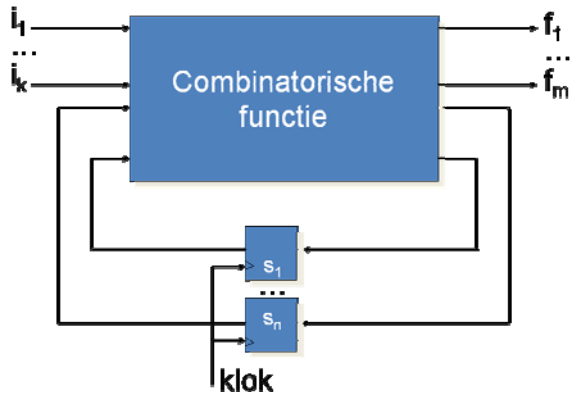
# Sequentiële logica

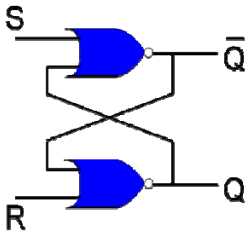
### Wat is sequentiële logica

We noemen dit ook wel final state machine. Het neemt een ingang en een huidige toestand en vertaalt die via een combinatorische functie in een uitgang en een nieuwe toestand.

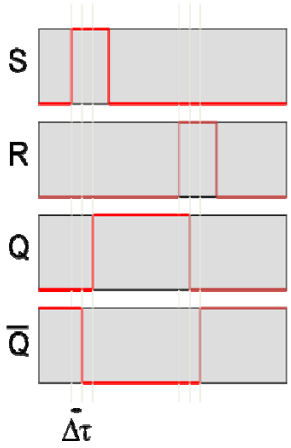



De geschiedenis van ingangen bepaalt mee de toestand en de uitgang.
Dit is het klassieke model van een toestandsautomaat. De combinatorische functie vertaalt de ingangsbits $i_1$ tot $i_k$ samen met de interne toestandsbits $s_1$-$s_n$ in nieuwe toestandsbits, en uitgangsbits $f_1$-$f_n$. De output van de combinatorische functie wordt bewaard door een speciale component en geeft deze op het gepaste ogenblik (klok!) door aan de input van het combinatorisch circuit. Mocht de klok niet aanwezig zijn, wordt de output meteen teruggekoppeld naar de input → ontstaan van een onstabiele situatie.

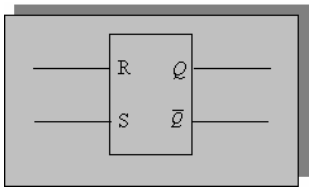
### S-R Latch




| A | B | A+B |
|---|---|---|
| 0 | 0 | 1 |
| 0 | 1 | 0 |
| 1 | 0 | 0 |
| 1 | 1 | 0 |




**Een geheugencel** is een verzameling van logische poorten die een stabiele uitgang kan bewaren zonder dat de ingangen actief moeten zijn. De uitgang van de geheugencel wordt bepaald door de huidige ingangen en de geschiedenis ervan. Een 1 bit geheugencel wordt een **latch** genoemd.
In Q bevindt zich de opgeslagen bit.
*Als S=R=0 dan bewaren de NOR poorten de huidige uitgangen Q en niet-Q.*

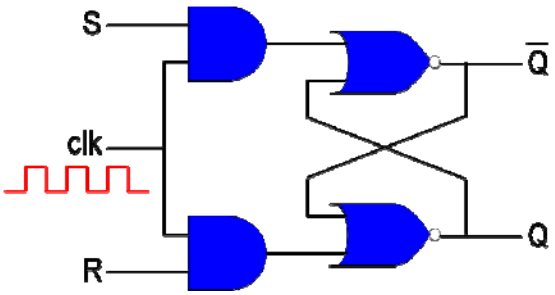



Als Q =R=S=0 en S(et) gaat naar 1, dan zal Q=1 (2 poortvertragingen) en nietQ = 0 (1 poortvertraging). Vanuit de situatie Q=1 S=R=0 kunnen we de oorspronkelijke situatie herstellen door R=1, die langer dan 2 poortvertragingen duurt.

| Ingangen | | Uitgangen | |
|---|---|---|---|
| R | S | Q | Q' |
| 0 | 0 | Willekeurig | |
| 0 | 1 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | Vorige stand | |

| $Q_t$ | S | R | $Q_{t+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | - |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | - |

### Geklokte S-R Latch



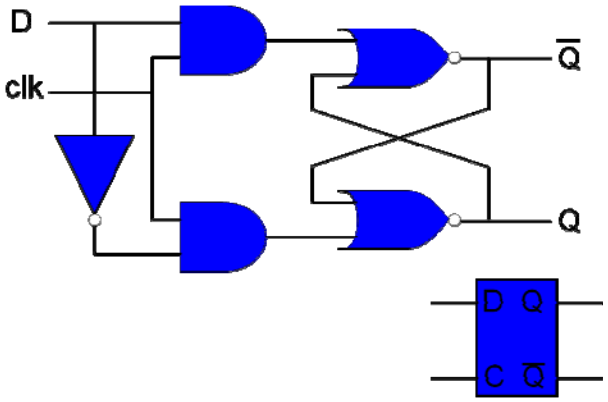

Indien S en R komen uit een ander en ingewikkeld circuit, is het mogelijk dat een aantal ongewenste overgangen gebeuren op die twee vooraleer ze de latch binnentreden. Hiervoor wordt een klok toegevoegd aan de latch, zodat veranderingen aan de bewaarde Q bit enkel gebeuren als S en R stabiel zijn telkens de klok op 1 staat.

De tijd tussen twee opgaande klokflanken = **cyclustijd**. De **kloksnelheid** is de inverse van de cyclustijd.

**D Latch**

Om een 1 of 0 te bewaren in de S-R latch, moet men een 1 aanleggen aan S of R. De D latch zorgt ervoor dat een 1 of een 0 slechts moet aangelegd worden aan 1 enkele input (D=Data). Het is eigenlijk niets meer als een geklokte S-R latch, waarbij S=D en R=niet-D. Wanneer de klok omhoog gaat, wordt de waarde van D bewaard.

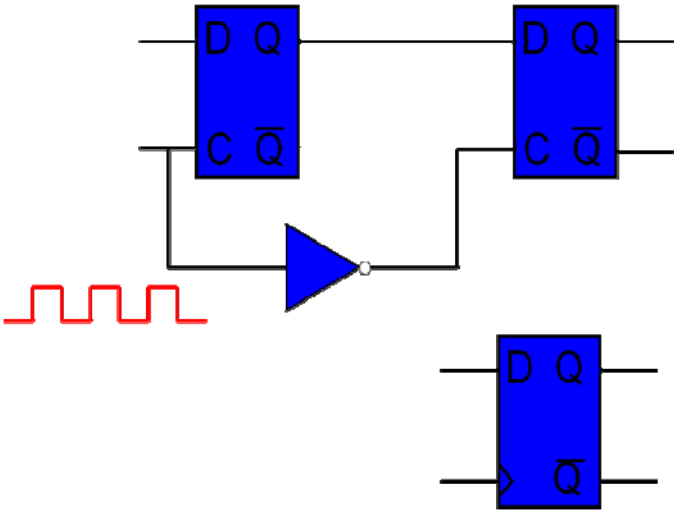
| C | D | $Q$ |
|---|---|---|
| 0 | 0 | $Q_{-1}$ |
| 0 | 1 | $Q_{-1}$ |
| 1 | 0 | 0 |
| 1 | 1 | 1 |




Indien de klok laag staat, blijft de uitgang dus stabiel, en wanneer de klok hoog staat, volgt de uitgang de D (natuurlijk met poortvertraging).

**Master-Slave D-Latch (D Flip Flop)**

Om er zeker van te zijn dat de D latch slechts 1 keer per klokpuls van toestand verandert, gebruiken we geen gewone D latch, maar een **master-slave D-latch of een D flip flop**.



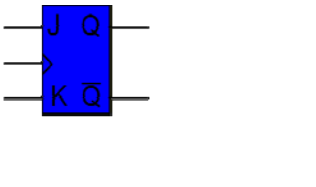

Bestaat uit 2 D latches in cascade, waarbij de tweede de geïnverteerde klok van de eerste gebruikt. De master verandert zijn toestand wanneer de klok hoog is, **de slave wanneer de klok laag** is. De klok moet dus eerst hoog gaan en dan laag vooraleer de waarde wordt opgeslagen in de slave latch.
Het driehoekje hiernaast duidt aan dat transities aan de uitgang enkel gebeuren bij stijgende/dalende klokflanken.
Op de dalende flank van de klok wordt dus de uitgang veranderd!

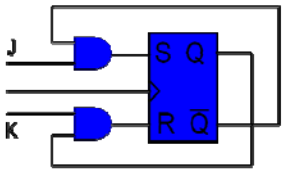
Wel is het zo dat de uitgang niet direct zal worden veranderd, omdat de flipflop wat tijd zal nodig hebben om een stabiele toestand aan te nemen. Men spreekt van de **setup tijd** van de flipflop. Gedurende minstens de setup tijd zal het signaal D stabiel moeten blijven. Sommige flipflops vereisen dat na de dalende flank het signaal D nog even bewaard wordt, we spreken van een **zeer korte!! hold-tijd**.

We spreken van **niveaugestuurde latches** als de toestand wordt gewijzigd wanneer de klok hoog (of laag) is, en van **flankgestuurde flip-flops** die de toestand veranderen tijdens kloktransities.
Een S-R flip flop kan zo ook opgebouwd worden.

**J-K Flip Flop**

J is ongeveer gelijk aan S, en K aan R.

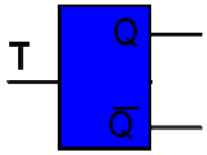


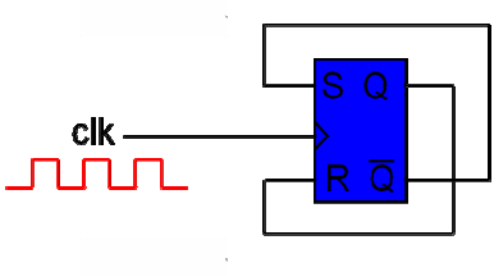

| $Q_t$ | J | K | $Q_{t+1}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 |
| 0 | 1 | 0 | 1 |
| 0 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 1 | 0 | 1 | 0 |
| 1 | 1 | 0 | 1 |
| 1 | 1 | 1 | 0 |

Er is wel een verschil! Wanneer J=K=1, dan slaat de toestand om (**toggle**). Deze flipflop realiseert 4 functies die men kan uitvoeren op 1 bit: omkeren, onveranderd laten, set (op 1 zetten), en reset (op 0 zetten).

**T Flip Flop**

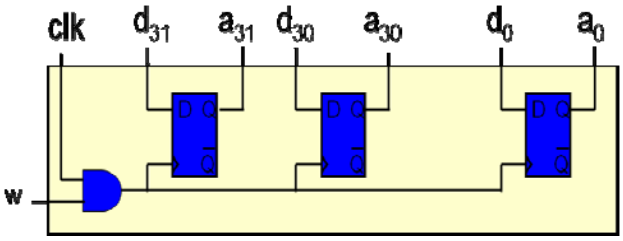



= Een J-K flip flop waarbij J=K=1. Deze flip flop zorgt dus altijd voor een toggle per klokperiode. Het signaal Q zal slechts de halve frequentie hebben van het kloksignaal.

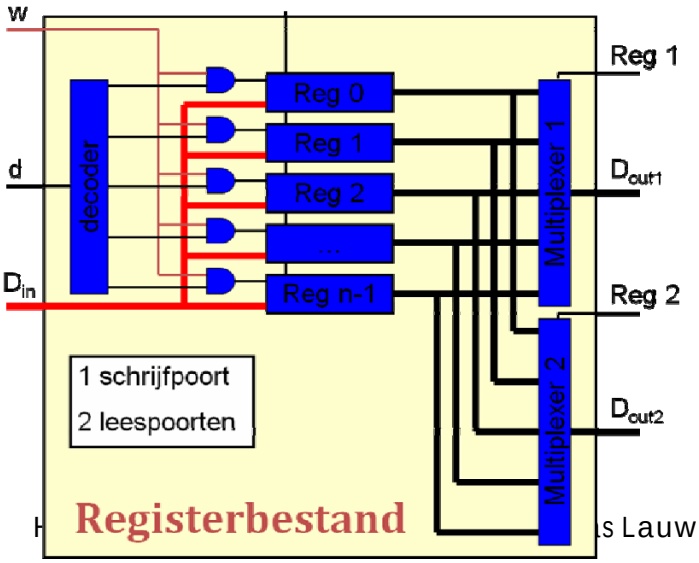



**Registers**

Een register is een zeer snel geheugen ter grootte van n bits. Een register bestaat uit 1 D flip flop per op te slaan bit. Het liet bij het begin van een cyclus een waarde kunnen beschikbaar stellen, en tegelijkertijd op het einde van die cyclus een nieuwe waarde kunnen opslaan. Dus een register kan binnen dezelfde cyclus belezen en beschreven worden.
Alle flip flops delen dezelfde controlesignalen, en het schrijfsignaal is een combinatie van klok en schrijfcommando “w”; indien w=1, dan wordt de waarde op de D-bus ingeschreven in het register bij dalende klokflank.




Een **registerbestand** bestaat uit een lijst van registers. Men spreekt hierbij van schrijf- en leespoorten. (hier: 1 schrijf, 2 lees)
“d” is het nummer van het register dat moet worden beschreven. Welke waarde moet worden beschreven zit in $D_{in}$. De decoder zendt dan signalen naar alle afzonderlijke registers, en samen met het AND-en van het “w”-signaal, zal dan in het gepaste register worden geschreven. (**bij dalende klokflank**)




Leespoorten zijn hier aangegeven door een MUX. Het aantal leespoorten kan men dus gemakkelijk uitgebreid worden, ware het niet dat ze registertoegang sterk vertragen. Ook schrijfpoorten kunnen worden uitgebreid, maar dat is al wat complexer omdat je dan ook MUX’en nodig hebt om het gepaste $D_{in}$ te kiezen.

# Het datapad

### De klok

Geeft het tempo aan waarmee interne en externe controlesignalen worden gegenereerd. Het kloksignaal van de processor wordt in de praktijk afgeleid van het kloksignaal van de **front side bus**. Deze is doorgaans lager dan die van de processor, maar er wordt een frequentievermenigvuldiger opgesmeten. Dit garandeert dat de klok van de processor en de bus aan elkaar gekoppeld blijven.

### De processor

De kern van de processor bestaat uit 2 essentiële onderdelen:
- Datapad met registers & ALU
- Controle-eenheid: regelt de aansturing van de controlepunten in het datapad en zet alle bewerkingen in de processor in gang + controleert alle processen.

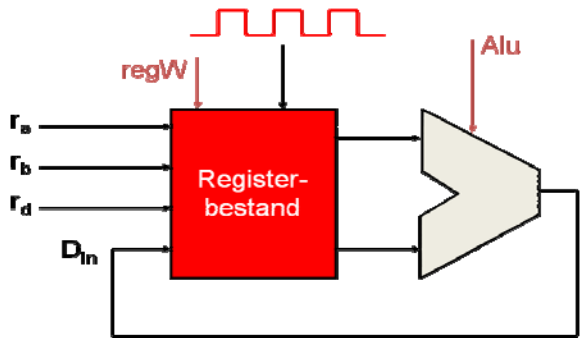
We zullen stap voor stap een datapad opbouwen, en beginnen daarbij bij het inladen van een instructie.

### Inladen van een instructie

Per klokcyclus zal deze schakeling een instructie ophalen uit het geheugen. Op de dalende flank van de klok zal de PC een nieuwe waarde krijgen, welke hij meteen naar het geheugen doorstuurt. Van zodra het geheugen het adres binnenkrijgt, zal het de instructie produceren aan de output die op dit adres opgeslagen ligt.
Simultaan wordt de waarde van PC ook naar een ALU gestuurd die altijd een constante 4 erbij optelt. Het resultaat gaat terug naar PC, waar het bij de volgende dalende klokflank de waarde van de vorige PC zal vervangen.

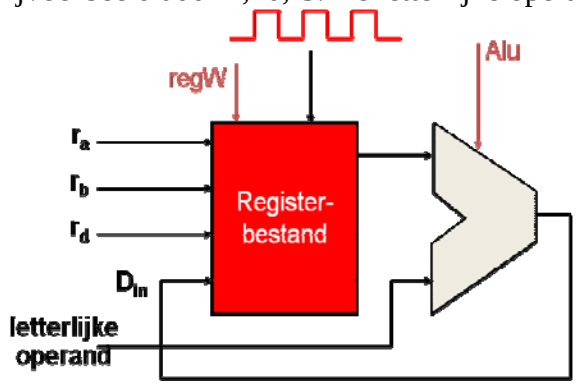
### ALU instructie RRR (2 bronnen, 1 doel uit register)

Uit de instructiebits worden de nummers van de 2 bronregisters en het doelregister van de instructie afgezonderd en naar de twee leespoorten, respectievelijk schrijfpoort van het registerbestand gestuurd. Via de leespoorten stelt het registerbestand dan de twee waarden ter beschikking aan een ALU die het resultaat berekent (bvb add r1,r2,r3). De operatie die moet worden uitgevoerd wordt aan de ALU meegedeeld door een aantal operatiebits.
Het resultaat wordt aan de poort $D_{in}$ van het registerbestand aangelegd. Als regW aanstaat en $D_{in}$ is stabiel , zal bij dalende klokflank het resultaat worden weggeschreven.



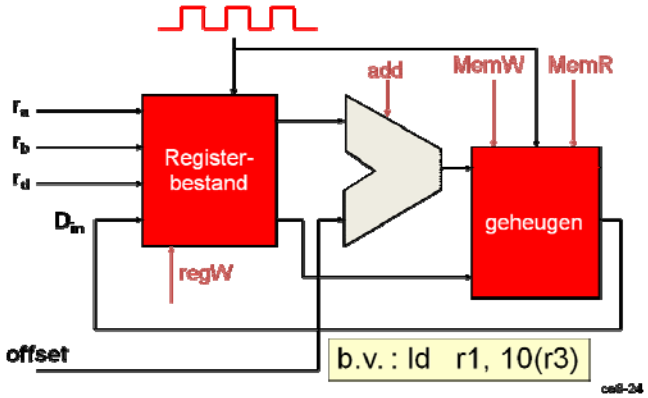
**ALU instructie: RRI (1 letterlijke operand)**

Bijvoorbeeld add r1,10,r3. De letterlijke operand komt uit de instructie zelf.

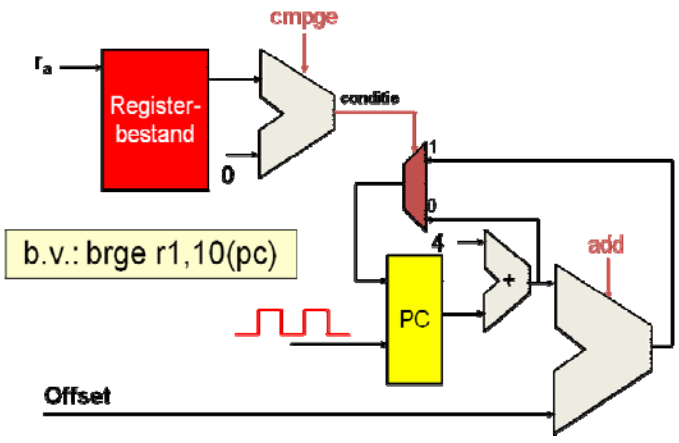



**MEM instructie**

Hier wordt de ALU gebruikt om een adresberekening te maken. Het resultaat ervan wordt dan aangelegd in het datageheugen. Bij het lezen zal MemR actief zijn, en zal het geheugen na een tijd een waarde produceren die aangelegd wordt aan de $D_{in}$ ingang van het registerbestand. Via de tweede leespoort wordt nu ook (simultaan met de adresberekening) de tweede operand ingelezen, en bij het schrijven aan het datageheugen aangelegd. MemW wordt actief, en de waarde wordt weggeschreven bij dalende klokflank.



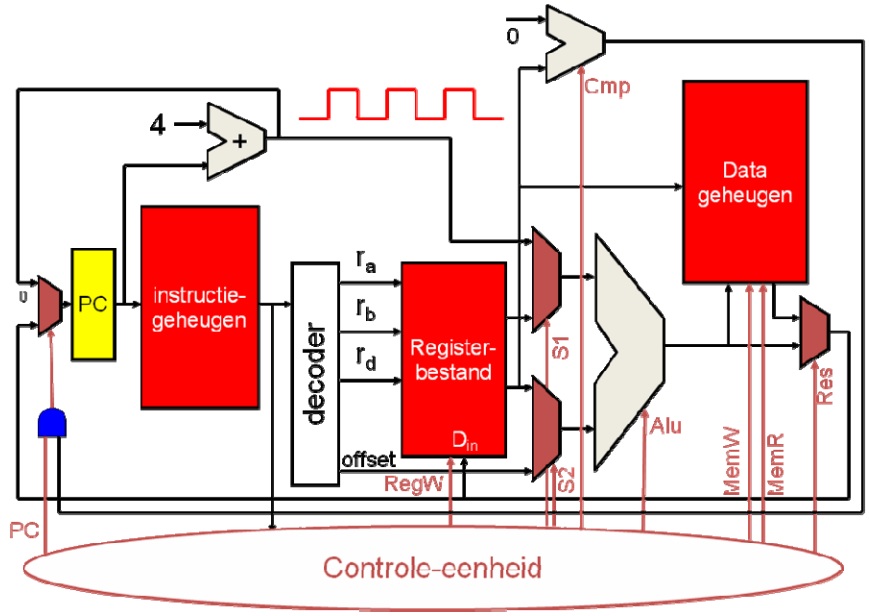

**Controletransfer instructie**




2 ALU operaties: ééntje om de sprongvoorwaarde vast te stellen, en een andere om het sprongadres vast te stellen. Aangezien de nieuwe waarde van PC nu PC+4 of PC+4+offset kan zijn, moet een MUX worden toegevoegd om een keuze te maken. Als er een 1tje staat aan de MUX, wil het zeggen dat dit wordt gekozen als de voorwaarde opgaat.

# Een cyclus per instructie machine

De zonet besproken datapaden kunnen allemaal worden samengevoegd tot 1 groot datapad, dat per cyclus 1 instructie uitvoert. Om dit mogelijk te maken werden er een aantal MUX'en toegevoegd, een decoder om registernummers en constanten uit de instructies te isoleren, en tenslotte ook nog een **controle-eenheid** die de werking van alle ALU's, MUX'en, geheugens en registers controleert. Normaal gezien is 0 de bovenste uitgang bij een MUX.




Hieronder vind je de controletabel om de verschillende controlepunten aan te sturen. Deze tabel kan gemakkelijk worden geïmplementeerd aan de hand van een geheugen met baucellen van (hier althans) 13 bits.

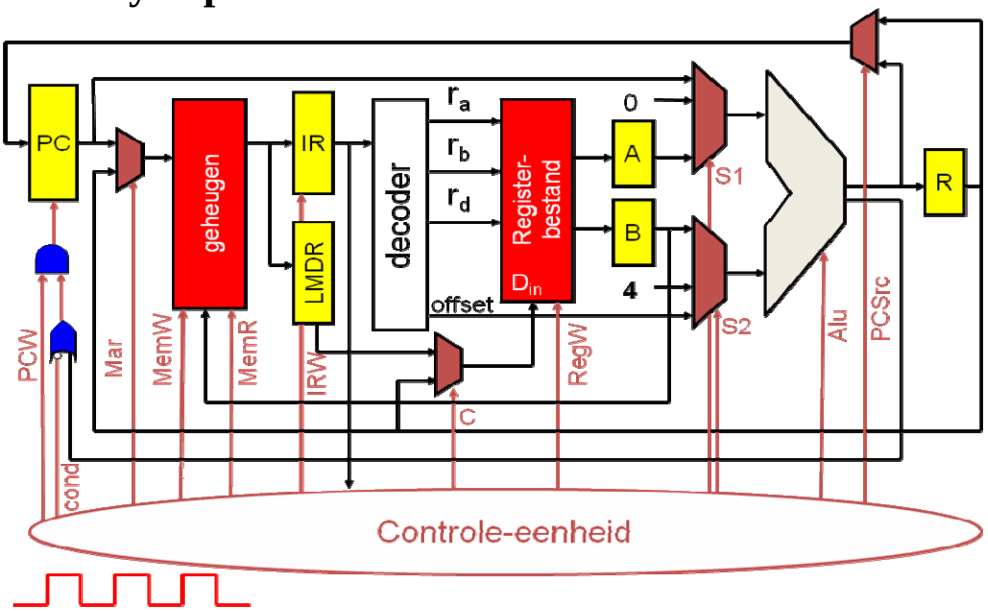
| | PC | RegW | S1 | S2 | Alu | Cmp | MemW | MemR | Res |
|---|---|---|---|---|---|---|---|---|---|
| add | 0 | 1 | 1 | 0 | 001 | xxx | 0 | 0 | 1 |
| addi | 0 | 1 | 1 | 1 | 001 | xxx | 0 | 0 | 1 |
| load | 0 | 1 | 1 | 1 | 001 | xxx | 0 | 1 | 0 |
| store | 0 | 0 | 1 | 1 | 001 | xxx | 1 | 0 | x |
| jump | 1 | 0 | 0 | 1 | 001 | 111 | 0 | 0 | 1 |
| brge | 1 | 0 | 0 | 1 | 001 | 110 | 0 | 0 | 1 |

| | |
|---|---|
| Add 000 | 0110001xxx0 01 |
| Addi 001 | 0111001xxx001 |
| Load 010 | 0111001xxx010 |
| Store 011 | 0011001xxx10x |
| Jump 100 | 1001001111001 |
| Brge 101 | 1001001110001 |

2 grote nadelen:

- Sommige onderdelen komen gedupliceerd voor (ALU)
- Alle instructies voeren even snel/traag uit, want de klokperiode kan niet kleiner gemaakt worden dan de tijd die de traagste instructie nodig heeft om uit te voeren.

# Meer cycli per instructie machine




Alle onderdelen komen hier nu slechts éénmaal voor, dus sommige zullen meermaals moeten gebruikt worden tijdens uitvoering van 1 instructie.

De eerste stap tijdens de uitvoering is identiek voor alle instructies. Tijdens de dalende flank van de klok wordt PC aangelegd aan het geheugen die een instructie ophaalt en doorspeelt aan zowel IR (instructieregister) als aan LMDR (load memory data register). Hierbij wordt ook PC doorgespeeld naar de ALU en telt er 4 bij op, en stuurt resultaat terug naar PC.
Bij de volgende dalende klokflank zal alles zijn uitgevoerd
**Actieve signalen**: MemR, IRW, S2, Alu=add, PCSrc, PCW
**Niet actieve signalen**: Cond, S1, Mar, C, RegW

De tweede stap is ook identiek. De instructie die is ingeladen, wordt nu gecodeerd door de instructiedecoder, die de registernummers en constanten in de instructie eruit haalt, en doorspeelt naar het registerbestand (registernummers, welke waarden leveren aan A en B) en controle-eenheid (controle-informatie ophalen nodig voor de uitvoering van de instructie). De gedecodeerde offset, samen met de aangepaste instructie wordt doorgegeven aan de Alu om te vermijden dat hij niets zou doen. We weten nog niet of er gesprongen dient te worden, dus deze berekening is louter speculatief. Het resultaat wordt aangeboden aan register R.
Bij volgende dalende klokflank zijn al deze operaties uitgevoerd, en waarden in A, B en R opgeslagen. Deze registers zullen immers reageren bij elke klokcyclus, PC en IR doen dit enkel wanneer hun controle-ingang hoog is.
**Actieve signalen**: S1=1, S2=2, Alu=add

**VERVOLG ADD/ADDI**
Add Derde Cyclus
Registers A en B worden naar de Alu gestuurd, opgeteld, en in R gestoken.
**Actieve signalen**: S1=2, S2=0, Alu=add

Addi Derde Cyclus
Enkel register A + offset naar Alu, resultaat in R.
**Actieve signalen**: S1=2, S2=2, Alu=add

Add/Addi Vierde Cyclus
Via de gepaste controlesignalen wordt de inhoud van R doorgestuurd naar de $D_{in}$-ingang van het registerbestand. Het signaal $r_d$ bepaalt dan waar de waarde moet worden opgeslagen. Dit stond al een hele tijd klaar, maar wordt nu pas gebruikt doordat RegW=1. Zoals eerder gezegd zijn ook $r_a$ en $r_b$ nog actief, waardoor A en B steeds dezelfde waarde krijgen.
**Actieve signalen**: RegW=1,C=1

**VERVOLG LOAD/STORE**
Load/Store Derde Cyclus
Berekening van de adresseermode, hier de som van register+hard gecodeerde offset. Resultaat komt in R.
**Actieve signalen**: S1=2, S2=2, Alu=add

Store Vierde Cyclus
Inhoud van B (weg te schrijven waarde) wordt naar ingang van het geheugen gebracht. Via Mar=1 wordt het effectief adres aan de i,gang van het geheugen gelegd.
Bij dalende klokflank zorgt MemW = 1 dat de waarde wordt weggeschreven in het geheugen.
**Actieve signalen**: Mar=1, MemW=1

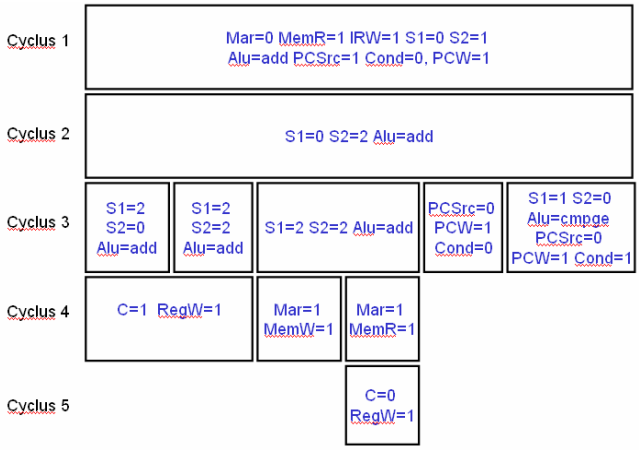
Load Vierde Cyclus
Analoog, maar er wordt niet geschreven, maar gelezen en het resultaat komt in LDMR terecht. Omdat IRW=0 komt het niet in IR terecht! Anders zou er info worden overschreven.
**Actieve signalen**: MemR=1, Mar=1

Load Vijfde Cyclus
Gelezen waarde wordt weggeschreven in een register.
**Actieve signalen**: RegW=1, C=0

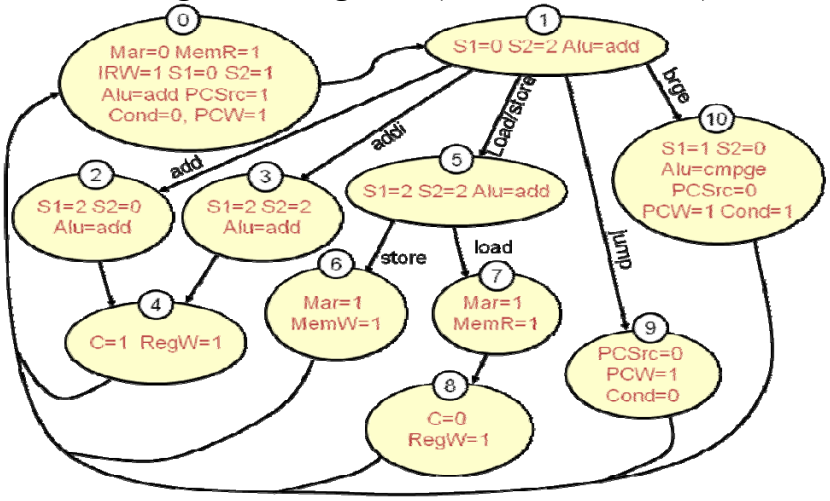
**VERVOLG JUMP/BRGE**
Jump Derde Cyclus
Het enige wat men hier hoeft te doen is de berekende speculatieve waarde doorsturen naar PC.
**Actieve signalen**: PCScr=0, PCW=1, Cond=0

Brge Derde Cyclus
Alu wordt gebruikt om te vergelijken met 0, en de resulterende conditiecode wordt gebruikt om het schrijfsignaal van PC te sturen. Cond wordt normaal op 0 gelaten, tot het verandert op 1, en pas dan zal de AND poort het schrijfsignaal naar de PC sturen.
**Actieve signalen**: PCSrc = 0, PCW=1, Cond=1, Alu=cmpge, S1=1, S2=0

**Samenvatting controlesignalen**



**Samenvatting controlesignalen (toestandsautomaat)**




De labels op de pijlen geven aan welke voorwaarden er moeten voldaan zijn om een bepaalde overgang te maken.

Naast de controlesignalen per toestand moet er ook informatie worden bijgehouden over het verloop van de controle doorheen de automaat: de overgangen tussen de verschillende toestanden. Onderstaande tabel geeft aan op welke voorwaarde er van één toestand naar een volgende kan gesprongen worden. In 8 bits zal dus begintoestand en operatie kunnen worden gecodeerd, indien we ervan uitgaan dat de operaties ook in 4 bits kunnen worden gecodeerd. Uitgaande van deze 8 bits kan men dan ondubbelzinnig de volgende toestand en alle controlesignalen afleiden. De generatie van al deze signalen gebeurt aan de hand van een tabel.

| toestand | voorwaarde | volgende toestand |
|---|---|---|
| 0 0 0 0 | | 0 0 0 1 |
| 0 0 0 1 | op = add | 0 0 1 0 |
| 0 0 0 1 | op = addi | 0 0 1 1 |
| 0 0 0 1 | op = load/store | 0 1 0 1 |
| 0 0 0 1 | op = jump | 1 0 0 1 |
| 0 0 0 1 | op = brge | 1 0 1 0 |
| 0 0 1 0 | | 0 1 0 0 |
| 0 0 1 1 | | 0 1 0 0 |
| 0 1 0 1 | op = store | 0 1 1 0 |
| 0 1 0 1 | op = load | 0 1 1 1 |
| 0 1 0 0 | | 0 0 0 0 |
| 0 1 1 1 | | 1 0 0 0 |
| 1 0 0 0 | | 0 0 0 0 |
| 1 0 0 1 | | 0 0 0 0 |
| 1 0 1 0 | | 0 0 0 0 |

**ROM implementatie**

Indien de tabel in Read-Only Memory wordt opgeslagen, zijn er lichte wijzigingen. De controlesignalen hangen enkel af van de toestand, niet van de operaties, dus beschouwen we ze beter afzonderlijk. Er zijn dan 16 mogelijke toestanden (11 daadwerkelijk gebruikt), en in elke toestand moeten 16 signalen worden gegenereerd → 256 mogelijke gevallen, die elk een nieuwe toestand van 4 bits moeten genereren.
Alle don't cares in bvb ROM1 worden vervangen door 0'en. De lijnen 12-16 worden nagenoeg niet gebruikt! (dus allemaal 0)

# Microcodering

Indien we gebruik maken van een sequencer, proberen we ervoor te zorgen dat daar, waar er geen keuzen worden gemaakt, de toestanden sequentieel oplopen. Het volstaat dan om een opteller met 1 te definiëren om te gaan naar de volgende toestand. Twee gevallen waar dit niet mogelijk is: **terugspringen naar 0 en waarbij er meer dan 1 opvolger is**.

→ Het terugspringen naar 0 kan worden opgelost door een reset-signaal te genereren.
→ Meer dan 1 opvolger: gebruik maken van sprongtabellen die uitgaande van de operatiebits de nieuwe toestand gaan bepalen.

De keuze tussen terugspringen naar 0, kiezen uit meer dan 1 opvolger, en de gewone sequentie moet worden gemaakt door een MUX aan het begin van de ROM. Hiervoor dienen echter 2 nieuwe controlebits worden gezonden naar de MUX, welke de aard van de operatie aanduiden. Hiervoor is het nodig dat ADD en ADDI worden opgesplitst.

Indien we nu niet willen dat ADD en ADDI worden opgesplitst, en indien we niet willen gebonden zijn aan de sequentiële nummering van de toestanden, dan kunnen we de **volgende toestand ook opnemen in de controlesignalen**.
In plaats van de toestand nu op 0 te dwingen, kunnen we hem eender welke waarde geven, welke gecodeerd staat in ROM1. **(escape-simulator model!)**
De ROM1 tabel wordt dan het microcodeprogramma genoemd, en de toestandsveranderlijke de microcodePC. De controle-eenheid voert dan het microcodeprogramma uit.

# RISC vs. CISC

### Evolutie van de instructiesets

Eerst was er in de jaren 70 de CISC: complex instruction set computer, welke werkte met complexe microgecodeerde instructiesets. Vervolgens in de jaren 80 de RISC: reduced instruction set computer.
De RISC herkent men aan volgende kenmerken:

- Voornamelijk “echt nodige”, effectief gebruikte, eenvoudig decodeerbare instructies
- Groot aantal registers, RRR machinemodel
- Geen complexe adresseermodes

### Instructiegebruik & Complexiteit

Een groot deel van de uitgevoerde instructies zijn héél eenvoudige instructies, waardoor de complexe CISC dus vervangen werd door een gereduceerde instructieset computer, de RISC. Ook is voor de meeste instructies de complexiteit niet erg groot; voor toewijzingen is 80% van de eenvoudigste vorm met 2 parameters: X=Y;

**Strafverhouding**

= een maat voor hoeveel beter een assemblerprogrammeur een bepaald probleem kan oplossen. In de CISC tijd was deze verhouding zeer gunstig, want:

- Optimaliserende compilers waren een uitzondering
- Programmeurs konden met succes gebruik maken van de particulariteiten van de processor, terwijl de compiler dit niet kon.

Bij de RISC viel het tweede weg, en door de regelmatige structuur in de instructieset, kon ook de compiler verder gaan optimaliseren. Het loont dus tegenwoordig niet langer om in assembler te programmeren, uitgezonderd de binnenste kern van spelletjes bijvoorbeeld, waar iedere extra cyclus telt.
De strafverhouding = **uitvoeringstijd machinetaal / uitvoeringstijd hoge-niveautaal**

**SPARC registervensters**

SPARC werkt met een zéér groot aantal registers. De processors met veel registers zorgen ervoor dat er bvb maar 32 per keer simultaan zichtbaar zijn, omdat veel registers niet echt goed is voor systeemsoftware, die bij een onderbreking deze in een register moet bewaren.

SPARC werkt met register**vensters** van 32 registers per keer:

- 8 globale
- 8 lokale
- 8 input (parameters voor de huidige functie)
- 8 output (parameterdoorgave)

Per procedureoproep worden 16 registers bewaard (lokale), en komen er 16 andere in de plaats (output). Tegelijk worden de outputregisters van niveau n, inputregisters van niveau n+1.