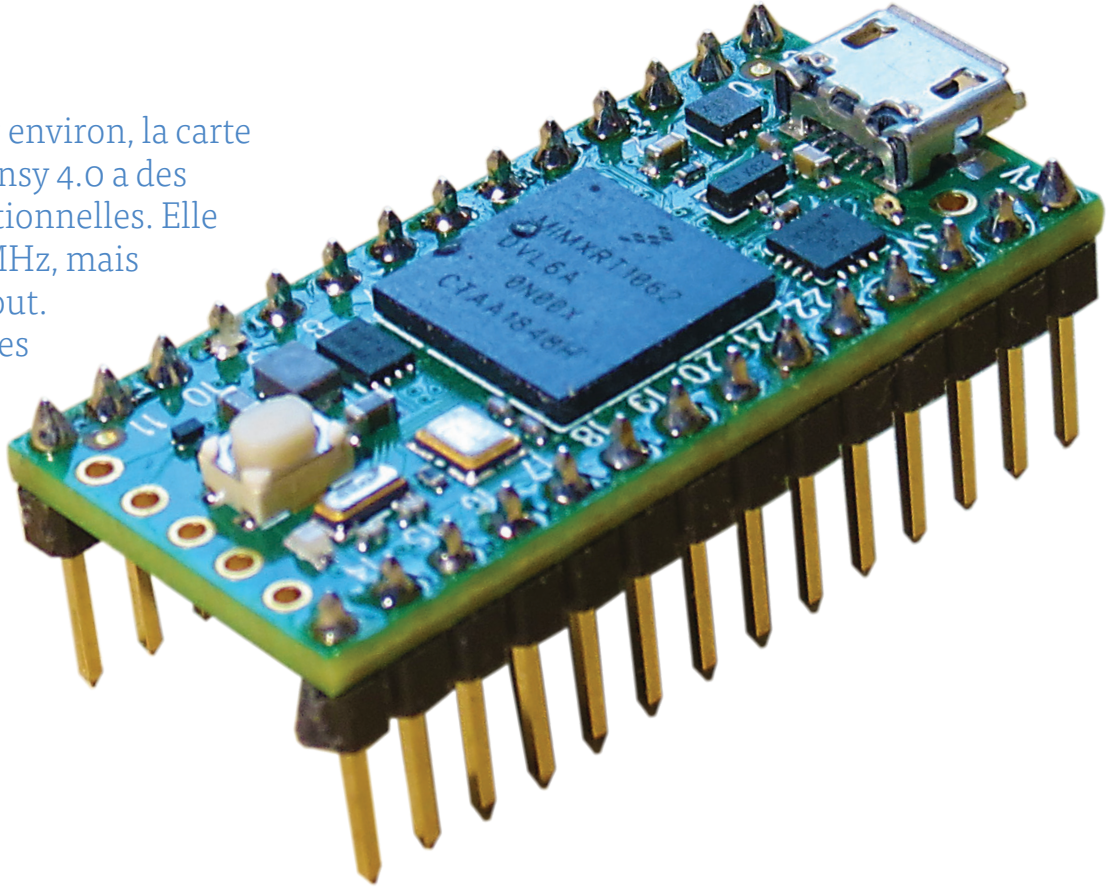
Teensy 4.0 – comment cette carte peut-elle être aussi rapide ?

Non, la vitesse, ce n'est pas sorcier !

Martin Ossmann (Allemagne)

Disponible pour 20 € environ, la carte microcontrôleur Teensy 4.0 a des performances exceptionnelles. Elle est cadencée à 600 MHz, mais cela n'explique pas tout. Au moyen de quelques essais, nous allons examiner quelles caractéristiques et quelles dispositions ont conduit à ce niveau de performances.



Le processeur utilisé, l'IMX-RT1062 de NXP, un ARM Cortex-M7, est plus proche, par son architecture, des processeurs de PC que des microcontrôleurs AVR. Pour le programmer, nous allons surtout utiliser l'EDI *Teensyduino*, largement compatible Arduino, en ayant quelquefois recours au C ou à l'assembleur.

Faire commuter une broche

Pour commencer, nous allons faire commuter une broche aussi rapidement que possible. Pour nous échauffer et à titre de comparaison, faisons l'opération sur un Nano cadencé à 16 MHz. Rien que par sa fréquence d'horloge, le Teensy est 600 / 16 = 37,5 fois plus rapide que l'Arduino. Pour faire clignoter la LED sur la broche 13 de l'Arduino, on peut se servir du programme du **listage 1**, censé allumer et éteindre la LED pendant une microseconde. On observe en fait que la LED est allumée pendant 3,5 µs et éteinte pendant 3,9 µs. Deux raisons à cela : la première est que la fonction `digitalWrite()` prend 2,5 µs environ, la seconde est qu'entre deux appels de `loop()`, l'Arduino exécute quelques tâches qui prennent 0,4 µs. Pour commuter plus rapidement, on peut et on doit programmer plus près du matériel comme le montre le **listage 2**.

Avec ce programme, une exécution de la boucle ne dure plus que 2,66 µs, soit 6 périodes d'horloge à une fréquence de 16 MHz. Pour savoir d'où viennent ces 6 périodes, on peut examiner le programme assembleur du **listage 3**.
La boucle se compose de trois instructions. Les instructions `sbi` et `cbi` allument et éteignent la LED. Avec l'instruction `rjmp` on réalise la boucle infinie souhaitée. Le manuel de référence de l'AVR donne les temps d'exécution des différentes

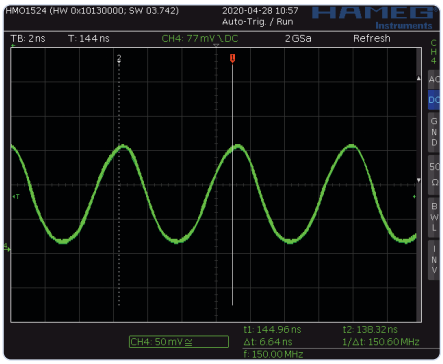


Fig. 1. Commutation de broche à 150 MHz.

instructions. Chaque instruction durant deux cycles d'horloge, on retrouve ainsi la durée observée. Il n'est guère possible d'obtenir par programme une commutation plus rapide et nous allons voir comment s'y compare la vitesse maximum obtenue avec une Teensy 4.0.

Commençons avec un programme dans le «style Arduino» (**listage 4**). Un cycle de la boucle infinie dure 135 ns. Par rapport à l'Arduino, c'est relativement rapide, mais ça prend tout de même 18 cycles d'horloge. L'exécution d'un `digitalWrite()` est assez lente et les «pertes» internes entre appels de `loop()` ne sont pas négligeables. Passons donc à la programmation proche du matériel (**listage 5**). L'exécution de la boucle `while` ne dure que 6,66 ns, soit 4 cycles.
L'oscillogramme de la **fig. 1** montre la tension sur la sortie LED. Comme le signal a une fréquence de 150 MHz, il occupe toute la bande passante de l'oscilloscope et ressemble davantage à une sinusoïde qu'à un carré. Pour une mesure précise des temps d'allumage et d'extinction de la LED, la fréquence d'horloge est réduite à 100 MHz. Ces temps sont de deux cycles, soit 20 ns à 100 MHz.

Pour savoir quelles instructions l'unité centrale (UC) exécute dans cette boucle, considérons le **listage 6**. Comme pour l'Arduino, la boucle se compose de trois instructions. Deux instructions d'écriture de mot (`str.w`) commutent le bit 13 et une instruction de branchement (`b.n`) ferme la boucle. On aimerait connaître le nombre de cycles consommé par chaque instruction, mais la documentation ARM reste muette sur ce sujet, avec raison d'ailleurs, puisque ce nombre dépend de nombreux effets

Listage 1. Commutation de broche dans le style Arduino.

```
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);
   }
```

Listage 2. Commutation de broche proche du matériel 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
     }
   }
```

Listage 3. Commutation de broche en assembleur.

```
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
```

Listage 4. Programme pour Teensy en style Arduino.

```
int led = 13;

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

Listage 5. Programme pour Teensy proche du matériel.

```
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(){
   }
```

Listage 6. Commutation rapide en assembleur.

```
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
```

Listage 7. Séquence d'instructions plus longue.

```
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(){
   }
```

Listage 8. Boucle de commutation finie.

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

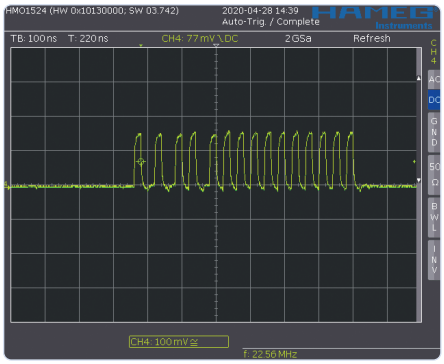


Fig. 2. La broche commute irrégulièrement.

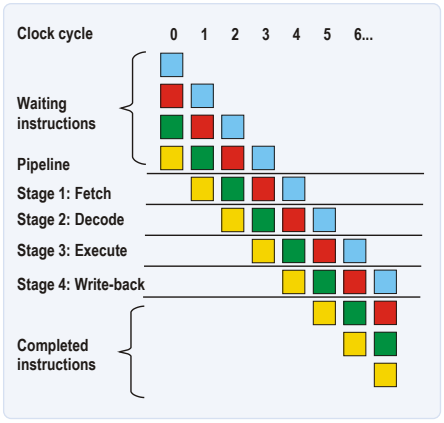


Fig. 3. Pipeline à 4 étages.

secondaires, ce qui suggère de se contenter d'une performance d'ensemble. Nous sommes têtus et voudrions quand même le savoir. Il semble que pour l'instruction `str`, il faille compter deux cycles, ce que confirme le **listage 7**.

Toute paire supplémentaire `set/clear` ne change rien à la fréquence de la tension sur la broche 13. Cela signifie qu'une paire `set/clear` allonge la boucle de 4 cycles. Dans l'exemple initial, les deux instructions `str` représentent donc 4 cycles, soit la durée totale de la boucle `while`. Il semble surprenant que l'exécution de l'instruction de saut n'y ajoute aucun cycle supplémentaire, mais on en verra la raison plus tard. Cela rend la carte Teensy à 4 cycles encore plus rapide que l'AVR à 6 cycles.

Ajoutons encore un peu à la confusion avec la boucle du **listage 8**. Pour que la tension sur la broche de la LED soit plus proche d'un carré, gardons la fréquence de 100 MHz. Le programme commute la broche de la LED 15 fois, fait une pause et affiche un texte, puis recommence.

La tension sur la broche de la LED a l'aspect de la **fig. 2**. Le rapport cyclique entre *haut* et *bas* varie. Au début, l'exécution de la boucle semble tantôt plus, tantôt moins longue, les instructions semblent avoir une durée d'exécution variable. Vers la fin, la boucle dure 4 cycles, comme avec la boucle infinie précédente. C'est étonnant, vu que la boucle contient maintenant une instruction de plus, `k1--` (listage 9)**,** qui ne semble n'exiger aucun temps supplémentaire et donc ne pas influencer la durée d'exécution de la boucle.

Prédiction de saut

La prédiction de saut avec exécution spéculative d'instructions fournit une première explication à la grande vitesse d'exécution de la boucle. Pour le comprendre, il faut avoir une connaissance précise de la manière dont l'UC exécute les instructions. Une première technique d'augmentation de la vitesse consiste en l'introduction de *pipelines*. Le pipeline contient les instructions en cours d'exécution. Chaque instruction est décomposée en plusieurs *étapes*, associées aux *étages* du pipeline. Si, par exemple, l'instruction 1 exécute l'étape 4, l'instruction 2 en est à l'étape 3, l'instruction 3 à l'étape 2 et l'instruction 4 à l'étape 1 (pipeline à 4 étages, **fig. 3**). Les instructions passent de la mémoire au pipeline que l'unité de lecture de la mémoire doit alimenter en permanence avec de nouvelles instructions en veillant à ce qu'il soit toujours bien rempli.

Il arrive toutefois qu'il faille purger le pipeline de son contenu parce que celui-ci est faux, lorsqu'il se produit un *aléa*. C'est le cas des sauts conditionnels si l'exécution ne se poursuit pas en séquence. Un autre

aléa est ce qu'on appelle un *pipeline stall* (suspension du transit). Il peut se produire lorsque l'activité d'un étage dépend du résultat élaboré par un autre étage. Dans le cas du saut conditionnel, l'UC devrait savoir si le saut va avoir lieu ou non pour remplir correctement le pipeline. C'est impossible, mais l'UC peut tenter de le prédire. Dans notre exemple (**listage 9**), le saut est exécuté 14 fois sur 15. L'UC observe ce qui se passe pour les premiers parcours de la boucle et note que le saut a lieu à chaque fois. Elle en déduit qu'il aura lieu aussi la fois suivante.

Après l'instruction de saut, le pipeline est donc toujours rempli avec les instructions `str`. L'UC a raison au début, mais pas à la fin, où il se produit un aléa. C'est cette technique qu'on appelle prédiction de saut avec exécution spéculative d'instructions. Comme le pipeline est rempli correctement la plupart du temps, l'instruction de saut ne coûte pratiquement plus de temps machine, ce qui explique le comportement du programme du listage 9. Au début, la prédiction est incorrecte et, à la fin, la boucle est exécutée à très grande vitesse. D'un autre côté, le calcul précis du temps d'exécution d'une partie de code à partir des cycles d'instructions devient très difficile, car le résultat de la prédiction dépend de nombreux facteurs, c'est pourquoi les nombres de cycles des instructions individuelles ne sont pas documentés. Une bonne prédiction de saut est un élément essentiel de l'architecture d'une UC à haute performance. Comme les UC modernes possèdent souvent des pipelines de grande taille, un aléa devient pénalisant et une prédiction correcte essentielle. Les *branch-predictors* courants atteignent un taux de réussite de plus de 98 % ! Les petits processeurs et les microcontrôleurs (les AVR, par ex.) en sont souvent dépourvus, mais les pipelines sont devenus des éléments standards.

Examinons maintenant le temps d'exécution de parties de code un peu plus compliquées. Notre premier programme calcule dans une boucle un produit scalaire (**listage 10**). Le calcul proprement dit est effectué par l'instruction `skp += x[k]*y[k];`. Avant cette instruction, allumons la LED sur la broche 13 et éteignons-la après. Le temps d'allumage de la LED devrait donc nous donner le temps précis d'exécution de notre instruction de test.

Nous constatons avec surprise que le temps d'allumage mesuré équivaut à deux cycles. Mais la commutation de la broche 13 prend déjà tout ce temps à elle seule et n'en laisse aucun à notre instruction de test. Ce qui se passe réellement n'est donc pas ce que nous pensions.

Pour en découvrir la raison, reportons-nous au code assembleur correspondant (**listage 11**).

On voit qu'il n'y a qu'une seule instruction entre les actions sur la broche. Le compilateur a déplacé les accès aux variables `x[k1++]` et `y[k2++]` devant l'allumage de la LED. Il est donc impossible de déduire le temps d'exécution de notre instruction du temps d'allumage de la LED. Le compilateur ne génère donc pas le code que nous pensions ; il lui arrive de modifier l'ordre de succession des instructions si cela améliore la performance et ne change rien au résultat. Pour la mesure des temps, il faut donc s'assurer que le compilateur ne procède pas à des optimisations intempestives.

La variable `ARM_DWT_CYCCNT` fournit le total des cycles écoulés, on peut s'en servir pour évaluer le temps d'exécution. C'est ainsi

Listage 9. Code assembleur de la boucle de commutation finie.

```
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
```

Listage 10. Produit scalaire pour test.

```
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 ;
```

Listage 11. Programme assembleur du produit scalaire.

```
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
```

Listage 12. Addition de 100 valeurs de fonction.

```
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 ;
   }
```

Listage 13. Code assembleur de la boucle d'addition.

```
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
```

Listage 14. Boucle plus complexe.

```
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 ;
    ...
```

Listage 15. Code assembleur du listage 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
```

Listage 16. Tampon alloué dynamiquement en mémoire RAM2.

```
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 ;
```

que, dans notre exemple et pour 1000 exécutions de la boucle, le nombre de cycles par boucle (le total / 1000) est de 7. Mesurons maintenant le temps total de 100 exécutions de la boucle du **listage 12**, en encadrant comme précédemment la boucle d'instructions de commutation de la LED. Dans cette boucle, on fait la somme des valeurs de la fonction `fun1(k)`.
Nous obtenons le résultat étonnant de 5 ns pour l'ensemble de la boucle, soit 3 cycles. Comment est-ce possible ? Une fois encore, l'explication nous est fournie par le code assembleur (**listage 13**).

Compilateur optimiseur

Entre les deux instructions de commutation de la LED, le compilateur n'en a inséré que deux simples, et pas une boucle complète. On ne retrouve aucune boucle dans le code assembleur. Une analyse plus poussée montre que le compilateur a éliminé la boucle et l'a remplacée par l'instruction `sum=826550` (= 0x000C9CB6H). Il a donc commencé par évaluer cette valeur et s'en est servi pour remplacer 100 exécutions de boucle par une assignation. Cela montre l'efficacité avec laquelle les compilateurs optimiseurs arrivent à traiter du code relativement compliqué. Le code exécuté peut donc ponctuellement différer du code source. Ici aussi, il faut procéder avec précaution pour ne pas en arriver à comparer des pommes et des poires.

Exécutons maintenant la boucle du **listage 14**. La carte Teensy effectue un parcours de cette boucle en 5 cycles. Ça semble relativement court, alors recourons encore une fois au code assembleur (**listage 15**). Nous y découvrons la longueur de la boucle : 9 instructions. La carte Teensy les exécute en 5 cycles, ce qui semble à première vue surprenant. La carte Teensy y parvient parce que son processeur est une «UC superscalaire avec *dual issue*». Dans une unité centrale superscalaire, certaines unités (additionneur,...) sont présentes en plusieurs exemplaires, de sorte que les parties de plusieurs instructions peuvent être traitées simultanément. Dans ce contexte, *dual issue* signifie que deux instructions sont transférées simultanément de la première partie du FIFO vers les unités superscalaires.

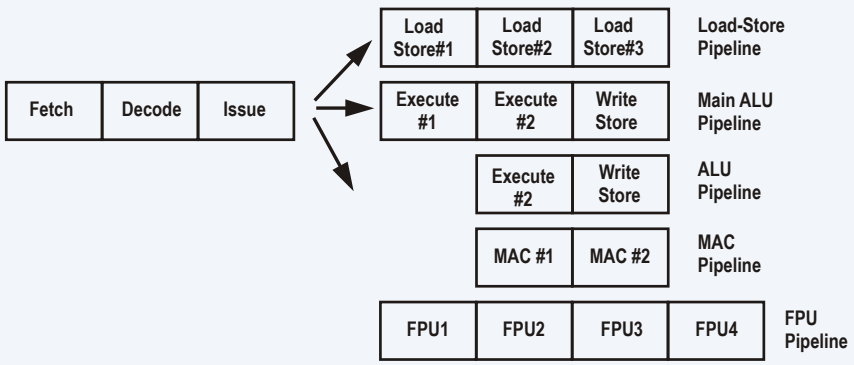
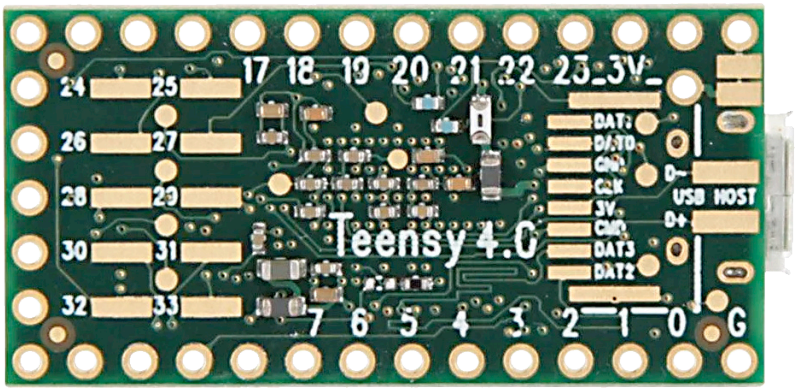


Fig. 4. Pipeline dual-issue.

ATOUTS DE L'UC IMXRT1062

- Fréquence d'horloge élevée
- Compilateur optimisé
- Prédiction de saut avec exécution spéculative d'instructions
- Structure RISC
- Pipeline
- Architecture superscalaire
- *Dual issue*
- Cache d'instructions
- Cache de données
- Unité de calcul flottant



Votre avis, s'il vous plaît…
Vous pouvez vous adresser à l'auteur par courriel en anglais (ou en allemand) : ossmann@fh-aachen.de.

Le pipeline a alors l'aspect de la **fig. 5**. De l'unité *Issue* deux instructions peuvent parvenir simultanément aux unités suivantes. Si nous neutralisons le *Dual Issue*, notre boucle va consommer 9 cycles pour 9 instructions, soit presque le double, ce qui montre l'importance et l'efficacité de ces techniques d'optimisation. Examinons maintenant les performances dans le cas d'une application réelle, la transformation de Fourier rapide (FFT). Commençons avec une FFT à 128 points. Sur un Arduino, son exécution prend 50 ms, sur Teensy 4.0, 23 µs seulement, ce qui est 50000 µs / 23 µs = 2200 fois plus rapide ! Le gain apporté par la fréquence d'horloge n'est que de 600 MHz / 16 MHz = 37, la différence, soit un facteur de 2200 / 37 = 60, provient de l'unité de calcul flottant et d'autres spécificités de l'architecture de la Teensy. Comme on peut couper chacune de ces spécificités (prédiction de saut, *dual issue*,…), on peut évaluer son effet avec précision.

La prédiction de saut apporte un gain de 21 %, le *dual issue*, de 41 %, ce qui montre que le traitement superscalaire de deux instructions est relativement fréquent. Le cache d'instructions n'apporte aucun gain dans notre exemple. Cela provient du fait que notre code réside dans la mémoire rapide. Si l'on met le code en mémoire flash et qu'on coupe le cache, le programme ralentit d'un facteur 7, qui est donc le mérite du cache. Mais si les données résident en RAM rapide, le cache de données n'apporte pas d'amélioration supplémentaire. Les zones de mémoire allouées dynamiquement par `malloc()` se situent dans une mémoire lente (RAM2). Si l'on y utilise des données et qu'on coupe le cache de données, le programme ralentit d'un facteur 7 parce que RAM2 est desservie par un bus plus lent.

Nous arrivons ainsi à la fin de notre voyage à travers les concepts d'amélioration de la performance. L'encadré reprend la liste des concepts mis en œuvre dans la Teensy. Peut-être que quelqu'un voudra faire le même travail avec le Raspberry Pi, dont la nouvelle version est cadencée à 1,5 GHz. Il y a encore quelques autres concepts, comme l'exécution dans le désordre (*out of order execution*), l'unité de gestion des pages (*paging memory management unit*), les caches L2 et L3, l'exécution en parallèle de fils multiples (*simultaneous multithreading*) ou le renommage de registres (*register renaming*) qui peuvent servir à apporter un supplément de gain de performances.

200190-02