Automates

Dans ce TD, on va construire une implémentation matérielle d'un algorithme, sous la forme d'un circuit séquentiel complexe avec séparation de la *partie contrôle* (automate à nombre fini d'états) et de la *partie données* (chemin de données). Pour gagner du temps, la partie données du circuit vous est fournie dans le sujet ; votre travail consistera à implémenter l'automate de contrôle, et surtout à exécuter le circuit pas à pas.

On veut construire un diviseur, c'est à dire, un composant qui calcule le quotient Q de la division entière d'un nombre positif ou nul N par un autre nombre positif P.

1 Sur papier !

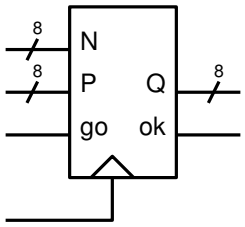
1.1 Algorithme

Afin de simplifier l'exercice, on va adopter un algorithme itératif simpliste, qui procède par additions successives en calculant p, $p+p$, $p+p+p$, $p+p+p+p$, et ainsi de suite jusqu'à dépasser N. Quand la somme partielle dépasse N, alors le résultat de la division est le nombre de fois qu'on a pu ajouter p à lui-même.
L'algorithme utilise deux variables auxiliaires, x pour la somme partielle, et q pour le quotient partiel. Lorsque $x+p$ dépasse n, alors on a terminé, et q est le résultat recherché.

```
n := entrée N
p := entrée P
x := 0
q := 0
tant que x+p ⩽ n
    x := x+p
    q := q+1
fin tant que
sortie Q := q
```

1.2 Interface

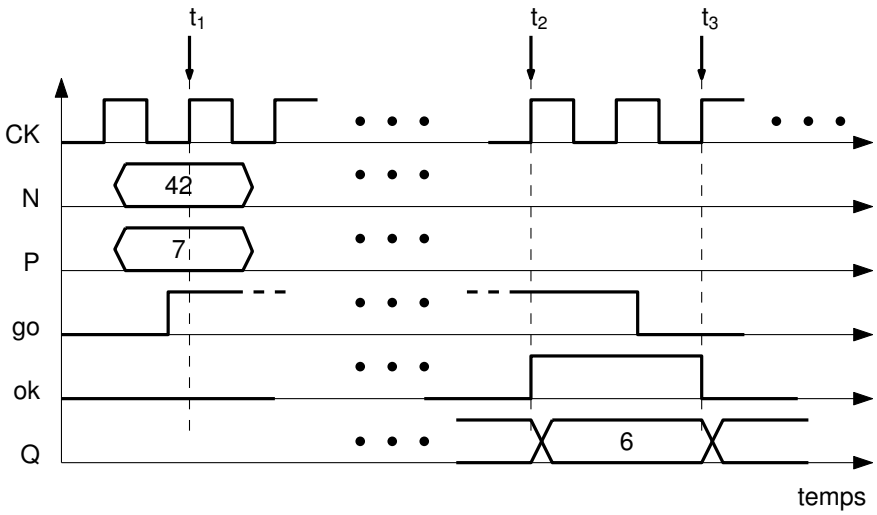
L'interface de notre circuit avec son environnement sera la suivante :



On suppose que les nombres N et P sont tous les deux positifs, et sont codés en binaire sur 8 bits. Le signal d'entrée *go* ordonne au composant de se mettre à travailler, et le signal de sortie *ok* permet au composant de signaler qu'il a terminé son calcul.

1.3 Protocole

Le protocole de communication avec notre diviseur est illustré dans le chronogramme ci-dessous, et expliqué dans le texte qui suit.

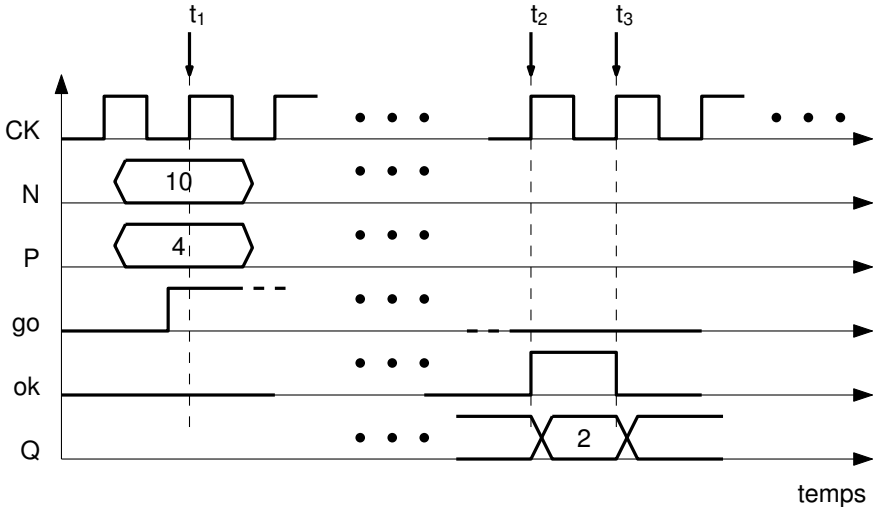


Pour demander une division, le client doit écrire *N* et *P* sur les entrées correspondantes, positionner *go* à 1 et attendre le prochain top d'horloge (instant t_1 sur le chronogramme). Le circuit mémorise alors ces entrées dans ses variables *n* et *p*, puis il déroule l'algorithme ci-dessus.

Lorsqu'il a terminé son calcul, le circuit positionne sa sortie *ok* à 1 (instant t_2) pour indiquer au client qu'il peut lire le résultat sur la sortie *Q* (d'autres valeurs intermédiaires ont peut-être été visibles sur *Q* entre-temps, mais elles ne sont pas significatives tant que *ok* n'est pas à 1).

Le circuit garde *ok* à 1 jusqu'à ce que le signal *go* soit retombé à zéro. La sortie *Q* est valide pendant ce temps-là. Lorsque *go* est de nouveau à zéro (instant t_3) le circuit retourne dans son état initial et attend une nouvelle commande. (Bien sûr le circuit ne réagit pas immédiatement lorsque *go* passe à zéro, mais seulement lors du top d'horloge suivant)

Note : Même si le client ne maintient pas *go* à 1 pendant tout le calcul, et que *go* est déjà à zéro au moment de t_2, notre circuit garde sa sortie valide pendant au moins un cycle d'horloge. Autrement dit, il y a toujours au moins un cycle d'écart entre les instants t_2 et t_3, comme illustré sur le chronogramme ci-dessous :



En résumé, on peut considérer que notre circuit implémente l'algorithme ci-dessous :

```
pour toujours faire
    attendre que go = 1
    n := entrée N
    p := entrée P
    x := 0
```

```
    q := 0
    tant que x+p ⩽ n
        x := x+p
        q := q+1
    fin tant que

    sortie ok := 1
    sortie Q := q
    attendre que go = 0
    sortie ok := 0
fin tant que
```

1.4 Chemin de données

Pour construire un tel circuit, on va essayer de séparer proprement le contrôle [1] des données, comme sur le schéma page suivante. La partie données vous est fournie : des registres pour les différentes variables, et des circuits combinatoires pour les calculs. Attention, tous les registres sont synchrones, comme dans le poly. Au moment du top d'horloge :

- si *reset* est vrai, le registre passe à zéro
- sinon, si *load* est vrai, le registre mémorise la valeur de D
- sinon, il garde sa valeur précédente.

Pour la partie contrôle, ce sera à vous de l'implémenter, mais les signaux de commande (loadN, loadX, ... resetQ) et de compte-rendu (pp) vous sont donnés. À défaut d'un meilleur nom, on appelle *pp* (pour «plus petit») le signal transmis par le comparateur à destination de l'automate.

1.5 Automate de contrôle

Question 1 Proposez, sous la forme d'un diagramme états-transitions, un automate capable de piloter ce chemin de données. Il s'agira d'un automate de Moore [2], donc vous indiquerez

- sur chaque transition, le symbole d'entrée attendu,
- sur chaque état, le symbole de sortie émis.

Question 2 Donnez les tables de vérité de la fonction de transition et de la fonction de sortie.

1.6 Exécution

Question 3 Complétez le chronogramme page 5, jusqu'à la fin de la division.

Les premières lignes représentent les données aux entrées et aux sorties du circuit. La ligne verticale pointillée repère le top d'horloge où l'algorithme démarre. Après cet instant, les entrées N et P ne sont plus utiles, leur valeur n'est donc pas importante. Pour le signal *go*, vous pouvez supposer que le client garde *go* à 1 jusqu'au bout de l'opération, ou qu'il le remet à zéro plus tôt.

Les lignes suivantes représentent les valeurs des registres du chemin de données. Leur valeur initiale est inconnue (notée « ? » sur le chronogramme).

Les deux lignes suivantes représentent l'état courant de l'automate de contrôle (traditionnellement noté q, à ne pas confondre avec le quotient de la division) et l'état suivant calculé par la fonction de transition, noté q'.

Les dernières lignes représentent les signaux de contrôle de de compte-rendu échangés entre la partie contrôle et le chemin de données.

1. Le terme "contrôle" est un vilain anglicisme pour *control* qui veut dire "commande". Mais on n'est pas là pour réparer le français.
2. ... c'est à dire comme dans le poly. Il y a aussi des automates de Mealy, vous pouvez demander au prof ce que c'est.

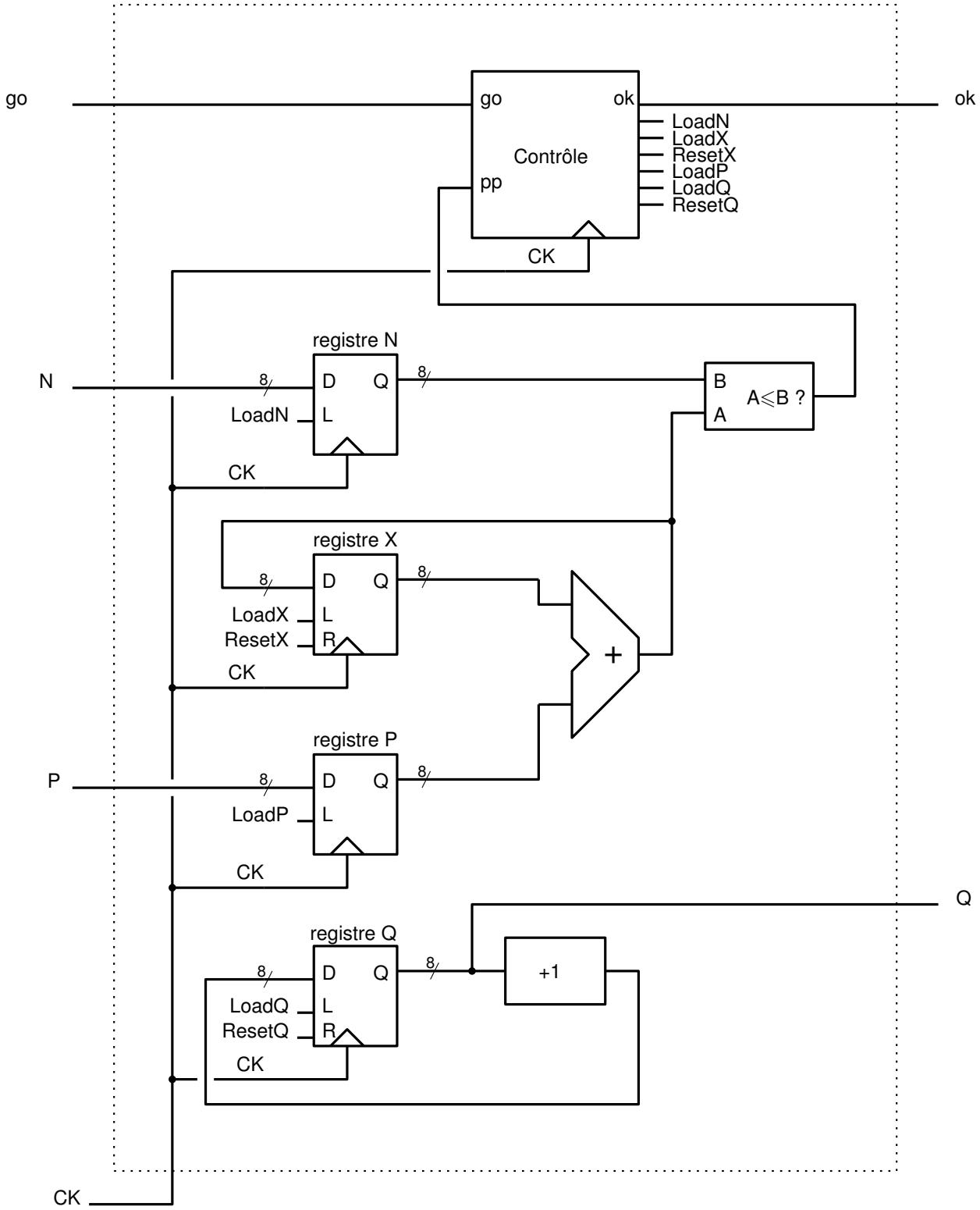


FIGURE 1 – Le circuit du diviseur


CK
entrée N
12
entrée P
5
go
ok
sortie Q
?
registre N
?
registre P
?
registre Q
?
registre X
?
état courant q
état suivant q'
pp
LoadN
LoadX
ResetX
LoadP
LoadQ
ResetQ

2 En Logisim

Dans cette partie, vous allez réaliser votre diviseur dans Logisim, comme un assemblage de portes élémentaires pour concevoir l'automate, puis d'un assemblage de composants plus gros que vous avez déjà conçu précédemment, pour le diviseur dans son ensemble.

2.1 Construction de l'automate

On va commencer par construire le circuit implémentant le comportement de l'automate construit à la section 1.5 en s'attaquant d'abord à la fonction de transition puis à sa fonction de sortie.

2.1.1 Fonction de transition

La fonction de transition prend pour entrée l'état courant q et les signaux *go* et *pp*. Elle produit l'état suivant q'. Si vous avez construit un automate avec 4 états, q' est codé sur 2 bits. La fonction de transition a donc deux bits en sorties, il vous faudra donc construire 2 équations.

Question 4 A partir des tables de vérités construites à la section 1.5, donner les équations booléennes pour q' en fonction de q.

Question 5 A partir des équations booléennes, construisez le circuit correspondant.

2.1.2 Fonction de sortie

Question 6 Procédez de la même façon (équations booléennes puis circuits) pour les sorties de votre automates, à savoir : *loadN*, *loadP*, *loadX*, *resetX*, *loadQ*, *resetQ* et *ok*.

2.2 Construction du diviseur

2.2.1 Il manque encore ...

Vous avez maintenant à votre disposition presque tous les composants nécessaires à la construction du diviseur : votre automate, des registres, des additionneurs. Il vous manque encore un **comparateur** qui vous permettent de réaliser la boite $A \leq B$?.

Question 7 Implémentez ce comparateur.

2.2.2 "Et Voilà"

Il n'y a plus qu'à assembler tout ce beau monde pour reproduire à l'identique la figure 1. C'est tellement gratifiant de voir son diviseur fonctionner :)
Note : c'est gratifiant, certes mais en même temps très laborieux. Dans le chapitre suivant, vous allez pouvoir apprécier la puissance de VHDL et vous rendre compte que derrière une syntaxe "aride" (mais quel langage n'est pas aride au début ?), construire votre diviseur va aller beaucoup plus vite ! ! !

2.3 Simulez et corrigez votre chronogramme si besoin

3 En VHDL

3.1 Les automates en VHDL

Vous trouverez dans `passage_a_niveau.vhdl` une implémentation de l'automate du passage à niveaux vu en cours (chapitre 6 du poly pour ceux qui dormaient). Elle n'est pas terminée. Comprenez la suffisamment pour la terminer par des copier-coller judicieux. Complétez aussi le banc de test.
Faites valider par un enseignant : on veut voir une fenêtre gtkwave montrant les deux capteurs, l'état et l'ampoule, dans différentes configurations de trains venant de droite et de gauche. En attendant que l'enseignant arrive, ajoutez une sortie appelée `alerte` qui passe à 1 dans l'état `cata`.
Remarque : Il y a des dizaines de manières correctes de décrire des automates en VHDL. Nous utilisons ici un style qui suit la méthodologie du poly :

- *le choix du codage des états est laissé à l'ingénieur ;*
- *on fait bien apparaître la fonction de transition, la fonction de sortie, et le registre d'état de la figure du poly ;*
- *par contre les fonctions de transitions et de sortie sont données en intention, de manière comportementale : on va laisser les outils de synthèse les compiler en des fonctions booléennes.*

Notez toutefois que si vous avez oublié des cas, le compilateur ne vous le dit pas, et vous obtiendrez des 'U' (undefined) dans vos simulations.
Remarquez aussi que ce style n'utilise qu'une seule entité pour tout l'automate. C'est plus concis et plus lisible.
Pour ceux qui veulent creuser : d'autres styles sont possibles. Pour en avoir une idée, vous pouvez googler "VHDL FSM". Par exemple il est possible de définir les états par un simple type énuméré, et laisser le choix du codage des états au compilateur. C'est plus simple mais on a moins de contrôle sur les détails d'implémentation. Il y a aussi des styles qui sont encore plus proches du dessin des patates, mais ils demandent de bien comprendre la logique des process.
Dans la suite de ce TP et du suivant, on conservera le style utilisé ici.

3.2 Diviseur

On reprend la figure 1. Vous trouverez dans `diviseur.vhdl` une implémentation incomplète de ce diviseur. Ici aussi, on utilise une seule entité pour tout le diviseur. C'est possible entre autres parce qu'on utilise une bibliothèque standard, `numeric_std`, qui permet de décrire un additionneur synthétisable par un simple "+" entre deux vecteurs de bits. Ces vecteurs ne doivent pas être déclarés comme `std_logic_vector` mais comme `signed` ou `unsigned`. Voyez le code de l'exemple pour plus de détails, en particulier pour voir comment on fait les conversions d'un type en l'autre.
On rappelle qu'on attrape le bit 3 d'un vecteur `toto` par `toto(3)`.
Commencez par bien comprendre le VHDL qui vous est donné. Il est conseillé de commencer par annoter chaque fil sur le dessin avec le nom du signal correspondant dans le VHDL. Remarquez que la fonction de transition est donnée en intention, et non pas au niveau de détail qu'on avait en logisim. C'est plus simple à écrire, et plus lisible à lire.
Puis completez-le jusqu'à observer la division dans le `testbench`.
Un détail technique : pour économiser une entrée de reset qui définit l'état de départ de l'automate, vous pourrez utiliser le signal `go`. Du coup (par rapport à la première section) il devient interdit de faire passer go à 0 avant le passage de done à 1. Oh, c'est un handshake.

4 Et pour les plus rapides

Posez une division en binaire (comme vous avez appris en CM2 en décimal, mais en binaire). Déduisez-en un algorithme plus rapide que celui que nous avons utilisé, et implémentez-le en logisim ou en VHDL.
Une fois l'algorithme compris, cela peut être une question d'un quart d'heure.