie minder dan twee operands heeft

Voorbeeld:

|  |  |
| --- | --- |
| add 8 3 <5> | Add: x = 0000 0000 ---1 0000  <>: klm = 110  a: 0000 0000 0000 1000  b: 0000 0000 0000 0011  c: 0000 0000 0000 0101 |

‘<>’ om een waarde maakt het van een immediate een pointer en is dus het tegenovergestelde van klm het vorige schema.

Nu een voorbeeld met hetzelfde programma als voor Overture, maar om het overzichtelijker te maken haal ik ben lange binaire getallen alle nullen waar het mee begint weg en bij alle operaties (x) in dit programma zijn dat de eerste acht. Het is belangrijk om te weten dat jumpNE, jump Not Equal betekent.

|  |  |  |  |
| --- | --- | --- | --- |
| Geef 32 de naam output\* | const output 20  const i 1  copy 9 i  label loop  copy <i> output  sub <i> 1 i  jumpne <i> 0 loop | add 9 0 1  add <1> 0 20  sub <1> 1 1  jumpne <1> 0 2 |  |
|  |  |
|  | Klm x a b c |
| i ← 9 (stiekem: i←9 + 0) | 000 10000 1001 0 1 |
|  |  |
| Loop = (regel) 2\*\* |  |
| Geef i als output | 100 10000 1 1 10100 |
| i ← i - 1 | 100 10010 1 1 1 |
| Ga naar loop als i niet gelijk aan 0 is | 100 00101 1 0 10 |

\*register 0x20 is het register waar je naar schrijft om een output te geven en dat is 32 in binair want alle getallen in mijn code zijn automatisch hexadecimaal als je er niet ‘0d’ voor zet.

\*\* label en const zijn geen instructies, dus telt ook niet mee voor regelnummer

Naast de veranderingen die aan deze voorbeelden duidelijk zijn is er nog andere en dat is het kunnen importeren van andere files om te gebruiken in je code. Als je het vorige programma bijvoorbeeld ‘cijfer.ovt’ noemt dan kun je je programma beginnen met ‘import cijfer.ovt’ en vanaf dan kun je in je code ‘cijfer.ovt’ callen.

### High

High heeft veel verbeteringen vergeleken met RASM, maar de duidelijkste is dat je niet meer al je variabelen een naam hoeft te geven via ‘const’ en de compiler dat nu voor je doet.

Hetzelfde programma als eerder in High wordt:

|  |  |
| --- | --- |
| i ← 9 | i = 9  while i != 0:      output = i      i = i – 1 |
| Zolang i niet 0 is: |
| Geef i als output |
| i ← i - 1 |

Het heeft geen zin om te laten zien hoe dit bitcode wordt, omdat niet elke regel een instructie is. Dit programma compiled naar:

Text

Description automatically generated

Het stuk code is duidelijk korter geworden, maar minder efficient door while

High heeft nog een hoop andere helpvolle keywords, zoals ‘array(0,1,2,3)’ waarmee je een lijst kan maken in een regel of ‘z[2]’ waarmee je snel een element uit een lijst kan krijgen, maar die zijn te uitgebreid om het over te hebben in dit verslag.

# Conclusie

Het is duidelijk dat er veel eisen zijn voor een computer om gebruiksvriendelijk te zijn in assembly. De eisen die ik bij het programmeren het belangrijkste vond waren memory gerelateerd. Het jongleren van adressen die ik bij Overture moest doen maakte alles onoverzichtelijk en het hebben van 8bit words was niet omheen te werken.

De volgende belangrijke punten zijn de hoeveelheid adressen, want dat zorgde in eerdere Overture en Regi V1 dat het onmogelijk was om grotere hoeveelheden data op te slaan voor tekst of plaatjes.

De verdere onmisbare punten waren het hebben van immediates en IO. Zonder immediates kun je geen getallen in je computer krijgen. Terwijl zonder IO het onmogelijk is om informatie van buitenaf te krijgen of om informatie af te geven waar er je geen beeldscherm kan gebruiken en moet je een nieuw programma schrijven elke keer dat je je variabelen wilt veranderen.

Het hebben van optellen en aftrekken, en het hebben van logische operaties is onmisbaar. Anders kun je sommige sommen niet uitvoeren. Andere instructies zijn ook belangrijk voor het simpeler maken, maar zijn omheen te werken.

Ik weet niet hoe belangrijk het is om een instructie te hebben per klokcyclus, maar ik denk dat het handig is om te hebben of het is in ieder geval moeilijk om het niet te hebben en nog steeds gebruiksvriendelijk te zijn.

Een extra punt waarmee ik uit dit onderzoek kom is dat het mij nog duidelijker is geworden waarom bijna niemand meer puur in assembly programmeert. Het is gewoon te onhandig, nadat ik mijn compiler voor Regi V2 heb geschreven ben ik (bijna) compleet gestopt met het programmeren in RASM, omdat een hogere taal overzichtelijker en sneller is om in te programmeren.

# Evaluatie

Het is mij uiteindelijk veel beter gelukt om een computer te maken dan ik van tevoren had gedacht. Ik had bijvoorbeeld gehoopt om simpele formules uit te rekenen, maar het is uiteindelijk gelukt om zelfs Pong te spelen en fractals te generen met de computer.

Er zijn natuurlijk nog wel punten die ik zou kunnen verbeteren zoals het toevoegen van een manier om (pseudo-)willekeurige getallen te genereren of een manier om te zien hoeveel instructies er zijn uitgevoerd om op die manier bijvoorbeeld de computer te timen. Ook had ik meer tijd kunnen steken in het precies uitleggen van hoe de computer werkt, want dat is wat ik het lastigste vond. Dit komt doordat de designs veel kleine iteraties hebben gehad in plaats van de drie grote die ik heb genoemd en ik heb niet elke kleine verandering opgeschreven waardoor ik nu niet meer mijn hele stappen proces kan uitleggen.

Als vervolgstudie op dit onderzoek is het mogelijk om verder te gaan met de software kant van dit onderzoek. Het maken van libraries om programmeren in high makkelijker te maken en het maken van een betere compiler zodat het compiled naar efficiëntere code

# Bronnen

1. **cloutier, felix.** x86. *felix cloutier.* [Online] mrt. 20, 2021. https://www.felixcloutier.com/x86/.

2. **Kjær, Olav Junker.** *nandgame.* [Online] mrt. 15, 2021. https://nandgame.com/.

3. **Steve.** Binary Numbers Explained. *steves internet guide.* [Online] nov. 11, 2022. http://www.steves-internet-guide.com/binary-numbers-explained/.

4. **Contributor, TechTarget.** logic gate (AND, OR, XOR, NOT, NAND, NOR and XNOR). *whatis.* [Online] nov. 5, 2022. https://www.techtarget.com/whatis/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR#:~:text=A%20logic%20gate%20is%20a,of%20logic%20gates%20in%20them.

5. **Rombauts, Karl.** Building an 8-bit computer in Logisim (Part 2— Arithmetic). *medium.* [Online] nov. 6, 2022. https://medium.com/@karlrombauts/building-an-8-bit-computer-in-logisim-part-2-arithmetic-ae7861c82e79.

6. **electronicsforu.** Flip Flop Basics – Types, Truth Table, Circuit, and Applications. *electronicsforu.* [Online] nov. 6, 2022. https://www.electronicsforu.com/technology-trends/learn-electronics/flip-flop-rs-jk-t-d.

7. **wikipedia.** Single-precision floating-point format. *wikipedia.* [Online] nov. 5, 2022. https://en.wikipedia.org/wiki/Single-precision\_floating-point\_format#:~:text=Single%2Dprecision%20floating%2Dpoint%20format%20(sometimes%20called%20FP32%20or,using%20a%20floating%20radix%20point.

# Bijlagen

### Logisim

Om logisim te gebruiken moet je eerst de installer gebruiken genaamd: ‘logisim-evolution-3.8.0-x86.msi’.

‘voorbeelden.circ’ bevat alle kleinere circuits die voorbeelden zijn in dit verslag, ‘OVERTURE.circ’ bevat alle onderdelen van Overture en ‘V232bitfloatstack - Copy - Copy.circ’ die van Regi V2 (De rare naam komt van alle versies die er zijn geweest).

Links in het paarse vierkant kun je met dubbelklik onderdelen selecteren, met de hand geselecteerd kun je inputs veranderen en links onder kun je de zoom aanpassen.

Graphical user interface, application

Description automatically generated

### Digital

Om

-moeilijke woorden

Plaatje dat misschien gebruikt gaat worden:

Diagram

Description automatically generated

TO-DO:

-uitleggen overture

-Documentatie Overture, RASM en High als bijlagen

-uitleggen hoe assembler/compiler/digital werkt

-Bijlage moeilijke woorden

-Bijlage code/programma’s (vermenigvuldigen + optellen boven 255: overture)

-bijlage logboek

6 left (7 met float)

Hoofdvragen toevoegen?