een FPGA programmeren

Figuur 1. Het Alchitry Au FPGA development board.

Gerelateerde producten

Bent u op zoek naar de belangrijkste producten die in dit artikel genoemd zijn? Elektor en SparkFun staan voor u klaar!

Alchitry Au FPGA Development Board

www.elektormagazine.nl/esfe-en-fpga1

Justin Rajewski (Alchitry)

In deze tutorial zullen we de principes bespreken van het maken van een ontwerp voor een FPGA, plus de fundamentele bouwstenen die u daarbij kunt gebruiken. Op het eerste gezicht lijkt een FPGA ontmoedigend complex, maar een goede combinatie van gebruiksvriendelijke software, veelzijdige hardware en een paar handige voorbeelden geeft u een zetje in de goede richting.

Even voor alle duidelijkheid voordat we beginnen: u programmeert geen FPGA's [1]. We zeggen voor het gemak vaak dat we dat doen – omdat het aanvoelt als programmeren: u schrijft wat tekst, die tekst wordt vertaald in een binair bestand, en dat binaire bestand wordt in de FPGA geladen. Maar u schrijft geen programma. U creëert een schakeling. En u gebruikt geen programmeertaal om schakelingen te maken; u gebruikt een hardwarebeschrijvingstaal (HDL, *Hardware Description Language*) [2].

Het is veel te ingewikkeld om grote en complexe ontwerpen als schema te tekenen; in plaats daarvan beschrijven we het gewenste gedrag van de schakeling, en de tools zoeken dan uit hoe dat in de praktijk geïmplementeerd kan worden. U moet altijd bij het creëren van een FPGA-schakeling in gedachten houden dat u *hardware* beschrijft en dat alles wat u schrijft uiteindelijk de vorm van een fysieke schakeling zal krijgen. Het is mogelijk om schakelingen te beschrijven die onmogelijk geïmplementeerd kunnen worden, of om iets te beschrijven dat eenvoudig lijkt maar waarvan de realisatie een enorme hoeveelheid resources zou vereisen. Daarom is het van cruciaal belang om een goed idee te hebben hoe de schakeling die u probeert te beschrijven, geïmplementeerd zou kunnen worden.

Om deze tutorial te volgen, hebt u een Alchitry Au FPGA Development Board (**figuur 1**) en een 'omkeerbare' USB A-naar-C-kabel nodig.

De structuur van een ontwerp

HDL's zijn meestal gebaseerd op het idee van een module. Een module is een deelschakeling ('blok') met een aantal in- en uitgangen plus wat logica om ze aan elkaar te plakken. Een module kan submodules bevatten of zelfstandig zijn – dit is vergelijkbaar met het opsplitsen van een programma in functies.

Hoewel het mogelijk zou zijn om een compleet ontwerp in een enkele module uit te voeren, is het beter om kleinere modules te gebruiken

die specifieke functies in uw ontwerp uitvoeren. Door uw project op te splitsen in modules, reduceert u de complexiteit van elk onderdeel waaraan u werkt. Sommige modules kunnen bijvoorbeeld algemene taken uitvoeren en steeds opnieuw worden gebruikt.
Wanneer u met een ontwerp begint, is het handig om eerst een blokschema te maken waarin u de verschillende modules schetst en hoe ze met elkaar zijn verbonden. Dit helpt om de omvang van uw ontwerp te definiëren en op lgische manier op te splitsen.

Lucid

Voor de rest van deze tutorial gebruiken we Lucid [3]. Dit is een HDL die speciaal voor FPGA's is gemaakt en veel van de valkuilen vermijdt die vaak voorkomen bij andere HDL's zoals Verilog en VHDL (en geloof me, er zijn veel valkuilen).
Lucid is een fantastisch uitgangspunt om met FPGA's te gaan werken. Ik word vaak benaderd door mensen die bezorgd zijn dat ze vastlopen met Lucid of die om een andere reden liever met Verilog of VHDL willen werken. Als u net begint, is Lucid het beste startpunt. Daar leert u de juiste principes van hardware-ontwerp, voordat u zich een weg door onhandige andere HDL's probeert te banen. En als u later naar iets anders wilt overschakelen, dan is dat niet zo moeilijk. Lucid is in grote lijnen gebaseerd op Verilog en Alchitry Labs kan Lucid voor u naar Verilog converteren als u uw superslimme modules ergens anders wilt gebruiken.
Laten we nu eerst eens kijken waaruit een module eigenlijk bestaat.

Anatomie van een module

Wanneer u een ontwerp maakt, begint u met een module op het hoogste niveau (top level). Dit is de module waarvan de in- en uitgangen de 'echte' in- en uitgangen zijn op de aansluitingen van de FPGA. Voor elk Alchitry-project is dit *cu_top.luc* of *au_top.luc*, afhankelijk van het board (Cu of Au) dat u gebruikt. De initiële modules op het hoogste niveau voor beide boards zien er in wezen hetzelfde uit (**listing 1**).
Het eerste deel van de module is de declaratie van de poorten. Hier declareert u de in- en uitgangen voor uw module. Dit zijn de signalen op het board zelf, aangezien het de top level-module betreft (**listing 2**).
Het is u misschien opgevallen dat achter *output led* een getal tussen rechte haken staat. Dat geeft aan dat het hier niet om één maar om acht afzonderlijke uitgangen gaat die als een array zijn gegroepeerd. We gaan later nog nader op de array-syntax in.
Een module kan ook een lijst met parameters hebben die kunnen worden gebruikt om de module aan te passen. Deze lijst is hier weggelaten en zou zinloos zijn op een top level-module, aangezien de parameters worden doorgegeven door de bovenliggende (*parent*) module die deze instantieert. De term *instantiëren* wordt gebruikt om te verwijzen naar het toevoegen van een module of andere resource aan uw ontwerp. Het betekent dat er een *instantie* van die module of andere resource wordt gemaakt.
Wanneer u code schrijft, worden telkens wanneer u een functie aanroept, exact dezelfde instructies telkens weer gebruikt, ongeacht hoe vaak u die functie aanroept. Maar iedere keer dat u een module instantieert, wordt de complete schakeling waaruit de module bestaat, gedupliceerd. Als u dezelfde resources voor meerdere taken wilt hergebruiken, moet u er zelf achter proberen te komen hoe u dat voor elkaar kunt krijgen. Over instantiëren gesproken, meestal instantieert u alles wat uw module nodig heeft direct na de declaratie van de poorten.
De eerste regel declareert een signaal met het sleutelwoord *sig*:

```
sig rst;                    // reset signal
```

Listing 1.

```
module au_top (
   input clk,      // 100MHz clock
   input rst_n,    // reset button (active low)
   output led [8], // 8 user controllable LEDs
   input usb_rx,   // USB->Serial input
   output usb_tx   // USB->Serial output
  ) {

  sig rst;          // reset signal

 .clk(clk) {
   // The reset conditioner is used to
   // synchronize the reset signal to
   // the FPGA clock. This ensures the
   // entire FPGA comes out of reset at
   // the same time.
   reset_conditioner reset_cond;
  }

  always {
   reset_cond.in = ~rst_n; // input raw inverted
                           // reset signal
   rst = reset_cond.out;   // conditioned reset
    led = 8h00;            // turn LEDs off
    usb_tx = usb_rx;       // echo the serial data
  }
}
```

Listing 2.

```
    input clk,          // 100MHz clock
    input rst_n,        // reset button (active low)
    output led [8],     // 8 user controllable LEDs
    input usb_rx,       // USB->Serial input
    output usb_tx       // USB->Serial output
```

Signalen zijn geen geheugen. Ze slaan geen waarden op. U kunt ze als draden beschouwen. Een draad kan een waarde hebben, maar het is eigenlijk gewoon een verbinding van het ene punt naar het andere. In dit ontwerp gebruiken we het signaal *rst* als een plaatshouder (tijdelijke aanduiding) voor de uitvoer van de *reset_conditioner* module. Het instantiëren van deze module gebeurt in de volgende regels:

```
.clk(clk) {
   // The reset conditioner is used to synchronize
   // the reset signal to the FPGA clock. This
   // ensures the entire FPGA comes out of reset
   // at the same time.
   reset_conditioner reset_cond;
 }
```

Om iets te instantiëren, gebruikt u eenvoudig de naam van de resource, gevolgd door de naam van deze specifieke instantie. Dus de regel `reset_conditioner reset_cond;` maakt een instantie van de *reset_conditioner* module met de naam *reset_cond*. Het blok waarin deze instantiatie is ingebed, wordt een verbindingsblok (connection block) genoemd. Hiermee kunt u een ingang of parameter met een bepaalde naam van meerdere modules op hetzelfde signaal aansluiten.

In ons geval verbinden we de ingang *clk* met het signaal *clk* (wat een input is voor onze module). De syntaxis is *.port (signal)* waarbij *port* de naam is van de ingang op de module die wordt geïnstantieerd en *signal* het signaal is dat erop moet worden aangesloten.
De *reset_conditioner* module heeft een ingang met de naam *clk*, dus is deze ingang direct verbonden met het signaal *clk* in onze module. U zou deze ook rechtstreeks met de module kunnen verbinden in de regel waar de instantiëring plaats vindt – in deze geest:

```
reset_conditioner reset_cond(.clk (clk));
```

Dat hebben we hier niet gedaan omdat de input *clk* deel uitmaakt van bijna elke module en het handig is om een blok zoals dit setup-blok te hebben, zodat we eenvoudig andere instantes kunnen toevoegen die met *clk* moeten worden verbonden.
Bijna elke module heeft een *clk*-ingang en vaak hebben ze ook een *rst*-ingang voor een reset. Waarvoor een klok en reset worden gebruikt, komt later aan de orde. U kunt als volgt meerdere verbindingen aan hetzelfde verbindingsblok toevoegen:

```
.clk(clk), .rst (rst) { ... }
```

U kunt de blokken ook 'nesten' en aangezien niet elk blok dat *clk* gebruikt ook *rst* zal gebruiken, ziet u aan het begin van een module meestal zoiets als dit:

```
.clk(clk) {
    // clk only instantiations
    .rst(rst) {
        // rst and clk instantiations
    }
}
```

Over het *always*-blok

De volgende sectie is het *always*-blok. Dit is waar het allemaal gebeurt. In de always-blokken beschrijft u alle logica die in uw module moet plaatsvinden. Ze bevatten wat bekend staat als combinatorische logica. Combinatorische logica is de benaming voor alle digitale schakelingen waarvan de uitvoer uitsluitend een functie is van de momentele invoer. Ze bezitten geen interne toestand of geheugen. Een goed voorbeeld hiervan is een opteller. De uitvoer wordt uitsluitend bepaald door de twee getallen die momenteel worden ingevoerd. Het maakt niet uit wat de laatst ingevoerde getallen waren of hoe vaak u de getallen hebt gewijzigd. De uitvoer is altijd een functie van de momentele invoer.

Het eerste FPGA-project: pret met PWM

Wanneer u een nieuw Alchitry Au [5] of Alchitry Cu [6] board koopt, creëert de standaard FPGA-configuratie een fraai golfeffect op de LED's. In de bijbehorende tutorial doorlopen we verschillende stappen om zoiets te maken. Deze tutorial toont de juiste benadering van een ontwerp en belicht verschillende zaken waarover we moeten nadenken, aangezien we met hardware werken.

Meer weten? Bekijk de tutorial:
https://learn.sparkfun.com/tutorials/first-fpga-project---getting-fancy-with-pwm

Binnen een *always*-block schrijven we *statements*. Hiervan kennen we vier hoofdtypen:

- toewijzingen;
- *if* statements;
- *case* statements;
- *for* lussen.

Toewijzingen
Toewijzingen komen verreweg het meest voor. Daarbij staat links een signaal, gevolgd door een isgelijkteken en een uitdrukking (expression).

```
signal = expression;
```

De kracht hiervan komt van de expression (uitdrukking). Hier kunt u diverse operatoren gebruiken om bits te manipuleren. Daartoe behoren enkele wiskundige operatoren zoals +, – en *. Merk op dat / voor deling niet kan worden gebruikt bij dynamische uitdrukkingen. De reden is dat deling voor de beschikbare tools te gecompliceerd is om dit op een acceptabele standaard-manier te doen. Deling blijft mogelijk, maar daarvoor is wat meer moeite en planning noodzakelijk.

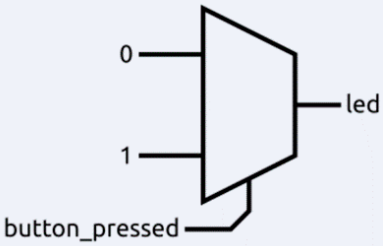
if statements
if statements hebben de gebruikelijke vorm:

```
if (expr) { ... } else { ... }
```

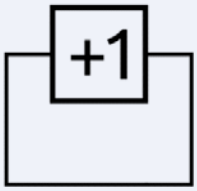
Als de uitdrukking na *if* waar is (ongelijk aan nul), dan is de eerste groep regels geldig. Als de uitdrukking niet waar is (gelijk aan nul), dan zijn de regels in het *else*-blok geldig. Het *else*-gedeelte is optioneel. Merk op dat ik het woord *geldig* gebruikte en niet *uitgevoerd*. Die fout kunt u gemakkelijk maken als u *if* statements en *for* loops hebt die u een programmeermentaliteit bezorgen. *if* statements krijgen in hardware meestal de vorm van een multiplexer. U selecteert dan gewoon een van de twee ingangen op basis van een bepaalde uitdrukking.
Wanneer u een signaal in een *always*-blok een waarde toewijst, dan moet er ook ongeacht de omstandigheden *altijd* een waarde aan worden toegewezen. Dit betekent meestal dat wanneer u een waarde toekent aan iets in een *if* statement, u een corresponderende toewijzing moet hebben in het *else*-gedeelte van het statement. De enige uitzondering op deze regel is de d-input van een dff- of fsm-type. Daar komen we later nog op terug.
Een andere manier om ervoor te zorgen dat u altijd een waarde toewijst, is door uw *always*-blok te beginnen met een aantal redelijke default-waarden. Zie bijvoorbeeld deze pseudo-code:

```
led = 0;
if (button_pressed)
    led = 1;
```

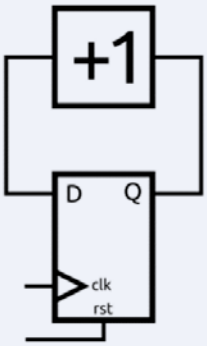
Als de knop niet is ingedrukt, heeft *led* de waarde 0. Maar wat gebeurt er als de knop wordt ingedrukt? Krijgt *led* eerst een waarde van 0 toegewezen die vervolgens geactualiseerd wordt met de waarde 1? Nee. Als de knop wordt ingedrukt, heeft *led* altijd de waarde 1. Toewijzingen verderop in een *always*-blok hebben voorrang op eerdere toewijzingen. U kunt zich dat zo voorstellen dat het blok altijd en ogenblikkelijk wordt geëvalueerd.
Stelt u zich nu voor dat we die defaultwaarde niet vóór het *if* statement stond. Welke waarde zou *led* dan hebben als de knop niet is



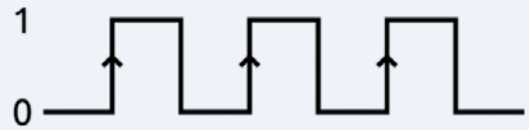
Figuur 2. FPGA-stijl knop aan / uit-bediening voor een LED.



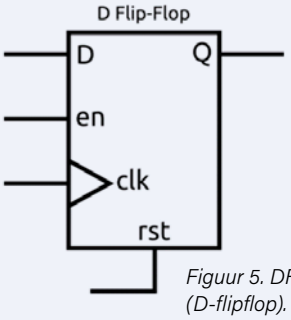
Figuur 3. Optelfunctie.



Figuur 6. Combinatie van DFF en teller.



Figuur 4. Kloksignaal waarbij de opgaande flanken met pijlsymbolen zijn gemarkeerd.



Figuur 5. DFF (D-flipflop).

ingedrukt? Het is verleidelijk om aan te nemen *led* het gewoon zijn vorige waarde zou behouden, maar vergeet niet dat signalen geen waarden kunnen opslaan. Het zijn eigenlijk gewoon draden die twee dingen met elkaar verbinden.
Met de defaultwaarde 0 zouden we dit in hardware kunnen realiseren met een multiplexer. Dit triviale geval kan nog worden vereenvoudigd door gewoon de signalen *led* en button_pressed te combineren zoals in **figuur 2**.

case statements
case statements hebben de volgende syntaxis:

```
case(expr) {
    value: statement;
    value: statement;
    default: statement;
}
```

Ze functioneren net als *if* statements, maar zijn handiger wanneer een enkele uitdrukking veel vertakkingen heeft. Het value-gedeelte van het *case* statement moet een constante zijn. De optionele default-vertakking is een 'catch-all.'
In tegenstelling tot code bieden *case* statements geen prestatieverbetering vergeleken met een groot aalnat *if* statements. Ze zijn uitsluitend bedoeld om de leesbaarheid van de code te verbeteren.

for lussen
En tenslotte hebben we *for* lussen (of loops). *For* loops hebben in Lucid eenzelfde syntaxis als *for* loops in C of Java *for* loops, maar hebben enkele beperkingen:

```
for (init; eval; increment) { ... }
```

Ze worden meestal gebruikt met een *var* type die wordt gebruikt om waarden op te slaan die niet direct in de schakeling verschijnen maar in de beschrijving worden gebruikt. De grote beperking van *for* loops in hardware is dat ze een constant aantal iteraties moeten hebben. Dat komt omdat de tools de loop (lus) moeten kunnen 'uitrollen.' Een *for* lus is eigenlijk niets anders dan het keer op keer kopiëren en plakken van het betreffende codegedeelte – maar is wel veel gemakkelijker te lezen. U doet er goed aan deze loops te vermijden, tenzij u een goede reden heeft om er een te gebruiken. Het is heel gemakkelijk om met *for* loops een zeer omvangrijke en trage schakeling te creëren.

Getallen

In Lucid is er een handvol manieren om een numerieke constante te definiëren. De eenvoudigste manier is om simpelweg een getal (bijvoorbeeld 14) in te typen. Als u een 'eenzaam' getal ziet, is het een decimaal getal (grondtal $_{10}$) en is het aantal bits dat wordt gebruikt om het te representeren het minimum dat vereist is in een tekenloos formaat, tenzij het negatief is.
Als u meer controle wilt over het aantal gebruikte bits, kunt u *xd* voor het getal zetten, waarbij *x* het aantal te gebruiken bits is. Zo is *8d14* de decimale waarde 14, gerepresenteerd met 8 bits. U kunt in plaats van *d* ook *h* schrijven voor een hexadecimale notatie (grondtal $_{16}$), of *b* voor binair (grondtal $_{2}$). Bij beide formaten kunt u vóór de letter *h* of *b* het aantal te gebruiken bits specificeren. Als u het aantal bits weglaat, gebruiken hexadecimale getallen standaard 4 bits per cijfer. *h08* gebruikt bijvoorbeeld 8 bits omdat ik twee cijfers heb gebruikt, hoewel voor de waarde 8 eigenlijk slechts 4 bits hadden volstaan.
Voor binaire getallen is het aantal bits (tenzij anders aangegeven) gewoon het aantal cijfers. Dus *b101001* heeft een breedte van 6 bits. Als met *d* een decimaal getal wordt aangegeven maar het aantal bits niet is gespecificeerd, gedraagt het zich alsof ook de *d* ook is weggelaten.

Arrays

Veel signalen die u tegenkomt, zijn multi-bit signalen – zoals *led* in onze top-level module. De bits in een array kunnen individueel worden geïndexeerd met de syntax *signal[bit]*, waar *bit* een uitdrukking is. U moet er zelf voor zorgen dat de waarde altijd binnen de grenzen van de array valt als het een dynamische waarde betreft.
U kunt ook toegang krijgen tot subgroepen van de bits, met behulp van de array-syntaxis *[max: min]*. Hier wordt het bereik van bits beginnend bij min en oplopend tot max geselecteerd (beide inclusief). Bij gebruik van deze syntaxis moeten beide waarden constanten zijn.

Als u dynamisch een subgroep bits wilt selecteren, kunt u de syntaxis *[start +:width]* gebruiken. Hier is *start* het laagste te selecteren bit en is *width* het aantal daarboven liggende te selecteren bits (inclusief het start-bit). Met deze syntaxis hoeft alleen *width* een constante te zijn. U kunt ook op deze notatie variëren: *[start-: width]*. Hier is *start* het hoogste bit van de selectie in plaats van het laagste.
Arrays kunnen in Lucid multidimensionaal zijn. Bij de declaratie plakt u er gewoon op deze manier extra dimensies aan:

```
sig my_array[dim1][dim2][dim3];
```

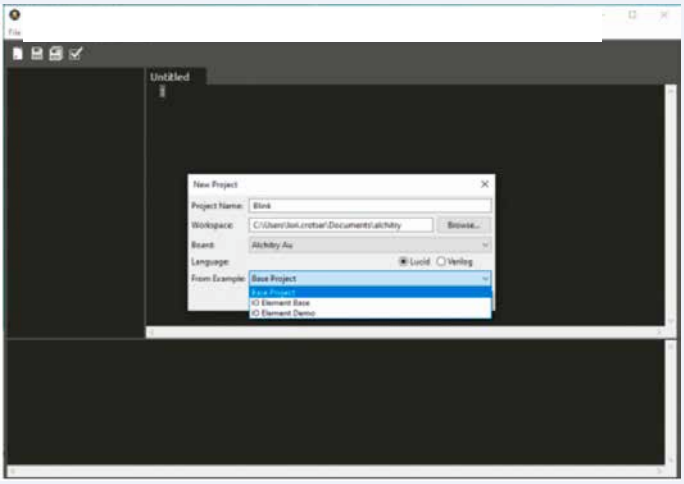
Alle dimensies van een array moeten worden gedeclareerd met constante waarden. U kunt vervolgens het array indexeren met behulp van de selectors zoals eerder. Merk op dat u de subarray-selecties uitsluitend als *laatste* selector kunt gebruiken.

Sequentiële logica en DFF's

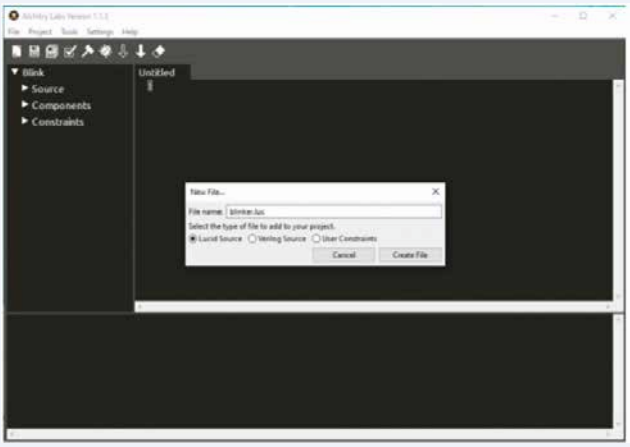
Nu wordt het echt interessant. Combinatorische logica is erg belangrijk, maar een systeem zonder enige toestand of geheugen is vrij beperkt. Maar hoe creëert u geheugen? In wezen hebt u gewoon een soort terugkoppeling nodig. Als u een teller wilt maken, voegt u gewoon één toe aan de uitkomst van de laatste optelling. Het probleem is: hoe beheert u dat? Op het eerste gezicht lijkt dit misschien niet problematisch, maar bij nader inzien blijkt dat we met een doos van Pandora te maken hebben.
Laten we het voorbeeld van onze teller eens bekijken. We zouden die kunnen maken met behulp van een opteller waarvan een van de ingangswaarden de vaste waarde 1 heeft. We doen dit omwille van de eenvoud in een enkel blok (**figuur 3**). Als we de ingang met de uitgang verbinden, verkrijgen we toch een omhoogteller – of niet soms?
Laten we daar eens over nadenken. Bij welke waarde begint deze teller en hoe snel telt hij? De beginwaarde zou afhangen van hoe de schakeling wordt ingeschakeld en hoe die is opgebouwd. Hij hangt waarschijnlijk ook af van de temperatuur en andere omgevingsfactoren. En we willen toch niet dat de werking van de schakeling van het weer afhangt…
En nog erger – deze schakeling zou niet eens werken. Dat komt omdat een optelschakeling, zoals de meeste combinatorische logica met multibit-uitgangen, onjuiste tussenresultaten levert. In het geval van een opteller wordt eerst het minst significante bit berekend, waarna elk volgende bit met behulp van het resultaat van het vorige bit wordt berekend. Aangezien er niets is dat wacht tot het resultaat geldig is, worden de verkeerde tussenresultaten naar de opteller teruggekoppeld, die dan nog meer onjuiste waarden produceert totdat het ding alleen nog maar rommel genereert. Hoe lossen we dat op? We hebben gewoon een manier nodig om de timing van de terugkoppellus te regelen. Dit is waar DFF's (D-flipflops) van pas komen.
Voordat we kijken wat een DFF precies is, moeten we eerst uitleggen wat een klok is. Een klok is gewoon een signaal dat met een bepaalde frequentie telkens van 0 naar 1 schakelt en weer terug, zoals geschetst in **figuur 4**. De klok op de Alchitry-boards heeft een frequentie van 100 MHz (schakelt 100 miljoen keer per seconde om). Dit regelmatige signaal kan worden gebruikt om onze schakelingen een gevoel van tijd te geven. De overgang van 0 naar 1 staat bekend als de opgaande flank, en dat is meestal het belangrijkste deel van het signaal. Deze flanken zijn in de figuur met pijlsymbolen aangegeven.
Terug naar de DFF (**figuur 5**). DFF's zijn een soort geheugen. Ze hebben een ingang (D) en een uitgang (Q). Wanneer de klokingang van 0 naar 1 gaat, wordt de waarde van D opgeslagen en uitgevoerd op Q – tot de volgende opgaande flank van de klok. Het maakt niet uit of D verandert *tussen* twee opgaande flanken van de klok, de waarde op Q blijft hetzelfde. In figuur 5 zien we een DFF met optionele enable- en resetsignalen. Enable kan worden gebruikt om te voorkomen dat de DFF een nieuwe waarde opslaat bij een stijgende flank. Het resetsignaal wordt gebruikt om de Q-waarde naar een bekende waarde te forceren. De DFF's in FPGA's kunnen worden geconfigureerd om na een reset een 0 of 1 op de uitgang te hebben.
We kunnen de DFF in onze teller gebruiken om de lus te besturen (**figuur 6**). Nu kunnen we het resetsignaal gebruiken om de beginwaarde in te stellen, bijvoorbeeld op 0. Dat betekent dat we weten dat Q gelijk aan 0 is. Als Q 0 is, dan is D 1. Op de opgaande flank van de klok krijgt Q de waarde van D. Dat betekent dat Q 1 wordt en dat betekent dat D 2 wordt. Bij elke opgaande flank zal Q met 1 toenemen. Dat is precies wat we wilden! De klokfrequentie bepaalt hoe snel onze teller zal tellen. Daar gelden natuurlijk enkele beperkingen. De klok moet langzaam genoeg zijn zodat de waarde bij D tijd heeft om te actualiseren nadat Q is gewijzigd. We willen niet dat onze DFF een van de ongeldige tussenwaarden opslaat die de opteller produceert. De tijd die nodig is om de uitvoer van de opteller geldig te laten zijn, wordt de *looptijdvertraging* genoemd. Dit is de hoeveelheid tijd tussen een wijziging van de invoer en het geldig en stabiel worden van de uitvoer. Hoe meer logica u toevoegt, hoe langer deze vertraging. De vertraging is ook een functie van de technologie die wordt gebruikt om het IC te fabriceren. De tools hebben modellen voor elke FPGA en als u de gewenste klokfrequentie aangeeft, zullen die tools proberen het ontwerp zo te construeren dat aan de timingvereisten wordt voldaan. In dit voorbeeld hebben we één set DFF's die door een blok combinatorische logica wordt gekoppeld. Het komt vaker voor de uitvoer van een set DFF's door een blok combinatorische logica naar een andere set DFF's loopt zodat een *pipeline* ontstaat.
In elk ontwerp is de langste looptijdvertraging bepalend voor de maximale klokfrequentie. Door uw ontwerp op te splitsen in blokken combinatorische logica met ruwweg gelijke timing, kunt u de klokfrequentie optimaliseren. De exacte timing kan behoorlijk ingewikkeld worden, maar bij de meeste ontwerpen gaat het goed als u dezelfde klok voor het hele ontwerp gebruikt; de tools nemen de rest voor hun rekening zolang er geen onmogelijk lange signaalwegen voorkomen. Bij 100 MHz is er tussen DFF's best wel veel mogelijk. Het wordt meestal pas een probleem wanneer u probeert te veel dingen aan elkaar te koppelen of een heleboel vermenigvuldigingen wilt uitvoeren.
Het voldoen aan de timingvereisten kan ook moeilijk worden wanneer u tegen de grenzen van de resources van de FPGA aan loopt. Omdat de tools steeds meer een FPGA van een bepaalde omvang moeten

Listing 3.

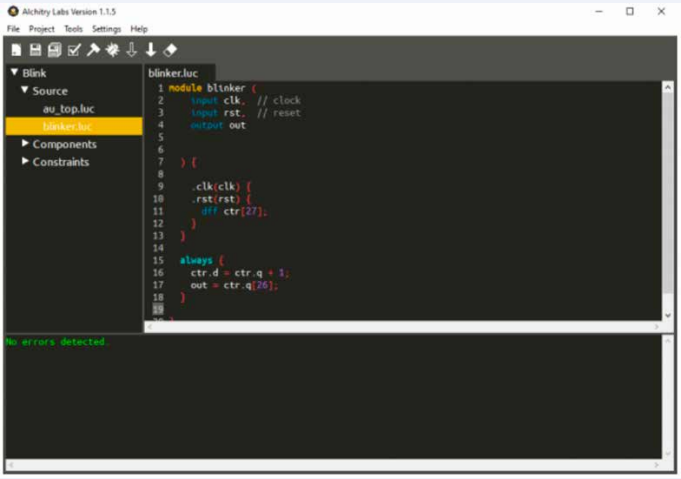
```
module blinker (
   input clk,  // clock
   input rst,  // reset
   output out
  ) {

  always {
    out = 0;
  }
}
```



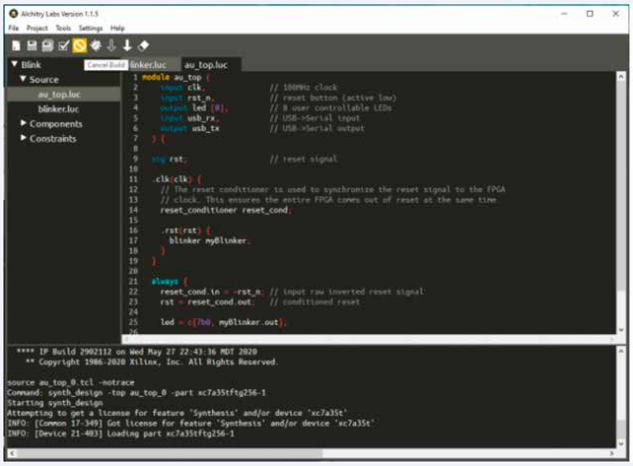
Figuur 7. Selectie van het Base Project in de Alchitry Labs-software.



Figuur 8. Een nieuw Lucid-bronbestand aanmaken.



Figuur 9. De 'module' op basis van het programma in listing 3.



Figuur 10. Uploaden van het project naar het Alchitry-board.

proppen, hebben ze minder mogelijkheden om een optimale layout te kiezen, waardoor eventueel niet meer aan de timingvereisten kan worden voldaan.

Een knipper-LED

Laten we dit alles nu combineren in een demoproject om een LED te laten knipperen. Maak een nieuw project aan in Alchitry Labs [4] en kies Base Project in de keuzelijst From Example (**figuur 7**). Klik op het pictogram New File in de werkbalk (pictogram uiterst links) en maak een nieuw Lucid Source-bestand aan met de naam *blinker.luc* (**figuur 8**). Zo creëert u een basismodule die eruit ziet als **listing 3** (code) en **figuur 9** (op het scherm).

De standaard modulesjabloon voegt klok- en reset-ingangen toe en een uitgang die op dit moment nog niets doet. Om een LED te laten knipperen moeten we een teller maken die de LED in- of uitschakelt. Als we de LED bij elke klokperiode domweg omschakelen, knippert deze veel te snel om te kunnen zien.

We kunnen verbindingsblokken voor de klok en de reset maken; het is dan eenvoudiger om de DFF daarin te declareren:

```
.clk(clk) {
  .rst(rst) {
    dff ctr[27];
  }
}
```

Dit creëert een array van 27 DFF's met de naam *ctr*. We kunnen dan de DFF aansluiten in het *always*-blok.

```
always {
  ctr.d = ctr.q + 1;
  out = ctr.q[26];
}
```

Om toegang te krijgen tot een module of signalen van DFF's gebruiken we de puntnotatie. De eerste regel van het *always*-blok verbindt de D-ingang van de DFF's met de Q-uitgang plus 1. Hierdoor zal de waarde van *ctr.q* elke klokcyclus toenemen. Merk op dat het *d*-signaal alleen-schrijven is en *q* alleen-lezen. De tweede regel neemt het meest significante bit (nummer 26) en verbindt dat met de uitgang. Aangezien *ctr* 27 bit breed is, lopen de indices van 0 tot 26. Aangezien *ctr* eenmaal per klokperiode wordt verhoogd en een 27-bit getal 2^{27} = 134.217.728 verschillende waarden kan aannemen, zal *ctr* bij een klokfrequentie van 100 MHz elke 1,34 seconden een overflow produceren. Gedurende de eerste helft van deze cyclus is het meest significante bit 0 en tijdens de tweede helft 1. Door dit bit aan te sluiten op de uitgang, schakelen we die elke 0,67 seconden om.

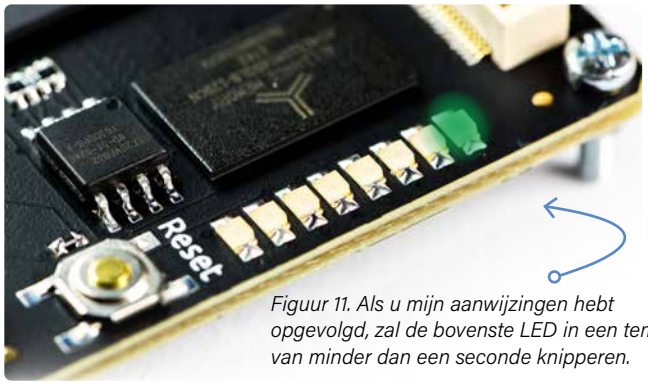
We kunnen nu naar de top-level module gaan en onze nieuwe module instantiëren. Merk op dat we een Au gebruiken; de module zou hetzelfde zijn voor de Cu, alleen de naam zou *cu_top* zou zijn in plaats van *au_top* (zie **listing 4**). Hier hebben we een verbindingsblok voor het resetsig-

Listing 4.

```
module au_top (
    input clk,          // 100MHz clock
    input rst_n,        // reset button (active
low)
    output led [8],     // 8 user controllable
LEDs
    input usb_rx,       // USB->Serial input
    output usb_tx       // USB->Serial output
  ) {

  sig rst;              // reset signal

  .clk(clk) {
    // The reset conditioner is used to
    // synchronize the reset signal to the FPGA
    // clock. This ensures the entire FPGA comes
    // out of reset at the same time.
    reset_conditioner reset_cond;

    .rst(rst) {
      blinker myBlinker;
    }
  }

  always {
    reset_cond.in = ~rst_n; // input raw inverted
reset signal
    rst = reset_cond.out;   // conditioned reset

    led = c;

    usb_tx = usb_rx;        // echo the serial data
  }
}
```

Listing 5.

```
module blinker #(
    MAX_VALUE = 50000000 : MAX_VALUE > 0
  )(
    input clk,  // clock
    input rst,  // reset
    output out
  ) {

  .clk(clk) {
    .rst(rst) {
      dff ctr[$clog2(MAX_VALUE)];
      dff led;
    }
  }

  always {
    ctr.d = ctr.q + 1;
    if (ctr.q == MAX_VALUE-1) {
      ctr.d = 0;
      led.d = ~led.q;
    }

    out = led.q;
  }
}
```



Figuur 11. Als u mijn aanwijzingen hebt opgevolgd, zal de bovenste LED in een tempo van minder dan een seconde knipperen.

naal toegevoegd en een instantie van de knipper-module gemaakt met de naam *myBlinker*. Daarna hebben we deze in het *always*-blok met de *led* uitgang verbonden met behulp van de *concatenation*-syntax. De elementen binnen c *{...}* worden aan elkaar geplakt tot een enkele array. Dus in ons geval combineren we zeven nullen met het *myBlinker.out*-bit. Hierdoor worden de 7 meest-significante LED's uitgeschakeld en wordt ons knippersignaal verbonden met de eerste LED. U kunt het project nu bouwen door op het hamer-pictogram te klikken en het vervolgens in uw board te laden door op het 'gevulde' pijl omlaag-pictogram te klikken (**figuur 10**). De bovenste LED moet nu iets langzamer dan één keer per seconde knipperen (**figuur 11**).

Verbeteringen aan de module

We kunnen onze knipper-LED module aanzienlijk verbeteren en nuttiger maken. Ten eerste knippert de LED met een onhandige frequentie. We kunnen hier precies een seconde van maken door tot 50.000.000 te tellen en dan de LED om te schakelen. Hiervoor hebben we nog een DFF nodig om de status van de LED op te slaan.

```
.clk(clk) {
  .rst(rst) {
    dff ctr[28];
    dff led;
  }
}
```

In het *always*-blok kunnen we nu controleren of *ctr.q* de waarde 49.999.999 heeft (we tellen immers van 0 tot 49.999.999) en als dat zo is, kunnen we resetten naar 0 en de LED schakelen.

```
always {
  ctr.d = ctr.q + 1;
  if (ctr.q == 49999999) {
    ctr.d = 0;
    led.d = ~led.q;
  }

  out = led.q;
}
```

Hier gebruikten we de bitgewijze inversie-operator, de tilde: ~. Hiermee wordt elk bit van een signaal geïnverteerd. Omdat *led.q* maar één bit breed is, wordt het bit gewoon omgekeerd.
Eerder hebben we opgemerkt dat een signaal onder alle omstandigheden een waarde moet krijgen, met uitzondering van DFF's. Aangezien DFF's hun waarde daadwerkelijk kunnen opslaan , zal de waarde van de DFF niet veranderen als u de *d*-input niet toewijst tijdens een klokcyclus.

Surf mee op de FPGA-golf

Het web is de ideale plek om meer te weten te komen over FPGA's. Hier een paar links om u op weg te helpen of om uw interesse te wekken.

- SparkFun en Alchitry, "Using FPGA's", www.sparkfun.com/fpga
- J. Rajewski, "How Does an FPGA Work?", SparkFun, https://learn.sparkfun.com/tutorials/how-does-an-fpga-work
- J. Rajewski, "First FPGA Project: Getting Fancy with PWM", SparkFun, www.elektormagazine.nl/esfe-en-fpga2
- J. Rajewski, "External IO ans Metastability", SparkFun, https://learn.sparkfun.com/tutorials/external-io-and-metastability
- S. Cass, "Painless FPGA Programming", IEEE Spectrum, november 2020, https://spectrum.ieee.org/geek-life/hands-on/peless-fpga-programming

Onze module slaat nu alleen de waarden 0...49.999.999 in *ctr* op, maar het is nog steeds een 28-bit array. Dit is een verspilling aangezien we slechts 26 bits nodig heben om deze waarden op te slaan. We kunnen de array-grootte natuurlijk gewoon wijzigen in 26, maar als we de maximale waarde willen wijzigen, moeten we deze waarde elke keer opnieuw berekenen. In plaats daarvan kunnen we Lucid-functies gebruiken om dit voor ons te berekenen. We kunnen *$clog2(50000000)* gebruiken (de ceiling-functie) – anders uitgedrukt: "hoeveel bits zijn nodig om zoveel combinaties op te slaan". In ons geval komen we uit op 26.

```
dff ctr[$clog2(50000000)];
```

Over het wijzigen van de maximale waarde gesproken, we kunnen onze module zo wijzigen dat deze als parameter wordt geaccepteerd, zodat deze kan worden gespecificeerd wanneer de module wordt geïnstantieerd. We kunnen dit doen door een *parameter list* aan onze module toe te voegen. Deze komt voor de *port list*.

```
module blinker #(
    MAX_VALUE = 50000000 : MAX_VALUE > 0
  )(
    input clk,  // clock
    input rst,  // reset
    output out
  ) {
```

De parameter list wordt gespecificeerd met de syntax #(param, param, param). Elke parameter kan een simpele naam zijn (in hoofdletters), of zo complex als in ons voorbeeld waar we een defaultwaarde van 50.000.000 instellen. Defaultwaarden zijn doorgaans een goed idee, tenzij u een waarde bij het instantiëren wilt forceren.
Na de default-toewijzing kunt u een voorwaarde specificeren waaraan de parameter moet voldoen. Deze voorwaarde kan (en moet) de parameter zelf gebruiken, evenals alle parameters die ervoor zijn gedeclareerd. Deze voorwaarde moet worden geëvalueerd als 'waar'. Als dat niet het geval is, verschijnt bij het instantiëren een foutmelding. Op deze manier kunt u uw module schrijven met enkele aannames over de parameterwaarde in de zekerheid dat deze zullen worden nageleefd. We kunnen dan overal in onze module "50.000.000" vervangen door MAX_VALUE zoals in **listing 5**.
Als u het project nu bouwt en laadt, moet de LED in een tempo van één keer per seconde knipperen. Als u echter teruggaat naar de top-level module, kunt u het instantiëren als volgt wijzigen:

```
blinker myBlinker(#MAX_VALUE(25000000));
```

Als u het project nu bouwt en laadt, knippert de LED twee keer per seconde.

Tot slot

Daarmee komen we aan het einde. FPGA-ontwerpen bestaan uit blokken van combinatorische logica die alle bewerkingen uitvoeren, en DFF's die waarden opslaan en de gegevensstroom regelen. De ontwerpen zelf zijn onderverdeeld in modules. Modules kunnen door andere modules worden gebruikt en kunnen zelfs parameters hebben om elke instantie ervan aan te passen. Elke keer dat een module wordt geïnstantieerd, wordt de betreffende schakeling gedupliceerd in de FPGA.
Bij het ontwerpen van hardware is het belangrijk om na te denken over hoe dat ontwerp in de vorm van een efficiënte schakeling kan worden geïmplementeerd. In het geval van onze knipper-LED zou (als de knipperfrequentie ons niet interesseerde) de eerste versie van de module bij de implementatie veel minder resources verbruiken. Dat komt omdat er dan geen comparator nodig is om de waarde van de teller te controleren. Er wordt gewoon steeds 1 opgeteld en de 'natuurlijke' overflow van de binaire optelling wordt gebruikt om de teller te resetten. ▶|

200646-04

Alchitry en Justin Rajewski

Met een forse boost van Kickstarter is Alchitry een van de meest deskundige en uitgebreide resources voor FPGA-technologie. Het SparkFun- team was in de wolken met de Alchitry Au- en Alchitry Cu-boards en werkt nauw samen met Justin Rajewski, oprichter en manus-van alles-ingenieur. Tegenwoordig richt Justin zich op de ontwikkeling nieuwe producten, de bouw van projecten en het genereren van content; hij heeft bij SparkFun de productie en logistiek opgekrikt om zijn boards te produceren. Bij SparkFun brengt Justin een ongeëvenaarde FPGA-expertise in.

WEBLINKS

[1] SparkFun, "Using FPGAs" : https://www.sparkfun.com/fpga
[2] Wikipedia, "Hardware Description Language" : https://en.wikipedia.org/wiki/Hardware_description_language
[3] Lucid: https://alchitry.com/pages/lucid-fpga-tutorials
[4] Alchitry Labs: https://alchitry.com/blogs/tutorials/getting-started-with-the-au
[5] Alchitry Au board: https://www.elektormagazine.nl/esfe-en-fpga1
[6] Alchitry Cu board: https://www.sparkfun.com/products/16526