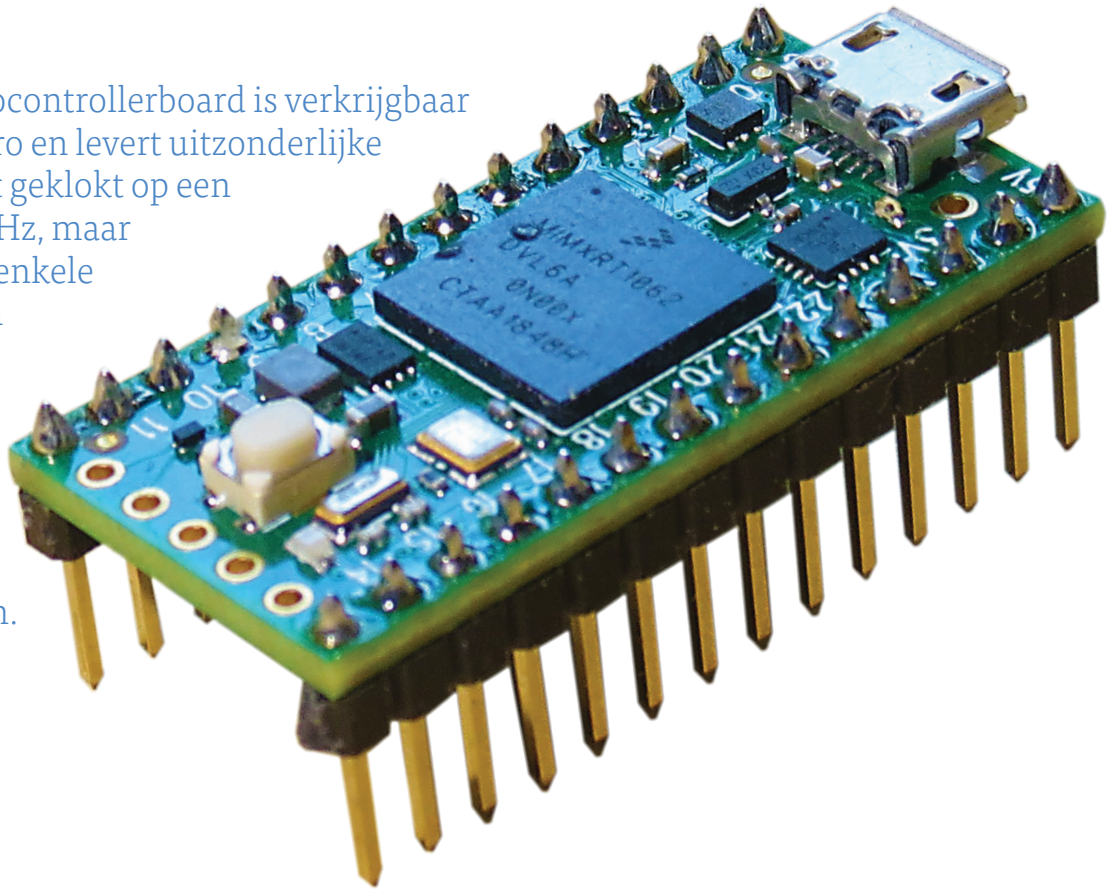
Teensy 4.0 -
waarom is dat board zo snel?
snelheid is geen hekserij!

Prof. Dr. Martin Oßmann (Duitsland)

Het Teensy 4.0 microcontrollerboard is verkrijgbaar voor ongeveer 20 euro en levert uitzonderlijke prestaties. Het wordt geklokt op een opmerkelijke 600 MHz, maar dat is niet alles. Met enkele experimenten willen we onderzoeken aan welke eigenschappen en maatregelen de hoge prestaties van de Teensy-boards te danken zijn.



De toegepaste processor IMX-RT1062 van NXP is een ARM Cortex-M7 CPU, waarvan de architectuur dichter bij PC-processoren staat dan bij AVR-microcontrollers. We gebruiken Teensyduino als programmeeromgeving (die is grotendeels compatibel met Arduino). In dit artikel gaan we echter ook (deels) programmeren met C of inline-assembler.

Toggle

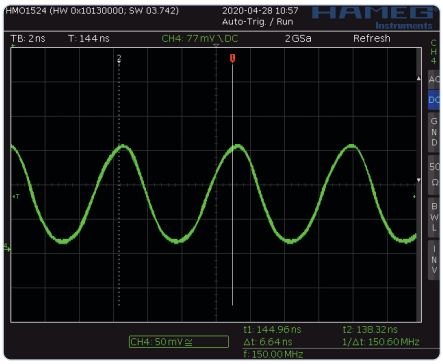
Het eerste dat we gaan doen, is een pin zo snel mogelijk omschakelen. Ter vergelijking doen we dit eerst met een Arduino Nano die geklokt wordt op 16 MHz. Alleen al vanwege de kloksnelheid is de Teensy 600/16 = 37,5 keer zo snel. Om de LED op pin 13 van de Arduino te schakelen, kunnen we eigenlijk volstaan met het programma van **listing 1**. Het was de bedoeling om de LED telkens één microseconde in en uit te schakelen. De LED is echter 3,5 µs aan en 3,9 µs uit. Dat heeft twee redenen. Enerzijds vergen de `digitalWrite()`-commando's relatief veel tijd (2,5 µs), anderzijds voert de Arduino eigen taken tussen de aanroepen van `loop()`, wat ongeveer 0,4 µs duurt. Om sneller te kunnen schakelen, kunt en moet u dichter bij de hardware programmeren, zoals in **listing 2**.

Met dit programma duurt een lus-doorloop slechts 2,66 µs, wat overeenkomt met zes cycli bij een klokfrequentie van 16 MHz. Om uit te leggen hoe deze 6 cycli verlopen, kunt u de assembler-**listing 3** bekijken.

De lus bestaat uit drie commando's. Met `sbi` en `cbi` wordt het LED-bit geset of gewist. Het `rjmp`-commando voert naar de gewenste oneindige lus. De tijdsduur van de afzonderlijke instructies is te vinden in de documentatie van de AVR-controller. Elke instructie duurt twee klokcycli, wat resulteert in de waargenomen timing.

Het omschakelen kan nauwelijks sneller worden geprogrammeerd; ter vergelijking zullen we zien wat de maximale snelheid van de Teensy 4.0 is.

Eerst programmeren we opnieuw in



Figuur 1. Pin-toggle op 150 MHz.

'Arduino-stijl' (**listing 4**). Eén cyclus van de oneindige lus duurt dan 135 ns. Vergeleken met de Arduino is dit in absolute zin relatief snel, maar een aantal van 18 cycli is toch relatief veel. De uitvoering van een `digitalWrite` vergt weer relatief veel tijd en het interne 'verlies' tussen de aanroepen van `loop` lijkt ook veel te zijn.

Daarom programmeren we het geheel nu op hardware-niveau (**listing 5**). De cyclustijd van de `while`-lus duurt nu 6,66 ns, dat zijn vier cycli.

De spanning op de LED-uitgang ziet er uit als het oscillogram van **figuur 1**. Omdat het signaal een frequentie heeft van 150 MHz, gebruikt het de volledige bandbreedte van de oscilloscoop en lijkt het meer op een sinus dan op een blokgolf. Om de aan- en uittijden van de LED nauwkeurig te kunnen meten, werd de CPU-klok gereduceerd tot 100 MHz. De aan- en uit-tijd van de LED bedragen elk twee cycli (overeenkomend met 20 ns bij een 100MHz-klok).

Om te zien welke instructies de CPU in deze lus uitvoert, werpen we een blik op assembler-**listing 6**. De lus bestaat uit drie instructies (zoals bij de Arduino). Twee word store-instructies (`str.w`) zetten of wissen bit 3. De lus wordt afgesloten door het branch-instructie (`b.n`). Nu zou het interessant zijn om te weten hoeveel cycli de individuele instructies nodig hebben, maar de ARM-documentatie geeft hierover geen uitsluitsel. En met reden, want het aantal cycli hangt van veel omstandigheden af. Kijk dus liever naar de *overall*-prestaties. Maar we zijn hardnekkig en willen het voor dit voorbeeld beslist weten. Het lijkt erop dat het uitvoeren van het str-commando hier twee cycli in beslag neemt. Dit kunnen we bevestigen met **listing 7**.

Elk extra `set`/`clr`-paartje verandert de frequentie van het signaal op pin 13 niet. Dit

Listing 1. Pin-toggle op de Arduino-manier.

```
void setup() {
   pinMode(led, OUTPUT);
   }
void loop() {
   digitalWrite(led, HIGH); //3.5 us high time
   delayMicroseconds(1);
   digitalWrite(led, LOW); // 3.9 us low time
   delayMicroseconds(1);
   }
```

Listing 2. Pin-toggle op hardware-niveau bij de Arduino AVR.

```
#define ledBit 5
 #define ledDDR DDRB
 #define ledPORT PORTB
void setup() {
   cli() ;
   ledDDR |= _BV(ledBit)    ; set output
   while(1){                ; loop time 2.6648 MHz = 6 cycles
     ledPORT |= _BV(ledBit) ; 2 cycles on
     ledPORT &= ~_BV(ledBit); 4 cycles off
     }
   }
```

Listing 3. Pin-toggle: assemblerlisting.

```
342: f8 94   cli            ; cli() ;
 344: 25 9a   sbi 0x04, 5    ; ledDDR |= _BV(ledBit) ;
                             ; while(1){ // 2.6648 MHz = 6 cycles
 346: 2d 9a   sbi 0x05, 5    ;   ledPORT |= _BV(ledBit) ;
 348: 2d 98   cbi 0x05, 5    ;   ledPORT &= ~_BV(ledBit) ;
 34a: fd cf   rjmp  .-6      ; jump to 0x346
```

Listing 4. Teensy programmeren in 'Arduino-stijl'.

```
int led = 13;

 void setup() {
   pinMode(led, OUTPUT);
   }
void loop(){
   digitalWrite(led,1) ;
   digitalWrite(led,0) ;
   }
```

Listing 5. Pin-toggle op de Teensy op hardware-niveau.

```
void setup() {
   pinMode(13, OUTPUT);
   cli() ;
   while(1){  // cycleTime 150 MHz = 4 cycles
     CORE_PIN13_PORTSET = CORE_PIN13_BITMASK ;  // on: 2 cycles
     CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK; // off 2 cycles
     }
   }
void loop(){
   }
```

Listing 6. Snel toggelen met assembler-instructies.

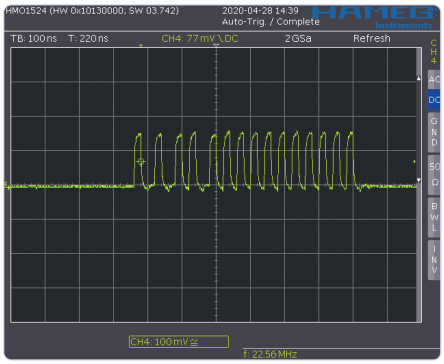
```
8c:    f8c2 3084     str.w    r3, [r2, #132]  ; set GPIO pin 13
 90:    f8c2 3088     str.w    r3, [r2, #136]  ; clear GPIO pin 13
 94:    e7fa          b.n      8c <setup+0x10> ; branch endless loop
```

Listing 7. Een langere reeks instructies.

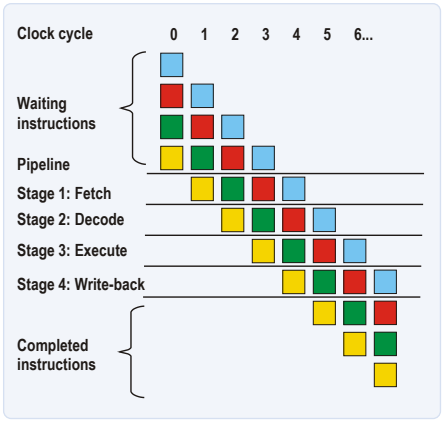
```
void setup() {
   pinMode(13, OUTPUT);
   cli() ;
   while(1){  // cycleTime 150 MHz = 4 cycles
     CORE_PIN13_PORTSET = CORE_PIN13_BITMASK ;
     CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK;
     CORE_PIN13_PORTSET = CORE_PIN13_BITMASK ;
     CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK;
     CORE_PIN13_PORTSET = CORE_PIN13_BITMASK ;
     CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK;
     . . . jeweils ein set/clear Paar
     }
   }
void loop(){
   }
```

isting 8. Eindige toggle-lus.

```
while(1){
   for(int k=0 ; k<15 ; k1--){
     CORE_PIN13_PORTSET = CORE_PIN13_BITMASK;
     CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK;
     }
   delay(100) ;
   Serial.println("test5\n") ;
   }
```



Figuur 2. De LED-pin schakelt ongelijkmatig.



Figuur 3. Viertraps pipeline.

betekent dat een `set`/`clr`-paar de lus met vier cycli verlengt. In het oorspronkelijke voorbeeld komen de beide `str`-opdrachten uit op vier cycli, wat de totale duur is van de `while`-lus. Er lijkt geen extra CPU-tijd nodig te zijn voor het uitvoeren van de sprongin-structie. Dit lijkt aanvankelijk verbazing-wekkend, maar we zullen de reden ervoor verderop leren kennen. Hiermee is het Teensy-board met vier cycli inderdaad sneller dan de AVR met zes cycli.

We gaan de verwarring nog wat groter maken door de while-lus zoals in **listing 8** uit te voeren. We draaien het programma opnieuw met een klokfrequentie van 100 MHz, zodat de spanning op de LED-pin er een beetje als een blokgolf uit ziet. Het programma schakelt de LED-pin 15 keer om, pauzeert vervolgens en voert een tekst uit. Dan begint alles opnieuw.

De spanning op de LED-pin ziet er dan uit als in **figuur 2**. Opmerkelijk genoeg is de aan/uit-verhouding niet constant. In het begin lijkt de lus soms langer en soms korter te duren. De instructies lijken geen constante uitvoeringstijd te hebben. Tegen het einde zien we een lus-cyclustijd van vier cycli, precies zoals hierboven voor de oneindige lus. Dit is des te verbazingwekkender omdat de lus nu nog een instructie meer bevat, namelijk `k1--`. Die lijkt echter geen extra tijd nodig te hebben, zodat hij geen invloed heeft op de tijd die de lus in beslag neemt.

Sprongvoorspelling

De zogenaamde sprongvoorspelling met speculatieve instructie-uitvoering geeft een eerste verklaring voor de hoge lussnelheid. Om dit te begrijpen, moet men precies begrijpen hoe een CPU de instructies uitvoert. Een eerste techniek om de snelheid te verhogen is het gebruik van instructie-pipelines. De opdrachten die op dit moment worden uitgevoerd, bevinden zich in een pipeline. Elke opdracht wordt in verschillende fasen verwerkt. Als instructie 1 bijvoorbeeld fase 4 uitvoert, voert instructie 2 fase 3 uit, voert instructie 3 fase 2 uit en instructie 4 fase 1 uit (viertraps pipeline zoals in **figuur 3**). De instructies komen vanuit het geheugen in de pijplijn en de leeseenheid van het geheugen moet ervoor zorgen dat de pipeline altijd goed gevuld is. Hiertoe haalt hij altijd de volgende instructies en zet die in de pipeline.

Nu zijn er situaties waarin de inhoud van de pipeline 'weggegooid' moet worden omdat ze incorrect zijn. Dit is bijvoorbeeld het geval bij (voorwaardelijke) sprongen wanneer de lineaire commandoreeks wordt verlaten. Zo'n zogenaamde *pipeline-stall* kan ook ontstaan wanneer de fasen van een pipeline afhangen van een resultaat dat een andere fase eerst moet berekenen. Om ervoor te zorgen dat de pipeline correct wordt gevuld bij voorwaardelijke sprongen, moet de CPU weten of een sprong al dan niet wordt uitgevoerd. Hij kan dat niet

zeker weten, maar hij kan proberen dat te voorspellen. In ons voorbeeld (**listing 9**) wordt de sprong 14 keer uitgevoerd en één keer niet. De CPU observeert nu wat de sprong doet tijdens de eerste keren dat de lus wordt doorlopen. De CPU merkt dat de sprong altijd als eerste wordt uitgevoerd en gaat er daarom van uit dat de volgende sprong altijd wordt uitgevoerd.

De pipeline wordt daarom altijd geladen met de `str`-instructies na de spronginstructie. Dan heeft de CPU aanvankelijk altijd gelijk, maar niet aan het einde waar een *stall* plaatsvindt. Deze techniek wordt sprongvoorspelling met speculatieve instructie-uitvoering genoemd. Omdat de pipeline meestal correct is gevuld, kost de spronginstructie als het ware geen CPU-tijd meer. We hebben hiermee het gedrag van het programma volgens listing 9 verklaard. Aan het begin is de sprongvoorspelling nog niet correct, maar aan het einde leidt deze tot de extreem snelle lusuitvoering. Aan de andere kant is een exacte berekening (in cycli) van de uitvoeringstijd van een stuk code bij een sprongvoorspelling erg moeilijk omdat het gedrag van de voorspelling van veel zaken afhangt. Dit maakt al duidelijk waarom er geen cyclustijden voor individuele instructies zijn gespecificeerd bij ARM-processoren van het M7-type. Een goede sprongvoorspelling is een cruciaal architectonisch kenmerk van krachtige CPU's. Aangezien moderne CPU's vaak relatief omvangrijke pipelines hebben, is een *stall* behoorlijk 'kostbaar' en is een correcte sprongvoorspelling essentieel. Moderne *branch-predictors* zijn in 98% van de gevallen correct! Kleinere processoren en microcontrollers (bijvoorbeeld AVR-controllers) hebben vaak geen sprongvoorspelling terwijl een pipeline zo ongeveer tot de standaarduitrusting behoort.

Laten we nu eens kijken naar de timing van enkele meer gecompliceerde codefragmenten. Ons eerste programma berekent in een lus een scalair product (inwendig product, **listing 10**). De daadwerkelijke berekening wordt uitgevoerd met de instructie `skp += x[k]*y[k];`. Voor dit statement hebben we LED-pin 13 geset. Daarna resetten we de pin weer. De aan-tijd van pin 13 zou dus precies gelijk moeten zijn aan de uitvoeringstijd van ons test-statement.

Vreemd genoeg zien we echter dat de inschakeltijd slechts twee cycli bedraagt. Die tijd is alleen nodig voor het instellen van de GPIO-pin, zodat er niets overblijft voor ons test-statement. Iets verloopt dus anders dan we denken.

Laten we dit tot op de bodem uitzoeken. We werpen daartoe een blik op de corresponderende assembler-**listing 11**.

Interessant genoeg staat er slechts één instructie tussen de port-instructies. De compiler heeft de variabelen-operaties `x[k1++]` en `y[k2++]` vóór de port set-instructie geplaatst. Dus door het LED-signaal te meten kunnen we de timing die ons interesseert niet te weten komen. De compiler genereert gewoon niet altijd de code die we willen. Instructies worden soms anders gerangschikt wanneer dit in C correct is en hogere prestaties oplevert. Bij het meten van de tijd moeten we ervoor zorgen dat de compiler de instructievolgorde niet ongevraagd verandert.

Overigens kunnen we het totale aantal verstreken cycli zien in de `ARM_DWT_CYCCNT`-tag en deze gebruiken om de timing te meten. In ons voorbeeld meet ik zo de totale lustijd (0...1000). Het zo bepaalde aantal cycli gedeeld door 1000 is dan het aantal cycli per lus. In dit geval is de lus zeven cycli lang. Vervolgens willen we

Listing 9. De eindige toggle-lus in assembler-code.

```
ca: 230f      movs     r3, #15           ; k1=15
 cc: 3b01      subs     r3, #1            ; k1--
 ce: f8c5 4084 str.w    r4, [r5, #132]    ; set pin 13
 d2: f8c5 4088 str.w    r4, [r5, #136]    ; clear pin 13
 d6: d1f9      bne.n    cc <test5()+0x10> ; if !=0 springe nach cc
```

Listing 10. Scalair product als testprogramma.

```
cyclesStart = ARM_DWT_CYCCNT ;
 skp=0 ;
 for(int k=0 ; k<nn ; k++){
   CORE_PIN13_PORTSET = CORE_PIN13_BITMASK;
   skp += x[k]*y[k] ;
   CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK;
   }
 cyclesStop = ARM_DWT_CYCCNT ;
```

Listing 11. Assembler-listing voor het scalair product.

```
for(int k=0 ; k<nn ; k++){

a0: ecf3 6a01 vldmia    r3!,      ; x[k1++]
 a4: ecb1 7a01 vldmia    r1!,       ; y[k2++]
 a8: 42a3      cmp       r3, r4          ; k1==1000 ?
 aa: f8c2 0084 str.w     r0, [r2, #132]  ; set pin 13
 ae: eee6 7a87 vfma.f32  s15, s13, s14   ; skp += x[k]*y[k] ;
 b2: f8c2 0088 str.w     r0, [r2, #136]  ; clear pin13
 b6: d1f3      bne.n     a0              ; branch on not equal
```

Listing 12. Optelling van 100 functiewaarden.

```
int NN=100 ;
CORE_PIN13_PORTSET = CORE_PIN13_BITMASK;
 sum=0 ;
 for(int k=0 ; k<NN ; k++){
   sum += fun1(k) ;
   }
 CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK;

 int fun1(int x){
   return 100*x+x*x+32 ;
   }
```

Listing 13. Assembler-code van de optellus.

```
1ea: f8c6 3084 str.w  r3, [r6, #132]     ; set pin13
 1ee: 4602      mov    r2, r0
 1f0: f8c5 9000 str.w  r9, [r5]          ; sum=0x000C9CB6H
 1f4: f8c6 3088 str.w  r3, [r6, #136]    ; clear pin13
```

Listing 14. Een complexere lus.

```
void loop(){
   int xx=42 ;
   int nn=256 ;
   int k ;
   int m=0 ;
   int vv=0 ;
   while(1){  // 5 cycles, 9 cycles if dualIssueDisabled
     cyclesStart = ARM_DWT_CYCCNT ;
     for( k=0 ; k<nn ; k++){
       CORE_PIN13_PORTSET = CORE_PIN13_BITMASK; // led-1
       CORE_PIN13_PORTCLEAR = CORE_PIN13_BITMASK; // led-2
       xx *= 105529 ;
       vv += m & 0x1234 ;
       m +=17 ;
       }
     cyclesStop = ARM_DWT_CYCCNT ;
    ...
```

Listing 15. Assembler-code van listing 14.

```
// r9=105529 ; r3=m ; r5=vv ; r6=xx

1d8: f241 2234  movw  r2, #4660       ; r2=0x1234
 1dc: f8c8 7084  str.w r7, [r8, #132]  ; set pin 13
 1e0: fb09 f606  mul.w r6, r9, r6      ; xx *= 105529 ;
 1e4: 401a       ands  r2, r3          ; r2= m & 0x1234 ; r2=1234h r3=m
 1e6: 3311       adds  r3, #17         ; m +=17 ;
 1e8: f8c8 7088  str.w r7, [r8, #136]  ; clear pin 13
 1ec: 4299       cmp   r1, r3          ; abbruchbedingung r1 <> m
 1ee: 4415       add   r5, r2          ; r5 += m & 0x1234 ; r5=vv
 1f0: d1f2       bne.n 1d8 <loop+0x1c> ; loop weitermachen
```

Listing 16. Dynamisch toegewezen geheugen in het RAM2-gebied van de Teensy.

```
uint8_t *RAM2buffer ;
 RAM2buffer=(uint8_t *)malloc(NN) ;
cyclesStart = ARM_DWT_CYCCNT ;
   int sum=0 ;
     for(int k=0 ; k<NN ; k++ ){
        sum += RAM2buffer[k] ;
        }
   cyclesStop = ARM_DWT_CYCCNT ;
```

de totale tijd meten die nodig is om de lus (k = 0…100) in het programma van **listing 12** uit te voeren. Daartoe hebben we port set-instructies voor en na de lus gezet. De functiewaarden `fun1(k)` worden in de lus opgeteld.

Verbazingwekkend genoeg meten we een tijd van 5 ns (komt overeen met 3 cycli) voor de hele lus . Maar dat kan niet. Een blik op assembler-**listing 13** levert de verklaring.

Compiler-geoptimaliseerd

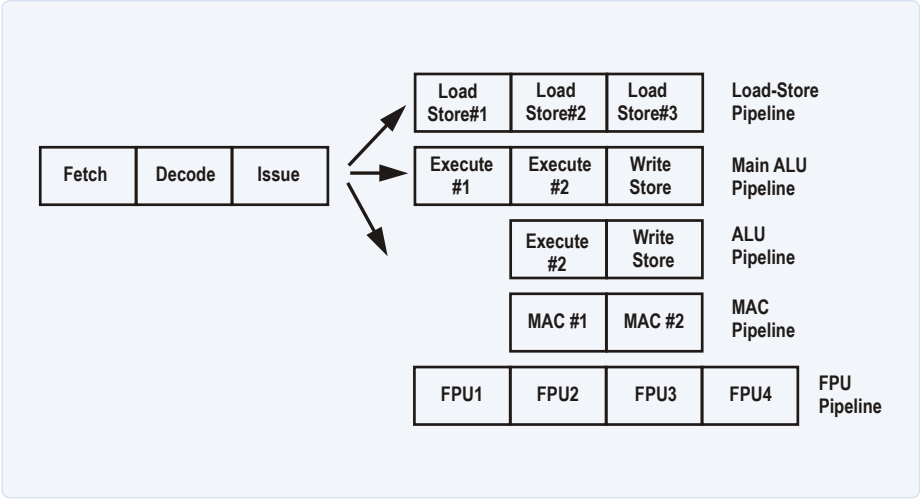
De compiler heeft tussen de twee port set-instructies slechts twee simpele instructies geplaatst en niet een hele lus. Die lus komt in de assembler-code niet voor. Als we dit nauwkeuriger analyseren, zien we dat de compiler de lus heeft geëlimineerd. Die is vervangen door de toewijzing `sum=826550` (= 0x000C9CB6H). De compiler heeft de waarde van tevoren berekend en 100 doorlopen vervangen door een toewijzing. Hieruit blijkt dat optimaliserende compilers inmiddels ook relatief complexe code effectief kunnen optimaliseren. De uitgevoerde code kan daarom op details verschillen van de broncode. Nogmaals: we moeten voorzichtig zijn bij het meten van de tijd, om te voorkomen dat we appels met peren vergelijken.

Laten we nu de lus van **listing 14** uitvoeren. Het Teensy-board heeft vijf cycli per doorloop nodig.

Dat lijkt relatief kort. We bestuderen daarom assembler-**listing 15**. Daar zien we dat de lus negen instructies lang is. Het Teensy-board voert de negen instructies uit in vijf cycli, wat in eerste instantie verbazingwekkend lijkt. Deze eigenschap wordt bereikt omdat de Teensy-processor een 'superscalaire CPU met dual issue' is. In het geval van een superscalaire CPU zijn meerdere functionele eenheden (optellers, …) meerdere keren aanwezig zodat de CPU meerdere subtaken uit meerdere instructies tegelijk kan uitvoeren. 'Dual issue' betekent in deze context dat twee instructies uit het eerste deel van de FIFO gelijktijdig worden overgedragen aan de superscalaire eenheden.

De pipeline ziet er dan uit als in **figuur 4**. Vanaf de uitgifte-eenheid kunnen twee instructies tegelijkertijd de volgende units bereiken. Als we de 'dual issue'-techniek uitschakelen, heeft onze bovenstaande lus negen cycli nodig voor negen instructies, wat bijna twee keer zo lang is als bij dual issue. Het voorbeeld laat zien hoe succesvol en belangrijk dergelijke optimalisatietechnieken zijn.

Nu gaan we de prestaties bekijken in een realistische applicatie, namelijk de snelle Fourier-transformatie FFT. We beginnen met een FFT van 128 punten. Op de Arduino duurt deze routine ongeveer 50 ms, op de Teensy 4.0 slechts 23 µs. Dat is 50.000 µs/23 µs = 2200 keer sneller!



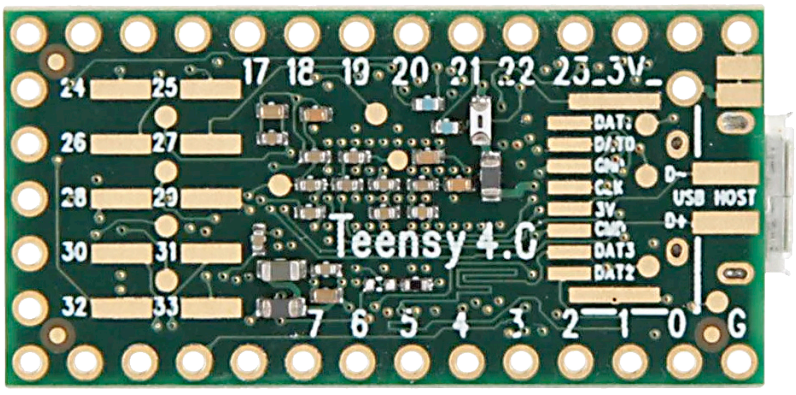
Figuur 4. Dual issue-pipeline.

CONCEPTEN IN DE IMXRT1062-CPU

- Hoge kloksnelheid
- Optimaliserende compiler
- Speculatieve instructie-uitvoering
- Sprongvoorspelling
- RISC-structuur
- Pipeline
- Superscalaire architectuur
- Dual issue
- Instructie-cache
- Datacache
- Floating Point Unit

De kloksnelheid is slechts voor een factor 600 MHz/16 MHz = 37 verantwoordelijk, de rest (2200/37 = 60) is te danken aan de bij de Teensy voorhanden floating-point unit en andere architectuur-eigenschappen. Aangezien de verschillende architectuur-trucs (sprongvoorspelling, dual issue, ...) ook uitgeschakeld kunnen worden, is het mogelijk om nauwkeuriger te bepalen hoeveel invloed ze hebben.
De sprongvoorspelling levert een snelheidswinst op van 21%, het dual issue-concept 41%. Hieruit blijkt dat twee instructies relatief vaak superscalair worden verwerkt. In ons voorbeeld levert de instructiecache geen snelheidswinst. Maar dat komt omdat de code zich in een snel geheugen bevindt. Als u de code in het flash-geheugen laadt en caching uitschakelt, wordt de routine een factor 7 langzamer. Als de code in flash staat, levert de instructiecache een factor 7 op. Maar aangezien de gegevens zich in het snelle RAM bevinden, veroorzaakt de datacache geen verdere verbetering. Geheugengebieden die dynamisch worden toegewezen via `malloc()` bevinden zich in een langzamer geheugengebied (RAM2). Als u daar data gebruikt en de cache uitschakelt, wordt het programma (**listing 16**) een factor 7 langzamer omdat RAM2 wordt geleverd door een tragere bus.
Hiermee eindigen we onze rondreis langs enkele concepten voor prestatieverhoging. In het **kader** worden de bij Teensy gerealiseerde concepten samengevat. Misschien onderzoekt iemand hetzelfde voor de Raspberry Pi, waarvan de nieuwe versies geklokt zijn op 1,5 GHz. Er zijn nog enkele concepten zoals out of order execution, paging memory management unit, L2- en L3-caches, simultaneous multithreading of register renaming die kunnen worden aangetroffen in PC-achtige CPU's waarmee de prestaties nog verder worden opgevoerd.

200190-04



GERELATEERDE PRODUCTEN

- **Teensy 4.0 Development Board**
  www.elektor.nl/19066
- **Teensy 4.1 Development Board**
  www.elektor.nl/19311

Vragen of opmerkingen?
Hebt u vragen of opmerkingen naar aanleiding van dit artikel?
Stuur een e-mail naar de auteur of naar de redactie van Elektor, via redactie@elektor.com.