BE : conception d'un accélérateur matériel pour le calcul de polynômes
solutions

Electronique numérique 1A

19 octobre 2021

On se propose ici de concevoir un accélérateur matériel, dédié au calcul de polynômes de rang 3. L'étude commence par une réflexion algorithmique, qui suggère des optimisations. Nous poursuivons la démarche par la manipulation abstraite de cet algorithme, à l'aide de *graphes*. L'accélérateur concu par le suite se présente comme une microarchitecture, composée classiquement d'un automate de contrôle et d'un chemin de données. On se limitera à des valeurs de x entières, codées sur 32 bits.

1 Réflexion algorithmique

La conception d'un accélérateur matériel naît souvent de la création d'un algorithme aux performances théoriques intéressantes (ex : compression par Transformée en Cosinus Discret), ou par la découverte d'une reformulation intéressante (ex : multiplication de Karatsuba [1]). Nous allons ici, en miniature, suivre cette démarche.
On considère un polynôme d'ordre 3 :

$$P_3(x) = a_3x^3 + a_2x^2 + a_1x + a_0 \tag{1}$$

1. **Question** : Combien de multiplications et d'additions sont-elles nécessaires ?

   Solution:
   Il en faut : 3+2+1 = 6 multiplications, et 3 additions.

2. **Question** : A l'aide du schéma de Hörner, normalement connu, montrer que l'on peut réduire le nombre de ces opérations.

   Solution:
   On peut réécrire le polynôme précédent de la manière suivante :

   $$P_3(x) = x(a_3x^2 + a_2x + a_1) + a_0 \tag{2}$$
   $$= x(x(a_3x + a_2) + a_1) + a_0 \tag{3}$$

   On en déduit que sous cette forme, seulement 3 multiplications sont nécessaires (en plus des 3 additions).

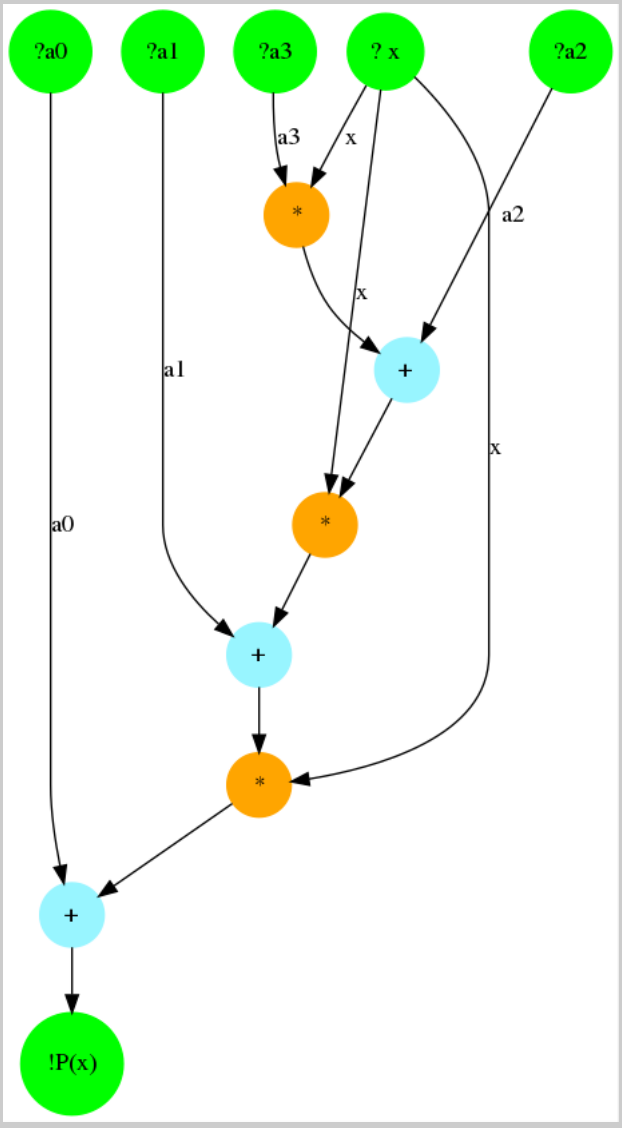
1. Nous vous invitons à vous renseigner sur ces deux curiosités mathématiques

2 Flot de données

L'étude du flot de données (ou DFG : dataflow graph) permet d'établir les dépendances du calcul. On représente généralement ces dépendances à l'aide d'un graphe orienté : chaque noeud représente une opération (ici arithmétique) ou une entrée ou une sortie. Chaque arc entre deux noeuds est étiqueté par le nom de la variable qui circule du noeud source au noeud destination. Dans notre cas, le graphe est acyclique. Il indique clairement l'enchaînement des opérations, mais il s'agit encore d'une vision abstraite du calcul, indépendante de choix d'architecture matérielle. On peut toutefois noter que ce DFG peut être vu comme une première tentative de réalisation purement combinatoire du circuit (mais son chemin critique est très certainement rédhibitoire).

1. **Question** : établir le graphe de flot de données issu du schéma de Hörner.

Solution:



3 Ordonnancement. Assignation et partage des ressources

La notion d'**ordonnancement** permet de découper le DFG en différentes étapes de calculs, appelées **control-step** (CSTEP). Prosaïquement, cela revient à dessiner différentes lignes qui

représentent effectivement cette découpe du DFG.

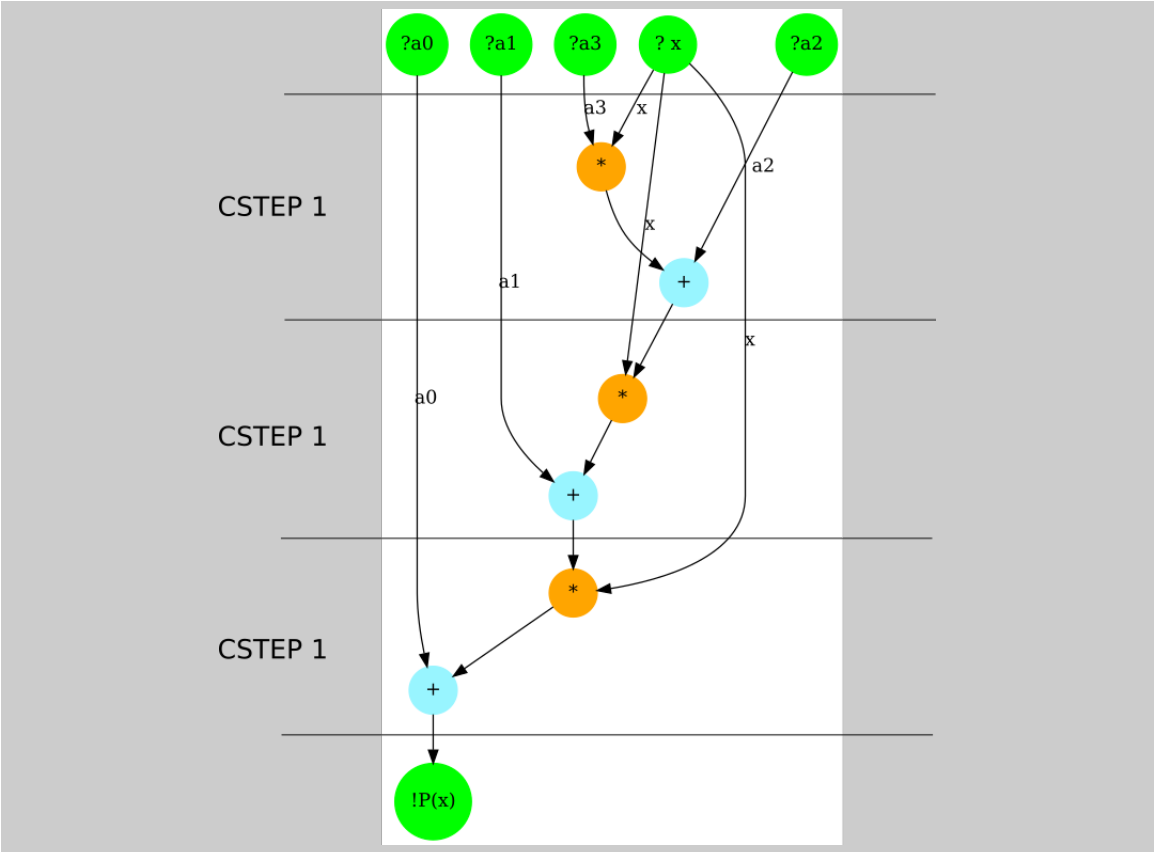
Chacune des étapes s'exécute en un seul cycle d'horloge. Selon la durée physique du cycle d'horloge, il est toutefois possible de réaliser des découpes du DFG plus ou moins "longues" : plus le cycle est long, plus il est possible d'envisager de **chaîner** un ensemble d'opérations. A la fin du cycle, on devra veiller à échantillonner les données produites : formulé autrement, on peut dire que la découpe en CSTEPs détermine où devront se trouver les *registres intermédiaires* de calcul.

En général, les opérateurs de calculs et les registres de calcul représentent la majeure partie de la surface occupée sur Silicium. Il est souvent souhaitable de les utiliser avec parcimonie et notamment de les partager dès que cela est possible. On réalise l'**assignation** des opérations aux opérateurs afin d'établir où se réaliseront effectivement les calculs du DFG abstrait. Bien évidemment, à la fin d'un CSTEP tous les opérateurs utilisés sont à nouveau disponibles pour le CSTEP suivant, ce qui donne des opportunités de partage de ressources entre opérations ordonnancées sur deux CSTEPs différents. Ces informations d'assignation peuvent être également annotées sur le DFG : à côté de l'opération abstraite du DFG, on peut indiquer sur quel opérateur matériel elle s'exécutera.

Avant même la phase d'assignation, le concepteur se donne par ailleurs un objectif en terme de nombre de ressources utilisées au total : c'est la phase d'**allocation de ressources**. Il peut se donner le droit d'utiliser tel nombre de multiplieurs, tel nombre d'additionneurs, tel nombre de registres, etc. Parmi les contraintes, on retrouve d'autres paramètres variés et en particulier la fréquence attendue du circuit. On se fixe ici une fréquence d'horloge de 100 Mhz.

1. **Question 4** : on s'alloue 1 multiplieur et un additionneur. Le multiplieur a un temps de calcul de 7 ns et l'additionneur de 2 ns. Concernant les registres, on considère que leur temps de setup et old sont très inférieurs, et non significatifs par rapport aux opérateurs. Proposer un ordonnancement qui vous semble judicieux. A combien de CSTEPS aboutissons-nous ?

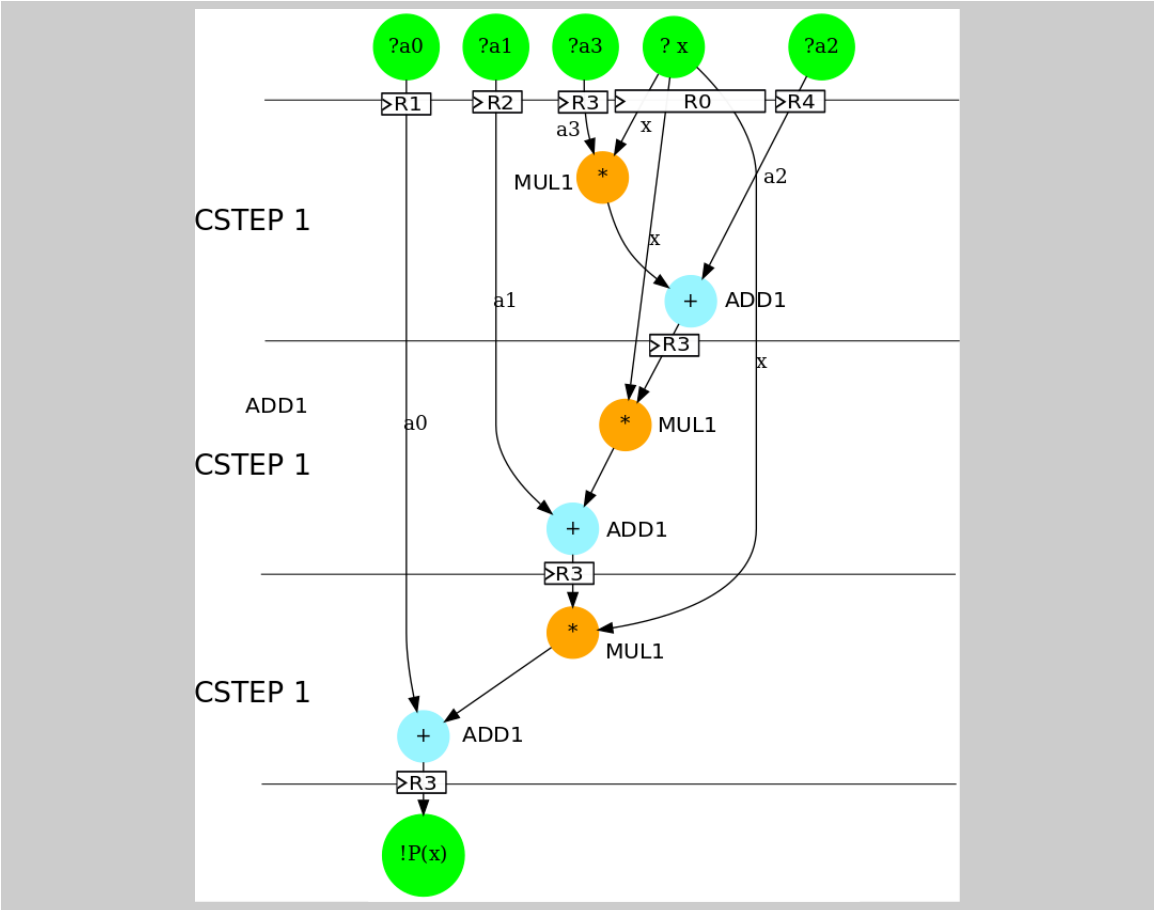
   Solution:
   On voit sur le DFG que le calcul de multiplication et d'addition se suivent. Le chemin critique de cet enchaînement est de $7 + 2 = 9ns$. Or, la fréquence de fonctionnement des registres est de 100 Mhz soit un temps de cycle de 10 ns : ce bugdet de 10 ns laisse parfaitement le temps aux signaux de traverser à la fois le multiplieur et l'additionneur, dans un même cycle d'horloge. Cela est d'autant plus vrai que les temps de setup et hold sont considérés comme négligeables dans l'énoncé. On peut donc chaîner les opérateurs matériel dans le même cycle. Par contre, à la fin de cette portion (répétée) de circuit, il faudra échantillonner les signaux, dans des bascules D (registres). Le schéma suivant donne 3 CSTEPS. On suppose que les entrées et sorties sont également "clockées" (échantillonnées), ce qui est une bonne pratique.



2. **Question 5** : réaliser l'assignation des ressources et notamment des registres.

Solution:

On annote les ressources sur le schéma : il s'agit désormais d'opérateurs matériels et non plus d'opérations abstraites. Ces opérateurs peuvent être réutilisés à chaque cycle (CSTEP). De même un registre dont on n'a plus besoin peut être réutilisé. C'est le cas du regsitre R3, qui contient initialement le premier coefficient lu. Après cette lecture, on peut utiliser R3 pour les calculs des variables temporaires (intermédiaires) de calcul.

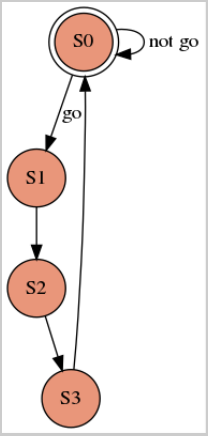


4 Microarchitecture : automate de contrôle et datapath

Dès lors qu'un ordonnancement est choisi, il est alors aisé de décrire un automate et un chemin de donnée. L'automate permet de séquencer les différentes étapes. Il réalise en outre, pour chacune d'entre elles, les actions appropriées. A l'issue de l'assignation, ces actions reviennent à contrôler les multiplexeurs d'un chemin de données (datapath) : ces multiplexeurs servent soit à échantillonner une donnée à l'entrée d'un registre, soit à router une donnée vers un opérateur matériel.

1. **Question 6** : dessiner l'automate de contrôle.

Solution:



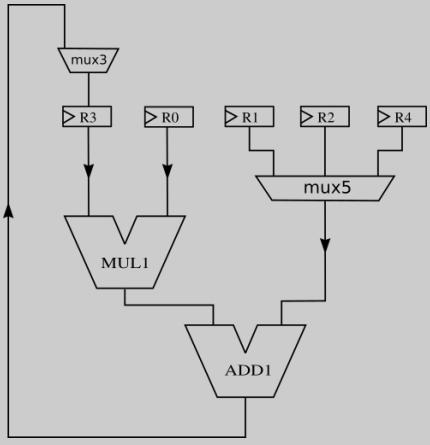
2. **Question 7** : dessiner un chemin de données

Solution:

Pour déterminer le chemin de données (datapath), on observe le DFG ordonnancé et alloué (question 5). Il s'agit essentiellement de déterminer les opérandes des opérateurs. Prenons le Multiplieur MUL1 : on voit que ses opérandes sont R0 et R3 lors du CSTEP 1, et à nouveau R0 et R3 aux CSTEPs 2 et 3. Le chemin entre les registres et l'entrée de MUL1 est donc très simple et direct. Observons maintenant ADD1 :

- dans le CSTEP 1, ADD1 opère sur le résultat combinatoire de MUL1 et sur R4
- dans le CSTEP 2, ADD1 opère sur le résultat combinatoire de MUL1 et sur R2
- dans le CSTEP 3, ADD1 opère sur le résultat combinatoire de MUL1 et sur R1

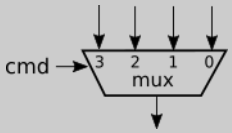
On peut en déduire que pour assurer ces 3 fonctionnements, il est nécessaire d'adjoindre un multiplexeur sur le second opérande de ADD1, dont les sources sont R4,R2 et R1. Notons que nos deux opérateurs sont commutatifs, ce qui a grandement simplifié notre analyse. On en déduit une version, encore incomplète, du datapath, présentée la figure suivante.



Quelques détails restent à régler : comment sont acheminés les entrées et calculs intermédiaires, vers ces registres ? Il faut bien penser à l'échantillonnage des entrées et leur stockage tout au long du calcul. Là encore on repose sur un multiplexeur, dont la commande peut jouer ce rôle. Afin de ne pas modifier la donnée stockée (potentiellement ré-échantillonnée à chaque cycle), on fait ainsi "recirculer" la donnée de la sortie Q à l'entrée D des bascules. Désormais notre datapath est complet. Reste à déterminer les instants de contrôles des multiplexeurs, émis par le contrôleur (fsm).

a_2

On numérote les entrées des multiplexeurs : ces valeurs correspondront aux valeurs du signal de commande qu'il faudra envoyer au multiplexeur pour qu'il fasse passer l'entrée correspondante sur sa sortie. Cela nous permettra de piloter le cheminement des données (question suivante).



3. **Question 8** : compléter l'automate pour indiquer les actions réalisées dans chaque état.

Solution:

On se propose de présenter cet automate sous la forme tabulée suivante. Ce tableau fait apparaître les *transfert de registres*, ainsi que les micro-actions correspondantes, sur les multiplexeurs.

état	effet algorithmique	transferts de registres	commandes des multiplexeurs
S0	arrivée des données	$R0 <= x$ $R1 <= a_0$ $R2 <= a_1$ $R3 <= a_3$ $R4 <= a_2$	$cmd_0 <= 1$ $cmd_1 <= 1$ $cmd_2 <= 1$ $cmd_3 <= 1$ $cmd_4 <= 1$ $cmd_5 <= 0$
S1	$t_1 = a_3 * x + a_2$	$R3 <= (R3 * R0) + R4$	$cmd_0 <= 0$ $cmd_1 <= 0$ $cmd_2 <= 0$ $cmd_3 <= 2$ $cmd_4 <= 0$ $cmd_5 <= 0$
S2	$t_2 = t_1 * x + a_1$	$R3 <= (R3 * R0) + R2$	$cmd_0 <= 0$ $cmd_1 <= 0$ $cmd_2 <= 0$ $cmd_3 <= 2$ $cmd_4 <= 0$ $cmd_5 <= 1$
S3	$P(x) = t_2 * x + a_0$	$R3 <= (R3 * R0) + R1$	$cmd_0 <= 0$ $cmd_1 <= 0$ $cmd_2 <= 0$ $cmd_3 <= 2$ $cmd_4 <= 0$ $cmd_5 <= 2$

Quelques remarques :

- On notera au passage que ces transferts de registres ressemblent à ce que réalise un langage de programmation de bas niveau : l'assembleur. Toutefois, nos transferts de registres sont ici plus puissants que l'assembleur : plusieurs transferts peuvent avoir lieu en même temps, en parallèle ,alors qu'en assembleur traditionnel chaque tranfert se fait séquentiellement.
- L'opération de multiplication suivie d'une addition est si fréquente dans le domaine scientifique, qu'il existe un opérateur dédié : le MULACC. Il est présent notamment dans les processeurs de traitement du signal, appelés DSP (digital signal processors,

les plus connus étant ceux de Texas Instruments). Les FPGA modernes, possèdent en plus de leur matrice de LUTs, un ensemble de tels opérateurs optimisés.

5 Dimensionnement des registres

Une difficulté récurrente lorsqu'on traduit un algorithme en Microarchitecture réside dans le dimensionnement des signaux, c'est-à-dire la détermination du nombre de bits nécessaires à la représentation des nombres, sans perte d'information préjudiciable. On parle aussi de leur *dynamique*. Il y a perte d'information lors d'une troncature (élimination brutale de bits à un certain rang).

1. **Question 9** : soient deux signaux s_n et $s_{n'}$ entiers codés respectivement sur n et n' bits. Déterminer le nombre de bits nécessaires à l'addition $s_n + s_{n'}$, sans perte d'information.

   Solution:
   Par exemple, avec $n = 8$ et $n' = 8$, $s_n, s_{n'} \in \{0..255\}$ et $s_n + s_{n'} \in \{0..510\}$. On se convainc facilement que le nombre de bits pour le résultat de l'addition est $max(n, n') + 1$.

2. **Question 10** : même question pour $s_n * s_{n'}$.

   Solution:
   Pour la multiplication le résultat est $n + n'$.

3. **Question 11** : en déduire la dynamique des signaux apparaissant dans le datapath.

   Solution:
   A la fin de chaque CSTEP, la dynamique des signaux calculés augmente. Si chaque entrée est codée sur un même nombre n de bits :
   - CSTEP 1 : R3 nécessite $2n + 1$ bits
   - CSTEP 2 : R3 nécessite $(2n + 1) + n + 1 = 3n + 2$ bits
   - CSTEP 3 : R3 nécessite $(3n + 2) + n + 1 = 4n + 3$ bits

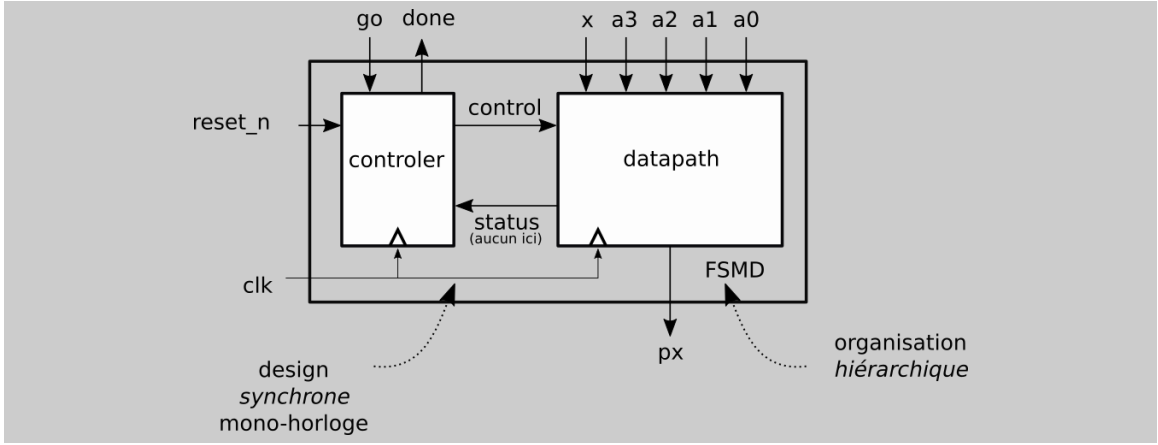
   A titre d'exemple, partant de $n = 8$ bits, on obtient 35 bits nécessaires en sortie.

6 Codage VHDL

Le code VHDL de cet accélérateur est fourni sur Moodle, ainsi qu'un banc de test. La syntaxe de VHDL est complexe et nouvelle pour vous, mais vous êtes à même d'en comprendre les grandes lignes.

1. **Question 12** : observez attentivement le code de l'accélérateur. Dessiner l'architecture générale du système. Discutez avec votre encadrant afin de bien vérifier que vous avez compris la correspondance avec vos schémas précédents.

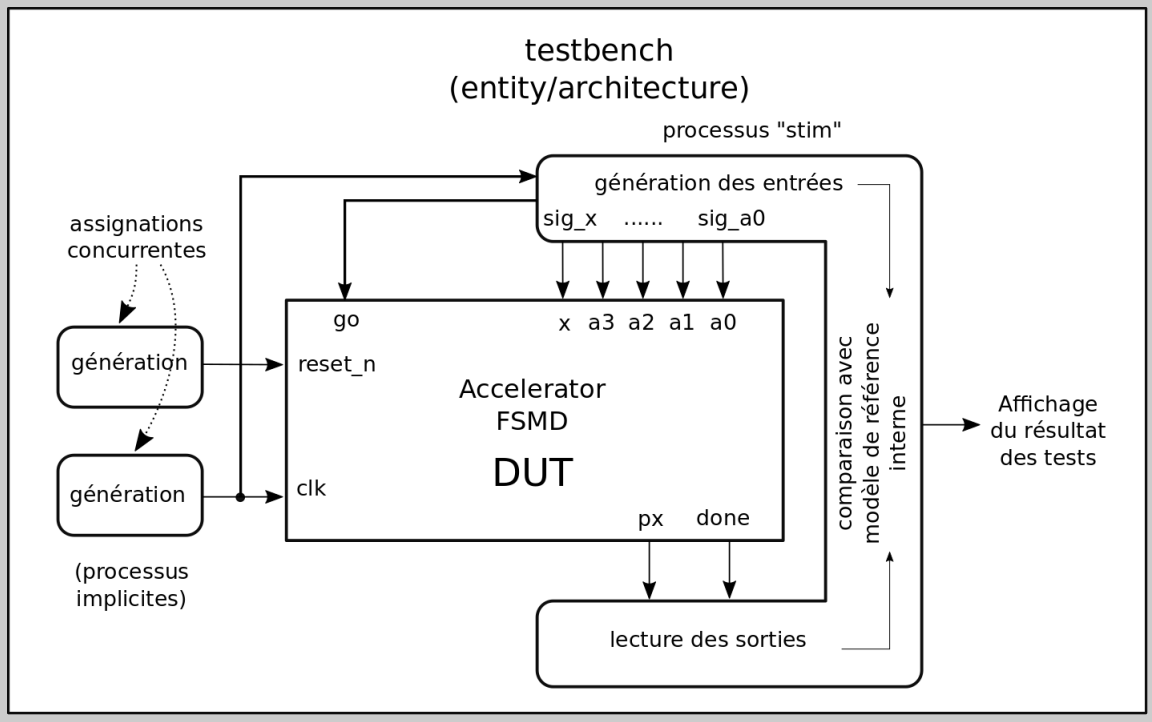
   Solution:



Le banc de test réalise tout d'abord la configuration de l'accélérateur, par l'envoi des paramètres du polynôme, puis réalise le calcul de $P_3(x)$ pour une succession de valeurs x.

2. **Question 13** : observez attentivement le code du banc de test. Discutez avec votre encadrant afin de bien vérifier que vous avez compris son fonctionnement.

Solution:
Le banc de test se présente comme un design classique : il est composé d'une entité et d'une architecture. Toutefois l'entité ne possède aucun port d'entrée-sortie. Cela est parfaitement normal, car notre testbench représente une "expérience virtuelle", dans un laboratoire tout aussi virtuel, entièrement équipé et autonome ! Il n'a besoin d'aucune information extérieure. L'architecture représente la paillasse où l'on prépare un "setup expérimental" : au centre, on retrouve bien entendu le circuit à tester, et tout autour de lui des instruments virtuels, sous forme de processus qui fonctionnent en parallèle, les uns des autres : générateurs de signaux, sondes, outils de diagnostique etc. Notre testbench réalise également une comparaison des signaux issu de notre circuit (done et px) avec un modèle de référence. Ce modèle de référence est ici calculé également par VHDL. Dans notre cas, VHDL est en effet capable de calculer un polynôme, de manière "informatique". I laurait été assez maladroit de recourir à un modèle de référence Python ou Matlab pour écrire une référence si simple. Toutefois, cela arrive fréquemment que les équipes en présence n'ait pas la même culture. Les algorithmiciens, qui fournissent de tels modèles de références, connaissent rarement VHDL.



3. **Question 14** : la simulation ne sera pas réalisée en séance, mais une video sera mise à disposition.

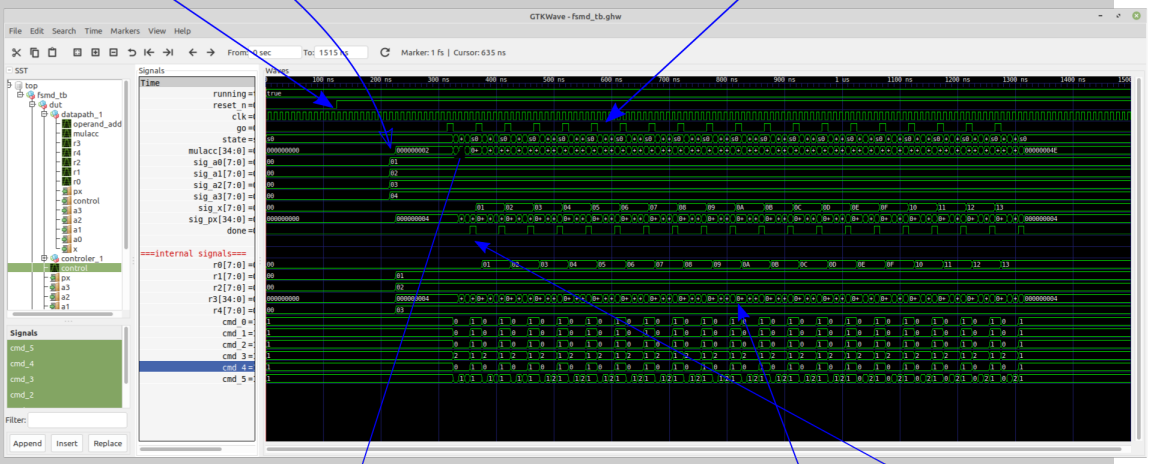
Solution:
La simulation se fait ici sous Linux, à l'aide de GHDL. GHDL permet de générer un simulateur compilé. Cette génération se fait en 3 grandes étapes :
- **Analyse** des fichiers avec l'option "-a" de ghdl.
- **Elaboration** du simulation avec l'option "-e" de ghdl.
- **Lancement** (**"run"**) du simulation avec l'option "-r" de ghdl.
- **Visualisation** des résultats de simulation avec GtkWave.

Ces commandes répétitives sont facilement automatisables sous Linux. Un script bash "compile.x" est fourni sur Moodle. On obtient le **chronogramme** suivant.

Le reset se relève à 123 ns.

Les coefficients du polynôme sont configurés

20 tests en boucle ordonnent un "go", avec des valeurs de "x" croissantes



l'automate évolue jusqu'à retourner à l'état S0, et émet alors un "done=1"

Le signal du registre r3 est très sollicité

Dans le terminal Linux apparaissent les messages du simulateur, ainsi que les ”reports” que l'on a décrit en VHDL :

```
~/JCLL/ens/UE1.2-Electronique/TD/9_BE_Horner/VHDL$ ./compile.x
============ cleaning  ============
======= analyzing vhdl files ======
=> controler_pkg.vhd
=> controler.vhd
=> datapath.vhd
=> fsmd.vhd
=> fsmd_tb.vhd
======= elaborating simulation ======
=> fsmd_tb
========= running simulation ========
@0ms    :(report note): running testbench for fsmd(rtl)
@0ms    :(report note): waiting for asynchronous reset
@215ns :(report note): p(x)=4x^3 +3x^2 + 2^x + 1
@215ns :(report note): sending parameters a0...a3
@315ns :(report note): expected p(0)=1
@365ns :(report note): actual : p(0)=1
@365ns :(report note): expected p(1)=10
@415ns :(report note): actual : p(1)=10
@415ns :(report note): expected p(2)=49
@465ns :(report note): actual : p(2)=49
@465ns :(report note): expected p(3)=142
@515ns :(report note): actual : p(3)=142
@515ns :(report note): expected p(4)=313
@565ns :(report note): actual : p(4)=313
@565ns :(report note): expected p(5)=586
@615ns :(report note): actual : p(5)=586
@615ns :(report note): expected p(6)=985
@665ns :(report note): actual : p(6)=985
@665ns :(report note): expected p(7)=1534
```

```
@715ns :(report note): actual : p(7)=1534
@715ns :(report note): expected p(8)=2257
@765ns :(report note): actual : p(8)=2257
@765ns :(report note): expected p(9)=3178
@815ns :(report note): actual : p(9)=3178
@815ns :(report note): expected p(10)=4321
@865ns :(report note): actual : p(10)=4321
@865ns :(report note): expected p(11)=5710
@915ns :(report note): actual : p(11)=5710
@915ns :(report note): expected p(12)=7369
@965ns :(report note): actual : p(12)=7369
@965ns :(report note): expected p(13)=9322
@1015ns:(report note): actual : p(13)=9322
@1015ns:(report note): expected p(14)=11593
@1065ns:(report note): actual : p(14)=11593
@1065ns:(report note): expected p(15)=14206
@1115ns:(report note): actual : p(15)=14206
@1115ns:(report note): expected p(16)=17185
@1165ns:(report note): actual : p(16)=17185
@1165ns:(report note): expected p(17)=20554
@1215ns:(report note): actual : p(17)=20554
@1215ns:(report note): expected p(18)=24337
@1265ns:(report note): actual : p(18)=24337
@1265ns:(report note): expected p(19)=28558
@1315ns:(report note): actual : p(19)=28558
@1315ns:(report note): end of simulation
@1315ns:(report note): number of errors : 0 / 20 tests
============== viewing ==============

GTKWave Analyzer v3.3.86 (w)1999-2017 BSI

[0] start time.
[1515000000] end time.
```