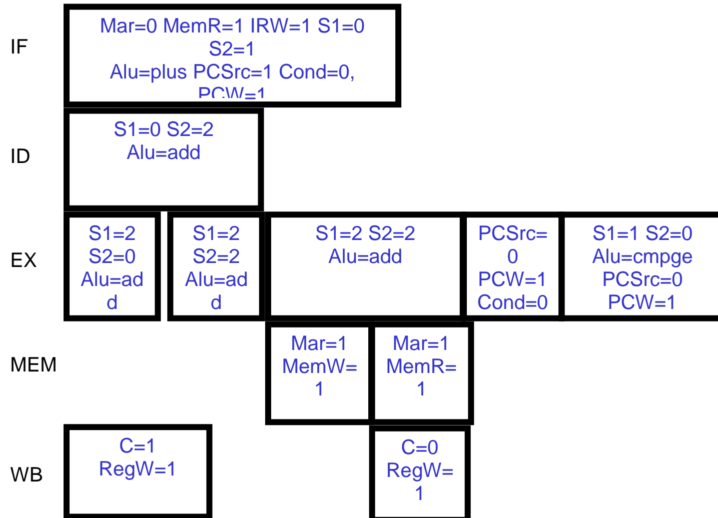
Sequentiële gepijplijnde machine

Samenvatting controlesignalen

Als we de controlesignalen van vorig hoofdstuk nemen, kunnen we per cyclus een naam geven aan de stap. We plaatsen hiervoor wel de registerbeschrijving RegW van cyclus4 in cyclus 5. M.a.w. de WB voor ALU instructies wordt slechts uitgevoerd in de vijfde cyclus!

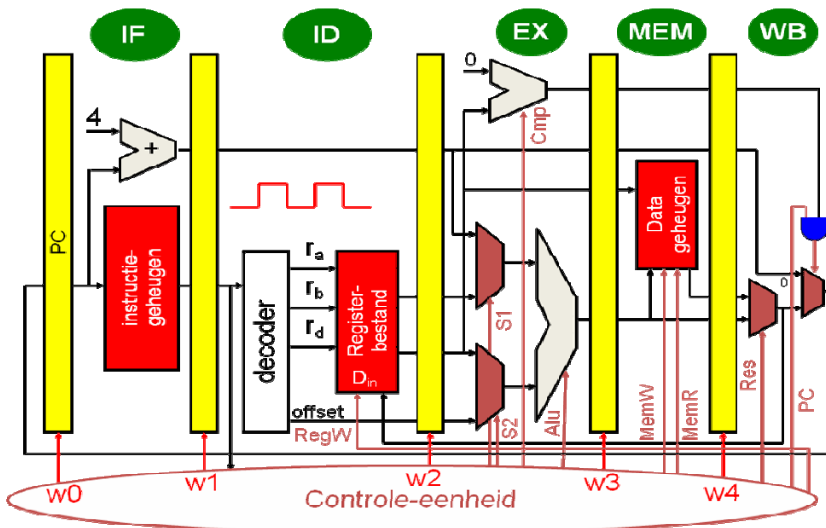


IF = Instruction Fetch, haalt de instructie op en verhoogt PC
ID = Instruction Decode, decodeert de instructie, haalt registerwaarden op en berekent speculatief het sprongadres.
EX = Execute, voert de semantiek van de instructie uit (betrekking van de ALU)
MEM = Memory, interactie met het datageheugen
WB = Write Back, resultaten wegschrijven in de gepaste registers

Indien we bij de meer-cycli-per-instructiemachine er willen voor zorgen dat iedere instructie in 5 trappen wordt uitgevoerd, zullen we er voor moeten zorgen dat:

- bij overgang van 1 cyclus naar de volgende, alle waarden die een overgang moeten overleven, ergens tijdelijk kunnen worden opgeslagen in een register (A,B,LDMR,IR,PC,R)
- moeten we extra registers bijplaatsen op het eind van elke trap.

Omdat we weten dat de bestemming van een sprong pas bekend is na de EX trap, of zelfs na de MEM trap (indien geheugenindirectie), wordt de berekening van de PC volledig verhuisd naar de WB trap!



Deze opstelling wordt een **pijplijn** genoemd. De instructies vertrekken aan de linkerzijde in de pijplijn, en verplaatsen zich naar rechts in een vast tempo. De registers tussen 2 pijplijntrappen, worden **pijplijnregisters** genoemd, aangestuurd door extra controlesignalen w0-w4, welke zullen bepalen wanneer een pijplijnregister waarden zal opnemen en doorgeven aan de volgende trap. Dus er wordt een 1tje gezonden naar w1 in de IF trap, w2 in de ID trap, w3 in de EX trap, w4 in de MEM trap, en w0 in de WB trap.

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

Aan de controletabel verandert ook iets! Het is nu niet meer nodig om de signalen gedurende de ganse uitvoering actief te houden. Er is toch maar 1 trap per keer actief, en tijdens die trap moeten de controlesignalen die bij die trap horen, geactiveerd worden.

Het **RegW** signaal wordt pas aangestuurd in de WB trap, ofschoon het een blok in de ID trap aanstuurt! Dit komt omdat het registerbestand 2 maal wordt gebruikt: éénmaal in de ID, en éénmaal in de WB trap.
De **IF en ID** trap staan hier niet in vermeld, omdat deze toch onafhankelijk van de instructie verlopen, en de controle kan dus vast zijn.

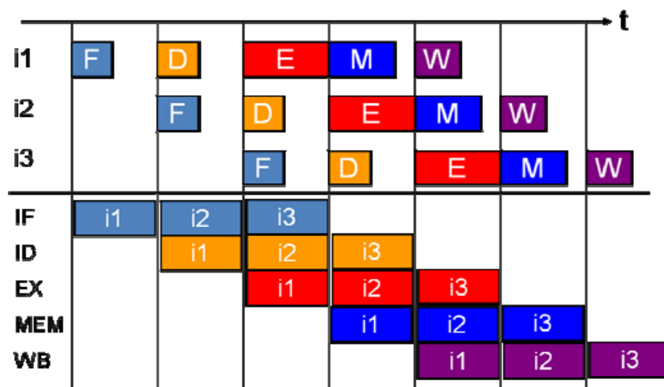
We kunnen de sturing van de controlesignalen ongelooflijk vereenvoudigen, indien we de signalen laten genereren in de ID trap, en de signalen via de pijplijnregisters doorgeven naar de juiste locatie. Merk op dat we het **RegW** signaal artificieel vertragen tot in de WB trap, om dan terug te sturen naar het registerbestand. We zullen hierbij ook het r_d signaal vertragen van aan de decoder tot in de WB trap, zodat het nergens moet worden bewaard tot aan de WB trap.

Voor het volgen van de verschillende ALU instructies, zie slide 9-15

De langst durende trap bij een sequentiële pijplijncyclus zal bepalen hoe groot een klokperiode mimimaal moet zijn. Een **volgende instructie** wordt pas aangevat **wanneer de vorige instructie volledig is verwerkt**. Indien we naar de individuele trappen van de pijplijn kijken, dan zien we dat een trap gemiddeld gezien maar om de 5 cycli eens gebruikt wordt, en ongebruikt blijft tijdens de overige cycli.

Parallelle gepijplijnde machine

We kunnen vermijden dat de pijplijntrappen ongebruikt blijven tijdens die overige cycli door de verschillende instructies na elkaar te verwerken. Dit werkt dankzij het feit dat in een pijplijn slechts 1 trap per keer actief is. Bovendien werkt dit enkel voor instructies die geen controletransfer instructies zijn, want de nieuwe pc wordt pas in EX (of MEM) bekend. Dit kan worden opgelost door **4 NOP instructies** te plaatsen na de controletransfer. (zie later)



Kenmerken van parallelle pijplijncyclus:

- **PC** + **4** wordt rechtstreeks doorgestuurd naar de WB trap, zodat in een volgende cyclus onmiddellijk een nieuwe instructie kan worden opgehaald.
- Hier bespreken we slechts een model met 5 trappen, maar er kunnen er veel meer zijn. We kunnen door deeltrappen te beschouwen, de maximale uitvoeringstijd per trap verder reduceren, en dus de klokfrequentie op te drijven.
- De instructies voeren **niet sneller uit**! Per instructie blijft de uitvoeringstijd 5 cycli! Men spreekt van **CPI = clocks per instruction** en **IPC = instructions per clockcycle**

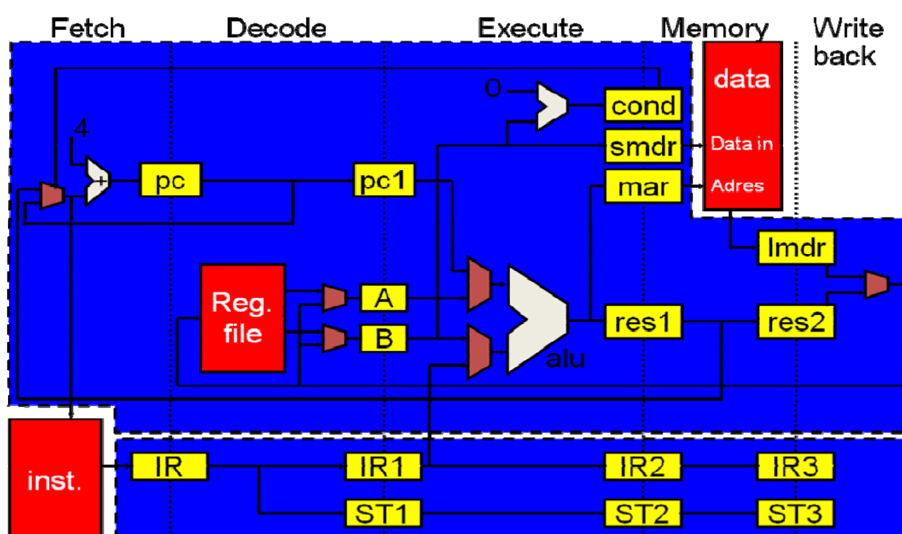
Aanpassingen:

- Eigenlijk maakt de berekening van de nieuwe pc enkel gebruik van informatie die in de EX trap wordt gegenereerd. De selectie van de PC kan nu worden verplaatst naar de EX trap, en meteen worden opgeslagen in het PC register. Nu moeten we geen 5 cycli meer wachten op de PC, en zijn er maar **2 NOP's** meer nodig bij controletransfers.

Het model van de Escape Simulator

Fetch: instructie ophalen & PC berekenen.
Decode: registernummers uit instructie gehaald, en juiste registers bepaald.
Execute: sprongadres / adresseermode / resultaat van de ALU operatie berekenen. Sprongadressen worden dus na 3 cycli al teruggevoerd naar PC.
Memory: interactie met het datageheugen
Writeback: resultaat naar register geschreven. Dit resultaat kan tegelijk in het registerbestand, en in registers A en B worden geschreven → registers kunnen in dezelfde cyclus een waarde opnemen, en in diezelfde cyclus (aan het eind) diezelfde waarde beschikbaar stellen. Dit is mogelijk door de klok van het registerbestand wat te vervroegen, zodat de op te slane waarde opgeslagen wordt en al zichtbaar wordt aan de uitgang, net voor het invullen van A en B. (cfr. de extra MUX)

Het bovenste blauwe vierkant is het datapad, het onderste het controlepad.



Hazards

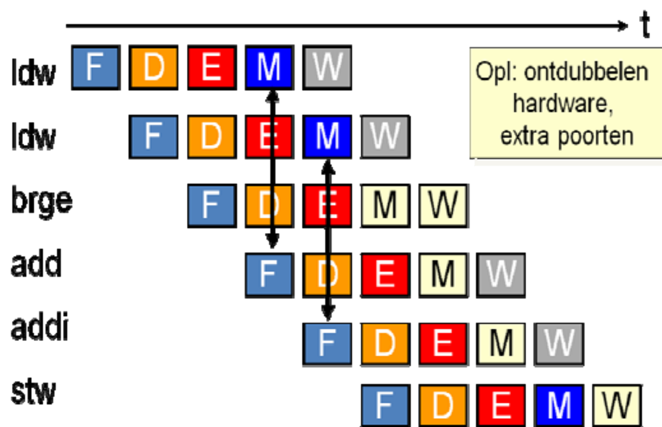
Een hazard is een afwijking van de standaard sequentiële interpretatie van de programmacode. Ze worden veroorzaakt door de overlappende uitvoering van instructies die na elkaar komen.

Structurele hazards

<u>Oorzaak</u>

- Schaarse hardware: bijvoorbeeld wanneer registerbestand niet simultaan zou kunnen lezen en schrijven, dan ontstaat een structurele hazard tussen WB en ID.
- Voornamelijk een probleem in micro architecturen waar niet alle EX trappen even lang duren, en parallel kunnen uitvoeren. Hierbij is het dus niet statisch te voorspellen wanneer een bepaalde trap een bepaald onderdeel van de processor zal nodig hebben.

<u>Situatie</u>



Hier wordt natuurlijk gesteld dat er slechts 1 geheugen is, en dat het geheugen maar 1 aanvraag per keer kan bedienen.

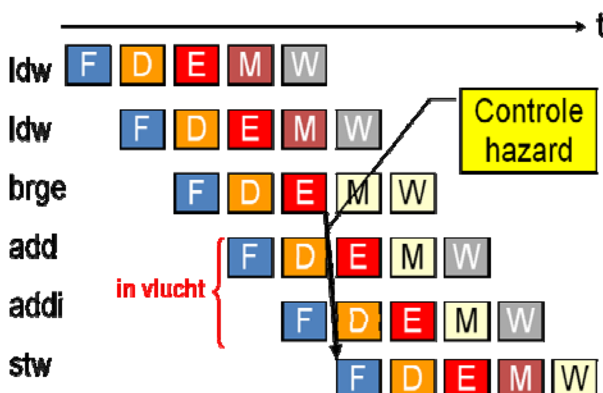
<u>Oplossing</u>
Onderdelen van de processor ontdubbelen, zodat 2 simultane aanvragen wel gewoon door kunnen gaan.

Controlehazards

<u>Oorzaak:</u>

- Vertraging bij berekenen van een sprong

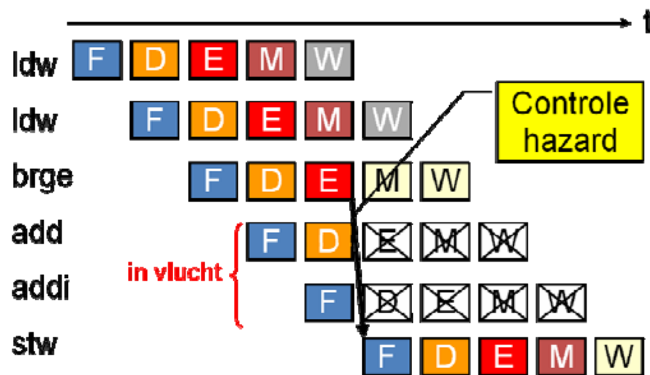
<u>Situatie</u>



Eigenlijk mogen add en addi hier niet worden uitgevoerd, maar dat weet de processor niet, want pas op het einde van de EX trap van het brge commando weet men pas dat er degelijk moet worden gesprongen. Instructies die aan hun uitvoering begonnen zijn noemen we **in vlucht**.

Oplossingen

- **Vervangen door NOP**: instructies die onterecht in uitvoering zijn, kunnen worden vervangen door NOP instructies, ze worden m.a.w. omgezet in **pijplijnbellen**. Dit is net op tijd vooraleer een register of geheugen wordt veranderd van toestand. In de Fetch en Decode trap gebeuren geen wijzigingen van de processortoestand, dus deze kunnen blijven bestaan. Het creëren ervan is eenvoudig. Het signaal welke aangeeft dat er moet gesprongen worden, zal de Fetch en Decode trap van de instructie in de **pijplijnregisters** annuleren door ze bvb op 0 te zetten. De reeds ingelezen en gedecodeerde instructies, zullen dus worden vervangen door NOP's, die ongestuurd verder door de pijplijn kunnen gaan.



- **Vertraagde controletransfer/sprong**: het probleem wordt verschoven naar de programmeur door aan te duiden dat 1 of 2 instructies na de sprong wel degelijk worden uitgevoerd, zodat men in de processor geen zorgen moet maken of controlehazards. De programmeur zal dus deze **instructieslots / delay slots** moeten invullen met 2 bewerkingen die sowieso mogen uitgevoerd worden. Als men geen gepaste bewerkingen vindt, kan men nog altijd NOP invoegen.

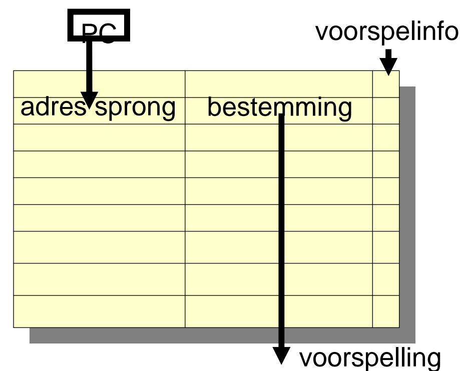
  We kunnen dit op een aantal manieren doen.
  Bij een **onvoorwaardelijke sprong** kun je de instructies die net voor de spronginstructie komen, of van op de bestemming van de sprong erin plaatsen.
  Bij een **voorwaardelijke sprong** mag je meestal geen instructies nemen van net voor de sprong, want ze helpen meestal mee de sprongconditie vast te leggen. Op de sprongbestemming dienen de instructies ook slechts ENKEL uitgevoerd te worden ALS de sprong wordt genomen. De SPARC architectuur biedt een oplossing: men schrijft een instructie van de bestemming in het instructieslot, maar bij de vertraagde sprong (bvb **breq next**) zet men een annulatiebit **breq, a next** zodat men een cyclus wint indien de sprong genomen wordt!

  Men poogt omwille van het laag houden van het aantal delay slots om de selectie van de volgende PC reeds in de ID-trap te laten gebeuren. We moeten in de architectuur dan op een aantal zaken letten wanneer we de twee ALU's betrokken bij berekenen van de nieuwe PC willen verplaatsen naar de ID stap:

  1. De optellingsALU geeft geen probleem, daar de offset reeds kort na het begin van de ID cyclus bekend is, en de PC bij het begin.
  2. De comparator is iets moeilijker omdat hij moet wachten op het resultaat van het registerbestand (B). Comparator kan alleen worden verplaatst naar de ID als de vergelijking eenvoudig is, en dus de waarde nog op tijd binnen dezelfde cyclus naar de MUX te sturen voor de PC.

- Soms lukt het dus niet om de **comparator** naar de ID trap te verplaatsen, en dan maakt men gebruik van **sprongvoorspellers**.

  Een sprongvoorspeller zal proberen, uitgaande van het voorbije gedrag van een programma, te voorspellen wat de volgende in te laden instructie zal zijn (op een precisie van 95%). Ze zal in de IF fase voorspellen of een sprong al dan niet genomen wordt, en het adres zal worden berekend in de ID trap. In sommige gevallen levert ons dit geen tijdswinst bij een vijftrapspijplijn, want de uitkomst kan soms worden berekend in 2 trappen. Bij een complexe vergelijking, levert ons dit wel winst op, omdat de sprong anders slechts in de 5de trap is bekend.
  Het berekenen van een adres kan ook in de IF trap. Hiervoor maken we gebruik van een **Branch Target Buffer**. Hiervoor dienen we naast info over het al dan niet nemen van een sprong (zie later) ook info bijhouden over de bestemming van een sprong. Dit neemt wel heel wat meer plaats in beslag per entry dan een klassieke voorspeller.



1. **Statische sprongvoorspellers**

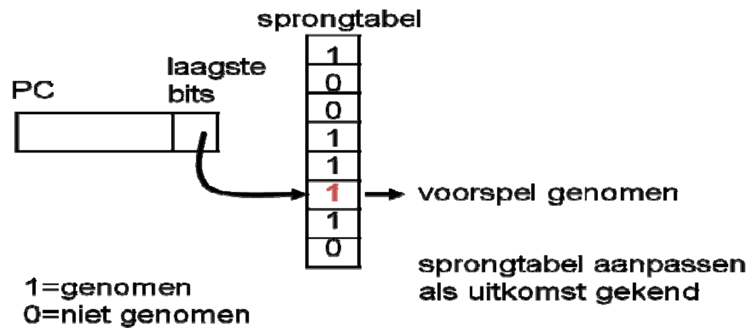
   1/ Voorspel niet-genomen
   Een voorwaardelijke sprong wordt in een lus meestal in het begin geplaatst zodat de voorspelling in de meeste gevallen juist is, want de instructie na een voorwaardelijke sprong wordt meestal altijd ingeladen, en de sprong wordt dus voorspeld als niet genomen.
   Een onvoorwaardelijke sprong wordt als genomen voorspeld.
   2/ Branch backward taken/ forward not taken
   Voorspelling wordt hier gemaakt in de richting waarin de sprong gaat. In dit geval wordt de voorwaardelijke sprong beter achteraan geplaatst in de lus.

2. **Dynamische sprongvoorspellers**

   1/ Eenvoudige dynamische voorspeller
   Maakt een voorspelling op basis van de inhoud van een sprongtabel. De tabel houdt hier bij of de sprong op een gegeven adres de vorige keer al dan niet genomen werd. Bij elke foute voorspelling wordt de waarde in de tabel aangepast.

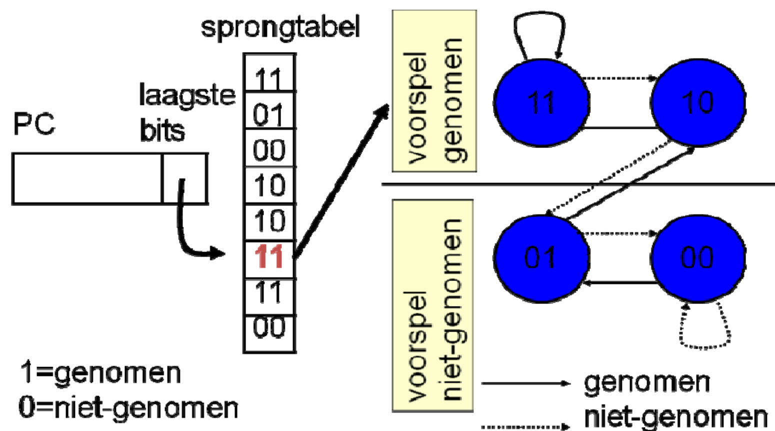
Probleempjes:

- Bij het verlaten van de lus, en bij het eerste keer heruitvoeren van de lus zal er verkeerd voorspeld worden. Indien de lus 10 keer wordt genomen, wordt dus 20% verkeerd voorspeld.
- De sprongtabel is véél kleiner dan het aantal adressen → waarden worden overschreven = **aliasing of verwarring**



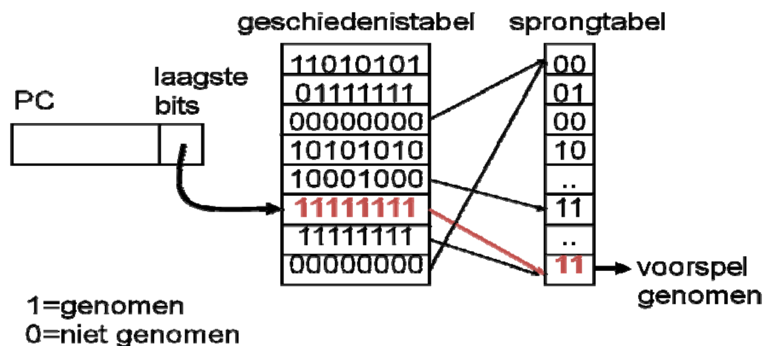
2/ 2-bit dynamische voorspeller

Er worden 2 bits bijgehouden in de sprongtabel. We gebruiken hier een **saturerende 2 bit opteller**.

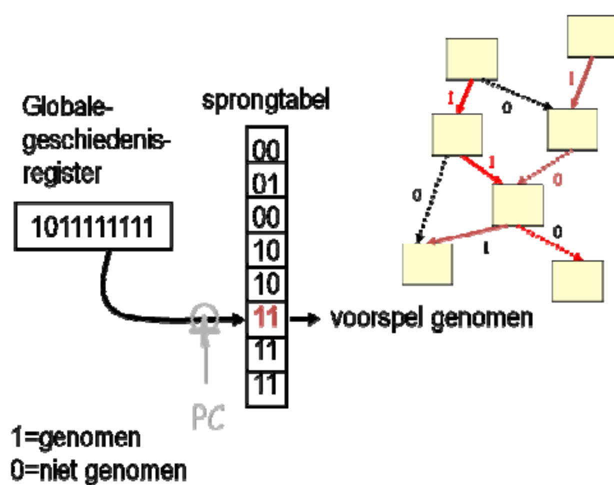


3/ Lokale dynamische voorspeller

We gaan nu de sprong voorspellen aan de hand van de geschiedenis van de sprongen. (hier bvb 8 bits) Per uitgevoerde sprong wordt in de geschiedenistabel een 1 ingeschoven. Op basis van dit patroon wordt er dan gekeken in de sprongtabel wat er bij dergelijk patroon gebeuren moet.



4/ Globale dynamische voorspeller



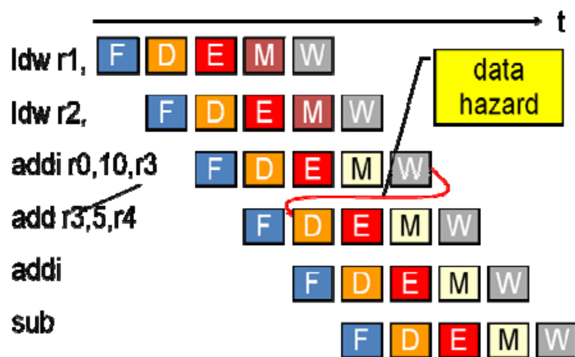
Op dezelfde manier kan er ook vertrokken worden van een globale geschiedenis zoals de processor ze aan zich ziet voltrekken. Per sprong die wordt ontmoet, wordt een bit toegevoegd aan het globale geschiedenisregister, welke dan wordt gebruikt om een waarde uit de sprongtabel aan te duiden. **Probleem:** om te vermijden dat twee globale geschiedenissen (afkomstig van 2 verschillende plaatsen in het programma) dezelfde voorspelling moeten opleveren, zullen we ze XOR'en met een aantal bits van PC, en zo geraken de indices in de sprongtabel ook meer verstrooid.

Datahazards

Oorzaak:

- Resultaten worden pas in de vijfde cyclus bewaard, maar er zijn soms instructies die die resultaten al vroeger nodig hebben

Situatie

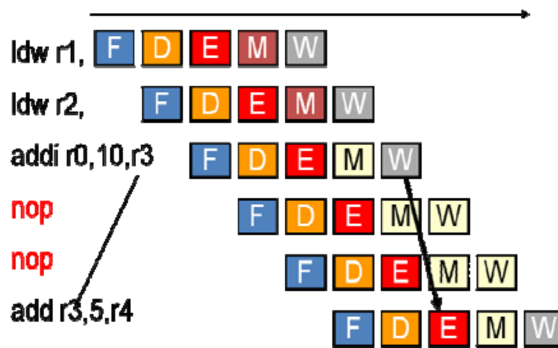


Wanneer twee nabije instructies hetzelfde object lezen of schrijven, ontstaat een afhankelijkheid.

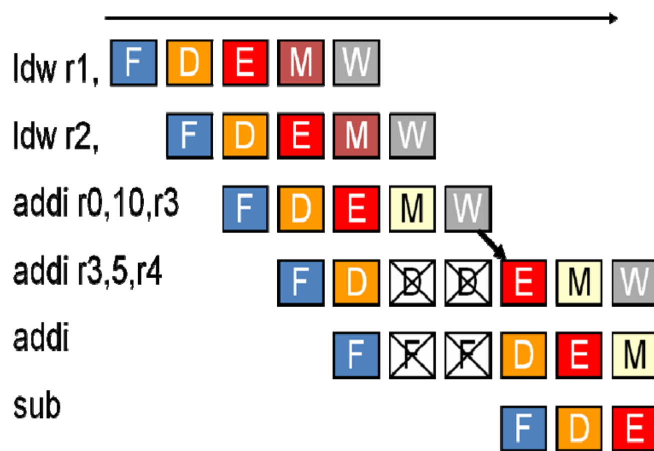
- **RAW**: object wordt eerst geschreven, dan gelezen (echte afhankelijkheid). Lezen moet wachten op schrijfoperatie
- **WAR**: object wordt eerst gelezen, dan geschreven (anti-afhankelijkheid). Schrijfoperatie moet wachten op leesoperatie, maar kan zijn waarde ook elders wegschrijven en op deze manier het wachten vermijden
- **WAW**: object wordt 2 maal geschreven (anti-afhankelijkheid). Tweede schrijfoperatie mag niet vóór de eerste gebeuren maar kan zijn waarde ook elders wegschrijven en op deze manier het wachten vermijden.
- **RAR**: object wordt 2 maal gelezen (**geen** afhankelijkheid). Leesoperaties mogen in een willekeurige volgorde worden uitgevoerd!

<u>Oplossingen</u>

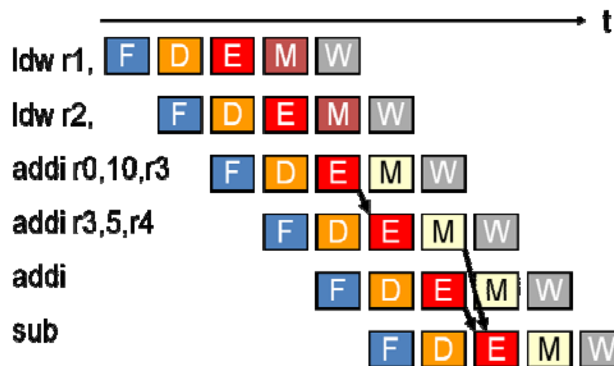
- Afhankelijke instructies ver genoeg uit elkaar zetten door invoegen van NOP.



- Pijplijn blokkeren tijdens de uitvoering telkens zich een datahazard voordoet (**stalls**). De volgende instructies worden onderbroken tot het resultaat beschikbaar is.



- Snelste oplossing: **forwarding**. Van zodra een resultaat bestaat op de processor zal ze beschikbaar zijn voor de volgende instructies. Forwarding kan gebeuren vanuit de output van de EX trap of vanuit de output van de MEM trap.



Om forwarding te realiseren, moeten een aantal extra verbindingen worden gelegd. De uitgang van de ALU na S1 en S2, wordt teruggekoppeld naar de ingang van S1 en S2; dus aangeboden aan dezelfde ALU. Ook wordt de uitgang van de MEM trap terug aangeboden aan diezelfde ALU (terugkoppeling van 2 van WB naar EX).

Principes van een in-order gepijplijnde processor

- IPC = 1 (eigenlijk een stuk onder 1 door hazards), CPI = 5
- 5 instructies tegelijkertijd in uitvoering
- Klokfrequentie hangt af van de **trapvertraging**: tragere trappen kan je opsplitsen in meerdere kleine, maar snellere delen, en aldus een hogere klokfrequentie maken.
- Indien men IPC > 1 wenst zal men moeten toelaten dat:
  - Instructies in een andere volgorde worden uitgevoerd dan deze van het prog
  - Per cyclus meer dan 1 instructie opgehaald & uitgevoerd wordt.

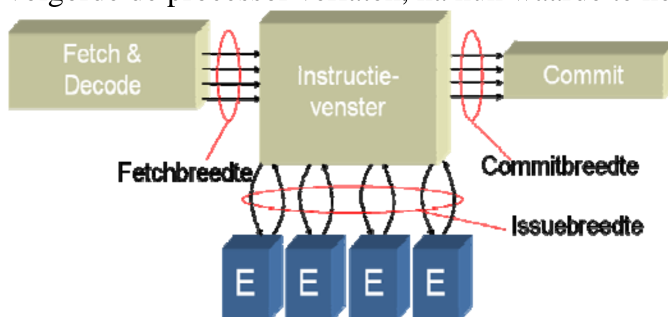
Andere microarchitecturen

Superscalaire architecturen

Zal meer dan 1 instructie per cyclus ophalen en uitvoeren → IPC > 1.

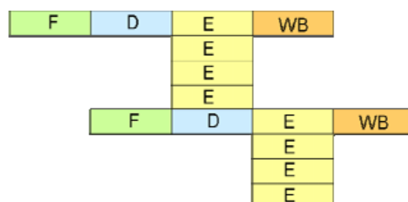
- **Ophalen:** Stel bvb 2 instructies ophalen per cyclus, dan moet geen 32 bit, maar 64 bit in 1 keer worden ingelezen en verwerkt.
- **Uitvoeren**: de trappen moeten nu meer dan 1 instructie per cyclus uitvoeren. De sequentiële semantiek mag hierbij niet verloren gaan! We zullen moeten rekening houden met **echte afhankelijkheden**, anti-afhankelijkheden zullen weggewerkt worden door het resultaat van die instructies tijdelijk op een andere plaats bij te houden en pas weg te schrijven in het registerbestand als alle voorgaande instructies zijn afgerond.

Er zullen – bij superscalaire architecturen – asynchroon instructies worden opgeladen, gedecodeerd en opgeslagen in een **instructievenster**. Hier wachten de instructies op beschikbaarheid van al hun argumenten, om dan uitgevoerd te worden op een van de functionele eenheden (E). Na uitvoering verhuizen ze naar de **commit trap**, waar ze terug in volgorde de processor verlaten, na hun waarde te hebben geproduceerd in registers & caches.



VLIW : Very Long Instruction Word Processors

Een andere manier om van de functionele eenheden nuttig gebruik te maken is om de hulp van de **compiler** in te roepen. Een instructie bestaat dan uit een aantal operaties die simultaan moeten kunnen worden uitgevoerd. De processor leest dan een instructie in, en stuurt de operaties in volgorde naar de betrokken functionele eenheid. De processor zal stellen dat de compiler heeft nagegaan op welk ogenblik de resultaten van de vorige instructie beschikbaar zullen komen in de registers, en neemt gewoon de waarde die op dat moment in de registers zitten. (**Philips Trimedia**)



EPIC : Explicitly Parallel Instruction Computing

Een EPIC instructie bestaat uit 2 operaties, waarbij het onderlinge parallellisme aangegeven wordt in een template, welke kan aangeven welke instructies parallel mogen worden uitgevoerd, en welke niet.

- Heel wat minder NOP nodig!
- Versneld de uitvoering door bvb een zeer groot aantal registers.