GRADUAAT IN HET PROGRAMMEREN

SEMESTER 2 & 3 ACADEMIEJAAR 2019-2020

COMPUTER-ARCHITECTUUR

Basic IT Skills

laatst bijgewerkt op 17 november 2019

howest.be

INHOUDSOPGAVE

1 INLEIDING 6

1.1 Wat is een computer? 6

1.2 Evolutie 7

1.3 Generaties 8

1.4 Soorten systemen 8

1.4.1 Supercomputers 9
1.4.2 Mainframe computers 9
1.4.3 Minicomputers 9
1.4.4 Microcomputers 9

2 COMPONENTEN 10

2.1 Inleiding 10

2.2 Computerbussen 11

2.2.1 Data bus 11
2.2.2 Adresbus 11
2.2.3 Controlebus 12
2.2.4 Breedte van een bus 12
2.2.5 Chipset 13

2.3 Central Processing Unit 14

2.3.1 Naamgevingen 14
2.3.2 Kloksnelheid 14
2.3.3 Evolutie van de werkelijke snelheid t.o.v. kloksnelheid 14

2.4 Geheugen 15

2.4.1 RAS en CAS 15
2.4.2 Geheugentypes 16
2.4.3 L1, L2 en L3 Caches 17

2.5 I/O 18

2.5.1 Controller 18
2.5.2 I/O-adressen 19
2.5.3 Vaste I/O-adressen 19
2.5.4 Interrupt Request 19
2.5.5 Prioriteit van de IRQ nummers 20
2.5.6 Zestien IRQ’s 20
2.5.7 Vaste IRQ's 20

2.6 Direct Memory Access 21

2.6.1 Geheugen 21
2.6.2 DMA-kanaal 21

3 WERKING VAN EEN COMPUTER 22

3.1 Componenten van de CPU 22

3.1.1 ALU 22
3.1.2 Control Unit 22
3.1.3 CPU Registers 22
3.1.4 Kloksnelheid 24

3.2 Instructieset 25

3.2.1 Machinetaal 25
3.2.2 Instructies 26
3.2.3 32-bit versus 64-bit 27

3.3 Uitvoering van instructies 28

3.3.1 Fetch 28
3.3.2 Decode 28
3.3.3 Execute 29
3.3.4 Concreet voorbeeld 29

4 GEHEUGENBEHEER 32

4.1 Inleiding 32

4.2 Geheugen hardware 32

4.2.1 Geheugenchips 32
4.2.2 CPU caching 32
4.2.3 Restricties voor gebruikersprocessen 32
4.2.4 Address binding 33
4.2.5 Logische versus fysieke adresruimte 34
4.2.6 Dynamic loading 35
4.2.7 Dynamic linking en gedeelde bibliotheken 35

4.3 Toekenning van geheugen 36

4.3.1 Memory allocation 37
4.3.2 Fragmentatie 37

4.4 Segmentatie 38

4.4.1 Programmasegmenten 38
4.4.2 Segment tabel 39

4.5 Paging 40

4.5.1 Principe 40
4.5.2 Voorbeeld 40
4.5.3 Eigenschappen 41
4.5.4 Protection 42
4.5.5 Gedeelde pages 43
4.5.6 Samenwerking paging – segmentering 44

4.6 Virtueel geheugen 44

4.6.1 Achtergrond 44
4.6.2 Demand paging 46
4.6.3 Thrashing 48

4.6.4 Windows 49

1 INLEIDING

1.1 WAT IS EEN COMPUTER?

Iedereen heeft tegenwoordig zowel een intuïtieve als visuele voorstelling van een computer. Deze voorstelling is echter vaak slechts ten dele correct en is daarenboven in de loop van de laatste vijftig jaren sterk geëvolueerd.

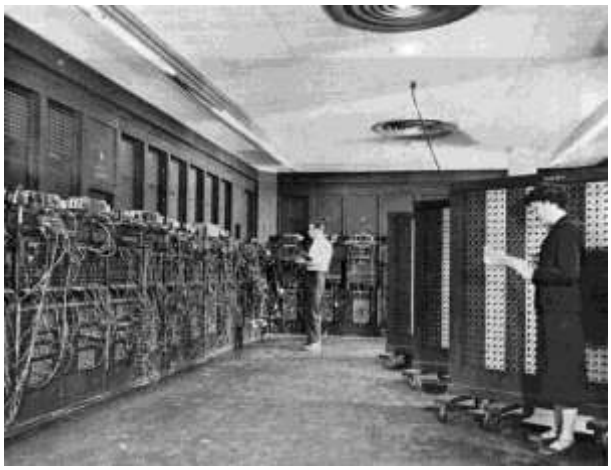
Daarom volgt hier een definitie van een computer die redelijk permanent is:

> Een computer is een machine met elektronische componenten waarmee gegevens van allerlei aard kunnen worden verwerkt tot informatie op een geautomatiseerde wijze door een programma.

De essentie van deze definitie houdt in dat **gegevens** worden omgezet tot **informatie**. Hierbij is het nodig dat deze twee begrippen zelf nauwkeurig worden omschreven:

- **gegevens (data):** De tekens of signalen, die op zichzelf niet noodzakelijk een betekenis hebben.
- **informatie:** De verwerkte, omgevormde gegevens, die in een bepaalde context geplaatst voor de gebruiker een betekenis krijgen die een waardevermeerdering inhoudt.

1.2 EVOLUTIE



De **Z3** van de Duitser **Zuse** was dan wel de eerste computer, maar alle latere computers zijn opvolgers van de Amerikaanse **ENIAC** (Electronic Numerical Integrator And Calculator). Deze machine nam alle wanden van een zaal van 6 bij 12 meter in beslag, woog 30 ton, bevatte 18 000 elektronenbuizen, had een geheugen waarin 20 decimale getallen worden opgeslagen en kon 5000 optellingen per seconde uitvoeren. Hij verbruikte een enorme hoeveelheid elektriciteit. Tegenover de huidige computers waren er naast deze fysische verschillen nog twee fundamentele conceptuele verschillen:

Het programma werd aangebracht via externe bedradingen (**hardwired**). Dit is een zeer onhandige wijze van programmeren.

Juist zoals de **Analytical Engine** van **Babbage** en de **Harvard Mark I** van **Aiken** werkte de **ENIAC** intern met decimale getallen. De **Z3** van **Zuse** en de hedendaagse computers werken intern met binaire getallen (enkel opgebouwd met de cijfers 0 en 1).

De Amerikaan **John von Neumann** legde de basis voor alle latere computers door twee eisen te stellen waaraan alle latere computers zouden voldoen:

- De gegevens worden intern **binair** voorgesteld.
- Het programmeren gebeurt niet via externe bedradingen maar **het programma wordt** voor de duur van de uitvoering intern **in het geheugen van den computer opgeslagen**, vaak ook zoals de te verwerken gegevens.

Computers die aan deze eisen van von Neumann voldoen worden **Stored-Program** computers of **von Neumann** computers genoemd.

Von Neumann ontwierp de **EDVAC** (afgewerkt in 1951) volgens deze principes, maar de Engelsman **Maurice Wilkes** was de eerste die een **Stored Program** computer bouwde: de **EDSAC** (in 1949).

Naast de hier behandelde **digitale computers**, die hun gegevens via digitale grootheden voorstellen, zijn er ook nog de **analoge computers**, die hun gegevens via op continue wijze variërende grootheden voorstellen. Deze worden echter bijna uitsluitend in industriële toepassingen gebruikt. Daarom worden ze in deze tekst niet behandeld.

1.3 GENERATIES

De eerste computers ontstonden in de jaren 1940. Het waren zeer omvangrijke mainframe computers, al was hun kracht zeer beperkt, zelfs vergeleken met de allerkleinste huidige computers. In het begin van de jaren 1960 verschenen de eerste minicomputers (3de generatie) en in de jaren 1970 de eerste microcomputers (4de generatie).

De evolutie van de computers wordt vaak aangegeven door de computergeneraties. De algemene tendens is dat computers bij elke generatie:

- **kleinere afmetingen** hebben.
- **sneller** en **krachtiger** worden.
- **energiezuiniger** worden.
- **goedkoper** worden.
- **betrouwbaarder** worden.

Het kenmerkende gegeven van een computergeneratie is meestal de **CPU-technologie** en het **centrale geheugen** zijn opgebouwd. In de onderstaande tabel worden enkele kenmerken van de computergeneraties weergegeven (de gegevens gelden voor typische grote computers):

Generatie	tijd	grootte	technologie	snelheid (instr./s)	centraal geheugen
1ste	1940-1956	kamer	elektronenbuizen	10^2	10 K
2de	1956-1963	kast	transistors:	10^3	100 K
3de	1964-1971	bureau	IC's (SSI, MSI)	10^6	1 M
4de	1979-heden	A4 blad, telefoon	IC's (LSI, VLSI)	$>10^7$	>10 M

Niet enkel de hardware van de computers evolueerden. Er was een gelijklopende reeks generaties aan programmeertalen (GL = Generation Language).

1GL: Eerste generatie: Machinetaal
2GL: Tweede generatie: Assembly taal
3GL: Derde generatie: Gestructureerde programmeertalen (C, COBOL, FORTRAN,…)
Vierde generatie: Programmeertalen ontworpen met een zeer specifiek doel in het achterhoofd (SQL, Coldfusion)

1.4 SOORTEN SYSTEMEN

De verschillende soorten computers worden vaak opgedeeld in vier categorieën:

- Supercomputers
- Mainframe computers
- Minicomputers
- Microcomputers

1.4.1 SUPERCOMPUTERS

Een supercomputer houdt zich bezig met heel **zware berekeningen** zoals weersvoorspellingen, astrofysica, nucleaire simulaties en andere wetenschappelijke onderzoeken. Men vindt er aan sommige (grote) universiteiten, bij militaire en bij meteorologische instellingen. Supercomputers zijn enorm duur en staan op de eerste rij wat betreft rekenkracht en capaciteit. De rekenkracht wordt uitgedrukt in floating point operaties per seconde (FLOPS). De belangrijkste firma die mainframe computers bouwt is Cray.

1.4.2 MAINFRAME COMPUTERS

Mainframes bevatten meestal meerdere CPU's. Ze worden typisch door honderden gebruikers tegelijk gebruikt. Ze kunnen dienen als host-computer van een netwerk, als database-server, als transactieverwerker en bij bedrijfsoverkoepelende systemen. Mainframes worden vaak onderverdeeld (van groot naar klein) in **grote**, **middelgrote** en **kleine mainframes**. De belangrijkste firma die mainframe computers bouwt is **IBM**.

1.4.3 MINICOMPUTERS

De minicomputer (andere benamingen zijn midrange computer, mini) situeert zich ergens in het midden van het computerspectrum: tussen de kleinste mainframe computers en de grootste microcomputers. Ze kunnen dienen als netwerkserver, krachtig werkstation. Ze hebben minder randapparaten en gelijktijdige gebruikers dan mainframes, en zijn vaak toegespitst op rekenwerk. Ze zijn ook heel wat goedkoper dan mainframes. Minicomputers worden vaak onderverdeeld in (van groot naar klein) **middelgrote systemen**, **supermini's** en **mini's**. De laatste jaren geraken de minicomputers wat in de verdrukking tussen de mainframes en de in netwerken verbonden microcomputers.

1.4.4 MICROCOMPUTERS

Tegenwoordig de meest bekende soort computer, gebruikt op het werk, op school, thuis of onderweg. De term “microcomputer” werd geïntroduceerd met de komst van de microprocessoren, en klopt eigen niet helemaal meer.

Traditionele microcomputers:

- Desktop computer – een case en monitor, om op of onder de bureautafel te plaatsen.
- Autocomputers – ingebouwd (embedded) in een auto, voor multimedia, navigatie, enz…
- Spelconsoles – gespecialiseerd voor multimedia en videospellen.

Een apart geval is de mobiele apparatuur:

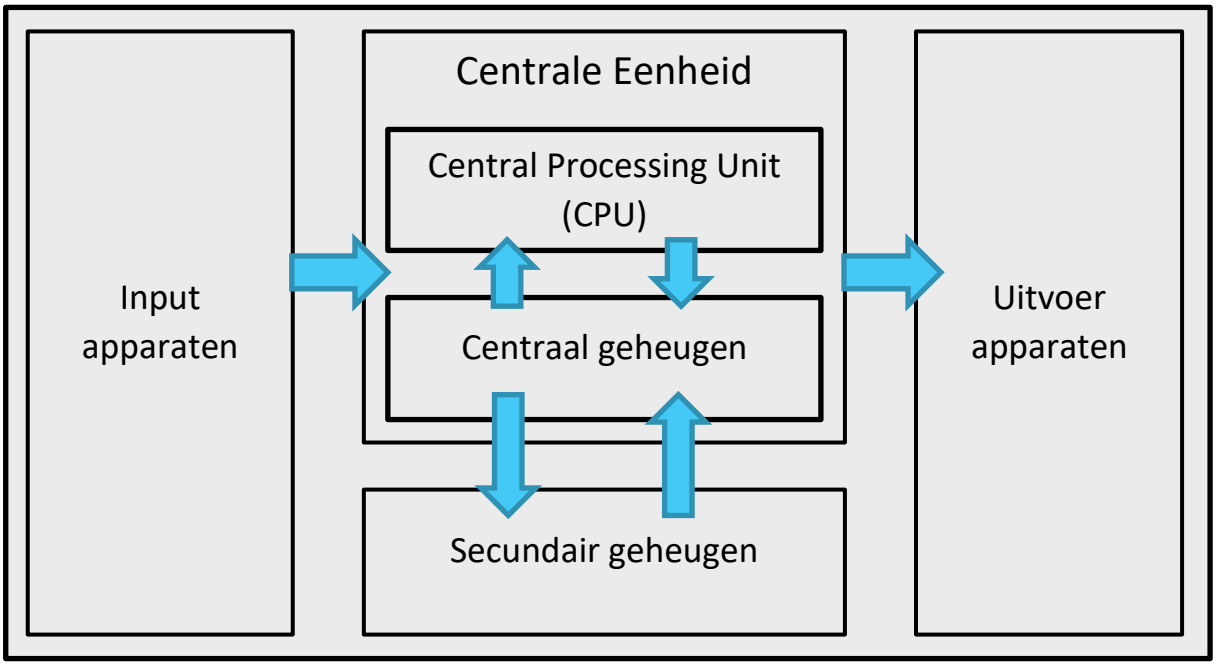
- Laptops, notebooks en palmtops – draagbare, volwaardige computers zonder beperkingen.
- Tablet computers – zoals een laptop met touch-screen, vaak met enige softwarebeperkingen.
- Smartphones, smartbooks, PDA’s – kleine computertjes met hardwarebeperkingen.

Men moet wel oppassen met de opdeling in “grote” mainframe, “middelgrote” mini- en “kleine” microcomputers. Vaak overlappen deze categorieën mekaar op het gebied van kracht en mogelijkheden. Daarenboven zullen microcomputers van nu op veel gebieden krachtiger zijn dan minicomputers en zelfs mainframes van 10 jaar geleden.

2 COMPONENTEN

2.1 INLEIDING

Een computer bestaat uit vijf functionele componenten met als functies: **besturing**, **verwerking**, **invoer**, **uitvoer** en **opslag**. De componenten voor besturing, verwerking en sommige componenten voor opslag bevinden zich in het inwendige van de computer en worden vaak de **centrale eenheid** of de eigenlijke computer genoemd. Externe componenten die dienen voor invoer, uitvoer noemen we vaak **randapparatuur**. Componenten die zorgen voor (permanente) opslag van gegevens worden ook wel het **secundair geheugen** genoemd.



- CPU: Het hart van elke moderne computer is de **CPU** of de **processor**, een rekenapparaat dat bestuurt wordt d.m.v. numerieke machinecodes die instructies vertegenwoordigen (vb: tel op, vergelijk, …).
- Centraal geheugen: Hetgeen we in de CPU pompen zijn reeksen machine codes afkomstig uit het centraal geheugen (ook: **main memory, RAM geheugen, werkgeheugen**). Deze codes bevinden zich in het werkgeheugen op een genummerde plaats (een **adres**).
- I/O apparaten: krijgen toegang tot het verwerkingsproces door middel van **computerbussen**. Het betreft zaken zoals hedit beeldscherm, het toetsenbord, maar ook minder evidente componenten zoals de netwerkkaart.
- Secundair geheugen: kan **niet rechtstreeks** door de CPU benaderd kan worden. De computer gebruikt I/O voor toegang tot de gegevens in dit geheugen. Bekende voorbeelden zijn de harde schijf, of flash geheugens zoals USB sticks.
- Computerbussen: Dit zijn de blauwe pijlen in de figuur. Een **bus** is een elektronische verbinding tussen de componenten waarmee **gegevens worden overgedragen**. Je kan dit vergelijken met een binaire snelweg.

2.2 COMPUTERBUSSEN

De **CPU** is door middel van computerbussen **verbonden** met het **werkgeheugen** en met de **randapparatuur**.

Je kan een bus makkelijk vergelijken met een snelweg waarop gegevens doorheen het systeem kunnen reizen. Het woord “bus” is afgeleid van het Latijnse *omnibus*, wat “voor alles” betekent.

Een **bus** is een fysieke verbinding:

- tussen de processor en het interne geheugen óf
- tussen de processor en de randapparatuur

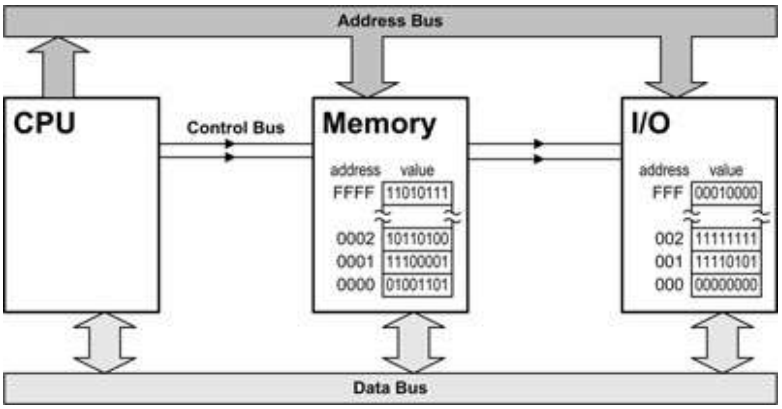
Dit betekent dat elke component rechtstreeks of onrechtstreeks verbonden is met de CPU en/of het intern geheugen door middel van een bus.

De belangrijkste bus is de **systeembus**. Andere namen voor de systeembus zijn ook wel de local bus, de geheugenbus of **front side bus** (FSB).

De **systeembus** verbindt **de processor** met het **interne geheugen**. De andere bussen die de communicatie tussen de randapparaten verzorgen, worden de I/O-bussen genoemd en vertakken vanuit de systeembus.

Binnen de FSB (en ook binnen de andere I/O-bussen) onderscheiden we drie verschillende bussen:

- **de databus**
- **de adresbus**
- **de controlebus**



2.2.1 DATA BUS

De **databus** of **gegevensbus** wordt gebruikt om gegevens te transporteren van en naar de **CPU** (en tussen de verschillende geheugens onderling). Wanneer er verkeer over de databus verloopt naar de **CPU** toe wordt dit een **lees** (**read**) bewerking genoemd; wanneer er verkeert over de databus verloopt van de **CPU** weg wordt dit een **schrijf** (**write**) bewerking genoemd. Over de databus kan in principe verkeer in beide richtingen verlopen, maar niet gelijktijdig. Er kan niet geschreven worden naar onderdelen die dit niet toelaten, zoals het ROM (Read-only memory) geheugen.

2.2.2 ADRESBUS

De **adresbus** wordt gebruikt voor het transport van adressen. Het **adres** van een geheugen (of poort) geeft aan waar een lees- of schrijfbewerking moet gebeuren. Dit adres wordt bepaald in de **CPU** en via de adresbus naar het geheugen (of de poort) getransporteerd bij de lees- of schrijfbewerking. Het verkeer over de adresbus verloopt steeds in één richting: weg van de **CPU**.

2.2.3 CONTROLEBUS

De **controlebus** of **besturingsbus** wordt gebruikt om allerlei signalen naar de aangesloten componenten te versturen. Deze signalen hebben allemaal een andere betekenis. Een component kan bijvoorbeeld de klok zijn of het geheugen. Via de controlebus wordt aangegeven of er data wordt geschreven of wordt gelezen.

2.2.4 BREEDTE VAN EEN BUS

De capaciteit van een bus, beter gekend als de **busbreedte**, bepaalt de hoeveelheid gegevens die in 1x getransporteerd kunnen worden.

Een bus bestaat altijd uit acht (of een veelvoud van acht) elektronische verbindingen naast elkaar. Bij deze elektrische verbindingen kan het gaan om draden, maar ook om koperverbindingen (sporen, zie afbeelding) die je op een printplaat ziet.

Bijvoorbeeld: een 16-bit bus kan maximum 16 bits per keer versturen, maar een 64-bit bus kan 4x zoveel gegevens per keer versturen. Denk nogmaals aan de vergelijking met de snelweg: hoe meer rijvakken, hoe meer auto's er gelijktijdig gebruik van kunnen maken.

De busbreedte mag niet verward worden met de **bandbreedte.** De bandbreedte wordt uitgedrukt in Mb/s (Megabit per seconde) en wordt berekend door de **busbreedte te vermenigvuldigen met de kloksnelheid** van die bus.

Bandbreedte (Mbit/s) = busbreedte (bit) x snelheid (Mhz)

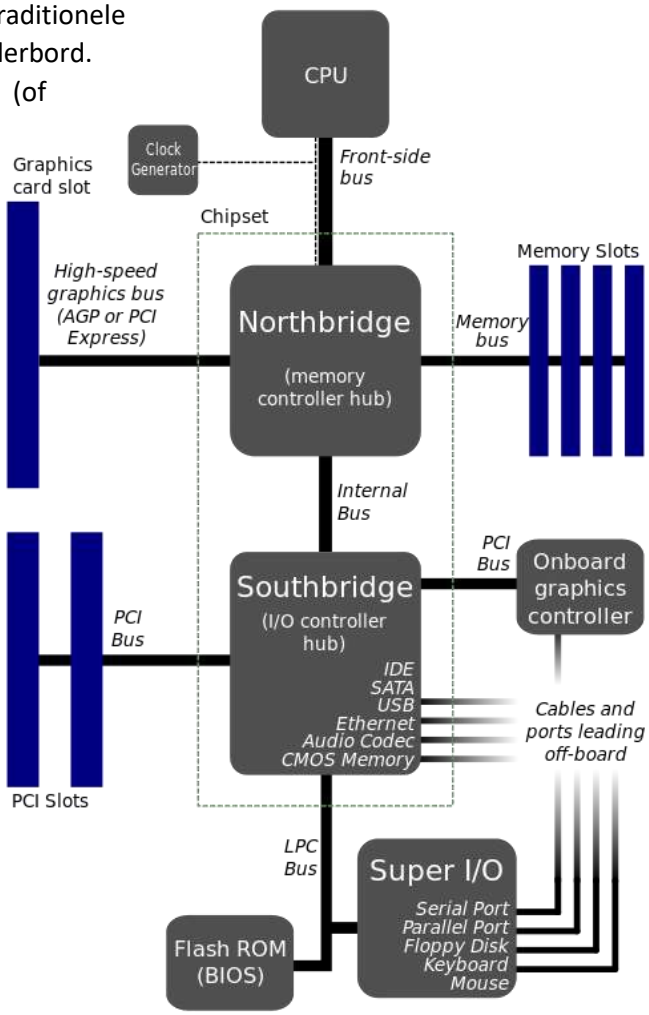
2.2.5 CHIPSET

De communicatie tussen de onderdelen van een traditionele PC wordt beheerd door de **chipset** op het moederbord. Deze bestaat uit twee chips: de **North Bridge** (of **memory controller**) en de **South Bridge** (oftewel de **I/O controller**). De chipset vormt het hart van een moederbord en bevat soms extra functionaliteiten zoals onboard graphics, geluid of netwerkapparaten.

Volgend schema toont de positionering van de chipset op een moederbord van een moderne microcomputer. Je merkt dat de FSB verre van de enige bus is in een modern computersysteem.

We onderscheiden o.m.:

- Geheugenbus
- Grafische bus
- PCI bus (Peripheral Component Interconnect)
- LPC bus (Low Pin Count)

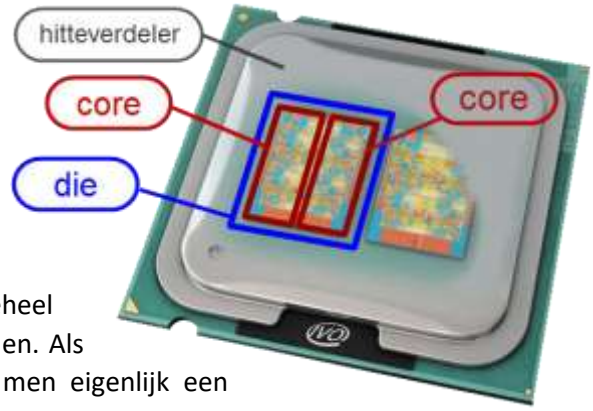


2.3 CENTRAL PROCESSING UNIT

De besturing en verwerking gebeurt in de **Central Processing Unit** (CPU) of centrale verwerkingseenheid (CVE). Processoren bevinden zich meestal op één **Integrated Circuit** (IC) of geïntegreerde schakeling.

2.3.1 NAAMGEVINGEN

In de figuur hiernaast zie je verschillende fysieke onderdelen van een typische "quad-core processor":



- Een **core** is een onafhankelijke verwerkings-eenheid dat één instructie per keer kan uitvoeren. Dit gebeurt parallel met andere cores die elk hun eigen programma verwerken.
- Een **die** is een vlak stukje (silicone) halfgeleider-materiaal waar één of meerdere cores op bevestigd worden.
- De **processorassemblage** is het vierkante geheel dat op het moederbord bevestigd kan worden. Als men een "processor" koopt, dan bedoelt men eigenlijk een assemblage.

Wanneer we in deze cursus spreken over een "processor", dan hebben we het steeds over een enkele processorkern of een **core**.

2.3.2 KLOKSNELHEID

Het doel van de processor is het verwerken van instructies. Om deze verwerking gelijkmatig te laten verlopen worden ze bepaald door een steeds herhalend signaal: de **klokimpuls**. Dit signaal bepaalt de overdracht van gegevens intern in de CPU.

Vroeger kon men makkelijk afleiden hoe performant de processor was door te kijken naar de snelheid waarmee dit signaal zich herhaalt, de **kloksnelheid** of **klokfrequentie**. Door voortdurende ontwikkeling in technologieën is dit echter niet langer het geval. De kloksnelheid wordt uitgedrukt in de eenheid van frequentie (Herz of Hz) en zit tegenwoordig in de **GigaHertz** grootorde.

De maximale kloksnelheid wordt bepaald door het elektrische voltage van het systeem en het ontwerp van de transistors. Wanneer een kloksignaal te snel gaat, dan kunnen de interne bussen niet meer mee en kunnen de signalen niet duidelijk gelezen worden.

2.3.3 EVOLUTIE VAN DE WERKELIJKE SNELHEID T.O.V. KLOKSNELHEID

Toen AMD processoren op de markt bracht die meer werk verrichtten per klokimpuls, kon men niet langer vaststellen hoe goed deze het deden t.o.v. andere processoren. Destijds verkocht men een Athlon van 1,8GHz onder de naam "Athlon 2400+", zijn equivalente snelheid met een 2,4GHz Intel. Inmiddels kreeg Intel hetzelfde probleem; de 2,2GHz "Pentium 4M" was trager dan de 1,6GHz "Pentium M".

Daarom wordt de kloksnelheid niet meer gebruik om de "kwaliteit" van de CPU te mee te delen en worden er andere naamgevingen gebruikt.

De beste manier om de performantie van een processor te weten te komen is tegenwoordig de vergelijkende tests te bekijken, te vinden op hardware sites zoals www.tomshardware.com.

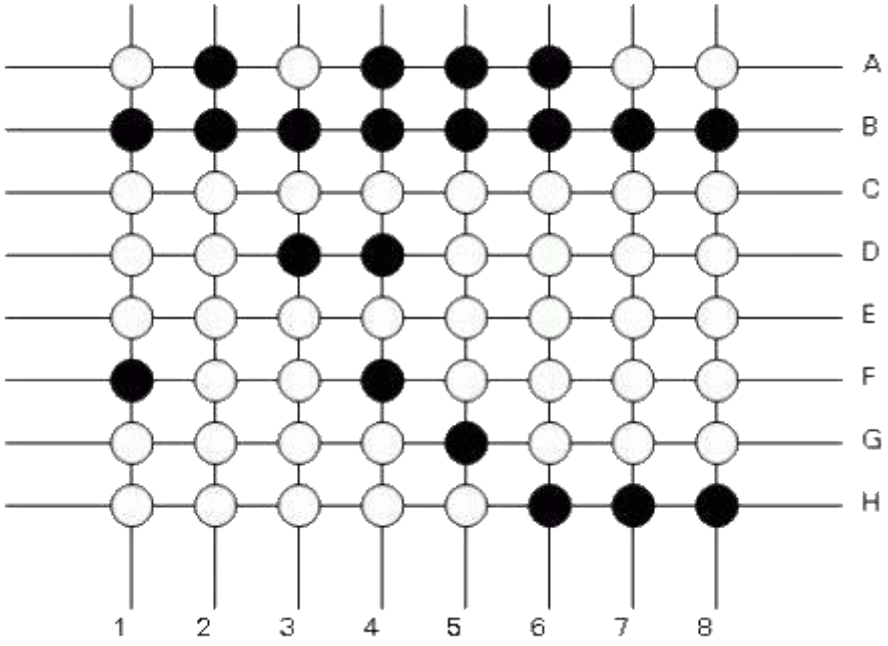
2.4 GEHEUGEN

Het interne geheugen (**RAM**, Random Access Memory) bestaat uit chips die op een insteekkaart gemonteerd zijn. Dit noemen we **geheugenmodules**.

De geheugenmodules passen in speciale sleuven op het moederbord: de **geheugenbanken**.

2.4.1 RAS EN CAS

In de modules worden de **gegevens** opgeslagen op plaatsen die aangeduid worden met een **adres**. De opbouw van een geheugenmodule kun je vergelijken met een raster.



In dit raster worden de gegevens in de vorm van nullen en enen opgeslagen in kolommen en rijen. De plaats waar een rij en een kolom elkaar kruisen is een *uniek geheugenadres*.

In de praktijk werkt dat als volgt:

1. De geheugencontroller zendt een signaal met het rijnummer naar de geheugenbank: **RAS** of Row Access Strobe.
2. Na enkele kloktikken (*RAS latency*) kan de controller het kolomnummer opgeven: **CAS** of Column Access Strobe.

Het aantal kloktikken dat zit tussen het uitzenden van het eerste signaal en het ontvangen van de data is de *CAS latency*; meestal zijn dat twee of drie kloktikken. Hoe lager de CAS latency des te sneller het geheugen. De CAS en RAS latency is in te stellen in het BIOS van de computer.

2.4.2 GEHEUGENTYPES

Door de steeds snellere processors is de snelheid van het geheugen steeds belangrijker. Een systeem met een snelle processor en geheugen van het langzaamste soort, zal in snelheid beperkt worden door het trage geheugen.

Er bestaan twee soorten geheugen:

1. **SRAM** maakt gebruik van **transistors** in de chip en kan daardoor de gegevens vast blijven houden zolang dat nodig is. SRAM-geheugen is erg duur en wordt dan ook weinig gebruikt. Het is de **snelste soort** en daarom worden deze geheugenchips als extern cachegeheugen op het moederbord geplaatst. In de cache worden de gegevens geplaatst die een programma het vaakst nodig heeft. Ook de interne L2-cache in een processor is SRAM.

2. **DRAM** heeft **condensatoren** in de chip die zichzelf steeds ontladen. Vergelijk het met een emmer met een gat in de bodem. Om de zoveel milliseconden moet de geheugencontroller de gegevens ophalen en herschrijven (een “refresh”) en dat kost tijd.

Enkele voorbeelden zijn:

- *SRAM* heeft een snelheid van ongeveer 2 ns.
- *DRAM* heeft een snelheid van ongeveer 60 ns.

De meest verspreide versie is de zogenaamde **Synchronous Dynamic RAM** of **SDRAM**. Er wordt gesproken van “synchrone” DRAM, omdat de geheugensnelheid synchroon loopt met deze van de **Front Side Bus** (de systeembus die de processor met de geheugencontroller/North Bridge verbindt).

DUAL-CHANNEL

De nieuwste processoren zijn zo snel dat de geheugensnelheid achterop hinkt ten opzichte van de interne kloksnelheid van de processor. De processor kan zijn data niet voldoende snel kwijt. Met bestaande technieken is het moeilijk – en duur – om nog sneller geheugen te maken; vandaar dat er voor een nieuw concept gekozen is: **dual channel technologie**.

Dual channel komt erop neer dat er twee onafhankelijke geheugenstromen zijn van de processor naar het geheugen en van het geheugen naar de processor. Er zijn dus **twee onafhankelijke geheugencontrollers**.

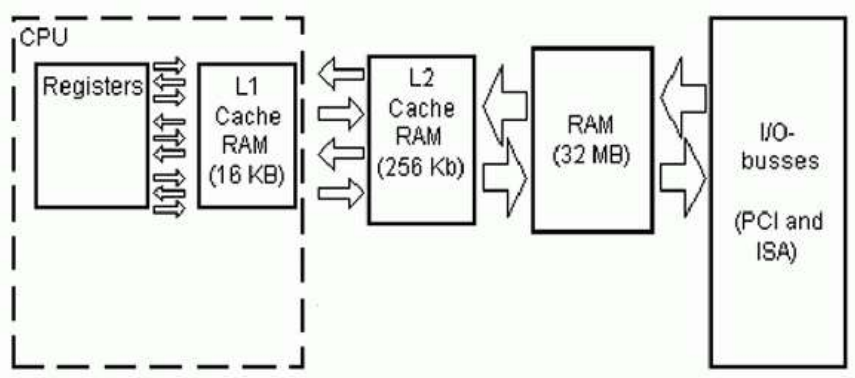
Hierdoor zijn veel grotere overdrachtssnelheden mogelijk. Een geheugenmodule heeft een breedte van 64 bit. Omdat tegelijkertijd geschreven kan worden naar twee geheugenbanken, kent een dual channel systeem dus een databusbreedte van 2 x 64 bits. Bij gelijkblijvende kloksnelheid betekent dat een verdubbeling van de doorgevoerde data.

DOUBLE DATA RATE (DDR)

Dit wordt vaak verward met het dual channel principe. Double Data Rate of DDR betekent dat het geheugen per klokpuls 2x geheugenoverdracht kan doen (in plaats van 1 overdracht per puls). Dit verdubbelt de effectieve bandbreedte van het geheugen.

2.4.3 L1, L2 EN L3 CACHES

Om de snelheid van de gegevensverwerking te verhogen is er in de processor wat extra supersnel geheugen ingebouwd. Dit wordt het **cachegeheugen** genoemd en is veel sneller (en dus ook veel duurder) dan het 'gewone' interne RAM. Omdat het geheugen in de *processor* zit, hoeft de processor niet steeds het RAM-geheugen aan te spreken en de gegevens via de databus te versturen.



SOORTEN CACHE

Cachegeheugen wordt verdeeld twee soorten:

1. **Interne cache (L1)**
   Als het cachegeheugen in de processor zelf zit, noemen we dat interne cache. Dit wordt ook wel *Level 1*, of L1, cache genoemd. De meeste processors hebben Level 1 cachegeheugen ingebouwd. Het is het enige geheugen in de pc dat de snelheid van de processor kan bijhouden omdat dit geheugen in de CPU zelf zit. Level 1 cache is maximaal 64 KB. Dat is niet erg groot en het geheugen zal ook vrij snel vol zijn.

2. **Externe cache (L2)**
   Het is erg duur om Level 1 cache groter te maken. Goedkoper is het om extra cachegeheugen tussen de processor en de databus te plaatsen. Dit wordt de externe cache genoemd, beter bekend als *Level 2* of L2. De toegangstijd tot het L2 cachegeheugen is groter dan die tot het L1 cachegeheugen. De externe cache is even snel als het moederbord en kan 64 KB tot 2 MB groot zijn, afhankelijk van de gebruikte processor.

De verbinding tussen de processor en het L2 cachegeheugen wordt ook wel de **Backside Bus** genoemd (BSB), naar analogie met de FSB.

WERKING VAN DE CACHE

Als de processor gegevens of instructies (in het algemeen data) nodig heeft, haalt hij die vanaf een extern geheugen naar het interne RAM-geheugen. De data die hij direct nodig heeft, komen in het cachegeheugen terecht zodat die onmiddellijk beschikbaar zijn.

De ervaring leert dat de processor vaak data nodig heeft die vlak bij de oorspronkelijke opgeslagen zijn. De processor haalt dan ook meer data naar de cache dan strikt genomen nodig is. Zo kan hij data die na de verwerking van de vorige data nodig zijn sneller vinden. Natuurlijk werkt dit niet altijd, maar in het algemeen is deze manier van cachegebruik snelheidsverhogend.

LEVEL 3 CACHE

Bij de nieuwste generatie processors wordt de L2 cache in de processor ingebouwd. De extra cache tussen processor en moederbord krijgt dan de naam L3 cache. In dat geval is de BSB dus de verbinding tussen L3 en de processor.

Als de instelling "*System BIOS cacheable*" op Enabled staat, kan het cachegeheugen gebruikt worden om BIOS-instellingen op te slaan als dat nodig is. Hierdoor wordt het systeem sneller.

2.5 I/O

Naast de interne en externe databus voor gegevenstransport naar en van het interne geheugen bestaan er ook I/O-bussen. Deze bussen verzorgen het gegevenstransport van en naar de randapparatuur. Ook in deze bussen is er sprake van een databus, een adresbus en een controlbus.

Op een I/O-bus worden apparaten aangesloten. De manieren om een apparaat aan te sluiten zijn:

- standaard uitbreidingsbussen (insteeksleuven) zoals:
    - ISA (Industry Standard Architecture)
    - EISA (Enhanced Industry Standard Architecture)
    - PCI(-Express)
    - AGP (Accelerated Graphics Port)
- externe poorten voor printer en muis
- aansluitingen voor een interne kabel zoals de kabels van een FDD, HDD of cd-r
- apparaat-specifieke aansluitingen voor bijvoorbeeld processor en geheugenmodules

Moderne videoadapters gebruiken de PCI-E bus (PCI Express)

2.5.1 CONTROLLER

Je kunt niet ieder apparaat op een willekeurige bus aansluiten omdat ze niet allemaal dezelfde betekenis aan de code geven. Er is een tolk nodig tussen het apparaat en de bus. Dit is de controller.

Elk apparaat heeft zijn eigen controller die geschikt is voor een bepaalde bus. De controller regelt onder andere het verkeer tussen de aansluitingen van apparaten en de databus.

2.5.2 I/O-ADRESSEN

Als de processor gegevens klaar heeft staan voor bijvoorbeeld de printer, plaatst hij ze op de databus om ze naar de printerpoort te verzenden. Het probleem is dat alle apparaten werken met dezelfde databus: de gegevens kunnen voor elk apparaat bestemd zijn.

De processor geeft daarom met een **I/O-adres** aan waarvoor de bewuste gegevens bedoeld zijn. Ieder apparaat dat aangesloten is op de databus heeft een I/O-adres.

Voordat de gegevens verstuurd worden, wordt dit I/O-adres vooruitgestuurd via de **adresbus**. Alle apparaten in de computer controleren of het om hun adres gaat. Is dit inderdaad zo, dan weet het apparaat dat de gegevens die verstuurd gaan worden voor hem bedoeld zijn. De apparaten met een ander I/O-adres zullen de gegevens negeren.

Om dit systeem goed te laten functioneren gelden de volgende regels:

- Ieder I/O-adres is uniek.
- Ieder apparaat bezit een I/O-adres.

2.5.3 VASTE I/O-ADRESSEN

Voor computeronderdelen die in vrijwel elke computer aanwezig zijn, is een aantal vaste I/O-adressen afgesproken:

Apparaat	I/O-adres
LPT1	378
LPT2	278
COM1	3F8
COM2	2F8
COM3	3,00E+08
COM4	2,00E+08
Joystickpoort	200
Netwerkkaart	300
Geluidskaart	220

2.5.4 INTERRUPT REQUEST

Veel apparaten willen regelmatig gegevens versturen naar de processor. Een apparaat kan deze gegevens niet zomaar op de databus zetten, want er kan op datzelfde moment al een ander apparaat aan het zenden zijn.

Hiervoor heeft ieder apparaat een **IRQ nummer**. Een apparaat geeft eigenlijk aan de processor de melding: “Stop, je moet nu eerst dit voor mij doen.” Het apparaat vraagt een onderbreking (interrupt) aan bij de processor om te kunnen melden dat hij klaar is met het verwerken van de gegevens, dan wel dat hij klaar is om ze te ontvangen. Hij kan dus de volgende datastroom ontvangen, of wil zelf een datastroom versturen.

De processor beslist of hij zijn andere werkzaamheden kan/wil onderbreken en geeft de aanvrager de gelegenheid de bus te gebruiken. **Enkel de apparaten die de processor kunnen onderbreken krijgen een IRQ-nummer**. Het gaat hierbij om de volgende apparaten:

- processorklok
- toetsenbord
- COM-poorten

- LPT-poorten
- datum/tijd-klok
- muis
- co-processor
- harde-schijfcontroller
- USB-controller
- uitbreidingskaarten (netwerk, geluid)

2.5.5 PRIORITEIT VAN DE IRQ NUMMERS

Vanaf de 16-bits systemen heeft iedere pc zestien **IRQ nummers** ter beschikking. Deze nummers hebben een bepaalde prioriteit: hoe lager het IRQ-nummer, hoe hoger de prioriteit. De interrupts met een hoge prioriteit hebben - uiteraard - voorrang op de interrupts met een lagere prioriteit.

Hierdoor kunnen aanvragen tegelijkertijd worden gedaan, maar de processor beslist welke het eerst uitgevoerd wordt.

Een IRQ-nummer wordt ook wel een **Interrupt Request Level** (IRQL) genoemd.

2.5.6 ZESTIEN IRQ'S

De zestien IRQ's zijn verdeeld in twee groepen:

1. de eerste groep: 0 tot en met 7;
2. de tweede groep: 8 tot en met 15.

De processor kijkt echter alleen naar de eerste acht IRQ's. Om ervoor te zorgen dat een IRQ uit de tweede groep ook door de processor gebruikt kan worden, is er een soort bruggetje gemaakt waardoor de twee groepen met elkaar in verbinding staan. Hiervoor zijn IRQ 2 uit de eerste groep en IRQ 9 uit de tweede groep aan elkaar verbonden.

Via IRQ 9 gaat een onderbrekingsaanvraag naar IRQ 2, die deze vervolgens doorgeeft aan de processor. Doordat de koppeling met IRQ 2 gemaakt is, hebben de IRQ's uit de tweede groep een hogere prioriteit gekregen dan IRQ 3 tot en met 7.

De volgorde van IRQ's wordt daarmee:

0 1 2 8 9 10 11 12 13 14 15 3 4 5 6 7

2.5.7 VASTE IRQ'S

Net als bij de I/O-adressen zijn ook bepaalde IRQ's vast ingedeeld:

IRQ	Standaardapparaat	Kaarttype	Meest gebruikte toepassing
0	Processorklok	-	
1	Toetsenbord	-	
2	Koppeling naar 9	-	
3	COM2 of COM4	8/16 bits	Modem
4	COM1 of COM3	8/16 bits	Seriële printer, scanner, muis
5	LPT2	8/16 bits	Geluidskaart
6	Floppydisk-controller	8/16 bits	Floppydrive
7	LPT1	8/16 bits	Parallelle printer

8	Systeemklok	-	
9	Koppeling naar 2	8 bits	
10	Vrij	16 bits	USB
11	Vrij	16 bits	Netwerkkaart / USB
12	PS/2	16 bits	Muis (of vrij)
13	Co-processor of FPU	-	
14	1e harde-schijfcontroller	16 bits	Harde schijf
15	2e harde-schijfcontroller	16 bits	Cd-romspeler

2.6 DIRECT MEMORY ACCESS

De processor treedt op als 'verkeersleider' bij het versturen van gegevens. Op basis van I/O-adres en **IRQ** wordt het verkeer geregeld. De gegevens gaan naar één van de volgende drie plaatsen:

- een ander apparaat op de databus
- de processor
- het geheugen

Direct Memory Access (**DMA**) is een alternatief voor het Interrupt systeem, wanneer een apparaat een geheugenbewerking wil uitvoeren.

2.6.1 GEHEUGEN

Het grootste gedeelte van de gegevens gaat naar het geheugen van de computer.

Het schrijven naar het geheugen wordt geregeld door de processor. Dit geregel kost de processor echter wel veel tijd. De processor kan zijn tijd wel beter besteden dan als doorgeefluik!

Daarom mogen bepaalde apparaten **direct naar het geheugen van de computer schrijven** zonder dat de processor dit hoeft te verzorgen. Dit maakt de computer sneller.

2.6.2 DMA-KANAAL

De apparaten die dit mogen, krijgen hiervoor een speciaal kanaal (een DMA-kanaal) toegewezen. Daarmee kunnen ze het geheugen benaderen.

Door spanning te zetten op één van de acht DMA-draden krijgt een speciale DMA-chip het verzoek gegevens direct in het geheugen te zetten. Deze chip kijkt aan de hand van het DMA-kanaalnummer wie dit verzoek indient en zet de gegevens in het geheugen. Intussen kan de processor ongestoord doorwerken.

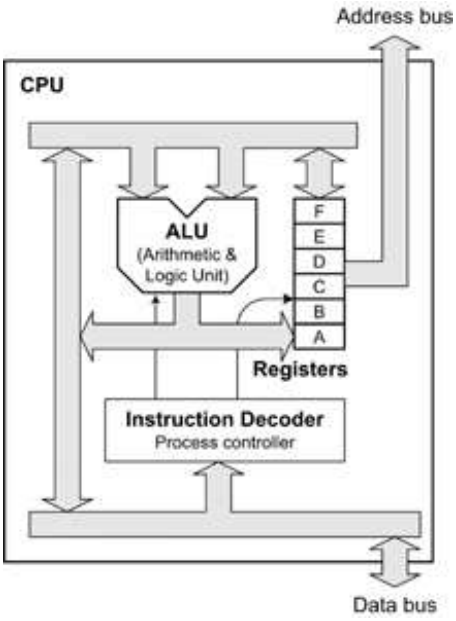
Elk DMA-kanaal is uniek, zodat het duidelijk is van welk apparaat het verzoek afkomt. De volgende tabel geeft aan welke DMA-kanalen standaard zijn toegewezen aan bepaalde apparaten.

DMA-kanaal	Apparaat
0	Geluidskaart
1	Geluidskaart of netwerk
2	Floppydisk-controller
3	LPT werkend volgens de ECP- of EPP-standaard
4	Cascade
5	Niet toegewezen
6	Niet toegewezen
7	Harddisk-controller

3 WERKING VAN EEN COMPUTER

3.1 COMPONENTEN VAN DE CPU

Een processorkern bestaat uit drie belangrijke componenten:



- een **Control Unit** (CU) of besturingseenheid:
- bestuurt de acties van de andere componenten zodat instructies in de juist volgorde kunnen uitgevoerd worden.
- **Registers**:
- Kleine interne geheugens die zeer snel gelezen of beschreven kunnen worden omdat ze zich in de CPU bevinden.
- **Arithmetic and Logical Unit** (ALU) of rekeneenheid:
- verzorgt rekenkundige en logische operaties, bijvoorbeeld het optellen van twee binaire getallen die zich in het geheugen of de registers bevinden.

3.1.1 ALU

De ALU voert **rekenkundige** (vb. optellen) en **logische** (vb. AND, OR, …) bewerkingen uit en vormt hiermee de laatste stap in het werk van de processor. Na deze bewerking wordt de resulterende informatie weer in een geheugen opgeslagen.

De ALU kan enkel rekenen met **gehele** getallen. Delingen leveren vaak getallen met cijfers na de komma op. Het werken met dergelijke rationale getallen is de specifieke taak van een **Floating Point Unit** (FPU).

3.1.2 CONTROL UNIT

De CU is het “brein in het brein” omdat het de **interne activiteiten in een processor dirigeert** (door middel van timing en signalen), die op zijn beurt de rest van de computer bestuurt. De taken toebedeelt aan deze component is in principe ongewijzigd sinds het ontwerp van de Von Neumann architectuur uit 1946.

De Control Unit bevat de **instructieset** die het mogelijk maakt om commando’s te geven aan de processor (tel op, ...). Het **ontvangt externe instructies** (afkomstig uit een programma) en converteert deze naar een reeks controlesignalen. Deze signalen bepalen de uitwisseling van gegevens tussen de rekeneenheid en de registers.

3.1.3 CPU REGISTERS

De **CPU** bevat een klein aantal **zeer snelle** geheugens (sneller dan alle ander interne en externe geheugens), die registers worden genoemd. Ze worden gebruikt bij de uitvoering van een **instructie** afkomstig uit een programma.

De belangrijkste registers van een eenvoudige **CPU** zijn:

- Het **Instruction Register** bevindt zich in de **Control Unit**.

Hierin wordt de instructie (in machinetaal, binair dus) geplaatst op het ogenblik waarop deze instructie moet worden uitgevoerd. De instructie bestaat uit een **operatiecode** (of **opcode**), die de informatie bevat welke bewerking moet worden uitgevoerd, en uit 0, 1 of meer **operands**. Het **operand** is het gegeven dat moet worden bewerkt bij de uitvoering van een instructie ofwel het adres dat aangeeft waar dat gegeven zich bevindt. De **Control Unit** zal de **opcode** en de **operands** van de instructie in het **Instruction Register** ontleden (decoderen).

- De **Instruction Pointer** (soms ook **Program Counter** of **Instruction Counter** genoemd) bevindt zich eveneens in de **Control Unit**. Hierin wordt het geheugenadres geplaatst van de eerstvolgende uit te voeren instructie. De **Program Counter** houdt dus bij hoe ver de uitvoering van een programma is gevorderd.

- De **Accumulator** bevindt zich in de **Arithmetic and Logical Unit**. Dit register bevat

    - Voor uitvoering: het **gegeven** dat de bewerking moet ondergaan
    - Na uitvoering: het **resultaat** van de bewerking

    Een elementaire processor heeft slechts één accumulator nodig maar de meeste processoren hebben er tegenwoordig tientallen, die dan gewone registers of **werkregisters** worden genoemd.

Een “word” wordt gedefinieerd door de grootte van de CPU registers. Deze term wordt ook gebruikt om de mogelijkheden van de processor. Een 16-bit CPU heeft dus vaak registers die maximaal 16 bits kunnen bevatten.

De databus heeft vaak dezelfde breedte als de registers. De adresbus kan echter een verschillende breedte hebben; wat bepalend is voor het maximaal adresseerbare geheugenbereik.

Om te kunnen programmeren hebben de registers een bepaalde naamgeving en doel. In de klassieke x86 architectuur vinden we deze registers terug:

register	*functie / naam*	
	werkregisters	
EAX	Primaire **A**ccumulator	De meeste basisbewerkingen van de ALU.
EBX	**B**ase accumulator	Geen specifiek doel, wordt gebruikt als extra opslagruimte
ECX	**C**ounter, accumulator	universele teller: een getal dat wordt bijhouden bij programmalussen (“i” in hoger programmeertalen).
EDX	**D**ata, accumulator	Een uitbreiding van de primaire accumulator, bewaart data gerelateerd aan huidige instructie.
	adresregisters	
EDI	**D**estination Index	elke lus die data genereert moet naar het geheugen schrijven. Deze index bevat het adres van die geheugenplaats.
ESI	**S**ource Index	elke lus die data gebruikt moet uit het geheugen lezen. Deze index bevat het adres van die geheugenplaats.
ESP	**S**tack Pointer	Bewaart het adres van de functie die de huidige instructie opgeroepen heeft (denk aan “return” in C#)
EBP	**B**ase Pointer	Gebruikt voor functies die parameters of variabelen nodig hebben
	Instruction pointer	
EIP	**I**nstruction Pointer	Bevat het geheugenadres van de eerstvolgende uit te voeren instructie. (niet toegankelijk voor de programmeur)

(bron: http://www.swansontec.com/sregisters.html)

Naast deze drie zijn er nog verschillende andere registers, die in deze tekst echter buiten beschouwing worden gelaten.

3.1.4 KLOKSNELHEID

De CPU is het snelst werkende onderdeel van de computer. Daarom is de **klok** zeer belangrijk, dit is een kristal dat zich buiten het IC van de microprocessor bevindt en zeer nauwkeurig de tijd bijhoudt. De snelheid van werken wordt niet alleen door de bitbreedte van de werkregisters, maar ook bepaald door de klok (kloksnelheid) en uitgedrukt in **megahertz** (MHz). 1 **MHz** = 10^6 Hz (1 miljoen Hertz) en 1 **Hz** (Hertz) is 1 cyclus (tik, klokslag) per seconde.

3.2 INSTRUCTIESET

Een instructieset (of **instruction set architecture**, ISA) is het gedeelte van de computerarchitectuur dat dient om te kunnen **programmeren**. Het specifieert de commando's (**opcodes**) die eigen zijn aan een bepaalde processor.

Instructieset = de reeks instructies die een CPU kan uitvoeren.

Een van de bekendste instructiesets is de zogenaamde **x86**, dat vele malen uitgebreid werd.

Computers met een verschillende architectuur kunnen dezelfde instructieset implementeren. Dit zorgt voor grotere compatibiliteit. De Intel Pentium en de AMD Athlon processoren implementeren een bijna identieke versie van de x86 instructieset, maar zijn qua intern ontwerp totaal verschillend.

3.2.1 MACHINETAAL

Dit zijn de **opcodes**, commando's die enkel begrepen worden door processor die de instructieset implementeert (die de woordschat machtig is).

Machinetaal = eenvoudige instructies die rechtstreeks door de CPU worden uitgevoerd.

Een programma in uitvoering geeft zijn instructies aan de CPU in **machinetaal**. Dit is moeilijk te lezen voor een menselijke programmeur gezien dit binaire codes is, en waarbij elke opcode (commando) vertegenwoordigd wordt door een nummer.

Voorbeeld (uit de x86 instructieset)

De opcode die vertelt aan de CPU om op te tellen is **0101** (decimaal getal 5).

Assembleertalen vergemakkelijken het programmeren door de opcodes een specifieke naam te geven. Met dergelijke "low-level talen" heeft een programmeur nauwkeurige controle over de registers en operaties van een CPU zonder binaire waarden te moeten schrijven.

Voorbeeld (uit de x86 instructieset)

De assemblercode die zich vertaalt naar **0101** is **ADD** (engels: optellen).

Onderstaand voorbeeld toont enkele lijnen assembler code:

```
LDA #23 ;laad het geheel getal 23 in de accumulator (EAX)
ADD #42 ;telt het getal 42 op bij de inhoud van de accumulator (resultaat = 65)
STO 34  ;bewaar de waarde van de accumulator in geheugenadres 34
```

3.2.2 INSTRUCTIES

Er zijn veel verschillende instructie die we kunnen gebruiken in machine code. We hebben al kennis gemaakt met drie ervan (LDA, ADD, STO), maar de meeste processors begrijpen er veel meer. Hier onder vinden we enkele vaak gebruikte instructies uit de x86 set:

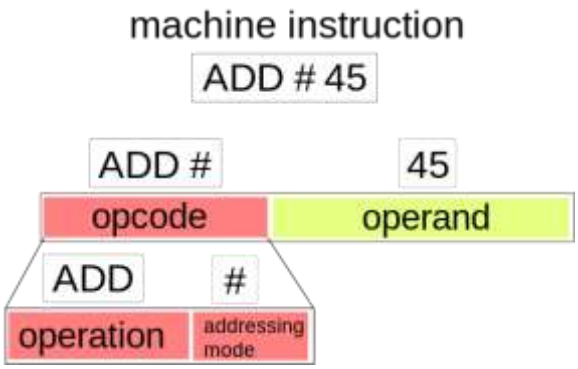
ADD	Tel een getal op bij een ander getal
SUB	Trek een getal af van een ander getal
INC	Verhoog een getal met 1 (incrementeren)
DEL	Verlaag een getal met 1 (decrementeren)
MUL	Vermenigvuldig een getal met een ander getal (multiply)
OR	Booleanse OF-berekening
AND	Booleanse EN-berekening
NOT	Booleanse NIET-berekening
XOR	Booleanse exlusieve OF-berekening
JNZ	Spring naar een andere instructie indien een getal niet 0 is. (**J**ump if **N**ot **Z**ero).
JZ	Spring naar een andere instructie indien een getal 0 is (**J**ump if **Z**ero).
JMP	Spring naar een andere instructie (**Jump**).

We bekijken even een iets uitgebreider voorbeeld van assembler instructies.

```
1. LDA #12 ;laad getal 12 in de accumulator.
2. MUL #2  ;vermenigvuldig de accumulator met 2 ( = 24).
3. SUB #6  ;verminder de waarde van de accumulator met 6 (24 – 6 = 18).
4. JNZ 6   ;als de waarde in de accumulator verschillend is dan 0, ga dan naar lijn 6.
5. SUB #5  ;verminder de acc met 5 (deze lijn wordt niet uitgevoerd!).
6. STO 34  ;bewaar het resultaat (18) op de geheugenplaats met adres 34.
```

Je merkt telkens dat de instructies uit twee delen bestaan:

- **opcode** – de naam van de instructie
- **operand** – gegevens of adres



Afhankelijk van de word size, zijn er meer of minder bits beschikbaar voor de opcode en de operand.

Sommige processoren kiezen voor veel instructies en een kleinere operand (Intel, AMD) en andere kiezen voor minder instructies en meer ruimte voor de operand (ARM).

Deze twee keuzes bepalen de classificatie van de processor:

- **CISC** (Complex instruction set computer):

- Deze sets hebben veel specifieke instructies, waarvan de kans klein is dat ze frequent gebruikt zullen worden door programma's.
- **RICS** (reduced instruction set computer):
- Dit vereenvoudigt de processor door enkel die instructies te implementeren die frequent gebruikt worden door een programma. Ongebruikelijke operaties worden dan geïmplementeerd als subroutines, die meer verwerkingstijd kosten.

3.2.3 32-BIT VERSUS 64-BIT

Er bestaan al 64-bit processors sinds 1992, en tegenwoordig zijn ze erg courant geworden. Zowel Intel en AMD hebben 64-bit chips geïntroduceerd, en de Mac G5 heeft ook een 64-bits CPU.

Dergelijke processors hebben 64-bit **ALU's**, **busbreedtes** van 64 bit en **registers** die 64 bits kunnen bijhouden.

Een goede reden waarom 64-bits processor noodzakelijk zijn is omdat ze meer geheugenadressen kunnen aanspreken (de adressen kunnen langer zijn). 32-bit chips kunnen slechts 2 of 4 GB RAM geheugen gebruiken. 64-bits processors kunnen veel meer aan (in combinatie met Windows Server 2012 bijvoorbeeld 4TB).

Op voorwaarde dat ook het moederbord over 64-bit bussen beschikt, kunnen 64-bit machines ook hogere I/O snelheden halen. Dit is bijzonder nuttig voor server machines.

Behalve de extra RAM geheugen capaciteit, heeft de "e-mail lezende gebruiker" geen zichtbaar voordeel. Wie echter veel videobewerking of fotobewerking doet en met grote bestanden werkt zal dit echter dadelijk merken. Ook videospellen kunnen er baat bij hebben, op voorwaarde dat ze werkelijk gebruik maken van de 64-bit functionaliteiten.

3.3 UITVOERING VAN INSTRUCTIES

Wanneer een programma wordt uitgevoerd, dan worden zowel de instructies en de gegevens waarmee het programma werkt gekopieerd naar het RAM geheugen. Dit is het geval bij de zogenaamde **Stored-Program** computers.

Wanneer de instructies en de data in het werkgeheugen zijn geladen, dan heeft zowel elke instructie en gegeven een eigen geheugenadres. We kunnen dit als volgt voorstellen:

GEHEUGENADRES	INHOUD	
500	LDA 1000	instructies
503	ADD 1001	
506	STO 1002	
…	…	
1000	03	data
1001	04	
1002	05	

Om nu een instructie die in het werkgeheugen zit uit te voeren moet de processor de **Fetch-Decode-Execute** cyclus doorlopen. Deze cyclus wordt ook wel de Von Neumann cyclus genoemd.

- **Fetch:** De instructie wordt uit het werkgeheugen gehaald
- **Decode:** De instructie wordt geïnterpreteerd, waarbij er signalen worden gestuurd naar
- andere relevante componenten (vb. de ALU).
- **Execute:** De instructie wordt uitgevoerd

Het is de **control unit** van de processor die deze **fetch-decode-execute** activiteiten coördineert. Bij elke klokimpuls kunnen er instructies en data uitgewisseld worden tussen de registers, werkgeheugen en I/O apparaten.

We hebben reeds gezien dat de registers in de processor een bijzondere taak hebben bij de verwerking van een instructie. Relevant in deze cyclus zijn de **Instruction Pointer** (EIP),

3.3.1 FeTCH

Elke cyclus begint met de Instruction Pointer. Dit register bevat steeds het geheugenadres van de **eerstvolgende uit te voeren instructie**.

1. De Control Unit plaatst het adres van de **instruction pointer** op de **adresbus**.
2. De Control Unit zorgt via een “read” signaal dat de instructie op dat geheugenadres in het **instruction register** geplaatst. (Lezen/schrijven van data gebeurt via de **databus**.

3.3.2 DECODE

De processor **interpreteert** de instructie dat zonet in het instruction register geplaatst werd. Het decoderen **controleert** ervoor dat de instructie wel degelijk mogelijk is en **zet** alles **klaar** om deze uit te voeren.

De instructie kan bijvoorbeeld “Add” zijn. In dien deze instructie in de instructieset aanwezig is zullen de operands opgehaald worden (in dit geval zijn dit de op te tellen waarden).

3.3.3 EXECUTE

De instructie wordt uitgevoerd door de betreffende component. Het voorbeeld met de **add** instructie wordt uitgevoerd door de ALU. Het resultaat wordt naar een register geschreven (write-back), bijvoorbeeld de EAX accumulator.

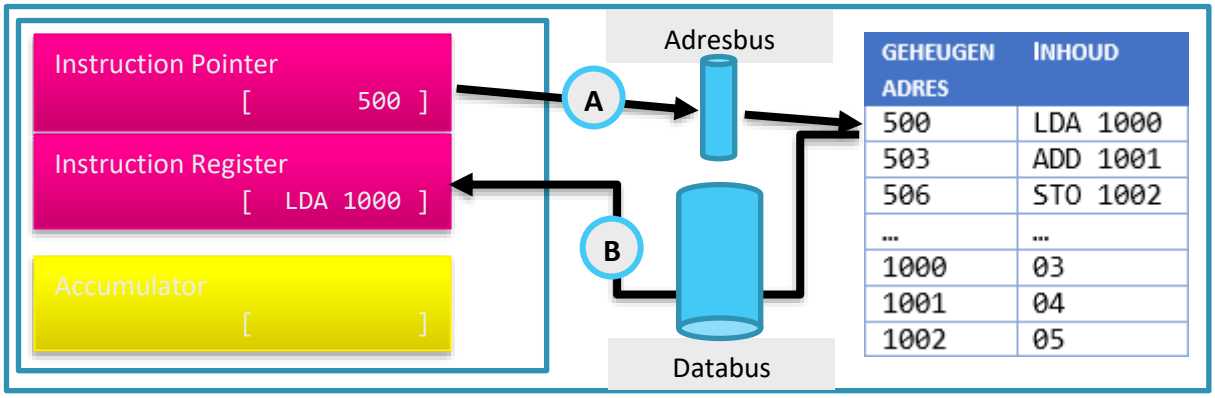
Zodra de instructie uitgevoerd wordt, kan het adres in de **instructie pointer** worden opgehoogd zodat het naar de daaropvolgende instructie verwijst. De cyclus kan opnieuw beginnen.

3.3.4 CONCREET VOORBEELD

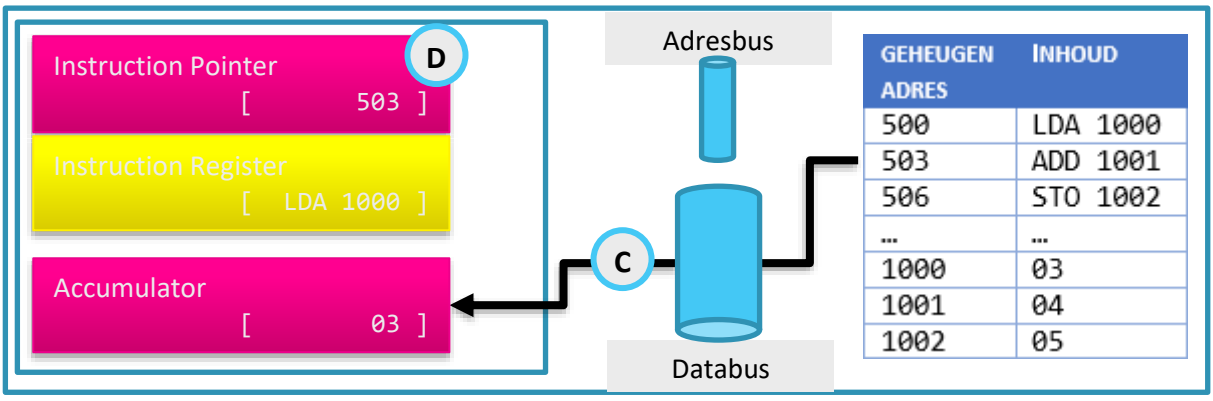
Onderstaande voorbeeld schets de uitvoering van een programma dat bestaat uit drie instructies, en drie gegevens die in het RAM geheugen zijn geladen.

Opmerking: Dit voorbeeld gebruikt een eenvoudige processorarchitectuur (8605) die slechts over 1 accumulator en een beperkte instructieset beschikt. Zie: http://www.6502.hk/6502-architecture

1. De instructie pointer wijst naar adres 500 in het geheugen. Dit is waar het programma begint.
2. FETCH:
   A. Het adres (`500`) wordt via de adresbus naar de geheugencontroller gestuurd.
   B. De inhoud (instructie) op dat adres wordt uitgelezen en via de databus in het instructie register geplaatst in het IR staat nu LDA `1000`.

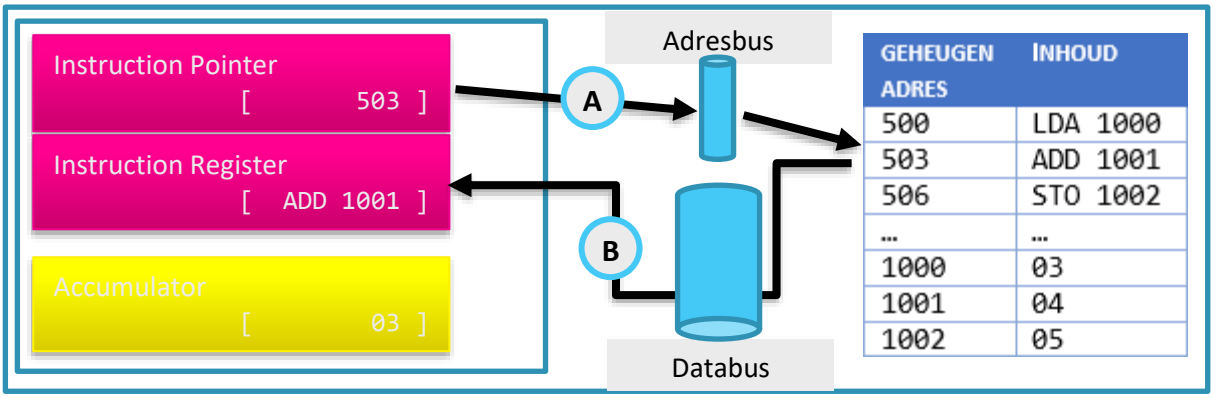


3. DECODE: De decoder herkent de instructienaam “LDA” (**L**oa**D** **A**ccumulator): er moet een waarde in een accumulator geladen worden, namelijk deze die op geheugenadres `1000` staat.
4. EXECUTE:
   C. De waarde van adres `1000` (namelijk het getal `03`) wordt in de accumulator geplaatst.
   D. De Instruction pointer wordt verhoogd zodat het naar de volgende instructie verwijst.

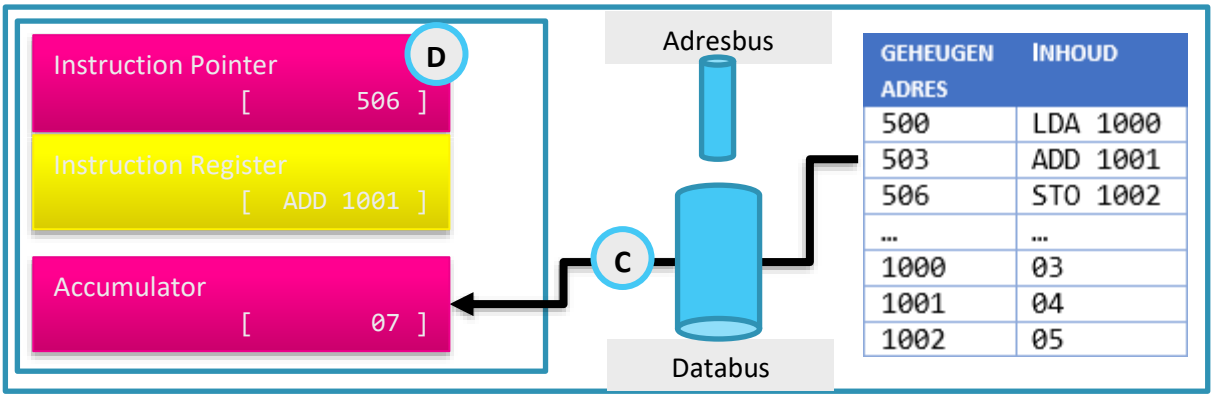


Deze cyclus wordt nu herhaald voor de overige instructies:

5. FETCH:
   A. Het IP (503) wordt verstuurd met de adresbus naar het geheugen.
   B. Het IR bevat nu ADD 1001.



6. DECODE: De decoder herkent de instructienaam “ADD” (optellen): er moet een waarde bij deze van de accumulator geteld, namelijk deze die op geheugenadres 1001 staat.
7. EXECUTE:
   C. De waarde van adres 1001 (04) wordt bij de waarde in de accumulator opgeteld (03) door de ALU. Dus: 03 + 04 = 07. Dit resultaat is de nieuwe waarde in de accumulator.
   D. De IP wordt verhoogd zodat het naar de volgende instructie verwijst.



Tenslotte wordt ook de instructie op adres 506 uitgevoerd:

8. FETCH:
    A. Het IP (506) wordt verstuurd met de adresbus naar het geheugen.
    B. Het IR bevat nu STO 1002.
9. DECODE: De decoder herkent de instructienaam “STO” (**STO**re): de waarde uit de accumulator moet opgeslagen worden in het geheugenadres, namelijk adres 1002.
10. EXECUTE:

- De waarde van de accumulator wordt via de databus gedragen naar geheugenadres 1002 (05). Het programma eindigt dus met resultaat 07 op geheugenadres 1002.

Bekijk een gelijkaardige uitvoering van een programma en de werking van de CPU a.d.h.v. volgende filmpjes:

- http://www.ib-computing.com/html/FlashMovies/FE_cycle.htm
- https://www.youtube.com/watch?v=cNN_tTXABUA

4 GEHEUGENBEHEER

4.1 INLEIDING

Geheugentoegang en beheer zijn een enorm belangrijk aspect van de werking van elke moderne computer. Elke instructie moet uit het geheugen opgehaalt worden voor dat het uitgevoerd kan worden. Daarbij komt dat de meeste instructies zelf opdracht geven om gegevens in en uit het geheugen te plaatsen.

Een multi-tasking OS vergroot de complexiteit van memory management. Zoals we al gezien hebben in een voorgaand hoofdstuk moeten we steeds een redelijk aantal processen in het geheugen geladen houden om de gebruiker het gevoel te geven dat er meerdere zaken tegelijkertijd gebeuren. De processen worden in en uit de CPU gewisseled, en dus moet ook hun code en gegevens in en uit het geheugen gewisseld worden. Alles moet met hoge snelheid gebeuren en zonder andere processen te verstoren.

Bovenop dit alles bestaan er dan nog technieken zoals gedeeld geheugen (shared memory), virtueel geheugen (virtual memory), read-only geheugen, en meer. Deze concepten worden toegelicht in dit hoofdstuk.

4.2 GEHEUGEN HARDWARE

4.2.1 GEHEUGENCHIPS

Vanuit het perspectief van de memorychip zijn alle geheugenadressen gelijk. De geheugenhardware weet niet voor welk doel een bepaald stuk geheugen gebruikt, en hoeft dit ook niet te weten.

De processor heeft enkel toegang tot zijn eigen registers en het werkgeheugen. Het heeft geen rechtstreekse toegang tot de gegevens op de harde schrijf, dus alle gegevens die gebruikt moeten worden moeten eerst naar chips van het RAM geheugen verplaatst worden.

Stuurprogramma's communiceren met hun hardware interrupts en geheugenadressen. Ze versturen korte instructies, bijvoorbeeld om gegevens van de harde schijf naar een bepaalde locatie in het geheugen te kopiëren. De schijfcontroller houdt de bus in de gaten voor dergelijke instructies, verstuurt de gegevens, en laat vervolgens de CPU weten wanneer de overdracht voltooid is met een andere interrupt. De CPU krijgt dus nooit rechtstreekse toegang tot de schijf.

4.2.2 CPU CACHING

Toegang vanaf de CPU naar zijn registers is zeer snel, meestal 1 clock tick (en een CPU kan meestal meer dan één instructie per kloksignaal uitvoeren). De toegang vanaf de CPU naar het werkgeheugen is iets trager, en kunnen een aantal kloksignalen in beslag nemen. Dit zou een ontoelaatbare wachttijd betekent voor de CPU, ware het niet dat er tussenliggende caches ingebouwd zijn in de moderne CPU's. Het idee hierachter is dat er stukjes gegevens vanuit het werkgeheugen naar de CPU cache worden gekopiëerd worden gekopiëerd, om vervolgens individuele geheugenlocaties te raadplegen vanaf de cache.

4.2.3 RESTRICTIES VOOR GEBRUIKERSPROCESSEN

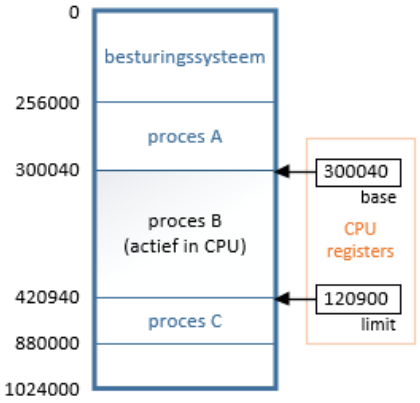
Er bestaan twee soorten processen; deze die uitgevoerd worden in **kernel-mode** en **gebruikersprocessen**. De processen in kernelmode omvatten quasi alle software die het gehele besturingssysteem uitmaken, en gebruikersprocessen zijn de toepassingen of apps die een gebruiker kan installeren en uitvoeren.

Een gebruikersproces moet gelimiteerd zijn zodat deze enkel geheugenlocaties kunnen manipuleren die aan dat proces zijn gekoppeld. Uit veiligheidsoverwegingen moeten we immers de kernelprocessen (en andere processen) kunnen beschermen van foutieve/kwaadaardige gebruikersprocessen.

De limieten voor een proces worden vastgelegd door een **base** adres en een **limit** op te geven. Het baseadres is een adreslocatie in het geheugen en de limit is het aantal adressen toegekend aan het proces. Het proces heeft dus een geheugenbereik vanaf adres "base" tot aan adres "base + limit". Het geheugenbereik van een proces wordt ook wel **logical address space** genoemd.

De base en limit waarde worden doorgaans bewaard in gelijknamige resiters. De CPU hardware waakt erover dat **elke** toegang vanuit een gebruikersproces binnen dit adresbereik valt. Indien dit niet zo is, dan wordt er een fatale fout gegenereerd. Een gebruikersproces kan dus nooit geheugenadressen buiten zijn eigen logical adress space aanspreken.

De kernel (het besturingssysteem) zelf heeft uiteraard toegang tot **alle** geheugenlocaties, wat nodig is om de code en gegevens van de processen in en out het geheugen te kunnenladen. Het wijzigen van dit base en limit adres is ook enkel mogelijk voor de kernel van het OS.



Figuur 1 – base en limit registers voor een

4.2.4 ADDRESS BINDING

Een programma staat doorgaans op de schijf als een binair, uitvoerbaar bestand (exectable). Om uitgevoerd te kunnen worden moet het in het geheugen geladen worden als een proces. Een proces dat uitgevoerd wordt krijgt toegang tot de instructies en gegevens in het geheugen. Wordt het beëindigd, dan wordt het geheugen vrijgegeven.

Bij de meeste systemen kunnen gebruikersprocessen op gelijk welke plaats in het fysieke geheugen komen te staan. Hoewel de geheugenadressering van een computer begint bij 00000, valt het eerste adres van het proces meestal op een andere plaats. In de programmalogica wordt vaak verwezen naar variabelen met een naam zoals *i*, *gemiddeldeTemperatuur* of *naam* of methoden zoals *berekenTemperatuur()* en *leesNaam()*.

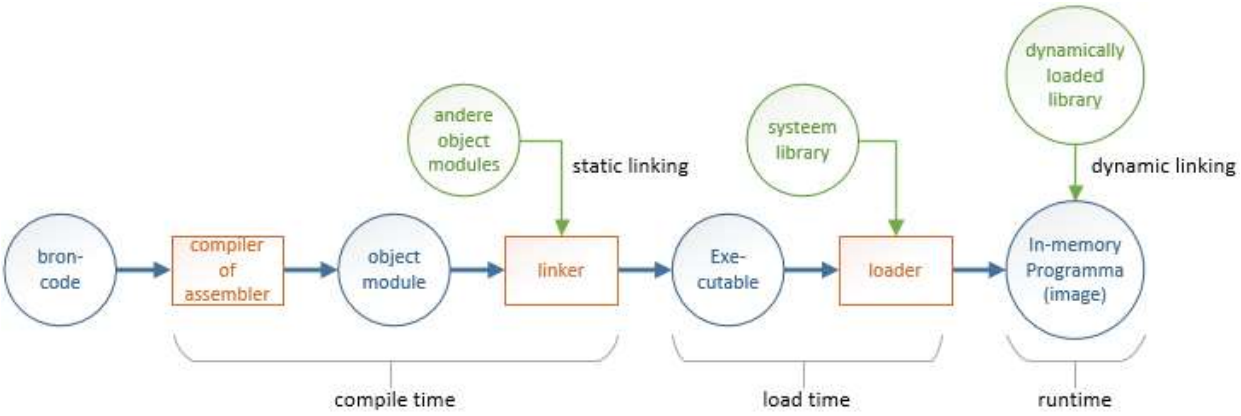
Met **address binding** bedoelen we de omzetting van deze geheugenlocaties naar werkelijke geheugenadressen.

Dit binden kan gebeuren tijdens verschillende fasen: tijdens programma ontwikkeling tot uitvoering.

- **Tijdens compile-time**
  Indien je reeds tijdens compile time weet waar het proces in het geheugen zal terecht komen, dan kan er zogenaamde *absolute code* gegenereerd worden. In absolute code wordt van uit gegaan dat het stuk geheugen steeds beschikbaar is. Dit is problematisch wanneer er meerdere user programs uitvoerd zouden moeten worden. De oude MS-DOS .COM programma's krijgen hun binding tijdens compile time.
- **Tijdens load-time**
  Een beter manier is de address binding uit te stellen tot op het punt dat het programma in het geheugen wordt geladen. De CPU genereert zogenaamde *relocatable code* (verplaatsbare code). Het startadres van het programma kan nu vrij gekozen worden: een stuk geheugen dat nog niet

in beslag is genomen door een ander proces. Een compiler die relocatable code aanmaakt werkt niet met absolute geheugenadressen, maar met *relatieve adressen* (bijvoorbeeld: variable i staat op 14 bytes vanaf start adress). De CPU vertaalt deze naar naar echte geheugenadressen tijdens load-time, wanneer het startadres van het programma gekend is.
Indien het programma echter verplaatst moet worden in het geheugen, dan moet het volledig opnieuw ingeladen worden, omdat het startadres (en alle opeenvolgende adressen) veranderd zullen zijn.

- **Tijdens run-time**
  Als het proces tijdens zijn uitvoering verplaatst moet kunnen worden in het geheugen, dan moet de adress binding nog verder uitgesteld worden. Na het inladen van het programma wordt een relocation register gevuld met het startadres van het programma (gelijkaardig aan het baseregister). Telkens wanneer er een geheugenopvraging is wordt het juiste adres gevormd (de eigenlijke binding) door naar het fysieke adres in het relocation register en het gevraagde, relatieve, adres te kijken.
  Door het absolute adres pas toe te kennen wanneer ze gevraagd wordt, kan het proces tijdens uitvoering nog verplaatst worden in het geheugen.



Figuur 2 – de verschillende stappen van een programma

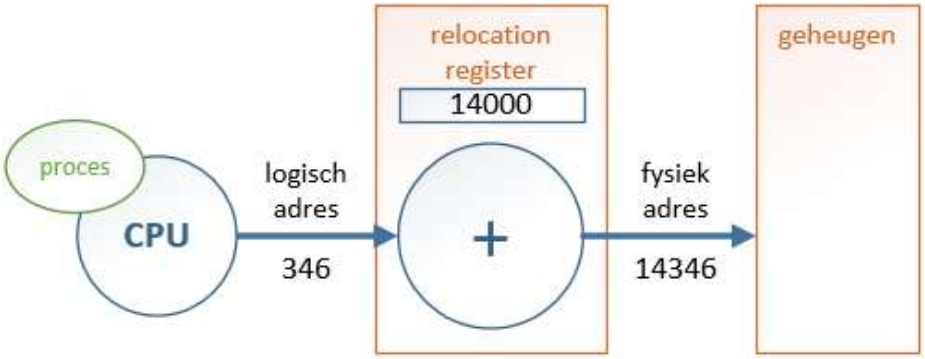
De meeste besturingsystemen gebruiken deze manier van binding.

Bovenstaande afbeelding toont de verschillende stappen waar een programma in kan verkeren, van broncode tot proces in het geheugen. Compile time, load time en runtime staan afgebeeld. Static en dynamic linking komt wat verderop aan bod.

4.2.5 LOGISCHE VERSUS FYSIEKE ADRESRUIMTE

Een geheugenadres dat gegenereerd is door de CPU tijdens de adress binding noemen we een logisch adres (*logical address*). Een adres zoals de geheugenmodule het ziet noemen we een fysiek adres (*physical adres*). Compile-time en load-time address binding genereert identieke logische en fysieke adressen. Run-time address binding resulteert echter in logisch adressen die verschillen van de fysieke adressen. Indit geval refereert men vaak naar logische adressen als zijnde *virtuele adressen*. In de syllabus behandelen we de termen logische en virtueel adres gewoon als hetzelfde. De reeks logische adressen die een programma genereert noemen we logische adresruimte of *logical address space*. Alle fysieke adressen van de geheugemodules die overeenkomen met deze logische adressen noemen we de fysieke adresruimte of *physical address space*.

De adress binding tijdens run-time, waarbij virtuele adressen naar fysieke adressen worden vertaald, wordt mogelijk gemaakt door een stukje hardware genaamde de *memory-management unit* of MMU. We illustreren dit met een eenvoudig MMU diagram.



Figuur 3 – MMU vertaalt geheugenadressen

Het *base* register noemen we nu een *relocation register*. De waarde in dit register wordt toegevoegd aan elk adres dat het gebruikersproces naar het geheugen stuurt. Als de base van een proces bijvoorbeeld 14000 is, en dat proces vraagt naar zijn virtueel adres 0, dan resulteert dit in het fysieke adres 14000. Een poging om virtueel adres 346 op te vragen resulteert in fysiek adres 14346.

Het programma krijgt dus nooit het werkelijke, fysieke adres te zien. Elke variabele, methode of geheugenlocatie in een gebruikersproces verwijst naar een logisch adres, die in werkelijkheid overeenkomt met een fysiek adres.

4.2.6 DYNAMIC LOADING

Tot dusver gingen we er steeds van uit dat het volledige programma –en alle gegevens van het proces- in het fysiek geheugen moet geladen zijn om het te kunnen uitvoeren. Dit heeft als gevolg dat de grootte van het programma niet groter kan zijn dat de geheugengrootte.

Dynamic loading biedt een oplossing, door enkel de routines in te laden die uitgevoerd moeten worden. Alle routines worden op de schijf gehouden in een reloceerbaar formaat. Wanneer een routine wordt opgeroepen, dan wordt deze eerst in het geheugen geladen, de adrestabellen worden geüpdatet, en de routinecode wordt uitgevoerd.

Het is de beslissing van de programmeur om dynamic loading te implementeren, niet deze van het besturingssysteem, hoewel het OS verschillende systeemfuncties ter beschikking kan stellen om dit te vergemakkelijken.

4.2.7 DYNAMIC LINKING EN GEDEELDE BIBLIOTHEKEN

Met libraries of *bibliotheken* bedoelen we meestal binaire bestanden met routines (functies, methodes), die door meer dan een programma benut kunnen worden benut. We herkennen deze bibliotheken vaak aan de **dll** of **lib** bestandsextensie.

Bibliotheken worden vaak gebruikt om veel uitgevoerde taken niet te moeten herschrijven, zoals functies voor databanktoegang, netwerktoegang, bestandsbeheer, enz.

Wanneer je een programma ontwikkelt, dan moeten er twee stappen ondernomen worden om de broncode van dat programma om te zetten naar een executable (exe bestand).

- De eerste stap is *compilng* (compileren): de broncode wordt omgezet naar object modules bestaande uit machinetaal.
- De tweede stap is *linking* (koppellen): de *module(s)* van jouw broncode worden samengevoegd, eventueel met *modules die door anderen geschreven* werden (maar waarvan je de broncode niet kent). Al deze modules worden samengevoegd tot de executable.

STATIC LINKING

Wanneer een dergelijke bestand (module) *statisch* gelinkt wordt aan een executable, dan wordt de letterlijke inhoud van dat bestand geïmporteerd in het programmabestand. Je programma (exe-bestand) bevat op die manier alle routines die het nodig heeft.

DYNAMIC LINKING

In dit geval wordt het linken (koppelen) tussen progamma en de bibliotheek *uitgesteld* tot op het moment van uitvoering. De executable bevat enkel geen routines uit de bibliotheek maar een stukje code (een *stub*) met *referenties* naar de externe bibliotheek.

Deze stub bevat alle informatie die het programma nodig heeft om

- De bibliotheek de lokaliseren.
- De nodige routines van de bibliotheek in het geheugen te laden (indien nog niet het geval).
- De adressen van de in het geheugen aanwezige routines te vinden.

De stub wordt tijdens uitvoering overschreven met het geheugenadres van de uit te voeren routine, waarna deze routine wordt uitgevoerd.

Dynamic linking biedt een aantal voordelen t.o.v. static linking

- Indien de bibliotheek een update krijgt, dan moet het programma niet opnieuw gelinket worden. De code zal namelijk automatisch linken aan de nieuwe dll tijdens de uitvoering.
- Er wordt schijfruimte bespaard, zeker in het geval van veel voorkomende bibliotheken, die niet met het programma mee moeten verhuizen.
- Er wordt geheugenruimte bespaard, want de bibliotheek hoeft maar eenmaal in het geheugen worden geladen, ongeacht het aantal processen die de routines uit de bibliotheek gebruiken.

Dit heeft echter ook een nadeel: indien de nieuwe versie van de blbiotheek niet compatibel is met het programma, dan kan het programma niet adequaat functioneren. Dit wordt ook wel eens “DLL-hell” genoemd. Het is de verantwoordelijkheid van de programmeur om een dergelijke situatie te voorkomen.

4.3 TOEKENNING VAN GEHEUGEN

Een mogelijke aanpak voor geheugenbeheer is om elk proces een uit één stuk bestaande geheugenruimte te geven. Het besturingssysteem komt eerst aan bod, en de overblijvende geheugenruimte wordt verdeeld tussen de gebruikersprocessen. Het besturingssysteem wordt meestal ingeladen op lage geheugenadressen, dicht bij de interrupt routines.

De moderne kernels en hardware laten toe om te werken met runtime binding, zodat processen tijdens hun uitvoering verplaatst kunnen worden van geheugenlocatie. Hierdoor krijgt het OS de ruimte om dynamisch te groeien of te krimpen in geheugengebruik.

4.3.1 MEMORY ALLOCATION

Geheugen "alloceren" is hetzelfde als geheugen toekennen. Een efficiënte allocatie van geheugen is één van de meest essentiele vereisten van een memory management systeem. Om elk proces stukken geheugen te kunnen geven dat uit één blok bestaat kunnen er verschillende strategiëen gebruikt worden. Meestal wordt er een lijst bijgehouden van ongebruikte geheugenblokken (free memory blocks). Wanneer een proces in het geheugen geladen wordt moet er een geschikt "gat" (ongebruikt blok) gevonden worden waar het proces het best in past.

De meest gebruikte strategieën zijn:

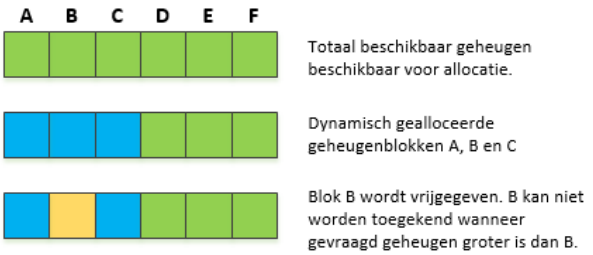
- **First fit:** alloceert het *eerste vrije* blok dat groot genoeg is om het proces in te plaatsen.
- **Best fit:** alloceert het *kleinste vrije* blok dat groot genoeg is. De volledige lijst met vrije blokken moet doorzocht worden, of de volledige lijst moet gesorteerd worden. De ruimte van het vrije blok wordt optimaal benut, maar er blijft een klein, onbruikbaar stukje geheugen over.
- **Worst fit:** alloceert het *grootste vrije* blok. De gehele lijst moet doorzocht of gesorteerd worden. Er blijft een groot, en wellicht nog bruikbaar stuk geheugen over na allocatie van het vrije blok.

Simulaties tonen aan dat first fit en best fit beter zijn dan worst fit. Ze zijn elkaars gelijken in wat betreft efficiëntie in plaatsgebruik, maar first fit is de snelste. Worst fit geeft op zijn beurt de minste fragmentatie (zie verder).

4.3.2 FRAGMENTATIE

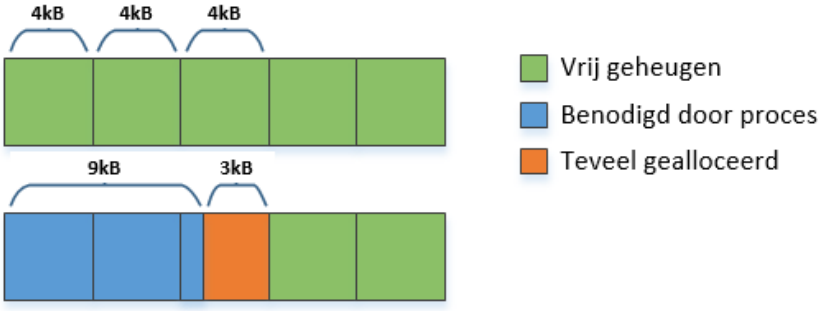
Alle memory allocation strategiën hebben last van *externe fragmentatie* (first en best fit hebben dit meer dan worst fit).

Externe fragmentatie betekent dat het vrije geheugen opgesplitst wordt in vele kleinere stukjes, en geen van deze stukjes is groot genoeg om het volgende proces te kunnen bevatten – hoewel de totale som van die stukjes wel groot genoeg is.



Figuur 4 - externe fragmentatie

Een ander fenomeen is *interne fragmentatie*. Dit wordt veroorzaakt door het feit dat geheugen toegekend wordt in blokken die deelbaar moeten zijn door 4, 8, 16. Het werkelijke benodigde geheugen voor een proces zal bijna nooit exact die grootte zijn. Een proces krijgt dus meestal te veel geheugen toegekend.



Figuur 5 - interne fragmentatie

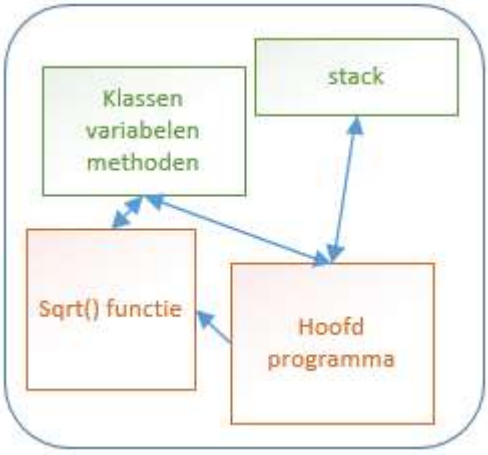
Het probleem van fragmentatie heeft een aantal oplossingen:

- **Compaction**
  vermindert het probleem van externe fragmentatie, door alle gealloceerde geheugenblokken op elkaar te laten volgen. Alle vrije blokken worden zo gecombineerd tot één groot stuk vrij geheugen. Dit is enkel mogelijk wanneer er gewerkt wordt met *relocatable* code, en wanneer de adres binding run-time gebeurd: enkel dan kunnen processen verplaatst worden in het geheugen tijdens hun uitvoering.
- **Paging**
  splitst het fysieke geheugen in blokken van vaste groottes. Deze blokken worden opgevuld met logisch toegekende geheugenblokken van dezelfde groote (zie verder).
- **Garbage collection**
  neemt een kijkje in de processen en ruimt alle ongebruikte variabelen op, om zo de ruimte vrij te geven. Dit gebeurt vrijwel alleen in *managed* code, geschreven met het .NET framework of Java.

4.4 SEGMENTATIE

De meeste gebruikers (programmeurs) zien hun programma's niet als een lineair stuk geheugenruimte.

Wanneer je een programma schrijft praat je wellicht over het "hoofdprogramma", "de stack" en de "math bibliotheek". Het maakt niet uit waar in het geheugen de stack of de sqrt() functie van de math bibliotheek staat. Programmeurs denken dus over hun programma alsof ze ingedeeld is in *segmenten* die elk een bepaalde rol vervullen.



Figuur 6 - hoe een programmeur zijn programma ziet

4.4.1 PROGRAMMASEGMENTEN

Geheugensegmentatie ondersteunt deze zienswijze. De logische adresruimte van een proces kan gezien worden als een *collectie van segmenten*. Elk van deze segmenten krijgt een *naam* en een lengte (*offset*). Elk segment begint bij een bepaald adres, *segment base adres* en is even groot als zijn *offset*.

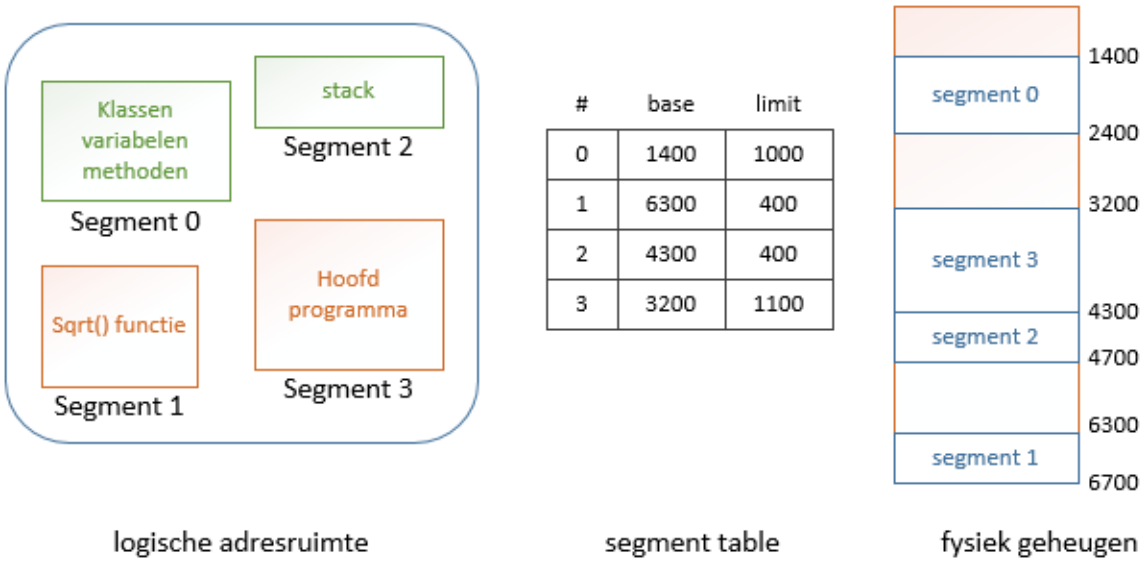
Een compiler maakt automatisch de segmenten aan wanneer het broncode compileert. Een C compiler kan bijvoorbeeld de volgende segmenten genereren:

1. De code
2. Statische variabelen (=globale variabelen)
3. De *heap* (dit is de ruimte waarin het programma alle variabelen kan alloceren)
4. De *stacks* (opeenvolging van subroutines) die gebruikt worden door elke thread
5. De standaard C bibliotheek

4.4.2 SEGMENT TABEL

De segmenten van het programma worden bijgehouden in een *segment tabel*. Deze tabel houdt het *segment base adres* bij, en het *segment limit*, om zo de verschillende segmenten af te bakenen. Een logisch adres bestaat hier dus eigenlijk uit een *segment naam* en *een offset*. Wanneer de offset bij het base adress van dat segment wordt geteld, verkrijgen we het fysieke adres van de gevraagde byte in het geheugen.

We nemen een voorbeeld. Stel dat we 4 segmenten hebben (we nummeren ze 0 t.e.m. 3 in plaats van een naam te geven). Deze segmenten hebben ruimte gekregen in het fysieke geheugen. De segment tabel heeft een aparte record voor elke segment: het beginadres van het segment in het fysiek geheugen (base), en de lengte van dat segment (limit). Segment 2 is bijvoorbeeld 400 bytes lang en begint op locatie 4300. Een bevraging naar byte 53 van segment 2 betekent dus de byte op fysiek geheugenadres 4353 (4300 + 53). Een bevraging naar byte 1222 van segment 0 resulteert in een fatale fout (*segmentation fault*) die doorgegeven wordt naar het besturingssysteem (segment 0 is slechts 1000 bytes lang).



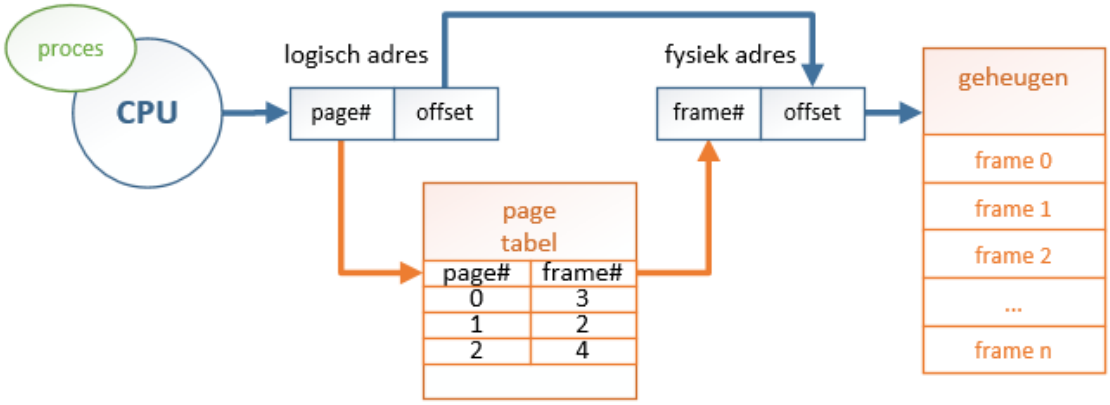
Figuur 7 - segmentatie voorbeeld

4.5 PAGING

Dankzij paging hoeft een memory mangement systeem niet in één doorlopend stuk geheugen te worden gealloceerd. Hierdoor wordt *externe fragmentatie* van het geheugen (en de noodzaak voor *compaction*) onbestaande. Dit is waarom de meeste moderne besturingsysteem beschikken over memory management met paging.

4.5.1 PRINCIPE

Paging splitst het *fysieke geheugen* op in blokken van gelijke groote (meestal 4 kB) die *frames* genoemd worden, het *logische geheugenruimte* wordt eveneens opgesplist in blokken van gelijke grootte, die *pages* genoemd worden. Een page van gelijk welk proces kan nu precies geplaatst worden in een frame dat nog niet in gebruik is.

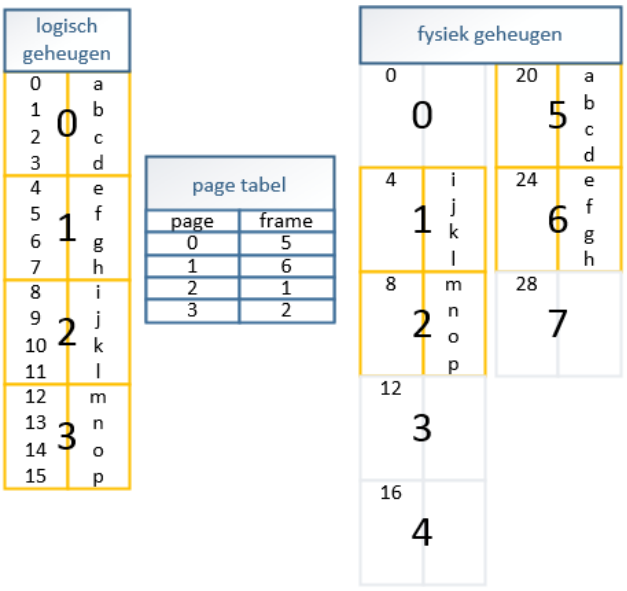


Figuur 8 - Paging hardware

Er wordt gebruik gemaakt van een *page table* om bij te houden welke frame een bepaalde page bevat. In de onderstaande afbeelding is page 2 van de logische adresruimte van het programma geplaatst in frame 3 van het fysieke geheugen.

4.5.2 VOORBEELD

Beschouw het volgende voorbeeld (op kleine schaal), waar een proces 16 bytes logische adresruimte toegekend krijgt. De logische adresruimte wordt opgesplitst in 4-byte pages. We hebben 32 bytes fysiek geheugen, en veronderstellen dat ander processen de overige vrije ruimte zullen innemen.



Figuur 9 - 32 byte geheugen met 16 byte proces

Je merkt dat paging zich gedraagt alsof we een ganse tabel relocation registers hebben, ééntje voor elke page van het logische geheugen.

4.5.3 EIGENSCHAPPEN

Met paging is er *geen externe fragmentatie*, omdat geheugen verdeeld wordt in blokken (frames) die even groot zijn aan de pages die moeten worden toegekend. Elk frame van fysiek geheugen kan gebruikt worden, zonder dat er “gaten” tussen bestaan of dat er een gat moet gevonden worden van adequate grootte.

- Toch is er sprake van *interne fragmentatie*, ook omdat het geheugen in blokken wordt toegekend. We kunnen stellen dat de laatste page van een proces nooit helemaal gevuld zal worden, waardoor dat stuk geheugen verloren gaat. Indien processen hun data en code segmenten in aparte pages bewaren, gaat er nog meer verloren.
- Hoe groter de *pagesize*, hoe meer geheugen er verloren gaat door interne fragmentatie. Grotere pages zijn echter efficiënter qua overhead voor de vertaling tussen logische en fysieke adressen. In systemen met veel geheugen vereist een kleine page size ook een zeer grote paging tabel.

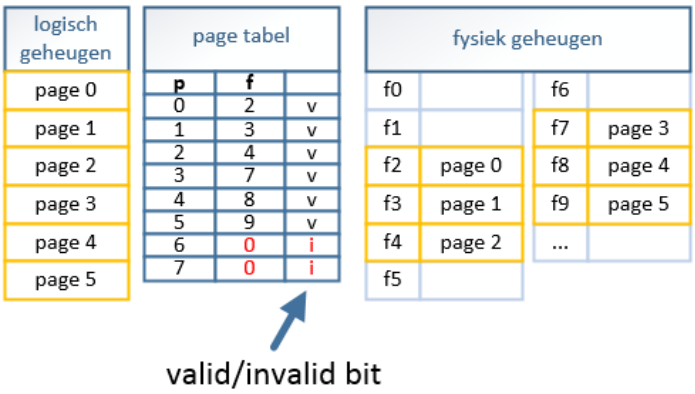
- De traditionele pagesize is 4 kB, maar sommige systemen hebben meerdere vaste page size om zo het beste van beide te verkrijgen.
- Pages die weinig geraadpleegd worden kunnen *geswapped* worden (zie hoofdstuk over scheduling). Met swapping bedoelen we: uit het geheugen halen en naar de harde schijf plaatsen. Hiermee is er meer plaats voor nieuwe processen en hun, wellicht, actievere pages.
- In Windows systemen worden geswappete pages bewaard in de *pagefile*, op linux systemen noemt dit de *swapfile*. Mobiele besturingssystemen swappen niet, maar vragen de apps om zelf geheugen vrij te maken (ze kunnen ook de apps gewoon sluiten).
  - Het besturingssysteem moet de *paging tabel* van elk process bijhouden en bijwerken wanneer de pages van het process in en uit het geheugen worden gehaald. Het moet ook de correct paging tabel toepassen wanneer er om een bepaald proces gevraagd word. Dit creëert veel overhead.

4.5.4 PROTECTION

De paging tabel kan er ook voor zorgen dat processen enkel het geheugen kunnen benaderen dat hen toegekend is. Dankzij paginering kunnen stukken van de logische adresruimte speciale beperkingen krijgen. Een page kan bijvoorbeeld read-write, read-only, of read-write-execute zijn.

De paging tabel definieert deze beperkingen met behulp van een extra veld per page. Wanneer het proces (de CPU) een geheugenbenadering maakt kan er vervolgens gecontroleert worden of de benadering in de correct mode valt.

Wanneer een proces bepaalde pages niet gebruikt, dan worden deze gemarkeert in de paging table met *valid/invalid bits*. Op die manier is er zekerheid dat er geen ongeldige geheugenadressen gerefereerd kunnen worden. Merk op dat deze manier niet alle ongeldige geheugenreferenties kan voorkomen, wegens de interne fragmentatie: de laatste page van de logische geheugenruimte is meestal niet helemaal in gebruikt.



Figuur 10 - valid/invalid bit kolom

4.5.5 GEDEELDE PAGES

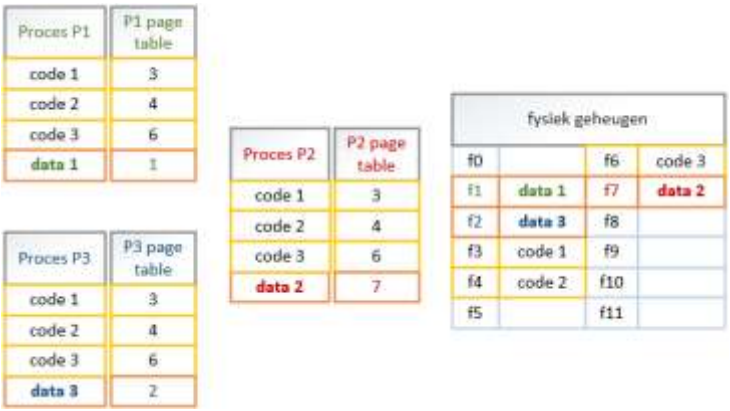
Een paging systeem faciliteert het *delen van geheugen* tussen processen door een framenummer (van een frame dat gegevens of code bevat) te kopiëren naar de paging tabellen van verschillende processen. Op dat moment hebben de processen toegang tot hetzelfde stuk geheugen.

Indien het om code gaat, kan deze enkel gedeeld worden indien de code niet gewijzigd kan worden tijdens run-time. In dat geval onstaat een beveiligingslek en mogelijk ongeldige geheugenverwijzigen voor processen die deze code gebruiken.

Stel dat we een server hebben die 40 gebruikers bedient. Elk van de gebruikers voert een tekstverwerkingsproces uit op het serversysteem. Als het tekstverwerkingsprogramma 150 kB aan code gebruikt, en 50 kB aan data, moet er in totaal 8000 kB geheugen gereserveerd worden om die 40 gebruikers te kunnen bedienen.

Als de editor geen self-modifying code bevat, dan kan de 150 kB aan code gedeeld worden tussen de 40 gebruikersprocessen. De code van het programma hoeft dan slechts 1x in geladen worden. Elk proces beschikt nog steeds over een eigen stuk geheugen voor data, in totaal 2000 kB (40 x 50 kB). Het totale benodigde geheugen is nu 2150 kB, wat veel minder is dan de oorspronkelijke 8000 kB.

Onderstaande afbeelding schematiseert drie gebruikers die elk een proces uitvoeren (een tekstverwerker). Er bestaan drie processen, maar elk van deze processen verwijst naar een stuk *gedeelde code*, namelijk dat van de tekstverwerker.

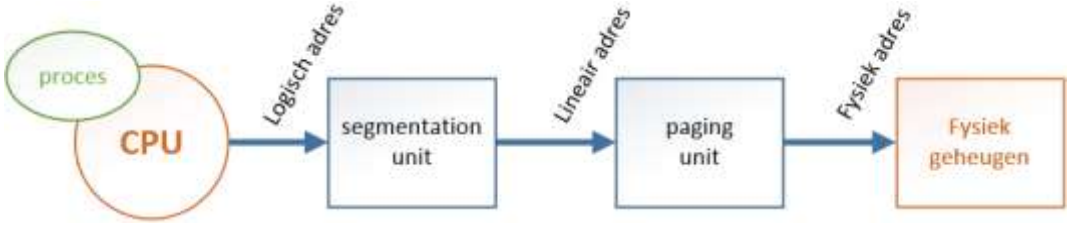


Figuur 11 – gedeelde pages

4.5.6 SAMENWERKING PAGING – SEGMENTERING

Pagineren en segmenteren zijn twee verschillende technieken die het logische geheugen opsplitst en verdeeld over het de fysieke adresruimte. Toch gaan ze hand in hand. Een proces is opgedeeld in segmenten, en het zijn eigenlijk deze segmenten die verspreid worden over (meerdere) pages.

Onderstaande afbeelding schetst de hardware aanwezig in de Pentium architectuur voor het omzetten van een logisch adres (zoals het proces het geheugen ziet) naar een fysiek adres (zoals de geheugencontroller de adressen kent). Uit een logisch adres wordt het juiste segment afgeleid waar deze zicht bevindt in de (lineaire) reeks pages. Eens de page en offset gekend is, kan het juiste frame worden gevonden en daarmee het fysieke adres.



Figuur 12 - Hardware voor adresomzetting

4.6 VIRTUEEL GEHEUGEN

In de voorgaande onderdelen hebben we gezien hoe geheugenfragmentatie voorkomen kan worden door de toegekende geheugenruimte aan processen op te splitsten in kleinere stukken (pages). Het volledige proces moest echter nog steeds in zijn totaliteit in het geheugen worden geladen.

4.6.1 ACHTERGROND

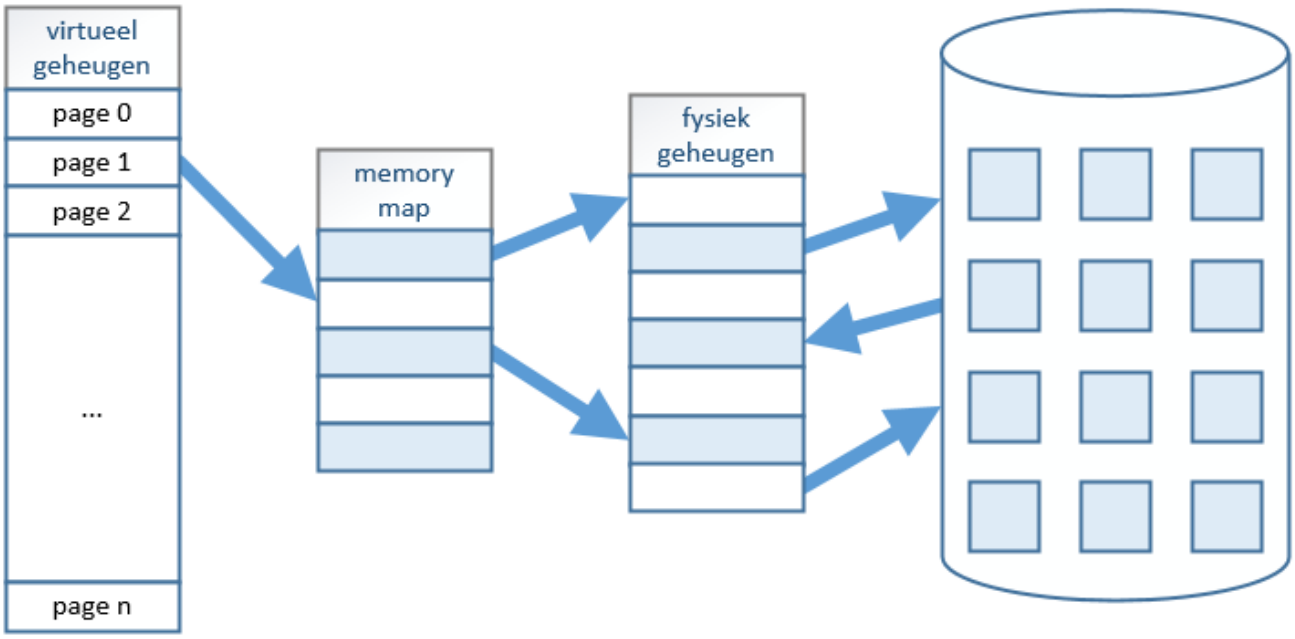
In werkelijkheid zijn er weinig processen die toegang moeten hebben tot *al* hun pages tegelijkertijd. Denk bijvoorbeeld aan de volgende dingen:

- Code om fouten af te handelen is niet nodig totdat er een fout voorkomt. Sommige fouten komen zelden voor.
- Arrays en collecties worden vaak groter dan nodig gemaakt, en enkel een klein stuk van die arrays worden bezet door gegevens.
- Bepaalde routines (methodes, functies,…) of objecten worden zelden gebruikt.

De mogelijkheid om enkel de benodigde stukken van een proces in het geheugen te laden (en enkel op het moment dat ze nodig zijn) heeft een aantal voordelen:

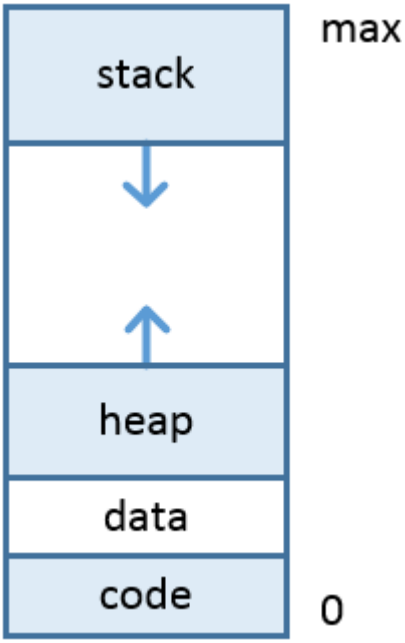
- Programma's kunnen veel meer geheugen toegekend krijgen (logische/virtuele geheugenruimte) dan wat er fysiek op de computer aanwezig is.
- Omdat elk proces slechts een stukje van hun totale adresruimte benut, is er meer fysiek geheugen over voor andere processen. Dit verbetert CPU gebruikt en doorvoersnelheden.
- Minder I/O is nodig wanneer een proces in en uit het RAM geheugen geswappet wordt.

Onderstaande afbeelding schetst de algemene layout van de ~~logische~~ *virtuele geheugenruimte* van een proces, en toont aan dat deze veel groter kan zijn dat het fysiek aanwezige geheugen.



Figuur 13 - virtueel geheugen > fysiek geheugen

We vergelijken dit eens met de manier waarop een programmeur de virtual address space van zijn programma ziet. Dit wordt getoond in de afbeelding hiernaast. De werkelijke (fysieke) layout wordt echter bepaald door de paging table van het proces.

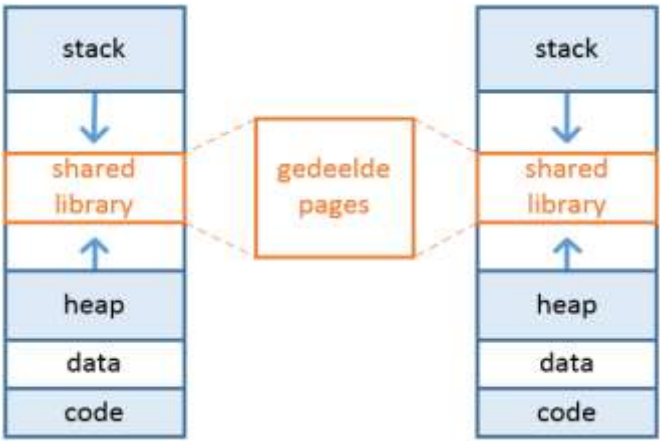


Figuur 14 – virtuele adresruimte (segmenten)

Merk op dat het toegekend geheugen aan het programma zeer ruim is: een grote leegt in het “midden” van de adresruimte wordt nooit gebruikt, tenzij de stack of de heap groeit:

- De **stack** groeit wanneer een routine een andere routine oproept. Is de lijst met de huidige actieve routines in het programma. Wanneer een routine afgelopen is (return statement) krimpt de stack.
- De **heap** groeit wanneer er meer variabelen of objecten worden aangemaakt. Wanneer variabelen niet langer gebruikt worden en opgeruimt worden, komt er weer heap-ruimte vrij.
- Het **data** segmenten bevat meestal resources zoals constanten, strings en binaire gegevens (bvb. programmaicoon, embedded geluid,…). Dit segment moet zelden gewijzigd worden.
- Het **code** segment bevat de machinetaal voor alle instructies: de eigenlijk compilatie van jouw broncode. Dit segment kan niet gewijzigd worden, behalve in processen met *self-modifying* code.

Omdat paging het mogelijk maakt om geheugen te delen, zou een gedeelde bibliotheek (bijvoorbeeld een systeemlibrary, of een .NET CLR bibliotheek er als volgt kunnen uitzien:

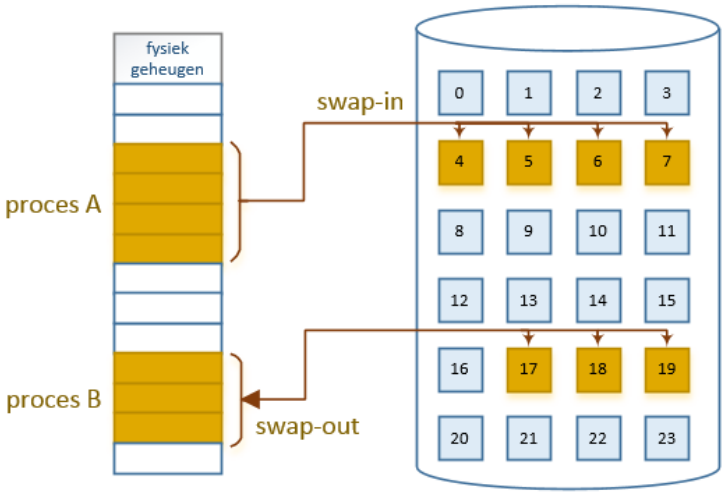


Figuur 15 - gedeelde codebibliotheek

Het proces hoeft zich echter niet bezig te houden met de werkelijke layout van deze structuur. Het is immers de taak van de MMU (memory management unit) om de logische pages te koppelen aan de fysieke frames in het geheugen.

4.6.2 DEMAND PAGING

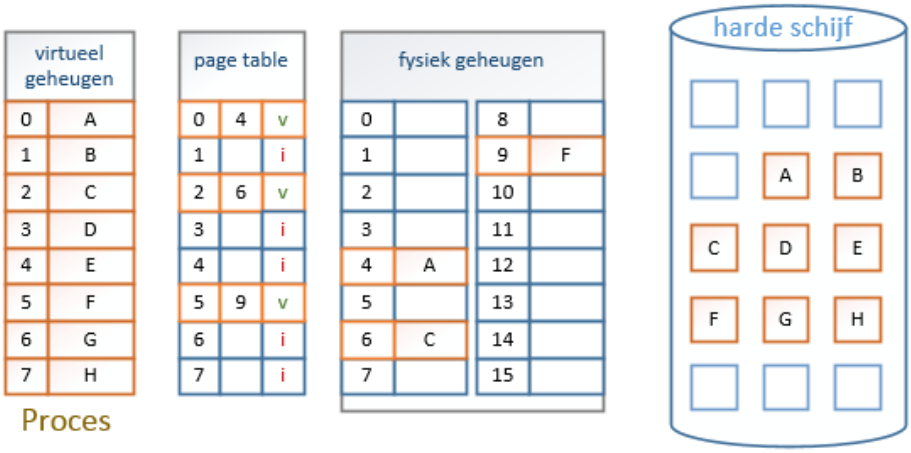
Het basisgedachte achter demand paging is dat wanneer een process een ingeswappet wordt (*swap in*: vanaf de schijf naar het geheugen gebracht wordt), niet *alle pages* tegelijk moeten ingeswappet worden. Enkel de pages die nodig zijn worden in het geheugen gebracht (on demand). Dit wordt ook wel *lazy swapping* genoemd.



Figuur 16 - swapping

GEVOLGEN VOOR DE PAGE TABLE

De page tabel moet bijhouden welke pages er in het geheugen gebracht zijn en markeert deze als “valid”. Pages dit niet in het geheugen staan worden “invalid” gemarkeerd. De page tabel kan extra informatie informatie bevatten omtrent waar het de niet-ingeladen pages kan terugvinden indien nodig.



Figuur 17 - virtueel geheugen van een proces

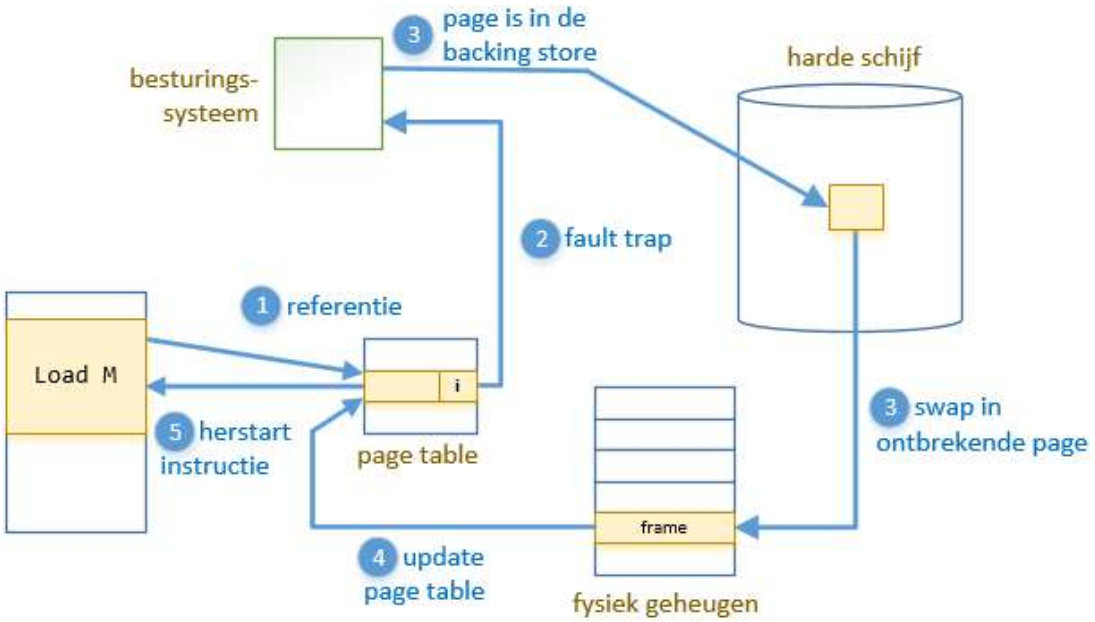
Er zijn nu twee mogelijke scenarios:

1. Als het process enkel pages benaderd die in het fysieke geheugen zijn geladen (*memory resident pages*), dan verloopt alles net alsof het gehele proces in het geheugen is geladen.
2. Als er een page benaderd moet worden die niet ingeladen is, dan wordt er *page fault* gegenereerd. Het besturingssysteem vangt deze fout op (via een zgn. *trap*), en moet dit afhandelen.

PAGE FAULTS

Een *page fault* vindt plaats wanneer een benodigde page niet in het geheugen geladen is. Het proces kan niet verder zonder dit stuk geheugen en moet wachten tot de volgende stappen voltooid zijn:

1. Het gevraagde geheugenadres wordt gecontroleert om zeker te zijn dat het een geldige referentie betreft
2. Als het een ongeldig adres is dan wordt het proces beëindigd, zoniet moet de page in het geheugen gebracht worden.
3. Er wordt een frame gezocht in het fysiek geheugen (ongebruikte frames staan meestal in een *free-frame list*)
4. Een schijf I/O operatie wordt ingepland om de page te kopiëren vanaf de schijf. Dit zal doorgaans het proces in de I/O waiting queue plaatsen zodat andere processen wat CPU-tijd krijgen.
5. Zodra de I/O operatie voltooid is wordt de page tabel van het proces geüpdatet met het nieuwe framenummer en page wordt als *geldig* gemarkeerd.
6. De instructie die de page fault veroorzaakte wordt opnieuw uitgevoerd zodra het proces opnieuw CPU-tijd krijgt.



Figuur 18 - afhandelen van een page fault

In uitzonderlijke gevallen worden er helemaal geen pages ingeswappet. Ze moeten eerst opgevraagd worden door page faults. Dit noemen we *pure demand paging*.

De benodigde hardware om virtueel geheugen te ondersteunen is hetzelfde als deze die nodig is voor paging en swapping: er is een *page table* nodig, en een *secundair geheugen* (meestal een harde schijf met swap space, bvb een pagefile).

GEVOLGEN VOOR PRESTATIES

Het spreekt voor zich dat de prestaties in het gedrang komen zodra er een page fault plaatsvindt. Het systeem moet nu immers het proces in de waiting queue plaatsen en een I/O operatie uitvoeren.

Er moet opgemerkt worden dat een speciaal toegewezen ruimte op de schijf voor swapping (*swap space*) snellere toegang biedt dan een regulier bestandssysteem, omdat het niet doorheen een mapstructuur hoeft te traverseren. Omwille van deze reden zullen sommige besturingssystemen een volledig proces van het bestandssysteem overbrengen naar de swap space alvorens het proces op te starten. Op die manier zijn alle pages reeds aanwezig in de (snellere) swap space.

4.6.3 THRASHING

Thrashing gebeurt wanneer virtuele geheugen constant bezig is met paging, het constant in en uit swappen van pages van en naar de harde schijf. Er blijft haast geen tijd meer voor de CPU om productief bezig te zijn, en de prestaties van het systeem gaan drastisch omlaag.

Wanneer er teveel pages in gebruik zijn door frames, dan wordt de geheugencapaciteit beperkt. Het kan niet langer alle benodigde pages beschikbaar houden om processen te geven wat ze op dat moment nodig hebben. Het moet voortdurend pages wisselen voor elk proces dat aan bod komt.

Hoe meer processen er uitgevoerd worden op het systeem, des te meer de page faults er zullen voorkomen, wat uiteindelijk trashing als gevolg kan hebben.

Oplossingen om trashing te voorkomen zijn de *working set strategie* en de *page fault frequency* aanpak.

WORKING SET

Het *working set* model legt op dat een proces enkel in het RAM geheugen mag staan wanneer alle pages dat het proces momenteel nodig heeft ook in het geheugen passen. Het model is een alles of niets strategie, wat betekent dat wanneer het aantal pages dat het proces nodig heeft toeneemt, maar er is geen geheugen vrij voor deze extra pages, dan wordt het proces uit het geheugen geswappet. De working set is dus de collectie processen die momenteel in het geheugen zijn geladen.

Wanneer een belaste computer veel processen in de wachtrij heeft staan dat, en mocht elk process een stukje CPU tijd toegekent krijgen terwijl niet alle benodigde pages in het geheugen staan, onstaat er trashing.

Door processen volledig uit het geheugen te swappen dan zullen de andere processen veel sneller eindigen dan mocht de computer ze allemaal tegelijk proberen uit te voeren. De processen zullen ook tijdiger hun uitvoering beëindigen omdat ze sneller vorderingen kunnen maken in plaats van te wachten op het inladen van pages.

De working set strategie voorkomt trashing en verhoogt de graad van multiprogramming, wat het CPU gebruik optimaliseert.

PAGE FAULT FREQUENCY

Een directere aanpak van het trashing probleem is het *tellen van de page-faults* die binnen een bepaalde tijd voorkomen in een proces en op basis hiervan handelen.

Een proces krijgt een vast aantal frames toegekend. Wanneer de page faults van dat proces boven een bepaalde limiet stijgen, dan moet dat proces meer frames toegekend krijgen. Als het aantal page faults van dat proces onder een bepaalde limiet komt, dan kunnen er een aantal frames van worden afgenomen om te verdelen aan andere processen.

Het aantal toegekende frames aan een proces wordt ook wel zijn memory pool genoemd.

Als de som van alle memory pools niet langer in het geheugen passen, dan moeten sommige processen geswappet worden naar de harde schijf.

4.6.4 WINDOWS

Vanaf Windows XP gebruikt het virtueel geheugensysteem *demand paging* met *clustering*. Page faults worden afgehandeld zoals bij demand paging, namelijk de page wordt pas opgevraagd nadat een page fault heeft plaatsgevonden, maar daarbij worden ook nog een aantal opeenvolgende pages opgehaald.

Waneer een process voor het eerst aangemaakt wordt, dan krijgt het een working set minimum en maximum. Het *working set minimum* is het aantal pages dat het process gegarandeerd in het geheugen krijgt. Indien er voldoende geheugen aanwezig is, dan mag het process nog meer pages toegekend krijgen, tot aan het *working set maximum*.

Voor de meeste applicaties zijn deze minimum en maximum limieten 50 en 345 pages. In sommige omstandigheden kan het maximum aantal pages van een process echter overschreden worden.

De virtuele geheugen manager onderhoudt een lijst met beschikbare frames in het fysieke geheugen. Bij deze lijst hoort een *drempelwaarde* die aanduidt of er voldoende vrije geheugenframes voldoende zijn.

Wanneer er een *page fault* voorkomt, en het proces zit onder zijn working set maximum, dan worden er meer pages toegekend voor dat proces. Zit het proces echter op zijn maximum, dan moeten er en aantal pages worden vervangen met de pages die het process effectief nodig heeft.

Wanneer het aantal beschikbare frames onder de drempelwaarde valt dan gebruikt Windows automatisch *working-set trimming* om het aantal beschikbaar frames boven de drempel te krijgen. Deze techniek werkt door het aantal pages die toegekend zijn aan processen te evalueren. Indien een proces boven zijn minimum limiet zit, dan verwijderd Windows pages tot dat proces op zijn minimum limiet is teruggevallen. De vrijgekomen frames kunnen nu ingezet worden voor processen die ze echt nodig hebben.

-