ASR1 – DM1 : À la découverte de VHDL

{ Jeremie.Detrey, Patrick.Loiseau, Nicolas.Veyrat-Charvillon } @ens-lyon.fr
http://perso.ens-lyon.fr/jeremie.detrey/06_asr1/
à rendre par mail avant le dimanche 3 décembre 2006, 23h59
(cachet du postmaster faisant foi)

*Ce DM est à faire en **binôme**. Dans le cas où vous seriez un nombre impair à suivre ce cours, nous n'accepterons qu'un unique trinôme. Mettez-vous donc bien d'accord entre vous là-dessus. Mis-à-part pour ce groupe, nous refuserons tout devoir qui n'aura pas été fait en binôme.*

Le but de ce DM est avant tout de vous faire découvrir le langage VHDL (ou tout du moins un sous-ensemble suffisant de ce langage). Dans un premier temps par la réalisation d'un simulateur puis par l'essai sur divers circuits de votre conception. Dans les deux cas, essayez autant que possible d'écrire du code lisible et bien commenté.

Enfin, vous devrez aussi nous rendre par mail un rapport imprimé (rédigé avec TEX ou LATEX) dans lequel vous détaillerez le fonctionnement de votre simulateur VHDL, ainsi que comment marchent vos composants.

Les sections 2 et 3 sont vaguement indépendantes. N'hésitez pas à les traiter dans le désordre. Par contre, évitez de vous répartir une section chacun dans votre binôme. Il faut bien que tout le monde touche à tout.

1 Présentation de VHDL

1.1 VHDL, qu'es acò ?

L'acronyme VHDL signifie *VHSIC Hardware Description Language*, VHSIC lui-même signifiant *Very High Speed Integrated Circuit*. Derrière ce nom à rallonge se cache un des langages les plus utilisés pour décrire des circuits, sa principale alternative étant Verilog.

Initialement conçu en 1987 sur commande de la défense américaine, il était censé décrire d'une façon formelle les circuits utilisés dans les équipements militaires, remplaçant ainsi les tonnes de documentations et de manuels pas toujours complets ou corrects et encore moins souvent lisibles. Le but de VHDL était donc à l'origine de décrire le *comportement* de circuits.

Très rapidement, des *simulateurs logiques* furent développés pour vérifier automatiquement le bon fonctionnement de ces circuits à partir de leur description VHDL. Peu après apparurent des *synthétiseurs logiques*, capables d'extraire la structure des circuits pour réaliser leur implémentation matérielle effective. À cette occasion, l'utilisation de VHDL, jusqu'alors limitée à la description *comportementale*, s'est étendue à la description *structurelle*, plus bas niveau et donc plus proche du matériel et de son implémentation par le synthétiseur.

Nous allons nous intéresser dans ce DM uniquement à la partie *structurelle* de VHDL.

1.2 Structure d'un fichier VHDL

Remarque : Cette introduction à VHDL est volontairement minimaliste et lacunaire, car elle se limite au sous-ensemble de VHDL que nous allons étudier lors de ce DM. Pour en savoir plus sur VHDL, allez

lire le VHDL Cookbook, *disponible en ligne d'un simple coup de Google.*

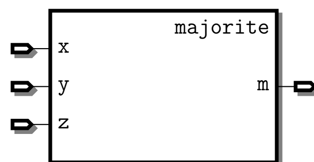
1.2.1 Entités

L'élément de base en VHDL est l'*entité* (ou le *composant*). Il s'agit en fait d'une boîte noire possédant des *ports* d'entrée et de sortie, et réalisant une fonction booléenne. Nous nous ne considèrerons dans le cadre de ce DM que des ports (en entrée comme en sortie) du type standard `std_logic`[1]. Cela signifie que seuls des signaux de ce type peuvent être connectés en entrée ou en sortie aux entités.

Par exemple, si l'on souhaite réaliser un circuit calculant la majorité sur trois bits x, y et z, on pourra définir l'entité suivante :

```
entity majorite is
  port ( x : in  std_logic;
         y : in  std_logic;
         z : in  std_logic;
         m : out std_logic );
end majorite;
```

Cela correspond en fait à la boîte noire suivante :



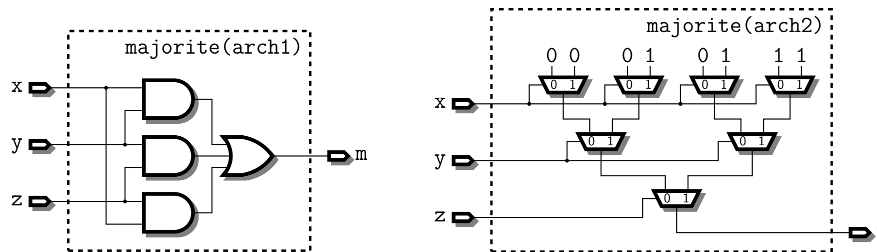
Nous retrouvons bien dans sa déclaration les trois ports en entrée x, y et z, ainsi que le port de sortie m correspondant au résultat de notre majorité. Par contre, la fonction de majorité n'est définie nulle part : l'entité n'est qu'une *interface*, un *prototype* (par analogie avec une fonction en C). Pour pouvoir *instancier* cette entité, il va falloir lui attribuer une *architecture* qui l'implémente.

1.2.2 Architectures

Chaque entité peut être réalisée par plusieurs architectures. Ainsi pour notre fonction de majorité, on peut soit implémenter la fonction booléenne à partir d'opérateurs de base (`arch1`), soit implémenter directement la table de vérité sous forme d'une *look-up table* adressée par les trois signaux x, y et z (`arch2`) comme sur la figure page suivante.

En pratique, c'est surtout utile pour fournir pour chaque entité une architecture comportementale (plus rapide à simuler) et une architecture structurelle (pour la synthèse du circuit).

[1]Ce type représente normalement l'ensemble des valeurs que peut prendre un signal dans un circuit logique, mais, pour simplifier, nous considèrerons dans ce DM qu'un signal peut être apparenté à un bit, c'est-à-dire soit dans l'état bas 0 (noté `'0'`) soit dans l'état haut 1 (noté `'1'`), soit enfin dans un état indéfini (noté `'U'` pour *Undefined*).



Chaque architecture est déclarée après l'entité à laquelle elle fait référence. Ainsi, pour l'architecture `arch1` de notre entité `majorite`, on a :

```
architecture arch1 of majorite is
  signal xy : std_logic;
  signal yz : std_logic;
  signal zx : std_logic;
begin
  xy <= x and y;
  yz <= y and z;
  zx <= z and x;
  m  <= xy or yz or zx;
end arch1;
```

Comme on peut le voir sur cet exemple, la déclaration d'une architecture s'effectue en deux temps.

Tout d'abord la décalaration des *signaux*, ainsi que leur type. Nous nous limitons toujours ici au seul type `std_logic`. Chaque signal représente en fait un fil dans le circuit. Il pourra donc être affecté (recevoir la valeur d'un autre signal) **une seule fois**, et plusieurs autres signaux pourront dépendre de sa valeur. Ainsi dans cet exemple, trois signaux sont déclarés : xy, yz et zx.

Lors de leur déclaration, les signaux peuvent de plus être initialisés à une certaine valeur. Cela signifie que ce signal aura cette valeur au tout début de la simulation. La syntaxe est la suivante :

`signal` *signal* `: std_logic :=` *val* `;`

avec *val* valant `'0'` ou `'1'`. En l'absence d'initialisation, le signal prend la valeur indéfinie `'U'`.

Entre les mots-clefs `begin` et `end` se trouve l'architecture proprement dite du circuit, sous forme d'une liste d'*instructions* qui seront "exécutées" en parallèle. Dans notre exemple, il s'agit de quatre instructions d'affectation.

1.2.3 Instructions

Nous ne considèrerons que deux types d'instructions dans ce DM :

Affectation L'affectation d'un signal ou d'un port de sortie *signal* est de la forme suivante :

signal `<=` *expr* `;`

L'expression *expr* peut-être :

– une constante de type `std_logic` (c'est-à-dire `'0'` ou `'1'`, la valeur indéfinie `'U'` n'étant pas “affectable”) ;
– un autre signal ou un port d'entrée ;
– la combinaison de deux autres expressions par un opérateur binaire (`or`, `and` ou `xor`) ;
– la négation d'une autre expression par l'opérateur unaire `not`.

Instanciation L'instanciation d'une autre entité *ent* permet de créer une instance de *ent* dans l'architecture courante et d'y connecter chacun de ses ports d'entrée et de sortie à des signaux locaux. L'instanciation de *ent* se passe donc en deux temps :
– Il faut d'abord déclarer *ent* en tant que *composant* de l'architecture. Cela se fait au même endroit que la déclaration des signaux, et consiste à recopier la déclaration de l'entité *ent*, en changeant le mot-clef `entity` en `component`.
– L'instanciation proprement dite est de la forme suivante :

$$inst \texttt{ : } ent \texttt{ port map (} port_1 \texttt{ => } signal_1\texttt{, } port_2 \texttt{ => } signal_2\texttt{, } \ldots \texttt{);}$$

L'identifiant *inst* est un nom unique donné à l'instance pour la différencier des autres. La connexion des ports se fait par un `port map` qui connecte chaque port d'entrée de *ent* à un signal ou une constante, et chaque port de sortie à un signal.

Ainsi, par exemple, on peut définir un multiplexeur par une entité `mux2` et son architecture comme suit :

```
entity mux2 is
  port ( d0 : in  std_logic; -- résultat quand x vaut '0'
         d1 : in  std_logic; -- résultat quand x vaut '1'
         x  : in  std_logic;
         r  : out std_logic );
end mux2;

architecture arch of mux2 is
begin
  r <= (x and d1) or (not x and d0);
end arch;
```

Cela donne alors l'architecture `arch2` pour `majorite` :

```
architecture arch2 of majorite is
  component mux2 is
    port ( d0 : in  std_logic;
           d1 : in  std_logic;
           x  : in  std_logic;
           r  : out std_logic );
  end mux2;

  signal x00 : std_logic;
  signal x10 : std_logic;
  signal x01 : std_logic;
  signal x11 : std_logic;
  signal xy0 : std_logic;
  signal xy1 : std_logic;
```

```
begin
  mux_x00 : mux2 port map ( d0 => '0', d1 => '0', x => x, r => x00 );
  mux_x10 : mux2 port map ( d0 => '0', d1 => '1', x => x, r => x10 );
  mux_x01 : mux2 port map ( d0 => '0', d1 => '1', x => x, r => x01 );
  mux_x11 : mux2 port map ( d0 => '1', d1 => '1', x => x, r => x11 );

  mux_xy0 : mux2 port map ( d0 => x00, d1 => x10, x => y, r => xy0 );
  mux_xy1 : mux2 port map ( d0 => x01, d1 => x11, x => y, r => xy1 );

  mux_xyz : mux2 port map ( d0 => xy0, d1 => xy1, x => z, r => m );
end arch2;
```

2 Un simulateur VHDL

2.1 Principe de base

Le but de cette partie est de réaliser un simulateur pour le sous-ensemble de VHDL présenté dans la section précédente. Du point de vue de l'utilisateur, le simulateur VHDL a un fonctionnement très simple : étant donné un circuit, passé au simulateur sous forme d'un composant VHDL, l'utilisateur peut fixer les valeurs en entrée des fils du circuit et observer les valeurs obtenues sur les ports de sortie.

Du point de vue du simulateur par contre, c'est un brin plus complexe. Il lui faut d'abord être capable de lire et d'interpréter le ou les fichiers VHDL qui lui sont passés en entrée. Ceci est réalisé grâce aux *lexer / parser*, chargés d'effectuer l'analyse lexicale et syntaxique de ces fichiers.

Les lexer et parser construisent un arbre syntaxique (AST) qui est ensuite passé à une étape de compilation, qui se contente de vérifier la cohérence de chaque entité et architecture (pas d'affectations multiples sur un même signal, pas d'affectation sur un port d'entrée, etc...). Le compilateur est aussi chargé de maintenir une liste des entités définies et de la ou des architectures qui leur sont associées.

Une fois tous les fichiers VHDL compilés, le simulateur demande à l'utilisateur l'entité et l'architecture à simuler. Cette entité est appelée *top-level*. Comme décrit dans la suite, la simulation d'un circuit nécessite que ce circuit soit représenté par une simple *netlist*, c'est-à-dire une liste de portes logiques de base et de fils reliant ces portes.

La création de cette netlist est réalisée grâce à une instanciation récursive des divers composants : d'abord le top-level, puis les composants instanciés par celui-ci, et ainsi de suite. L'idée est de mettre le circuit complètement à plat, de sorte à ne plus avoir une seule instanciation. Lors de l'instanciation des composants, chaque affectation est aussi décomposée en sous-opérations de base, ne faisant intervenir que des portes logiques simples (ET, OU et XOR à deux entrées ou NON à une entrée).

Enfin vient l'étape de simulation proprement dite : les valeurs imprimées sur les ports d'entrée du circuit sont propagées à travers ce circuit. Cette propagation n'est pas instantanée : chaque porte met un certain temps pour effectuer une transition sur sa sortie, et de même chaque fil met un certain temps pour propager l'information. Pour simplifier, on considère que l'information est transmise instantanément sur les fils, mais que les portes de base mettent chacune 1 ns à changer d'état.

Par exemple, une session typique avec notre entité de majorité pourrait être :

```
$ ./vhdlsim majorite.vhd
Simulate which entity: mux2, majorite? majorite
Simulate which architecture for entity <majorite>: arch1, arch2? arch2
Instantiating architecture <majorite(arch2)> as <sim://>...
Instantiating architecture <mux2(arch)> as <sim://mux_x00/>...
Instantiating architecture <mux2(arch)> as <sim://mux_x10/>...
Instantiating architecture <mux2(arch)> as <sim://mux_x01/>...
Instantiating architecture <mux2(arch)> as <sim://mux_x11/>...
Instantiating architecture <mux2(arch)> as <sim://mux_xy0/>...
Instantiating architecture <mux2(arch)> as <sim://mux_xy1/>...
Instantiating architecture <mux2(arch)> as <sim://mux_xyz/>...

-- Time: 0. ns --------------------------------------
Value for output signal <sim://m>: 'U'

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 1
Value for input signal <sim://z>? 0
Stabilized in 6. ns.

-- Time: 6. ns --------------------------------------
Value for output signal <sim://m>: '0'

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 1
Value for input signal <sim://z>? 1
Stabilized in 2. ns.

-- Time: 8. ns --------------------------------------
Value for output signal <sim://m>: '1'
. . .
```

2.2 Le point de départ

Comme vos TD-men vénérés ne sont pas vaches, ils vous ont un peu pré-mâché le boulot. Vous trouverez donc sur la page `http://perso.ens-lyon.fr/jeremie.detrey/06_asr1/` une archive avec une ébauche de simulateur, consistant uniquement pour l'instant en un lexer / parser. Le tout est écrit en Caml et vient avec un Makefile bien ficelé.

Remarque : Si vous êtes personnellement fâché-e avec Caml, vous pouvez aussi recoder tout ça en un autre langage de votre choix[2], même si nous pensons que cela serait une perte de temps (ainsi qu'une source non négligeable d'erreurs).

Voici un bref descriptif des fichiers présents dans l'archive :

[2]Pas trop exotique non plus, il faut quand même que nous arrivions à comprendre ce que vous avez écrit. Mettons comme impératif qu'il doit compiler sur les machines Linux des salles libre-service.

– `vhdl.ml` : C'est dans ce module que sont définis tous les types permettant de représenter en Caml le contenu d'un fichier VHDL. L'objet de base (de type `Vhdl.t`) peut être soit une entité, soit une architecture. Le module `Vhdl` contient aussi toute une batterie de fonctions pour afficher de manière structurée entités et architectures.
– `global.ml` : Pas grand chose dans ce module, si ce n'est la définition de quelques exceptions et autres variables globales nécessaire aux étapes d'analyse lexicale et syntaxique.
– `lexer.mll` : Le lexer (analyseur lexical). Rien d'intéressant ici, tout du moins du point de vue du cours d'architecture. Le lexer produit une suite de *tokens* qui est ensuite structurée par le parser.
– `parser.mly` : Le parser (analyseur syntaxique). C'est ici que la transformation entre le fichier VHDL et le format de représentation interne (l'arbre syntaxique, ou AST) est effectuée.
– `vhdlsim.ml` : Ce module contient la fonction principale du simulateur. Pour l'instant, celle-ci se contente d'appeler le lexer / parser sur chacun des fichiers passés en arguments, puis d'afficher les entités et architectures ainsi analysées.
– `Makefile` : Sans commentaire. Modifiez la variable `SRC_ML` pour rajouter d'autres fichiers sources.

Pour vous familiariser avec tout ça, commencez par saisir les entités et architectures présentées dans la section précédente dans des fichiers VHDL, puis lancez le programme `vhdlsim` sur ces fichiers.

2.3 Compilation

Comme déjà mentionné, la phase de compilation permet de passer de l'arbre syntaxique à la liste des entités effectivement décrites, et pour chacune d'entre elles l'architecture qui lui correspond. Pour mettre cela en œuvre, il suffit de passer chaque objet sortant du parser à une fonction chargée de maintenir une liste de toutes les entités définies et d'associer les architectures correspondantes à ces entités.

1. Implémentez donc cette fonction. Vous aurez peut-être besoin d'enrichir un peu certains types du module `Vhdl`.

2.4 Générer la netlist

L'instanciation du circuit s'effectue une fois tous les fichiers VHDL compilés. Il faut lui spécifier quelle est l'entité à simuler et possiblement aussi l'architecture, si plusieurs architectures ont été définies pour cette entité.

1. Discutez ainsi d'un format acceptable pour représenter la netlist correspondant au circuit simulé.

Gardez bien en tête qu'une architecture peut instancier plusieurs fois un même composant sous des noms différents, ce qui peut poser des soucis au niveau du nommage des fils de ces différentes instances.

De même, songez à comment représenter au niveau de la netlist une affectation directe de signaux (c'est-à-dire sans porte logique), comme par exemple :

```
x <= y;
```

2. Une fois que vous avez convergé vers une représentation pour la netlist, implémentez l'étape d'instanciation récursive des composants du circuit à simuler.

2.5 Simulation

Lors de la création de la netlist, tous les fils sont initialisés à la valeur inconnue `'U'`, sauf ceux qui ont été explicitement initialisés dans leur déclaration VHDL.

La simulation du circuit quant à elle est effectuée grâce à une file d'événements (transitions) ordonnés selon leur chronologie. Ainsi, lorsqu'un événement arrivant au temps t déclenche la transition de la sortie d'une porte, cette transition est insérée dans la file d'événement au temps $t + 1$ ns. On parle ici de *simulation à événements discrets*.

Dans un premier temps, votre simulateur ne devra rendre la main à l'utilisateur qu'une fois le circuit stabilisé, c'est-à-dire lorsque la file d'événements est vide. Ainsi, après avoir fixé la valeur des ports d'entrée, le simulateur tourne jusqu'à stabilisation éventuelle du circuit. C'est le fonctionnement qui est présenté dans la section précédente :

```
. . .
-- Time: 0. ns --------------------------------------
Value for output signal <sim://m>: 'U'

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 1
Value for input signal <sim://z>? 0
Stabilized in 6. ns.

-- Time: 6. ns --------------------------------------
Value for output signal <sim://m>: '0'

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 1
Value for input signal <sim://z>? 1
Stabilized in 2. ns.

-- Time: 8. ns --------------------------------------
Value for output signal <sim://m>: '1'
. . .
```

1. Implémentez ce module de simulation.
2. Est-il nécessaire de gérer les dépendances cycliques dans un circuit ? Si vous hésitez, jetez un œil à la section 3.2. Assurez vous que cela fonctionne bien pour votre simulateur.
3. Donnez un exemple de circuit qui ne se stabilise jamais. Peut-on détecter ce cas ? Si oui, comment ?

2.6 Simulation précise

Supposer que les entrées sont stables tant que le circuit n'est pas stabilisé est une hypothèse bien large. Modifiez donc votre simulateur pour que l'utilisateur puisse spécifier le temps que sont maintenues ces entrées. Une fois ce temps écoulé, la simulation s'interrompt pour lui permettre de modifier les valeurs appliquées aux entrées. Ainsi, par exemple, pour l'architecture `arch1` de la majorité :

```
. . .
-- Time: 0. ns --------------------------------------
Value for output signal <sim://m>: 'U'

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 0
Value for input signal <sim://z>? 1
Wait for? (ns) 0.5

-- Time: 0.5 ns -------------------------------------
```

```
Value for output signal <sim://m>: ’U’

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 1
Value for input signal <sim://z>? 1
Wait for? (ns) 2.5

-- Time: 3. ns ------------------------------------
Value for output signal <sim://m>: ’0’

Value for input signal <sim://x>? 0
Value for input signal <sim://y>? 1
Value for input signal <sim://z>? 1
Wait for? (ns) 0.5

-- Time: 3.5 ns -----------------------------------
Value for output signal <sim://m>: ’1’
. . .
```

1. Rajoutez cette fonctionnalité à votre simulateur.

2.7 *Bonus* : De jolis chronogrammes

L'idée est ici de pouvoir passer au simulateur une série de bits pour chaque port d'entrée, représentant les différentes valeurs appliquées à ce port au cours du temps, et de pouvoir obtenir en sortie un chronogramme représentant l'évolution des valeurs des signaux en entrée et en sortie du circuit.

La syntaxe utilisée pour spécifier les valeurs appliquées à un port d'entrée du circuit est la suivante :

$$val_1 \;\; délai_1 \;\; val_2 \;\; délai_2 \;\; val_3 \;\; \ldots \;\; délai_{n-1} \;\; val_n$$

On alterne de cette manière les valeurs val_i appliquées au port et les délais $délai_i$ (en nanosecondes) durant lesquels ces valeurs sont appliquées.

Ainsi, pour l'architecture `arch1` de la majorité, on peut avoir le chronogramme suivant :

```
Values for input signal <sim://x>? 0
Values for input signal <sim://y>? 0 0.5 1 0.5 0
Values for input signal <sim://z>? 1

<sim://x>:    __________________________________________________________________________________

<sim://y>:    __________/‾‾‾‾‾‾‾‾‾\_____________________________________________________________

<sim://z>:    ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

<sim://m>:    ------------------------------------------------------------._________/‾‾‾‾‾‾‾‾‾\__

Time:         | ’ ’ ’ ’ ! ’ ’ ’ ’ | ’ ’ ’ ’ ! ’ ’ ’ ’ | ’ ’ ’ ’ ! ’ ’ ’ ’ | ’ ’ ’ ’ ! ’ ’ ’ ’ | ’
              0. ns     0.5 ns    1. ns     1.5 ns    2. ns     2.5 ns    3. ns     3.5 ns    4. ns
```

1. Dans un premier temps, n'affichez pas le chronogramme mais juste la liste des diverses transitions sur les ports de sortie (et des instants auxquels elles se produisent). Ainsi, en reprenant l'exemple précédent :

   ```
   <sim://m>: ’U’ -> ’0’ at 3. ns
              ’0’ -> ’1’ at 3.5 ns
              ’1’ -> ’0’ at 4 ns
   ```

2. Passez au dessin du chronogramme.

3. Rajoutez la possibilité pour l'utilisateur de choisir la résolution du chronogramme. Ainsi, le chronogramme précédent était affiché avec une résolution de $0{,}1$ ns, mais on peut très bien vouloir une résolution moins fine pour afficher des chronogrammes plus grands. Par exemple, pour `majorite<arch2>`, avec une résolution de 1 ns, cela donne :

```
Values for input signal <sim://x>? 0
Values for input signal <sim://y>? 0 4.0 1 6.0 0
Values for input signal <sim://z>? 1

<sim://x>:     _______________________________

<sim://y>:     ________/‾‾‾‾‾‾‾‾‾‾‾\__________

<sim://z>:     ‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾‾

<sim://m>:     ------------.___/‾‾‾‾‾‾‾‾‾‾‾\__

Time:          | ' ' ' ' ! ' ' ' ' | ' ' ' ' !
               0. ns     5. ns     10. ns    15. ns
```

4. Au lieu d'afficher le chronogramme en ASCII-art, faites en sorte que votre simulateur génère un fichier représentant ce chronogramme sous forme vectorielle dans un format de votre choix (PostScript, XFig, MetaPost, etc...).

3 À vous de jouer

Pour chacune des entités que vous allez réaliser dans cette section, pensez bien à la tester exhaustivement à l'aide de votre simulateur.

3.1 Arithmétique

1. Décrivez une cellule HA (*half-adder*), prenant en entrée deux bits a et b, et calculant s le bit somme de a et b ainsi que co (*carry-out*) la retenue sortante.
2. Décrivez une cellule FA (*full-adder*), prenant en entrée deux bits a et b ainsi qu'une retenue entrante ci (*carry-in*), et calculant s le bit somme de a, b et ci ainsi que co la retenue sortante.
3. À l'aide de ces deux cellules, décrivez un additionneur à propagation de retenue prenant en entrée deux mots de 16 bits et renvoyant leur somme sur 17 bits.

3.2 Mémorisation

1. Décrivez une bascule RS (*RS latch*).
2. Décrivez une bascule D (*D latch*).
3. Décrivez un registre D (*D flip-flop*).
4. Observez et déterminez les caractéristiques temporelles de votre flip-flop : que se passe-t-il lorsque la valeur en entrée change juste avant le front montant ? Ou bien juste après celui-ci ? Combien de temps faut-il attendre pour que la donnée en sortie soit valide ?

Pour information, les vrais registres en VHDL ne se font pas du tout comme ça, mais à l'aide d'un *process*. Allez jeter un œil au *VHDL Cookbook* histoire de prendre peur.

3.3 *Bonus* : Composants asynchrones

On se place ici dans le cadre du protocole asynchrone 4-phases avec encodage double-rail.

1. Décrivez une porte de Muller (ou C-élément).

2. Décrivez un demi-buffer.

3. Construisez un pipeline de 6 demi-buffers en séries.

4. Combien de bits de données ce pipeline peut-il stocker au maximum ? Comment le vérifier avec votre simulateur ?

5. Décrivez les portes de base (ET, OU, XOR et NON) de la logique asynchrone suivant l'approche DIMS (*Delay-Insensitive Minterm Synthesis*), comme vu en TD.

6. Construisez les différents additionneurs asynchrones (fortement indicatif, faiblement indicatif et additionneur de Martin) vus lors du TD et observez les vitesses de convergence des états valides ou invalides dans ces additionneurs.

4 *Bonus* : Vers l'infini et au-delà

1. Proposez des extensions à votre simulateur VHDL parmi la foultitude de possibilités qu'offre ce langage[3]. Par exemple :
- les vecteurs de bits (`std_logic_vector`) ;
- les constructions de la forme `with` *signal* `select` ou encore les expressions conditionnelle du type *expr* `when` *condition* `else` *expr* ;
- la gestion des *packages* et *libraries* ;
- les *processes* ;
- ...

2. Faites un peu de recherche sur le type `std_logic` tel qu'il est décrit dans le standard IEEE 1164. Commentez ce que vous trouverez.

3. Réfléchissez à la réalisation d'un synthétiseur VHDL.

[3]Regardez le *VHDL Cookbook* pour une description plus complète.