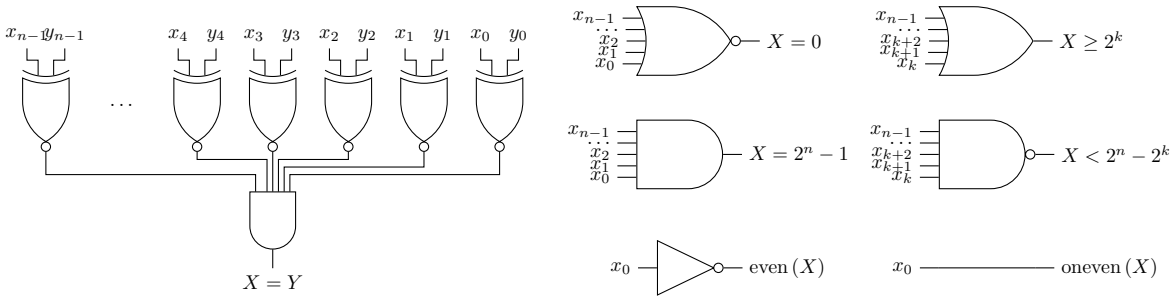
op dat we eenzelfde hoeveelheid hardware gebruiken. Deze schakeling laat echter toe om verschillende delen van het getal tegelijk te vergelijken.

Speciale gevallen

Testen op gelijkheid We kunnen twee getallen altijd vergelijken met een vergelijker zoals eerder beschreven. Soms willen we echter één of twee getallen testen op specifieke eigenschappen, bijvoorbeeld of twee getallen gelijk zijn. Dit kunnen we natuurlijk realiseren met een vergelijker, maar dit introduceert extra hardware en vertraging. In het geval van gelijkheid kunnen we dan ook gebruik maken van een rij van XNOR-poorten waarbij elke poort één bit van het ene en één bit van het andere getal als invoer krijgt. De uitgangen van al deze XNOR-poorten vormen vervolgens de invoer voor een AND-poort gaan. De uitvoer van deze AND poort toont ons dan of X gelijk is aan Y. Figuur 3.32(a) toont een realisatie van dit concept voor twee n-bit getallen.

3.3



(a) Gelijkheid (b) Testen met constanten

Figuur 3.32: Speciale gevallen van vergelijkers

Vergelijken met constanten We willen een getal niet altijd vergelijken met andere getal, maar soms met een constante. In dat geval zouden we natuurlijk ook van de vergelijker kunnen gebruikmaken, en de Y zelf samenstellen. Aangezien Y echter op voorhand gekend is spreekt het voor zichzelf dat we de implementatie echter grondig kunnen verbeteren. Bovendien kunnen we ook testen ontwerpen die we niet kunnen uitvoeren met een vergelijker[27]. Dergelijke componenten kunnen bouwen getuigt ook van enig inzicht. Alle speciale gevallen beschouwen is moeilijk. Op Figuur 3.32(b) geven we enkele voorbeelden. Deze opsommig is verre van exhaustief en het verklaren van deze schakelingen wordt aan de lezer overgelaten.

3.3.6 Schuifoperator

Een laatste set van operaties die vaak gebruikt wordt zijn **schuifoperaties**. Vele processoren implementeren schuifoperaties. Een ARM processor laat bijvoorbeeld toe om tijdens elke operatie de operand over enkele plaatsen te schuiven.

Er bestaan verschillende vormen van schuifoperaties, Daarom zullen we in deze paragraaf eerst de verschillende varianten bespreken. In het algemeen houdt een schuifoperatie in dat de waarde van de bit die eerst op plaats i stond, nu op een plaats $i + m$ staat. Of formeler: Y is het resultaat van een schuifoperatie van X over m plaatsen naar links indien

$$\forall i \in \mathbb{N} : i \in [0, n] \wedge i + m \in [0, n] \Rightarrow y_{i+m} = x_i \tag{3.43}$$

De definitie toont al dat i en $i + m$ beide binnen het bereik moeten liggen. Wat we doen met de bits die buiten de grenzen vallen, en met wat we de nieuwe plaatsen opvullen, wordt niet door Vergelijking (3.43)

[27] Bijvoorbeeld deelbaarheid door 2.

gespecificeerd. Een tweede opmerking is dat m ook negatief kan zijn, in dat geval voeren we een schuifoperatie naar rechts uit. Voor dit probleem bestaan er drie populaire oplossingen, de varianten van de schuifoperaties.

- Bij het **schuiven** negeren we de bits die buiten de grenzen vallen. Er bestaan twee vormen van schuiven die variëren in wat we op de vrijgekomen plaatsen zetten:
  - **Logisch schuiven**: indien we logisch schuiven hebben we een extra parameter nodig, namelijk welke waarde we op de nieuwe plaatsen zetten. Soms is dit zelfs een rij van bits. We maken dus de invoer groot genoeg om geen vrije plaatsen meer over te houden. Logisch schuiven wordt in de C/C++/Java taalfamilies vaak voorgesteld met de operatoren <<< en >>>.
  - **Aritmetisch schuiven**: hierbij willen we eigenlijk een wiskundige functie realiseren namelijk vermenigvuldigen of delen met een macht van 2. Hoe we dus aritmetisch schuiven hangt vooral af van de getalvoorstelling. Indien we naar links schuiven betekent dit meestal dat we de nieuwe plaatsen vullen met nullen. Indien we naar rechts schuiven zal bij een 2-complement voorstelling de hoogste bit (MSB) van het originele getal ingevoegd worden. Bij een "unsigned" voorstelling vullen we de vrije plaatsen ook met nullen op. Aritmetisch schuiven is vrij populair in programmeertalen. Programmeertalen die tot de C/C++/Java taalfamilie behoren, introduceren daarom 2 functies: voor links (<<) en rechts (>>) aritmetisch schuiven deze worden gedefinieerd als:

$$\begin{cases} X{<<}M \equiv X \times 2^M \\ X{>>}M \equiv X \div 2^M \end{cases} \tag{3.44}$$

    Men kan meestal niet schuiven met een negatieve M. De richting van schuiven dient men dus op voorhand te kennen.

- Bij het **roteren** vangen we de bits op die er langs één kant afvallen en plaatsen we deze op de vrijgekomen plaatsen aan de andere kant.

Implementatie van schuifoperaties

In deze subsubsectie zullen we twee schakelingen realiseren. De eerste is een component die alle schuifoperaties kan realiseren op 4-bit getallen, maar slechts over één plaats naar links of rechts. Het tweede component is 8-bit **barrel left rotator**. Dit component roteert 8-bit getallen naar links over een variabel aantal plaatsen. Merk op dat het bij een rotatie het eigenlijk niet uitmaakt in welke richting we roteren: een n-bit getal m plaatsen naar links roteren is net hetzelfde als het getal $n - m$ plaatsen naar rechts roteren. Deze tweede component illustreert verder hoe we schuifoperaties efficiënt over meerdere posities kunnen uitvoeren.

Schuifoperaties over 1 bit Indien we een component maken die meerdere operaties kan uitvoeren moeten we altijd eerst een instructieset definiëren, net zoals we dat ook deden in in Subsubsectie 3.2.6. De instructieset dient zowel een onderscheid te maken tussen schuiven of roteren, aritmetisch of logisch[28] en links of rechts. Verder willen we ook een bit voorzien die aangeeft of er überhaupt een schuifoperatie moet worden uitgevoerd: een soort enable-ingang. We zullen dus een 4-bit instructieset gebruiken. Tabel 3.11 toont de betekenis van elke bit. Op basis van deze instructieset kunnen we nu een component bouwen. Voor de

Signaal	0	1
s_3	geen schuifoperatie	schuifoperatie
s_2	links	rechts
s_1	schuiven	roteren
s_0	aritmetisch	logisch

Tabel 3.11: Instructieset voor de schuifoperaties over 1 bit.

berekening van de uitgangsbits gebruiken we multiplexers. Immers kan elke uitgang y_i maar drie mogelijke

[28] Enkel in geval van schuiven is dit relevant.

uitgangen hebben: x_{i-1}, x_i en x_{i+1}. Bij de uitgangen aan de rand dienen we alleen een andere interpretatie voor sommige x_j te vinden. Welke van deze drie ingangen we kiezen hangt verder alleen af van twee bits uit het instructiewoord: s_3 en s_2. In het geval dat $s_3 = 0$ geldt $y_i = x_i$. Bijgevolg zetten we x_i zowel op de d_0 en d_1 ingang van de multiplexer voor y_i. Indien $s_3 = 1$ en $s_2 = 0$ schuiven we naar links. We zetten dus x_{i+1} en x_{i-1} respectievelijk op de d_2 en d_3 ingangen. Tot slot dienen we nog de randgevallen op te lossen. Deze randgevallen beslaan enkel x_{i+1} voor y_3 en x_{i-1} voor y_0. Deze waarden zullen we respectievelijk noteren als x_4 en x_{-1} en vallen dus buiten de grenzen van de ingangen. In geval van rotatie geldt:

$$\begin{cases} x_{-1} = x_3 \\ x_4 = x_0 \end{cases} \quad \text{(rotatie)} \tag{3.45}$$

Dit dwingen we af met twee 2-naar-1 multiplexers. Deze multiplexers hebben als schakelelement instructiebit s_1. We dienen nu enkel nog het geval te behandelen waarin we schuiven. Indien we logisch schuiven, schuiven we de bits L_{in} en R_{in} in. Bij arithmetisch zullen we aan de rechterzijde een 0 inschuiven. Aan de linkerkant is dit ook het geval tenzij we schuiven met een 2-complement voorstelling. In dat laatste geval bepaalt de hoogste bit immers ook het teken van het getal. In dat geval moeten we de waarde van de hoogste bit dus nogmaals inschuiven. We kunnen bovenstaande beschrijvingen formaliseren tot volgende formules:

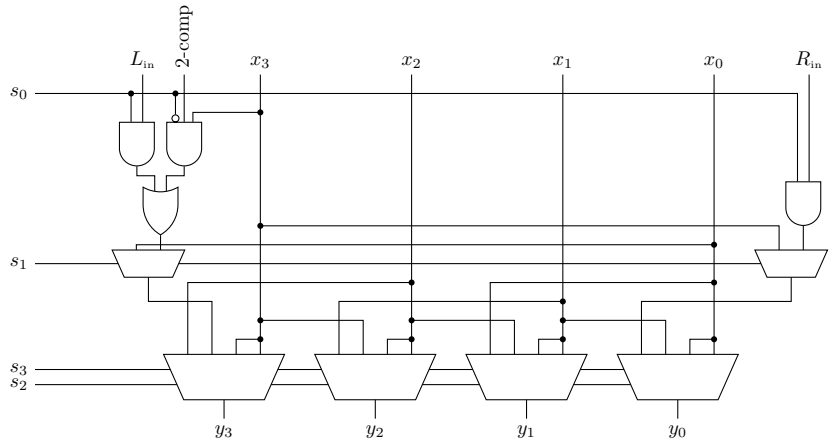
3.3

$$\begin{cases} x_{-1} = R_{\text{in}} \\ x_4 = L_{\text{in}} \end{cases} \quad \text{(schuiven, aritmetisch)}$$

$$\begin{cases} x_{-1} = 0 \\ x_4 = 0 \end{cases} \quad \text{(schuiven, logisch, unsigned)} \tag{3.46}$$

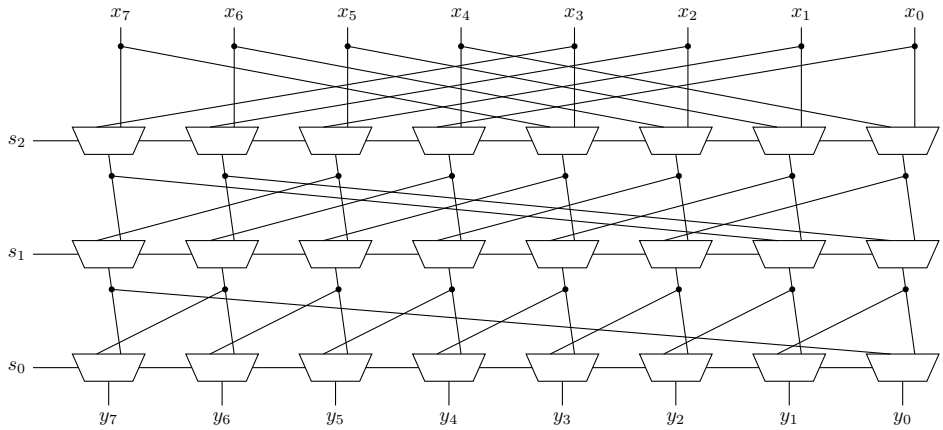
$$\begin{cases} x_{-1} = 0 \\ x_4 = x_3 \end{cases} \quad \text{(schuiven, logisch, 2-complement)}$$

Deze logica kunnen we vervolgens implementeren met OR-AND-poorten. Het resultaat van deze volledige implementatie staat op Figuur 3.33.



Figuur 3.33: Implementatie van een schuifoperator over 1 bit.

8-bit barrel left rotator In de vorige paragraaf hebben we een schuifoperator gebouwd die over 1 bit kan schuiven. Door verschillende van deze schuifoperatoren na elkaar te plaatsen kunnen we schuiven over meerdere plaatsen. Toch is dit niet erg praktisch: om m plaatsen te schuiven zouden we m van deze schuifoperatoren na elkaar moeten plaatsen, wat grote vertragingen zou impliceren. In deze paragraaf zullen we een rotator bouwen die de vertraging beperkt tot $\log_2 m$, met m het maximaal aantal plaatsen dat kan opgeschoven worden. Het concept is toepasbaar op algemene schuifoperaties, dus ook bijvoorbeeld logisch schuiven. Figuur 3.34 geeft het concept weer.



Figuur 3.34: Implementatie van een 8-bit barrel left rotator.

Deze barrel left rotator is gebaseerd op het additief principe. Het additief principe stelt dat indien we een getal willen schuiven of roteren over m bits, we dit ook kunnen verwezenlijken door eerst te schuiven/roteren over m_1 bits en vervolgens over m_2 bits met $m = m_1 + m_2$. Indien we onze rotator bouwen volgens een structuur waarbij elke niveau i een rotatie uitvoert over 2^i plaatsen kunnen we elke rotatie-operatie uitvoeren. Gedurende dit hoofdstuk hebben we immers alle getallen binair kunnen voorstellen. Op Figuur 3.34 zien we dat het niveau s_2, vier plaatsen naar links roteert. Het volgende niveau twee en het laatste één. Elk van deze niveaus heeft dezelfde vertraging. Indien we dus een n-bit barrel left rotator willen bouwen die maximaal m plaatsen kan opschuiven moeten we dus $\lceil \log_2 m \rceil$ niveaus synthetiseren. Een niveau i roteert 2^i plaatsen voor $\forall i = 0, 1, \ldots, \lceil \log_2 m \rceil - 1$. Dit concept is eenvoudig te veralgemenen naar een volledige schuifoperator. Merk op dat het aantal niveaus dus theoretisch niet afhangt van het aantal bit in het getal. Op de meeste processoren neemt men echter $m = n$. Een andere mooie eigenschap is dat de volgorde van de niveaus niet relevant is.