Voortschrijdend gemiddelde

Probleem

Geef het gedrag en implementatie van een systeem dat het voortschrijdend gemiddelde bepaald van de vorige n (generic) waarden.
De entity beschrijving is gegeven.
Valid is een bit dat aangeeft dat tenminste na een reset *n* waarden zijn ingelezen.

```
ENTITY voortschrijdend_gemiddelde IS
  GENERIC (n : positive := 16);
  PORT (i     : IN integer RANGE 0 TO 15;
        reset : IN bit; -- asynchroon
        clk   : IN bit;
        o     : OUT integer RANGE 0 TO 15;
        valid : OUT bit);
END voortschrijdend_gemiddelde;
```

Gedragsbeschrijving

Het doel van de gedragsbeschrijving is alleen het gedrag van het systeem te beschrijven. Het hoeft dus niet synthetiseerbaar te zijn. Het moet daarentegen wel duidelijk het gewenste gedrag beschrijven.
Het gedrag is eenvoudig.

```
TYPE integer_array IS ARRAY (natural RANGE <>) OF integer RANGE 0 TO 15;
```

Beschrijft een unconstrained array waarvan elk element een integer is met bereik van 0 t/m 15.
Vervolgens wordt een functie “gemiddeld” gedefinieerd die de elementen van het array sommeert en vervolgens deelt door het aantal elementen.
In het vet is de eigen gedragsbeschrijving gegeven.

- Reset: Pas nadat *n* keer een waarde is ingelezen is de uitgang geldig, dus *valid* moet ‘0’ worden. Verder wordt de variable *aantal* gebruikt om tot *n* te tellen. Deze variable moet ook weer op nul worden gezet. Vervolgens wordt *inp_array* en de output *o* ook 0 gemaakt. Dit hoeft op zich niet. Immers de *o* is pas geldig nadat n waarden zijn gelezen.
- Het eigenlijke gedrag is ook eenvoudig beschreven. De data die het langst geleden is ingelezen wordt verwijderd uit *inp_array* en de meest recente data wordt in *inp_array* geplaatst. Vervolgens wordt dit array aan de functie *gemiddeld* aangeboden.

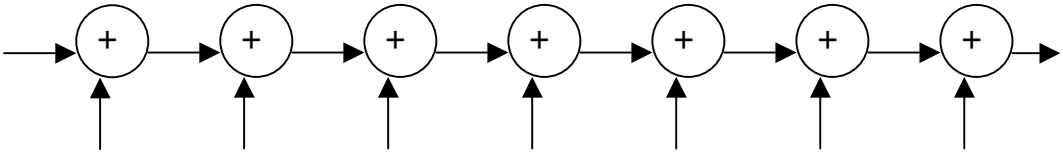
```
ARCHITECTURE gedrag OF voorstschrijdend_gemiddelde IS
BEGIN
  PROCESS (reset,clk)
    TYPE integer_array IS ARRAY (natural RANGE <>) OF integer RANGE 0 TO 15;
    FUNCTION gemiddeld(inp : integer_array)
             RETURN integer IS
      VARIABLE som : integer RANGE 0 TO inp'LENGTH*15;
    BEGIN
      som := 0;
      FOR i IN inp'RANGE LOOP som:=som+inp(i); END LOOP;
      RETURN som/inp'LENGTH;
    END gemiddeld;
    VARIABLE inp_array : integer_array(1 TO n);
    VARIABLE aantal : integer RANGE 0 TO n*15;
  BEGIN
    IF reset='1'
      THEN valid <='0'; aantal := 0; inp_array := (OTHERS=>0); o <= 0;
    ELSIF clk='1' and clk'EVENT THEN
        IF aantal < n
          THEN aantal:=aantal+1;
          ELSE valid <= '1';
        END IF;
        inp_array := i & inp_array(1 TO n-1);
        o <= gemiddeld (inp_array);
    END IF;
  END PROCESS;
END gedrag;
```

Synthesetools zijn de afgelopen jaren steeds krachtiger geworden. Ze nemen steeds meer (detail)werk uit handen.
De meeste synthesetools zijn technologieonafhankelijk. Bekende tools zijn o.a. *Design Compiler* en *FPGA Express* van Synopsys, *Synplicity* van Synplify, *Leonardo* van Mentor Graphics,..
In dit voorbeeld is gebruik gemaakt van FPGAExpress 3.3.1. Het versie nummer is belangrijk omdat de ontwikkeling nog steeds doorgaat en een nieuwere versie er ongetwijfeld weer een aantal trucjes heeft bijgeleerd.

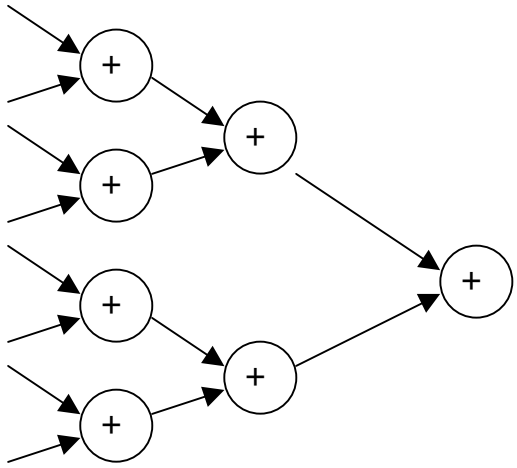
FPGA Express ondersteunt vele technologieën. In dit document is gebruik gemaakt van Xilinx. FPGA Express levert een EDIF (Electronic Design Intermediate Format) als output en dit wordt door de software van Xilinx (Foundation Series F1.5) ingelezen. Dan vindt pas de echte realisatie plaats. Na deze fase is de snelheid van het systeem bekend. Samengevat:

Synthesis tool:	FPGA Express: 3.3.1
Xilinx Foundation series	F 1.5
Component Xilinx	4010XL

Zoals aangegeven is de gedragsbeschrijving synthetiseerbaar. Dit was dan wel niet de bedoeling, maar het is mooi meegenomen. Als het maar voldoet aan de gewenste snelheid, grootte etc. Hoe zou een synthesetool dit synthetiseren?
Het gedrag geeft dan als het ware het algoritme .. d.w.z. hoe het geïmplementeerd moet worden. In feite is het een schuifregister waarbij elke positie een integer waarde bevat met een bereik van 0 t/m 15, d.w.z. 4 bits. Vervolgens worden alle ‘integer’ waarden uit de registers bij elkaar opgeteld. Voor *n* is 8 in file *gem_gedrag_schema4.pdf* het gegenereerde schema gegeven. In dit schema is duidelijk te zien dat de optellers voor grote waarden van n het kritieke pad ongunstig zal beïnvloeden.



In plaats van deze lineaire structuur zal in een boomstructuur de vertraging minder snel oplopen.



Het is te verwachten dat synthesetools, voor zover ze dit trucje nu nog niet kennen, dat straks wel kunnen! Hierover later meer.

	N=4	N=8	N=16	N=32
Gedrag	19.132ns.	31.925ns.	57.146ns	106.217ns.

Voor verschillende waarden van N is het kritiek pad bepaald. Zoals verwacht neemt de vertraging nagenoeg lineair toe met de grootte van N.

Recursieve beschrijving

Hoe kunnen we het gewenste gedrag realiseren? Wellicht de eerste voor de hand liggende oplossing is om het optellen efficiënter te laten plaats vinden. M.a.w. de optelling moet middels de boomstructuur worden gerealiseerd. Hoe beschrijven we dit zodanig dat een synthesetool dit begrijpt? Een recursieve aanpak voor het optellen. In de onderstaande recursieve beschrijving is dit in het vet weergegeven.

```
ARCHITECTURE recursief OF voorstschrijdend_gemiddelde IS
BEGIN
  PROCESS (reset,clk)
    TYPE integer_array IS ARRAY (natural RANGE <>) OF integer RANGE 0 TO 15;
    FUNCTION som(inp : integer_array)
             RETURN integer IS
      CONSTANT midden : integer := inp'LENGTH/2;
--    CONSTANT inpi : integer_array(1 TO inp'LENGTH) := inp;  -- synthesis complains
      VARIABLE inpi : integer_array(1 TO inp'LENGTH);
    BEGIN
      inpi := inp;
      IF inpi'LENGTH=1 THEN
        RETURN inpi(1);
      ELSE
        RETURN som(inpi(1 TO midden)) + som(inpi(midden+1 TO inpi'LENGTH));
      END IF;
    END som;
    VARIABLE inp_array : integer_array(1 TO n);
    VARIABLE aantal : integer RANGE 0 TO n*15;
  BEGIN
    IF reset='1'
```

```
      THEN valid <='0'; aantal := 0; inp_array := (OTHERS=>0); o <= 0;
    ELSIF clk='1' and clk'EVENT THEN
        IF aantal < n
          THEN aantal:=aantal+1;
          ELSE valid <= '1';
        END IF;
        inp_array := i & inp_array(1 TO n-1);
        o <= som(inp_array)/n;
    END IF;
  END PROCESS;
END recursief;
```

Ook hier zijn voor enkele waarden van N de lengte van het kritieke pad bepaald en vergeleken met de gedragsbeschrijving.
Duidelijk is te zien dat de recursieve oplossing goede resultaten oplevert, althans vergeleken met de gedragsbeschrijving.

	N=4	N=8	N=16	N=32
Gedrag	19.132ns.	31.925ns.	57.146ns	106.217ns.
Recursief	14.321ns.	20.347ns.	28.224ns.	32.986ns.

Maar er zijn meer manieren om het gewenst gedrag te realiseren!

Alleen sommeren wat nodig is.
In de recursieve oplossing werd het kritieke pad aanzienlijk ingekort bij grotere waarden van N. Maar is het noodzakelijk om alle N waarden steeds weer opnieuw op te tellen? Nee.
Waarom niet een extra variable (in de realisatie een extra register!) waarin de som van alle registerwaarden komt te staan en als er nieuwe data wordt ingelezen wordt uiteraard ook nu weer de langst gelezen ingelezen data verwijderd. De som wordt dus opgehoogd met de nieuwe data en verminderd met de data die uit de buffer wordt verwijderd. Dit beperkt het aantal optellers aanzienlijk. Sterker nog als het klopt zal de vertraging nauwelijks worden beïnvloed door de grootte van N. Nauwelijks, immers als N groter wordt worden de optellers wel breder en dus trager.
De VHDL Beschrijving is verrassend eenvoudig.

```
ARCHITECTURE algoritme OF voorstschrijdend_gemiddelde IS
BEGIN
  PROCESS (reset,clk)
    TYPE integer_array IS ARRAY (natural RANGE <>) OF integer RANGE 0 TO 15;
    VARIABLE inp_array : integer_array(0 TO n-1);
    VARIABLE nmb : integer RANGE 0 TO n;
    VARIABLE sum : integer RANGE 0 TO 15*n;
  BEGIN
    IF reset='1' THEN
      valid <='0'; nmb := 0; sum := 0; index:=0; o<=0; inp_array:=(OTHERS => 0);
    ELSIF clk='1' AND clk'EVENT THEN
      IF nmb < n-1 THEN
        nmb:=nmb+1;
      ELSE
        valid <= '1';
      END IF;
      sum := sum + i - inp_array(n-1);
      o <= sum / n;
      inp_array:= i & inp_array(0 TO n-2);
    END IF;
  END PROCESS;
END algoritme;
```

Ook hiervan is weer voor verschillende waarden van N de lengte van het kritieke pas bepaald.

	N=4	N=8	N=16	N=32
Gedrag	19.132ns.	31.925ns.	57.146ns	106.217ns.
Recursief	14.321ns.	20.347ns.	28.224ns.	32.986ns.
Algoritme2	11.984ns.	12.322ns.	12.397ns.	12.575ns.

Het resultaat is inderdaad verrassend(?) goed. Door het toevoegen van een extra register voor de som hebben we de snelheid aanzienlijk verbeterd en nagenoeg onafhankelijk van N.

Data in een geheugen

In de vorige oplossing werd de data door een schuifregister heen geschoven. In plaats van de data te verplaatsen kan de data ook in een geheugen worden geplaatst waarbij nieuwe data op de plaats wordt geschreven van de langst aanwezige data.
Hieronder de VHDL beschrijving.

```
ARCHITECTURE geheugen OF voorstschrijdend_gemiddelde IS
BEGIN
  PROCESS (reset,clk)
    TYPE integer_array IS ARRAY (natural RANGE <>) OF integer RANGE 0 TO 15;
    VARIABLE inp_array : integer_array(0 TO n-1);
    VARIABLE index : integer RANGE 0 TO n-1;
    VARIABLE nmb : integer RANGE 0 TO n;
    VARIABLE sum : integer RANGE 0 TO 15*n;
  BEGIN
    IF reset='1' THEN
      valid <='0'; nmb := 0; sum := 0; index:=0; o<=0; inp_array:=(OTHERS => 0);
    ELSIF clk='1' AND clk'EVENT THEN
      IF nmb < n-1 THEN
        nmb:=nmb+1;
      ELSE
        valid <= '1';
      END IF;
      index := (index + 1) MOD n;
      sum := sum + i - inp_array(index);
      o <= sum / n;
      inp_array(index):= i;
    END IF;
  END PROCESS;
END geheugen;
```

Ook hier zijn weer voor enkele waarden van N de lengte van het kritieke pad bepaald. Eigenlijk valt het resultaat tegen. Waarom? Immers ook hier is het aantal optelling beperkt. Een ding is duidelijk FPGA Express heeft niet in de gaten dat de data in een geheugen geplaatst moest worden. En veel synthesetools hebben het daar moeilijk mee. Maar er zijn ook synthesetools die dit wel kunnen, zoals synplicity.

	N=4	N=8	N=16	N=32
Gedrag	19.132ns.	31.925ns.	57.146ns	106.217ns.
Recursief	14.321ns.	20.347ns.	28.224ns.	32.986ns.
Algoritme2	11.984ns.	12.322ns.	12.397ns.	12.575ns.
Geheugen	15.532ns.	27.046ns.	26.947ns.	32.869ns.