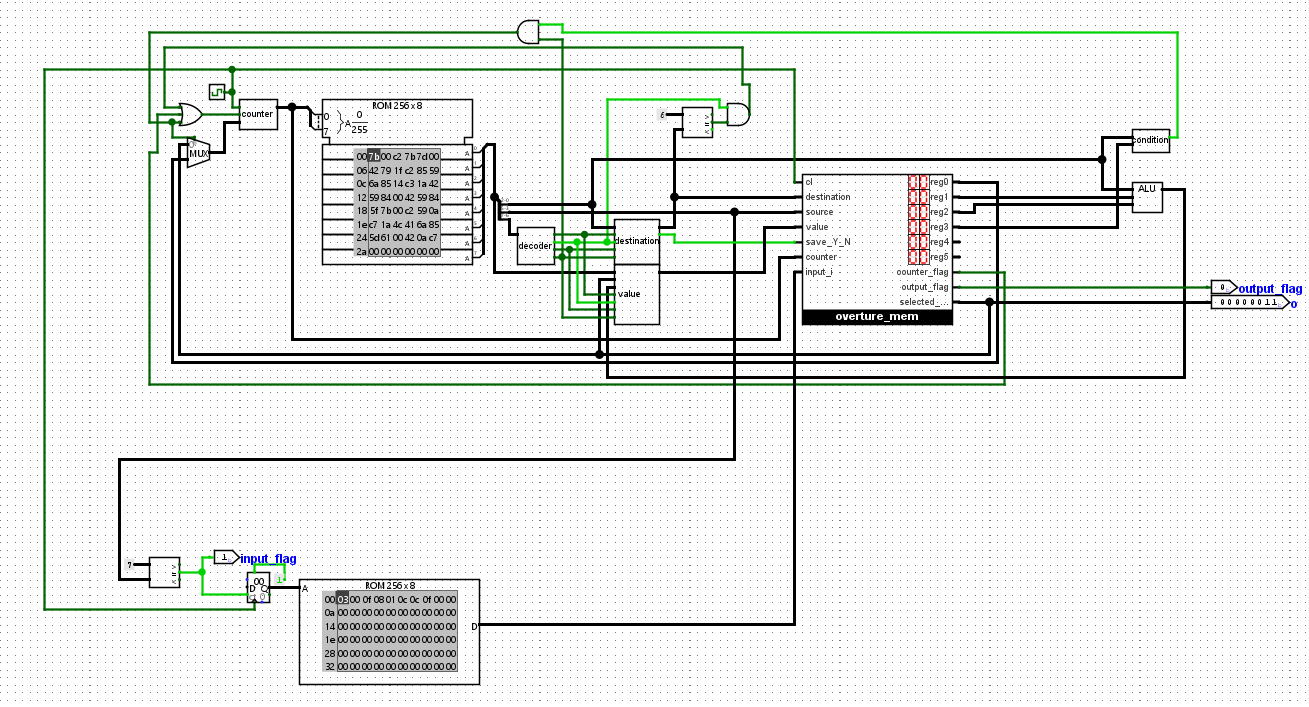
Computerdesigns

Designs for whole computers and computer specifics components



# Inhoud

[Inhoud 2](#_Toc99444942)

[Inleiding 3](#_Toc99444943)

[Indeling 3](#_Toc99444944)

[Doelen 4](#_Toc99444945)

[Architectuur 5](#_Toc99444946)

[Assembly 6](#_Toc99444947)

[Overture: 6](#_Toc99444948)

[OVT: 6](#_Toc99444949)

[Regi: 8](#_Toc99444950)

[RASM: 8](#_Toc99444951)

[Componenten 10](#_Toc99444952)

[Logic gates 11](#_Toc99444953)

[Plexers 11](#_Toc99444954)

[Rekenkundig 11](#_Toc99444955)

[Half Adder (HA) 11](#_Toc99444956)

[Full Adders (FA) 11](#_Toc99444957)

[Memory 13](#_Toc99444958)

[Set-Reset flip flop (SR flip-flop) 14](#_Toc99444959)

[Data flip-flop (D flip-flop) 14](#_Toc99444960)

[Data flip-flop met klok (DCl flip-flop) 15](#_Toc99444961)

[Logic Units 15](#_Toc99444962)

[Binary Logic Unit (BLU) 15](#_Toc99444963)

[Arithmatic Logic Unit (ALU) 15](#_Toc99444964)

[OVERTURE 15](#_Toc99444965)

[Appendix A: Code 16](#_Toc99444966)

[OVT: 16](#_Toc99444967)

[Multiplication: 16](#_Toc99444968)

[Appendix B: diagrams 17](#_Toc99444969)

[Appendix C: Truth tables 17](#_Toc99444970)

[Appendix D: terminologie 18](#_Toc99444971)

# Inleiding

## Indeling

This document is divided into three mean parts: architecture, assembly and components. Each part is separated into chapters. At the end of this document are appendices with bigger diagrams and explanation of terminology.

Architecture is about the design of whole computers and will often refer to ‘Components’ for simpler components.

Assembly is the languages that’s closest to machine code. In this part explains all instructions and workings for the languages made by me for the computers. Het bevat ook de documentatie voor mijn zelfgemaakte taal |||||| dat een higher level language[[1]](#footnote-2) ten opzichte van assembly.

In the part components are simpler designs made from logic gates. These designs do simple things like addition or holding information.

## Goals

I want to work towards a computer that meets the following points:

1. At least 8bit words.[[2]](#footnote-3)
2. Words don’t have to be loaded into a specific register before an operation.[[3]](#footnote-4)
3. Can do the most used operations:

* add (ADD) and subtract (SUB)
* copy (COPY)
* compare two numbers (COMP) and looping if a condition is met (JUMP)
* store a number directly from an instruction
* taking inputs (IN) and giving outputs (OUT) to things outside of the computer
* Logic operations (AND, NAND, OR, NOR, XOR, XNOR, NOT, TEST) [cross-reference]

1. (Bijna) alle instructies in een cycle[[4]](#footnote-5)

The words that are within parentheses are the keywords that are often used in assembly.

Deze specificaties zijn gekozen om een computer te maken die zo gebruiksvriendelijk mogelijk is. Words van 8 bits of meer zorgt voor het kunnen opslaan van een karakter in een adres. Het niet hoeven laden zorgt ervoor dat je veel minder ‘memory management’ hoeft te doen en dat is normaal een van de dingen is die je code veel minder overzichtelijk maakt. Het gebruiken van de meest gebruikte operations zorgt ervoor dat je mijn versie van assembly veel sneller kan leren als je x86 assembly[[5]](#footnote-6) al kent en het zorgt ervoor dat je voor veel gebruikte programma’s niet om missende instructies hoeft heen te werken. Door de laatste punt hoef je niet rekening te houden met instructies waarvan je de uitkomst nog niet hebt bij het uitvoeren van de volgende instructie.

# Architectuur

# Assembly

This chapter is about the programming languages for all my computers:

* Overture
* Regi

Keyword: Een keyword is een woord dat aangeeft wat voor instructie of wat voor optie van een instructie wordt bedoeld, dus als ergens het keyword ‘copy’ staat dan geeft het aan dat je de instructie om te kopiëren bedoelt.

Immediate: Een waarde die direct via instructie wordt geladen in een de computer. ‘Imme 3’ zou bijvoorbeeld ‘3’ meteen laden als waarde (‘Imme’ is de keyword voor immediate).

Pointer: A value that ‘points’ at a register.

## Overture:

This part has all the languages I made for Overture.

OVT is the simplest language I made, it’s an assembly language.

### OVT:

OVT has its own extension ‘.ovt’.

The instructions in OVT are divided into four categories:

* 00xx xxxx: immediate (imme)
* 01xx xyyy: copy
* 10-- -xxx: operations (ops)
* 11-- -xxx: jump

A dash means that that bit doesn’t have any effect on the instruction, and ‘x’ and ‘y’ are numbers that control what the instruction does.

Instructions:

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| Imme <x> | Load a 6bit number into reg0 | 00xx xxxx |
| Copy <x> <y> | Copy reg(x) to reg(y) | 01xx xyyy |
| Or | Or reg1 and reg2, store the result in reg3 | 10-- -000 |
| Nand | Nand reg1 and reg2, store the result in reg3 | 10-- -001 |
| Nor | Nor reg1 and reg2, store the result in reg3 | 10-- -010 |
| And | And reg1 and reg2, store the result in reg3 | 10-- -011 |
| Add | Add reg1 and reg2, store the result in reg3 | 10-- 100 |
| Sub | Sub reg1 and reg2, store the result in reg3 | 10-- -101 |
| Jump false | Don’t jump. | 1100 0000 |
| Jump >0 | If reg3 >0, jump to reg0 | 1100 0001 |
| Jump =0 | If reg3 =0, jump to reg0 | 1100 0010 |
| Jump >=0 | If reg3 >=0, jump to reg0 | 1100 0011 |
| Jump <0 | If reg3 <0, jump to reg0 | 1100 0100 |
| Jump !=0 | If reg3 ≠0, jump to reg0 | 1100 0101 |
| Jump <=0 | If reg3 <=0, jump to reg0 | 1100 0110 |
| Jump | Jump to reg0 | 1100 0111 |

OVT isn’t case sensitive so both ‘Nand’ and ‘NAND’ work.

Registers:

|  |  |
| --- | --- |
| Name | Function |
| Reg0 | Immediate |
| Reg1 | First operand |
| Reg2 | Second operand |
| Reg3 | Result of operation |
| Reg4 | - |
| Reg5 | - |
| Reg6 | PROGRAM COUNTER, this controls which instruction is executed. |
| Reg7 | Not really a register. This is an input from outside the computer when used as source and an output when used as destination. |

Const <name> <value>: Every .ovt program can start with a few lines that start with ‘const’.

assigns the value <value> to <name>. From then on you use <name> instead of <value>.

Label <name>: assigns the line number of the next instruction to <name>. This is useful for loops.

; (comments): everything after ‘;’ is seen as a comment (until a new line).

Example of code with and without consts, labels and comments:

|  |  |
| --- | --- |
| Code without | Code with |
| copy 7 3  imme 0  jump =0  copy 3 7 | const input 7    ;input as name for reg7  const output 7   ;output "  "    "   "  label loop       ;loop until input !=0  copy input 3  imme loop  jump =0  copy 3 output   ;output the first non-zero input |

This program copies an input (reg7) to reg3 and uses the immediate value ‘loop’ (0) to loop. It loops until the input isn’t 0 and then outputs that input.

The second version of the code looks worse for small snippets like this but for larger programs it’s a vast improvement. (Example: 14)

The above code is compiled to the following machine code:

* Binary: 01111011 00000000 11000010 01011111
* Hex: 7b 00 c2 5

All numbers in the code are hexadecimal. In ‘Imme 11’, ‘11’ = 0x11 = 17

## Regi:

### RASM:

RASM differs from OVT in that it has more instructions but more importantly it has 4 bytes per instruction.

The function of the bytes is as followed:

1. Instruction byte (opcode)
2. Source byte for the first operand (src1)
3. Source byte for the second operand (scr2)
4. Destination for where the result should be stored (dst)

This Regi also has more complex memory so a few notation differences:

|  |  |  |
| --- | --- | --- |
|  | operand | Result |
| a | The value a | Store at a |
| <a> | The value at register a | Store at the value in register a. (store at <a>) |
| [a] | Option to choose between a and <a> | Option to choose between a and <a> |

Some instructions start with a, b and/or c (i.e., abc01000). This signifies where the inputs come from and where the result is stored.

|  |  |  |
| --- | --- | --- |
| a | =0 | First operand is from the register where src1 points. (<src1>) |
|  | =1 | First operand is an immediate with value src1. |
| b | =0 | Second operand is from the register where src1 points. (<src2>) |
|  | =1 | Second operand is an immediate with value src1. |
| c | =0 | Result is saved at the register that is given by register dst. (<<dst>>) |
|  | =1 | Result is saved at the register dst. (<dst>) |

-> [z]: store at [z]

Like OVT, keywords aren’t case sensitive, but variables assigned with const or label are.

The instructions are divided into four categories:

1. xxx00xxx: loops & functions (& halt/exit)
2. xxx01xxx: (bitwise) logic
3. xxx10xxx: arithmetic
4. xxx11xxx: bit shifts & stack

A dash means that that bit doesn’t have any effect on the instruction, and ‘x’ and ‘y’ are numbers that control what the instruction does.

loops & functions

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| Call [x] | Go to function [x][[6]](#footnote-7) | a000 0000 |
| Ret | Return to where call/icall was last used | 0010 0000 |
| ICall [x] | Go to function [x] if there has been an interrupt since last time icall/iret was used | a100 0000 |
| IRet | Return to where call/icall was last used if there has been an interrupt since last time icall/iret was used | 1010 0000 |
| Hlt | Halt the program until next interrupt | 0110 0000 |
| Exit | Exit program | 1110 0000 |
| JumpGt [x] [y] [z] | If x > y, jump to z (greater than) | abc0 0001 |
| JumpEq [x] [y] [z] | If x = y, jump to z (equal) | abc0 0010 |
| JumpGE [x] [y] [z] | If x >= y, jump to z (greater equal) | abc0 0011 |
| JumpLt [x] [y] [z] | If x = y, jump to z (less than) | abc0 0100 |
| JumpNE [x] [y] [z] | If x ≠ y, jump to z (not equal) | abc0 0101 |
| JumpLE [x] [y] [z] | If x <= y, jump to z (less equal) | abc0 0110 |
| Jump [x] [z] | Jump to x||z | a0c0 0111 |
| CopyPtr [x] [y] | Copy <<x>> y | a1c0 0111 |

(nog niet toegevoegd: Call en icall lezen bij a=1, twee data bytes zodat je ook functions kan hebben na line 256) (ook niet toegevoegd: jump leest bij a=1, twee data bytes voor het jumpen naar lines na line 256)

(bitwise) logic

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| and [x] [y] [z] | x&y->z | abc0 1000 |
| or [x] [y] [z] | x|y->z | abc0 1001 |
| nand [x] [y] [z] | ¬(x&y)->z | abc0 1010 |
| nor [x] [y] [z] | ¬(x|y)->z | abc0 1011 |
| xor [x] [y] [z] | x^y->z | abc0 1100 |
| xnor [x] [y] [z] | ¬(x^y)->z | abc0 1101 |
| test [x] [y] [z] | Checks if the bit with index y in x is a 1. If it is a one: 1->z, else 0->z | abc0 1110 |
| not [x] [z] | ¬x->z | a0c0 1111 |
| high [x] [z] | Stores the index of the highest bit in z. If there is no 1 then it stores 16 (0x10000). | a1c0 1111 |

arithmetic

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| add [x] [y] [z] | x+y->z (set carry flag when overflowed) | abc1 0000 |
| addc [x] [y] [z] | Add x, y and the carry flag then store in z (set carry flag when overflowed) | abc1 0001 |
| Sub [x] [y] [z] | x-y->z (set carry flag when underflowed) | abc1 0010 |
| subb [x] [y] [z] | Subtract y and carry flag from x then store in z (set carry flag when underflowed) | abc1 0011 |
| mult [x] [y] [z] | x\*y->z (overflow -> carry register) | abc1 0100 |
| div [x] [y] [z] | x/y->z (x mod y -> carry register) | abc1 0101 |
| inc [x] [z] | x+1->z | a-c1 0110 |
| dec [x] [z] | x-1->z | a-c1 0111 |

Bit shift & stack

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
| shl [x] [y] [z] | Shift x a total of y places to the left->z (0<=y<=16)(set carry register) | abc1 1000 |
| shlc [x] [y] [z] | shift x a total of y places to the left and fill from the right from carry ->z (0<=y<=16)(set carry register) | abc1 1001 |
| shr [x] [y] [z] | Shift x a total of y places to the right->z (0<=y<=16)(set carry register) | abc1 1010 |
| shrc [x] [y] [z] | shift x a total of y places to the right and fill from the left from carry ->z (0<=y<=16)(set carry register) | abc1 1011 |
| Rotl [x] [y] [z] | Rotate x a total of y places to the left -> z | abc1 1100 |
| Rotr [x] [y] [z] | Rotate x a total of y places to the right -> z | abc1 1101 |
| copy [x] [y] [z] | X||y ->z | abc1 1110 |
| Push [x] | Increment stack pointer and then push x to stack | a001 1111 |
| Pop [x] | Pop from stack to x and then decrement stack pointer | a011 1111 |
| Exec [x] | Execute reg(x)||reg(x+1) | a101 1111 |

The stack grows down from memory address 0 and shares the memory with all other operations.

Conditionals (only after RASM v2.0)

|  |  |  |
| --- | --- | --- |
| keyword | description | instruction |
|  |  | 1 a000 0000 |
|  |  | 1 0010 0000 |
|  |  | 1 a100 0000 |
|  |  | 1 1010 0000 |
|  |  | 1 0110 0000 |
|  |  | 1 1110 0000 |
| uJumpGt [x] [y] [z] | Unsigned If x > y, jump to z (greater than) | 1 abc0 0001 |
| uJumpEq [x] [y] [z] | Unsigned If x = y, jump to z (equal) | 1 abc0 0010 |
| uJumpGE [x] [y] [z] | Unsigned If x >= y, jump to z (greater equal) | 1 abc0 0011 |
| uJumpLt [x] [y] [z] | Unsigned If x = y, jump to z (less than) | 1 abc0 0100 |
| uJumpNE [x] [y] [z] | Unsigned If x ≠ y, jump to z (not equal) | 1 abc0 0101 |
| uJumpLE [x] [y] [z] | Unsigned If x <= y, jump to z (less equal) | 1 abc0 0110 |
| uJump [x] [z] | Unsigned Jump to x||z | 1 a0c0 0111 |
|  |  | 1 a1c0 0111 |

Registers:

* reg0-31: registers
* Reg32: carry
* Reg33-35: input/output
* Reg36-37: output
* reg32768 (0x8000): (read only) ram or (write only) set ram address
* reg 40960-49151 (0xA000-0xBFFF): write to specified address in ram
* Reg49152 (0xC000): (write only) counter

Const <name> <value>: Every .ovt program can start with a few lines that start with ‘const’.

assigns the value <value> to <name>. From then on you use <name> instead of <value>.

Label <name>: assigns the line number of the next instruction to <name>. This is useful for loops.

; (comments): everything after ‘;’ is seen as a comment (until a new line).

# Components

Dit hoofdstuk gaat over alle componenten die ik gebruik in de architectuur van de computers. Elk onderdeel wordt uitgelegd met een plaatje, een ‘truth table’ en een stukje tekst.

Een ‘truth table’ is een tabel dat laat zien welke input leidt tot welke uitput. Hierbij: 1=aan en 0=uit.

|  |  |  |
| --- | --- | --- |
| a | b | o |
| 0 | 0 | 0 |
| 0 | 1 | 1 |
| 1 | 0 | 1 |
| 1 | 1 | 0 |

Dit is de ‘truth table’ van een XOR-gate, een XOR is alleen aan als precies een van de inputs een 1 is. Dit kun je zien doordat alleen in een rij met maar een 1 in de eerste twee vakjes een 1 in het derde vakje heeft.

Dit hoofdstuk bestaat uit een paar onderdelen:

* Logic gates: uitleg van hoe logic gates werken
* Plexers: componenten die kiezen van welke draad informatie doorgaat of naar welke draad informatie toe gaat
* Rekenkundig: componenten gemaakt van logic gates die simpel rekenkundige operaties uitvoeren
* Memory: componenten die informatie kunnen opslaan
* Logic Units: componenten die ingewikkeldere operaties uitvoeren en uit eerdere componenten bestaan

Als input uit meer dan een bits bestaan dan krijgen ze een nummer erbij die de significantie aangeeft. Zo heeft a0 een waarde van 1 en a2 een waarde van 4. De waarde is 2^n.

Grotere diagrammen en truth tables plaats ik niet in de uitleg zelf, want daar zijn ze vaak te groot voor, daarom verwijs ik dan naar de bijlage voor dat figuur.

Alle diagrammen zijn ingeleverd in een los bestand van Logisim.

## Logic gates

|  |  |
| --- | --- |
| ­­­­­­ ¬ | not |
| & | And |
| | | Or |
| ^ | xor |

## Plexers

## Rekenkundig

### Half Adder (HA)

Diagram

Description automatically generated

Een half adder kan twee 1bit getallen bij elkaar optellen. De carry is voor als de uitkomst niet in een bit ‘past’, dus bij ’ in binair. Dit is hetzelfde als wanneer je in decimaal hebt en dus een extra getal nodig hebt.

|  |  |  |  |
| --- | --- | --- | --- |
| a | b | sum | carry |
| 0 | 0 | 0 | 0 |
| 0 | 1 | 1 | 0 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |

### Full Adders (FA)

**1bit Full Adder:**

Diagram

Description automatically generated

De 1bit full adder kan drie 1bit getallen bij elkaar optellen, dit is nodig om later een 2bit full adder te maken door de carry van een eerdere full adder met de carry\_in van de tweede te verbinden

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| C\_in | a | b | sum | carry |
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

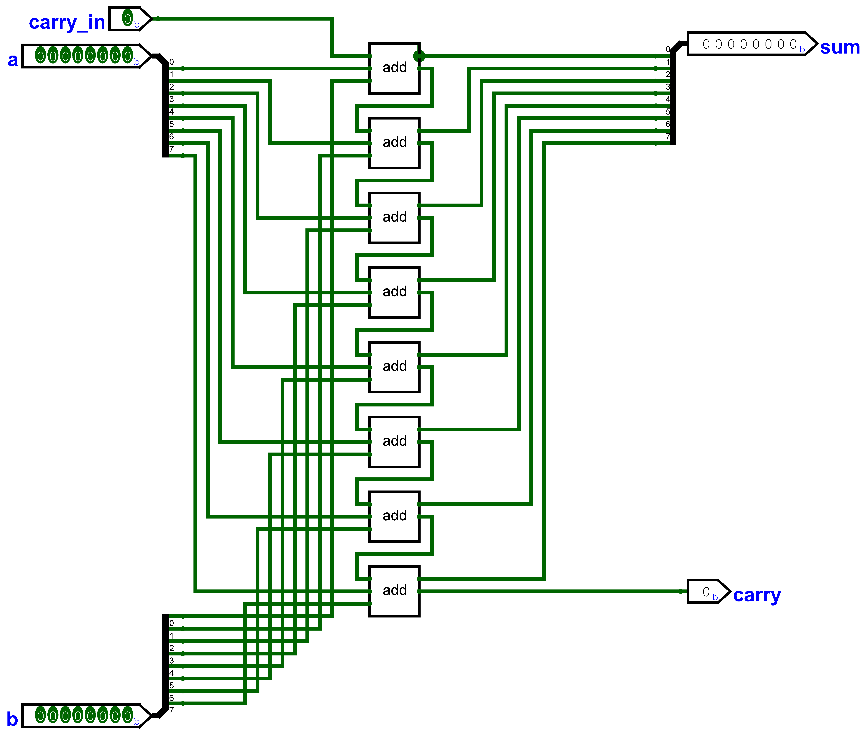
**2bit Full Adder:**

Diagram

Description automatically generated

De 2bit adder werkt door twee full adders samen te voegen. De bovenste telt alle eenheden op en de onderste krijgt de ‘een onthouden’ van de bovenste en telt die samen met alle tweetallen (niet tientallen, want het is binair) op.[[7]](#footnote-8)

**Byte Adder:**



De byte Adder laat goed zien dat de Full Adder herhaaldelijk onder elkaar kan worden geplaatst en op die manier steeds grotere getallen bij elkaar kan optellen.

*n*-bit Full Adder

Diagram

Description automatically generated

Een *n*-bit adder is te maken met twee adder die *n/2*-bit zijn. De ‘aLaag’ en ‘bLaag’ zijn dan de minst significante helft van de inputs a en b. De twee die eindigen met ‘Hoog’ zijn de meest significante helften.

## Memory

Memory in het specifiek heeft ingewikkelde Truth tables, omdat de uitkomst van een input afhangt van eerdere inputs, daarom geven Truth tables in dit gedeelte soms niet de uitkomst, maar in plaats daarvan hoe de uitkomst veranderd.

|  |  |
| --- | --- |
| a | S |
| 0 | S |

Deze Truth table geeft aan dat als a=0 dan is de S hetzelfde als eerder (~S zou betekenen het tegenovergestelde van S)

Doordat memory ook echt voor langere tijd veranderd en zichzelf beïnvloed is het belangrijk dat het ook echt alleen iets met de input doet als je klaar staat met de juiste inputs, daarom hebben veel memory onderdelen een klok (cl) die dat bestuurd.

### Set-Reset flip flop (SR flip-flop)

Diagram, schematic

Description automatically generated

De SR flip-flop heeft twee inputs. De set input zet de Q output naar een 1 en de reset input brengt het terug naar een 0. De flip-flop onthoudt de toestand als beide uitstaan. De nQ hoort altijd het tegenovergestelde te zijn van Q, maar als je beide inputs tegelijk aanzet dan staan beide Q en nQ aan. Dit is gevaarlijk, want als je dan beide inputs tegelijk uitzet dan weet je niet wat je output wordt.

|  |  |  |  |
| --- | --- | --- | --- |
| Set | Reset | Q | nQ |
| 0 | 0 | pQ | pnQ |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 1 | 1 |

### Data flip-flop (D flip-flop)

Diagram

Description automatically generated

De D flip-flop heeft normaal een clock op de plaats van save, waardoor de output veranderd naar de value waarde elke keer dat de klok hoog is. Dat gebruik ik niet in mijn computer, maar door de klok te veranderen naar een flag die bestuurd of er gesaved moet worden. Dit is handig in de computer om ervoor te zorgen dat bijvoorbeeld als je gekozen hebt op een adres iets op te slaan dat het niet ook wordt opgeslagen op alle andere adressen.

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

### Data flip-flop met klok (DCl flip-flop)

Diagram, schematic

Description automatically generated

De DCl flip-flop is een D flip-flop, maar dan met een klok erbij. Dit zorgt ervoor dat de opgeslagen waarde op vaste tijdstippen wordt veranderd.

|  |  |  |  |
| --- | --- | --- | --- |
| Cl | Save | Value | Q |
| 0 | 0 | 0 | Q |
| 0 | 0 | 1 | Q |
| 0 | 1 | 0 | Q |
| 0 | 1 | 1 | Q |
| 1 | 0 | 0 | Q |
| 1 | 0 | 1 | Q |
| 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 1 |

## Logic Units

### Binary Logic Unit (BLU)

### Arithmatic Logic Unit (ALU)

## OVERTURE

# Appendix A: Code

## OVT:

### Multiplication:

An example of good looking versus bad looking code.

|  |  |
| --- | --- |
| const input 7   ;register 7 is  const output 7  ;input & output  copy input 0    ;first operand (a)  copy 0 1  imme 1  copy 0 2  sub  imme eind  jump =0         ;skip to the end if 0 (a\*0=0)  copy input 0    ;second operand (b)  copy 0 1  copy 0 2  imme 1  copy 0 4  imme loop  label loop  copy 3 5  add             ;som+=a  copy 4 2  copy 1 4  copy 5 1  copy 3 5  sub             ;b=b-1  copy 4 1  copy 2 4  copy 5 2  jump !=0        ;loop until b=0  label eind  copy 5 output   ;output som | copy 7 0  copy 0 1  imme 1  copy 0 2  sub  imme 25  jump =0  copy 7 0  copy 0 1  copy 0 2  imme 1  copy 0 4  imme 14  copy 3 5  add  copy 4 2  copy 1 4  copy 5 1  copy 3 5  sub  copy 4 1  copy 2 4  copy 5 2  jump !=0  copy 5 7 |

# Appendix B: diagrams

#### 2bit Full Adder (only logic gates)

Diagram, schematic

Description automatically generated

The  is a constant with the value 0.

# Appendix C: Truth tables

#### 2bit Full Adder (Truth table)

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| b1 | b0 | a1 | a0 | carry | S1 | S0 |
| 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| 0 | 0 | 1 | 1 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| 0 | 1 | 1 | 1 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 1 | 0 | 1 | 1 |
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 1 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 0 | 1 | 1 | 0 | 0 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | 0 |

# Appendix D: terminologie

(Symbols and words)

Hoofdvraag: Hoe maak je een gebruiksvriendelijke computer (voor een programmeur op een laag level)?

* Deelvragen:
  + Wat voor soort architectuur is het makkelijkst te gebruiken?
    - Wat is beter load-store of register-memory
  + Wat voor soort assembly zorgt voor het overzichtelijkst programmeren bij deze architectuur?
    - Single instructie per loc of meerdere instructies per loc?
  + Welke opties zijn er qua hardware?
    - TTL logic vs CMOS logic

Planning:

Week 11: Dit document opstellen en het maken van een groot deel van het hoofdstuk ‘componenten’

Week 12: Het maken van ‘OVERTURE’ in het hoofdstuk ‘Architectuur’

Week 13: Het programmeren van assembly in Python

Week 14: vervolg van het maken van assembly

Het maken van een groot deel van de componenten maak ik voordat ik een computerdesign heb, omdat die universeel zijn voor elke computer (bijv. elke computer heeft een onderdeel dat twee inputs neemt, ze bij elkaar optelt en dan dat als output geeft.). Onderdelen zoals RAM maak ik pas zodra ik een design heb, omdat die heel specifieke eigenschappen kan hebben (zoals: Het hebben van 8 registers, het worden aangestuurd door 3 bits aan instructie of het intern kunnen kopiëren tussen registers).

Ik heb eerst met de specificaties van Alan Turing’s OVERTURE een computer gemaakt die dat kan zodat ik weet hoe het maken van een computer ongeveer werkt en wat wel of niet kan. Daarna maak ik zelf specificaties met deze kennis en maak ik opnieuw een design.

Het maken van assembly kan zodra de specificaties voor de computer zijn gemaakt (dus het kan technisch al voor het designen), want dan weet ik wat de computer precies moet doen bij welke instructies. Het maken van assembly doe ik pas zodra het design helemaal af heb, want dan kan ik de assembly ook echt testen in een simulatieprogramma.

Logboek:

15/03: 4 uur aan dit document gewerkt

16/03: 2 uur aan componenten

17/03: 1 uur, maken van OVERTURE

18-20/03: ? uur , afmaken van OVERTURE

27/03: 2,5 uur, assembly regex (is een bitch)

28/03: 4,5 uur, assembly custom syntax highlighting, OVT documentation and V2 documentation

04/04: 2 uur, willekeurige problemen verzinnen en oplossen met logic gates om beter te worden met logic gates

05/04: 3 uur, hetzelfde als gisteren ^

09/04: 1,5 uur, maken van interpreter voor OVERTURE

10/04: 2,5 uur, afmaken interpreter

14/04: 2 uur, interpreter sneller maken (was onnodig)

16/04: 2 uur, herschrijven van interpreter

17/04: 1 uur, laatste aanpassingen aan de interpreter waardoor die nu een stuk korter is

18/06: 2 uur, kleine verbeteringen aan de compiler

19/06: 2 uur, maken van visualisering in obsidian zodat ik weet hoe alles samen gaat werken

26/06: 3 uur, decoder en comparison

02/07: 4 uur, afmaken van ALU

03/07: 2 uur, begin ‘mem block’

04/07: 2 uur, afmaken ‘mem block’

05/07: 3 uur, samenvoegen laatste design computer

06/07: 2 uur, maken van beter werkende compiler

07/07: 4 uur, werken aan meer registers en laatste testen om te kijken of alles werkt

09/07: 3 uur, floats geimplementeerd, samen met subroutines

10/07: 2 uur, programeren van een cirkel plotter

11/07: 4 uur, gewerkt aan het ‘includen’ van andere .ovt files. Gemaakt dat je floats kan rounden. Syntax highlighting aangepast zodat het werkt met de nieuwe aanpassingen.

(11-12)/07 (ergens ‘s nachts): 2,5 uur, maken van stack

12/07: 1,5 uur, ervoor gezorgd dat bij assembly alles de juiste hoeveelheid argumenten vraagt (bijv. Ret dat om 0 argumenten hoort te vragen, maar vroeg om 1)

12-25/10: 10 uren omzetten simulatie naar ander programma en maken van programma’s (pong en beter priem generator)

5/11: 3 uur aan ruwe versie PWS

6/11: 3 uur aan ruwe versie PWS

Totaal: 81 uur

ER MISSEN VEEL UREN TUSSENDOOR (maken van assembler voor RASM, maken van compiler voor high, vele uren aan debugging van het uiteindelijke design van de computer, het maken van test programma’s, het maken van schetsen op school)

TODO:

* + - * Explain binary arithmatic
      * Explaining how Karnaugh map works
      * Documentation V2
      * Translation of the parts that are in Dutch.
      * Uitleggen floats

Syntax:

Vscode: C:\Users\pvano\ovt\syntaxes\ovt.tmLanguage.json

Powershell: PS C:\Users\pvano> copy ./ovt .\.vscode\extensions -recurse -force

Ctrl shift p: 

part -> chapter -> section

**theoretisch deel:**

For my PWS there isn’t a lot of information that is usable that I can find, because my computer designs are made for a specific purpose (low level user experience), so computers aren’t made for my use case. This means I can only use the very simple computer made in ‘Nandgame’.

The prerequisite knowledge that I used for this project are:

* + - * Python: how to make simple programs and how to use some of the more used modules. ([www.w3schools.com](http://www.w3schools.com), I didn’t use this website to learn python, but it is a good place to learn it)

Logic gates: how logic gates work. ([www.techtarget.com/whatis/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR](http://www.techtarget.com/whatis/definition/logic-gate-AND-OR-XOR-NOT-NAND-NOR-and-XNOR))

The things I needed to learn are:

* + - * How python regex (the ‘re’ module), this is for matching and replacing for assembly and just the matching for the syntax highlighting ([www.w3schools.com/python/python\_regex.asp](http://www.w3schools.com/python/python_regex.asp))
      * The components that I didn’t make myself and a simple example computer architecture I got from [www.nandgame.com](http://www.nandgame.com) (Only the hardware part)
      * The circuits for float operations I made myself as well, but how floats work I got from <https://www.geeksforgeeks.org/ieee-standard-754-floating-point-numbers/>
      * The research for my decision for what instructions to implement I got from: [The x86 instruction reference](https://www.felixcloutier.com/x86/) (Around 1000 different instructions) and then deciding if they are simple enough and useful.

The things that I will need to search for later are:

* + - * How components can be improved (i.e. [Ripple-carry\_adder](https://en.wikipedia.org/wiki/Adder_(electronics)#Ripple-carry_adder) vs [Carry-lookahead\_adder](https://en.wikipedia.org/wiki/Carry-lookahead_adder))

Nandgame screenshots (I made all components myself, but what components to make was provided by nandgame):

Nand: Graphical user interface, application

Description automatically generated not: A picture containing diagram

Description automatically generated and: Diagram

Description automatically generated with low confidence or: Graphical user interface

Description automatically generated with low confidence

Xor: Diagram

Description automatically generated with medium confidence half-adder: Diagram

Description automatically generated with low confidence full-adder: Diagram, schematic

Description automatically generated with medium confidence

2bit-adder: Diagram

Description automatically generated increment: Diagram

Description automatically generated subtract: Diagram

Description automatically generated with low confidence

=0: A picture containing icon

Description automatically generated <0: Diagram

Description automatically generated with low confidence selector: Diagram

Description automatically generatedMap

Description automatically generated with medium confidence

Switch: A screenshot of a video game

Description automatically generated with medium confidence latch: A screenshot of a cell phone

Description automatically generated with medium confidence data flip-flop: A picture containing map

Description automatically generated

2bit register: Diagram

Description automatically generated counter: Diagram

Description automatically generated 2x16 RAM: A picture containing timeline

Description automatically generated

Unary logic unit: Diagram

Description automatically generated ALU: Diagram

Description automatically generated

Condition: Diagram

Description automatically generated combined memory: Diagram

Description automatically generated

Instruction decoder: Diagram

Description automatically generated control unit: Diagram

Description automatically generated

Computer: Diagram

Description automatically generated

1. [Higher level](https://www.geeksforgeeks.org/difference-between-high-level-and-low-level-languages/) Betekent dat het verder van pure computertaal afstaat en niet direct een op een te vertalen is. Het is makkelijker voor een mens om te begrijpen, maar soms wel daardoor minder efficiënt in uitvoering. [↑](#footnote-ref-2)
2. Een 'word' de eenheid aan data wat door de meeste instructies wordt gebruikt. Alle data neemt een veelvoud van de 'word size' in aan ruimte. [↑](#footnote-ref-3)
3. Veel computers gebruiken in plaats hiervan [load-store](https://en.wikipedia.org/wiki/Load%E2%80%93store_architecture) architecture waarbij je eerst van memory (grote opslag) naar register (kleine opslag) moet kopiëren voordat je operaties kan uitvoeren. Ik gebruik [register-memory](https://en.wikipedia.org/wiki/Register%E2%80%93memory_architecture#:~:text=In%20computer%20engineering%2C%20a%20register,%22register%20plus%20memory%22%20architecture.). [↑](#footnote-ref-4)
4. Een [cycle](https://www.techopedia.com/definition/5498/clock-cycle#:~:text=In%20computers%2C%20the%20clock%20cycle,processor%20activity%20is%20carried%20out.) (ookwel ‘clock cycle’ of ‘clock tick’) is de tijd waarin de klok een cyclus van hoog en laag voltage heeft afgemaakt [↑](#footnote-ref-5)
5. x86 is de meest gebruikte instructie set [↑](#footnote-ref-6)
6. Call workings: call [a], jump to line a, at line a is a jump statement that brings jumps to function a. The instruction you call at is saved so you can later use ‘ret’ to return. This is saved by pushing it to the stack. [↑](#footnote-ref-7)
7. *2bit Full Adder (alle logic gates)*; *2bit Full Adder (Truth table)* [↑](#footnote-ref-8)