V Méthodes de synthèse

Au cours des quinze dernières années, les méthodes de conception des fonctions numériques ont subi une évolution importante. Dans les années soixante dix, la majorité des applications étaient construites autour de circuits intégrés standard, souvent pris dans la famille TTL. Au début des années quatre vingt apparurent parallèlement :

- les premiers circuits programmables par l'utilisateur (PALs), du côté des circuits simples,
- les circuits intégrés spécifiques (ASICs)[1], pour les fonctions complexes fabriquées en grande série.

La complexité des seconds a nécessité la création d'outils logiciels de haut niveau, qui sont à la description structurelle (schémas au niveau des portes élémentaires) ce que les langages évolués sont au langage machine dans le domaine de la programmation.

Les premières générations de circuits programmables étaient conçus au moyen de simples programmes de traduction d'équations logiques en table de fusibles. A l'heure actuelle, l'écart de complexité entre circuits programmables et ASICs s'est restreint : on trouve une gamme continue de circuits qui vont des héritiers des premiers PALs, équivalents de quelques centaines de portes, à des FPGAs ou des LCAs[2] de quelques dizaines de milliers de portes équivalentes. Les outils d'aide à la conception se sont unifiés, un même langage, VHDL par exemple, peut être employé quels que soient les circuits utilisés, des PALs aux ASICs.

Le remplacement, dans la plupart des applications, des fonctions standard complexes par des circuits programmables, s'accompagne d'un changement dans les méthodes de conception :

- On constate un « retour aux sources » : le concepteur d'une application élabore sa solution en descendant au niveau des bascules élémentaires, au même titre que l'architecte d'un circuit intégré.
- L'utilisation systématique d'outils de conception assistée par ordinateur (C.A.O.), sans lesquels la tâche serait irréalisable, rend caducs les fastidieux

[1]Programmable Array Logic, Application Specific Integrated Circuit.
[2]Field Programmable Logic Array, Logic Cell Array.

calculs de minimisation d'équations logiques. Le concepteur peut se consacrer entièrement aux choix d'architecture qui sont, eux, essentiels.

– La complexité des fonctions réalisables dans un seul circuit pose le problème du test. Les outils traditionnels de tests de cartes imprimées, du simple oscilloscope à la « planche à clous » en passant par l'analyseur d'états logiques, ne sont plus d'un grand secours, dès lors que la grande majorité des équipotentielles sont inaccessibles de l'extérieur. Là encore, la C.A.O. joue un rôle essentiel. Encore faut-il que les solutions choisies soient analysables de façon sûre. Cela interdit formellement certaines astuces, parfois rencontrées dans des schémas traditionnels de logique câblée, comme des commandes asynchrones utilisées autrement que pour une initialisation lors de la mise sous tension, par exemple[3].
– Les langages de haut niveau, comme VHDL, privilégient une approche globale des solutions. Dès lors que l'architecture générale d'une application est arrêtée, que les algorithmes qui décrivent le fonctionnement de chaque partie sont élaborés, le reste du travail de synthèse est extrêmement simple et rapide.

Nous tenterons ci-dessous de mettre en évidence quelques règles de conception et de donner au lecteur les clés de compréhension de la littérature spécialisée, notamment les notices des fabriquants de circuits et les notes d'application qui les accompagnent.

V.1. Les règles générales

Avant de présenter les outils de base du concepteur, il n'est sans doute pas inutile de préciser quelques règles, qui pourront sembler de simple bon sens, mais dont le non respect a conduit beaucoup de réalisations vers le cimetière des projets morts avant d'être nés.

Du général au particulier, une approche descendante

L'erreur de méthode la plus fréquente, et la plus pénalisante, que commettent beaucoup de débutants dans la conception des systèmes électroniques, qu'ils soient analogiques ou numériques, est sans doute de dessiner des schémas, voire de les câbler, avant même d'avoir une vision claire de l'ensemble de la tâche à accomplir. Le travail de réflexion sur la structure générale d'une application est primordial.

Ce que l'on appelle traditionnellement la méthode descendante (*top down design*), n'est rien d'autre que l'application de cette règle simple : quand on conçoit un ensemble, *on va du général au particulier*, on ne s'occupe des détails que quand le cahier des charges a été mûrement réfléchi, et que le plan général de la solution a été établi.

[3]Même à l'époque du règne des fonctions standard, ces pratiques étaient éminemment douteuses.

Si, au cours de la descente vers les détails, on découvre qu'une difficulté imprévue apparaît, il faut revenir au niveau général pour voir comment la réponse à cette difficulté s'insère dans le plan d'ensemble.

Diviser pour règner

Le premier réflexe à avoir, face à un problème, un tant soit peu complexe à résoudre, est de le couper en deux. La démarche précédente est répétée, pour chaque demi-problème, jusqu'à obtenir des sous-ensembles dont la réalisation tient en quelques circuits élémentaires, en quelques lignes de code source dans un langage ou dans un diagramme de transitions qui ne dépasse pas une dizaine d'états différents.

L'un des auteurs de ce livre garde un souvenir cuisant de la première introduction d'un système de CAO, pour réaliser un projet d'électronique numérique, auprès d'étudiants d'IUT que nous n'avions pas suffisamment averti des pièges liés à la puissance de l'outil. Ce système pouvait assurer automatiquement la répartition d'une application dans plusieurs circuits programmables dont on avait, au préalable, établi la liste.

Le problème posé était relativement simple ; il s'agissait de créer un automate pilotant un circuit de multiplication, suivant un algorithme séquentiel, et assurant l'interface entre ce circuit et un microprocesseur. Trois sous-ensembles en interaction devaient être créés :

- Un interface avec le bus du microprocesseur, qui assure le bon respect du protocole d'échange.
- Un compteur qui permet de savoir où en est la multiplication : à chaque impulsion d'horloge le multiplieur fournit un chiffre binaire du résultat, si le produit est codé sur 16 bits, il faut attendre 16 périodes d'horloge pour l'obtenir.
- L'automate séquentiel qui pilote le tout.

Trois blocs, dont la réalisation nécessite, en tout, une douzaine de circuits standard TTL.

La fusion des trois blocs dans une grande boite « fourre tout », ne rentre pas dans un seul circuit de type 22V10, ce qui est normal. Si l'utilisateur du système de CAO laisse ce dernier se charger lui-même du découpage de la réalisation en sous-ensembles, le résultat est une carte qui contient, outre le multiplieur lui-même, une ribambelle de circuits programmables, utilisés à 50% de leur capacité.

Le simple fait de subdiviser la solution en petits modules, ce qui permet de guider le logiciel de placement, divise par deux le nombre de circuits nécessaires et, soit dit en passant, permet d'en contrôler facilement le bon fonctionnement.

Mener de front l'aspect structurel et l'aspect fonctionnel

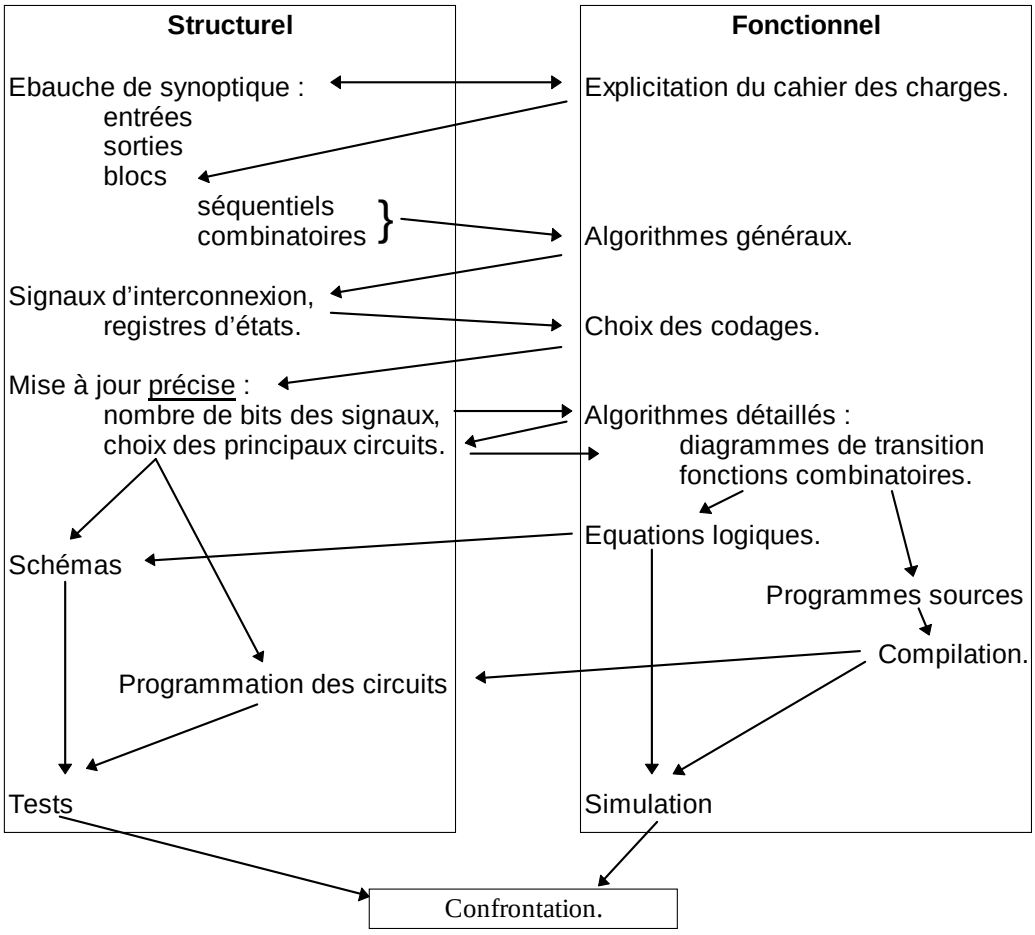
La réalisation d'un ensemble électronique se mène sur deux plans parallèles, structurel et fonctionnel, qui doivent, à chaque étape, être cohérents entre eux.

Le plan structurel précise les subdivisions en blocs, combinatoires ou séquentiels, fixe les signaux d'interconnexions entre ces blocs, pour aboutir, en

dernière étape, à un éventuel schéma. Une fois réalisés, chaque bloc et l'ensemble sont soumis à des tests de validation.

Le plan fonctionnel précise l'algorithme utilisé dans chaque bloc, fixe le codage des signaux, pour aboutir, en dernière étape, à des équations logiques ou à des modules de programme dans le langage choisi. Le fonctionnement de chaque sous-ensemble et du système complet peut être simulé.

Le tableau qui suit illustre le processus :



Citons quelques incohérences graves parfois rencontrées entre les deux plans :

- Les noms de signaux ne correspondent pas.
- Les tailles des registres d'états sont incompatibles avec les diagrammes de transitions qui les décrivent ; une bascule se voit pourvue de plus de deux états, ou, inversement, un diagramme qui contient cinq états est censé représenter le fonctionnement d'un registre de quatre bascules.
- Les équations de certaines commandes de circuits complexes ne sont pas précisées.

Faciles à corriger au début, de telles incohérences sont sources d'erreurs difficiles à identifier quand elles apparaissent lors des tests de validation.

Aléas et testabilité : privilégier les solutions synchrones

Une plaie de trop de réalisations rencontrées est le mélange, dans une même unité fonctionnelle, des commandes asynchrones et synchrones. Citons quelques exemples :

Mises à zéro ou chargements parallèles

Les mises à zéro ou les chargements parallèles de registres, par des commandes à action directe, c'est à dire indépendamment de l'horloge, sont la source de nombreux ennuis ultérieurs ; ce type de pratique est à condamner sans appel. L'utilisation de telles commandes en fonctionnement normal conduit à la génération d'impulsions de durées inconnues, souvent très faibles, donc difficiles à observer. Les outils de test et de simulation gèrent fort mal ces commandes, il devient impossible de valider correctement la fonctionnalité d'un équipement qui les utilise.

L'utilisation de ces commandes asynchrones conduit parfois à un résultat qui, s'il peut être instructif dans un contexte d'enseignement, est catastrophique dans une réalisation : une carte qui semble donner toute satisfaction quand on l'observe, par exemple avec un oscilloscope, cesse de fonctionner dès que l'on retire l'appareil de mesure. L'origine du phénomène tient à la charge capacitive supplémentaire apportée par la sonde de mesure. Cette charge peut, si elle est bien placée, modifier la durée et l'amplitude d'impulsions étroites dont l'existence est déterminante pour le bon fonctionnement de l'ensemble. Il est, nous l'espérons, inutile de préciser que le dépannage d'un tel objet relève plus de la divination que d'une méthodologie raisonnée[4].

Les signaux d'horloges

Le bloquage, par exemple par une porte, des signaux d'horloge pour maintenir l'état d'un registre, est une autre erreur que l'on rencontre parfois. Cette faute, qui provoque des décalages temporels entre les signaux d'horloge (*clock skew*) appliqués aux différentes parties d'une carte, ou d'un circuit, risque de conduire à des violations de temps de maintien ou de prépositionnement, d'où des comportements imprévisibles des registres concernés.

Un autre effet pervers des circuits combinatoires de « calcul » des signaux d'horloge, est la génération, difficile à contrôler, d'impulsions parasites sur ces signaux. La recherche de ces impulsions, suffisamment larges pour faire commuter les circuits actifs sur des fronts, mais suffisamment étroites pour ne pas être vues lors d'un examen rapide avec un oscilloscope, est un passe temps dont on se lasse très vite.

Quand il est nécessaire d'appliquer à différentes parties d'un ensemble des signaux d'horloges différents, il est indispensable de traiter à part, et de façon

[4]Un effet réciproque peut également être rencontré : la carte qui fonctionne quand on ne la regarde pas, et qui tombe en panne quand on l'observe, la sonde de l'appareil ayant « gommé » une impulsion essentielle, bien que fragile, à la bonne santé de l'ensemble.

méticuleuse, la réalisation du distributeur d'horloge correspondant[5]. Notons, en passant, que pour ces fonctions il convient de surveiller de très près les modifications apportées par les optimiseurs ; ces derniers ont la fâcheuse tendance d'éliminer les portes inutiles d'un point de vue algébrique, même si elles sont utiles d'un point de vue circuit.

Les bascules asynchrones (D Latch, R S)

Ce chapitre est consacré aux méthodes de conception des automates séquentiels. Ces opérateurs ont la particularité de fonctionner en boucle fermée : l'état futur dépend de l'état initial. Ce mode de fonctionnement exclut à priori l'usage de bascules ou registres asynchrones dans la réalisation de tels systèmes.

Les bascules D latch ou R-S ont, parfois, leur place à la périphérie des systèmes, elles servent alors d'interfaces entre deux ensembles indépendants, pilotés par des horloges différentes, qui échangent des informations en respectant un protocole bien défini.

V.2. Les machines synchrones à nombre fini d'états

Les machines à nombre fini d'états (*Finite state machines*), en abrégé machines d'états, ou automates finis ou, encore, séquenceurs câblés[6], sont largement utilisées dans les fonctions logiques de contrôle, qui forment le coeur de nombreux systèmes numériques : arbitres de bus, circuits d'interfaces des systèmes à base de microprocesseurs, circuits de gestion des protocoles de transmission, systèmes de cryptages etc. Plus prosaïquement, la quasi totalité des fonctions séquentielles standard, compteurs, par exemple, peuvent être analysées, ou synthétisées, en adoptant le point de vue « machine d'états ».

Une machine d'états est un système dynamique (i.e. évolutif) qui peut se trouver, à chaque instant, dans une position parmi un nombre fini de positions possibles. Elle parcourt des cycles, en changeant éventuellement d'état lors des transitions actives de l'horloge, dans un ordre qui dépend des entrées externes, de façon à fixer sur ses sorties des séquences déterminées par l'application à contrôler.

---

[5]Le seul argument sérieux qui peut, parfois, et avec une extrême prudence, conduire un concepteur à utiliser le bloquage de l'horloge, est la consommation : un circuit séquentiel, surtout en technologie CMOS, dont on arrête l'horloge, consomme moins que si le maintien est généré de façon synchrone. Comme quoi les règles doivent être édictées, mais parfois transgressées. Mieux vaut, dans ces cas, avoir conscience qu'il y a transgression, donc danger.

[6]Les différences de terminologie correspondent à des différences de point de vue : l'électronicien s'intéresse à la réalisation de la machine, donc à son fonctionnement interne. Il concentre son attention sur le fonctionnement d'un registre qui peut passer d'un état interne à un autre, en fonction de commandes qui lui sont fournies. L'automaticien et l'informaticien s'intéressent, de prime abord, au fonctionnement externe du même objet : ils en attendent des actions (automates) en sortie correspondant à une séquence (séquenceurs) fixée par l'application à laquelle la machine est destinée. Peu importent, à ce niveau et jusqu'à un certain point, les détails de réalisation interne du séquenceur.

Un programmateur de machine à laver en est l'illustration typique : suite à la mise en marche, le programmateur contrôle que la porte est fermée, si oui il commande l'ouverture de la vanne d'arrivée de l'eau, quand la machine est pleine etc.

L'architecture générale d'une machine d'états simple est celle de la figure V-1 :

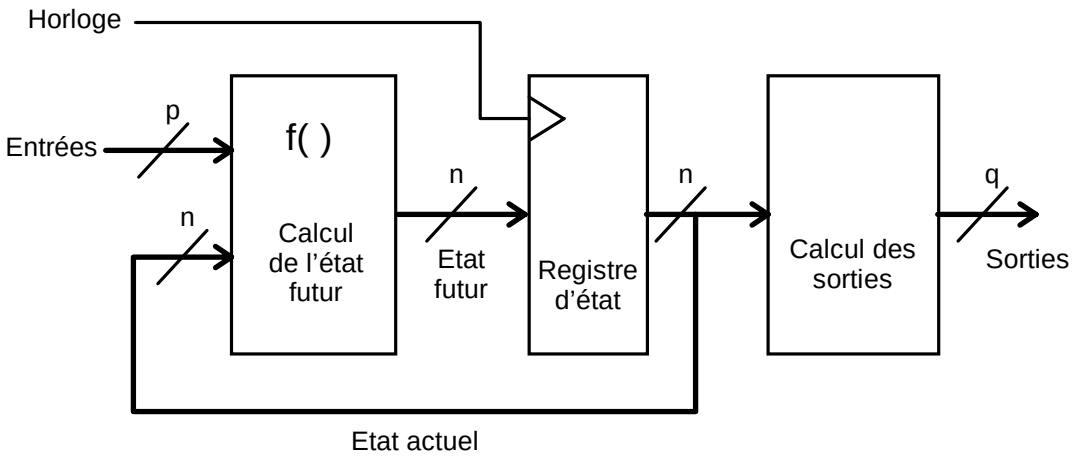


Figure V-1

Ce synoptique général va nous servir, éventuellement légèrement modifié, de support dans les explications qui suivent.

V.2.1 Horloge, registre d'état et transitions

Le registre d'état, piloté par son horloge, constitue le coeur d'une machine d'états, les autres blocs fonctionnels, bien que généralement nettement plus complexes, sont « à son service ».

Le registre d'état

Le registre d'état est constitué de n bascules synchrones, nous admettrons dans ce qui suit, sauf précision contraire, que ce sont des bascules D, ce qui n'impose aucune restriction de principe, puisque nous savons passer d'un type de bascule à un autre.

Etat présent et état futur.

Le contenu du registre d'état représente l'état de la machine, il s'agit d'un nombre codé en binaire sur n bits, dans un code dont nous aurons à reparler.

L'entrée du registre d'état constitue l'état futur, celui qui sera chargé lors de la prochaine transition active de l'horloge. Le registre d'état constitue la mémoire de la machine, l'élément qui matérialise l'histoire de son évolution.

L'importance de ce registre est telle que beaucoup de circuits programmables offrent la possibilité de le charger, à des fins de tests, avec une valeur arbitraire, indépendante du fonctionnement normal de la machine. Toute procédure de test devra s'appuyer sur la connaissance du contenu de ce registre[7], par une mesure réelle, ou en simulation.

Taille du registre et nombre d'états

La taille du registre d'état fixe évidemment le nombre d'états accessibles à la machine. Si n est le nombre de bascules et N le nombre d'états accessibles, ces deux nombres sont reliés par la relation :

$$N = 2^n$$

Cette relation, qui ne présente guère de difficulté, est pourtant trop souvent oubliée des concepteurs débutants :

- En synthèse le nombre d'états nécessaires est issu du cahier des charges de la réalisation, on en déduit aisément une taille minimum du registre.
- Quand tous les états disponibles ne sont pas utilisés, l'oubli des états inutilisés peut conduire à de cruelles déconvenues si on oublie de prévoir leur évolution.

Le rôle de l'horloge

Le rôle de l'horloge, dans une réalisation synchrone, est de supprimer toute possibilité d'aléas dans l'évolution de l'état. Idéalement, entre deux transitions actives de l'horloge le système est figé, en position mémoire, son état ne peut pas changer. Dans la réalité, le temps de latence qui sépare deux fronts consécutifs du signal de l'horloge est mis à profit pour permettre aux circuits combinatoires d'effectuer leurs calculs, sans que les temps de retards, toujours non nuls, ne risquent de provoquer d'ambiguïté dans le résultat.

Ce qui a été dit à propos des bascules élémentaires se généralise : chaque transition d'horloge est suivie d'une période de grand trouble dans la valeur de l'état futur, le concepteur doit s'assurer que cette valeur est stable quand survient la transition d'horloge suivante (figure V-2)[8] :

---

[7]Ce point peut être moins trivial qu'il n'y paraît, les sorties des bascules ne sont pas toujours disponibles en sortie d'un circuit, on parle alors de bascules enterrées. Dans de tels cas les circuits complexes offrent de plus en plus souvent la possibilité de « ressortir », par une procédure particulière, les contenus de ces bascules (c'est l'une des fonctions possibles des automates de test dits « *boundary scan* » qui sont intégrés dans certains circuits).

[8]Ce qui est dit ici est en étroite relation avec le contenu du paragraphe II-3, qui aboutit au calcul de la fréquence maximum de fonctionnement d'un circuit dans lequel interviennent des rétro couplages.

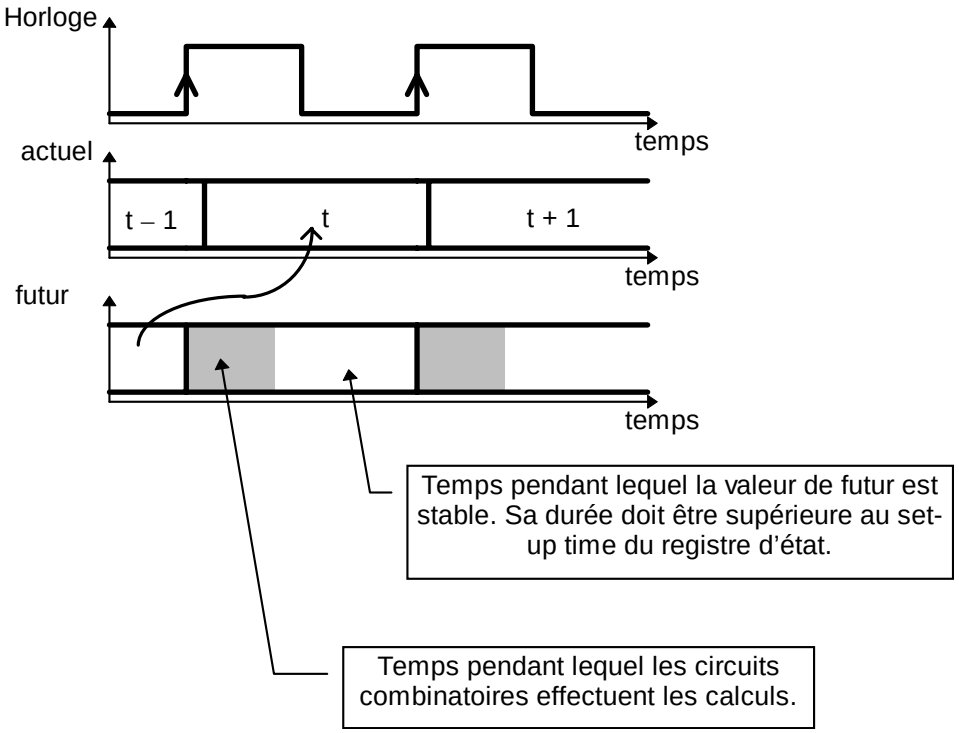


Figure V-2

Le temps devient une variable discrète

Sauf pour le calcul des limites de fonctionnement du système, le temps devient une variable discrète, un nombre entier qui indique simplement combien de périodes d'horloges se sont écoulées depuis l'instant pris comme origine.

Une machine d'états est complètement décrite par une équation de récurrence, qui permet de connaître le contenu du registre d'état à l'instant t, t entier, en fonction de ses valeurs précédentes et de celles des entrées.

Les entrées et l'état actuel déterminent l'état futur

Dans une architecture comme celle de la figure V-1, l'équation de récurrence évoquée précédemment est du premier ordre (la portée temporelle de la mémoire est égale à une période d'horloge) :

$$Etat\_actuel(t) = Etat\_futur(t - 1)$$

Or la fonction logique combinatoire, f( ), qui calcule l'état futur, fournit une valeur qui dépend de l'état présent et des entrées, d'où la relation générale qui décrit l'évolution d'une machine d'états :

$$Etat\_actuel(t) = f(Etat\_actuel(t - 1) , Entrées(t - 1))$$

Cette équation est la généralisation de celles qui ont été introduites à propos des bascules élémentaires.

Il est important de noter que, si l'équation qui régit l'évolution du registre d'état, pris dans son ensemble, est du premier ordre, ce n'est pas vrai pour l'équation d'évolution d'*une* bascule qui est, elle, plus complexe. Toutes les bascules du registre d'état sont, en général, couplées. L'ordre de l'équation d'évolution d'une bascule peut atteindre n, où n est la taille du registre d'état[9].

Exemple : un diviseur de fréquence par 3 ou 4

Pour illustrer ce qui précède, considérons le schéma de la figure V-3 :

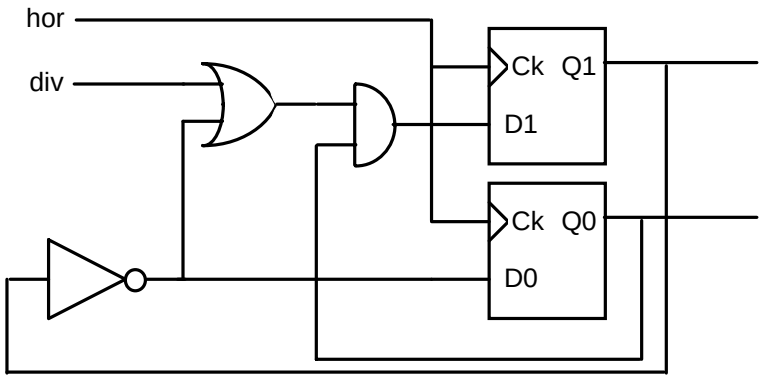


Figure V-3

L'état du système est (Q1,Q0), ensemble constitué des états individuels des deux bascules. On admet que l'entrée extérieure div est synchrone de l'horloge. A chaque front montant du signal d'horloge hor, le système évolue suivant le système d'équations :

$$Q1(t) = Q0(t - 1) * (\overline{Q1(t - 1)} + div(t - 1))$$
$$Q0(t) = \overline{Q1(t - 1)}$$

On peut remarquer, en passant, que l'équation de chaque bascule est effectivement du deuxième ordre, par exemple pour Q1 :

[9]On peut rapprocher ce point des méthodes d'analyse des circuits analogiques : la présentation « variables d'état » conduit à une équation différentielle du premier ordre, qui porte sur un vecteur de dimension n, la présentation traditionnelle considère chaque grandeur électrique comme obéïssant individuellement à une équation différentielle dont l'ordre peut atteindre n. Le passage d'un mode de représentation à l'autre n'est simple que dans le cas des équations linéaires. Les équations des systèmes numériques sont généralement non linéaires, avec une exception notable : les générateurs de séquences pseudo aléatoires qui utilisent un registre à décalage rebouclé par des sommes modulo 2.

$$Q1(t) = \overline{Q1(t-2)} * (\overline{Q1(t-1)} + div(t-1))$$

Sur le chronogramme de la figure V-4, construit à partir des équations précédentes, on voit que :

- si div = '0' les sorties Q1 et Q0 sont des signaux périodiques, de fréquence égale au tiers de la fréquence d'horloge,
- si div = '1' la fréquence des signaux Q1 et Q0 est égale au quart de la fréquence d'horloge.

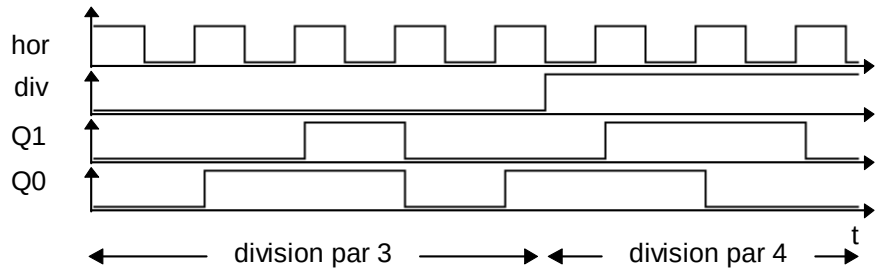


Figure V-4

Si les chronogrammes permettent d'illustrer un fonctionnement, ils ne constituent pas une méthode efficace d'analyse, et, encore moins, de synthèse. Dans l'étude de machines d'états simples, les diagrammes de transitions constituent une approche plus compacte, donc plus puissante, tout en fournissant exactement le même niveau de détails.

Les diagrammes de transitions

Dans un diagramme de transitions, on associe à *chaque* valeur possible du registre d'état, une case. L'évolution du système est représentée par des flèches, les transitions, qui vont d'un état à un autre[10]. Comme pour les bascules élémentaires, une transition est effectuée si trois conditions sont réunies :

1. Le système est dans l'état « source » de la transition considérée,
2. une éventuelle condition de réalisation sur les entrées doit être vraie,
3. un front actif d'horloge survient.

[10]Les automaticiens utilisent souvent un diagramme similaire, le GRAFCET, dont les étapes peuvent être matérialisées par les états d'une machine. Nous adoptons les diagrammes de transitions plutôt que le GRAFCET, car seuls les premiers sont utilisés, et le sont beaucoup, dans la littérature professionnelle électronique (notices d'applications de circuits, manuels des systèmes de CAO, etc.). Le passage d'un type de diagramme à l'autre ne présente guère de difficulté.

Si aucune transition n'est active, le système reste dans son état initial. S'il n'y a pas d'ambiguïté le signal d'horloge est généralement omis (horloge unique), mais il conditionne toutes les transitions.

Reprenant l'exemple précédent, nous représentons, figure V-5, le diagramme de transitions du diviseur par trois ou quatre. Le registre d'état contient deux bascules, soit quatre états accessibles. Pour chaque état initial, les équations du diviseur fournissent l'état d'arrivée.

Dans cet exemple, quel que soit l'état de départ, et quelle que soit la valeur de l'entrée div, il y a toujours une transition active ; le système change donc d'état à chaque front montant du signal d'horloge.

Quand le diviseur est dans l'état 3 le chemin parcouru dépend de la valeur de l'entrée div. Si div = '0' un cycle complet contient trois états, d'où la division de fréquence par trois, si div = '1', un cycle complet comporte quatre états, d'où la division de fréquence par quatre[11].

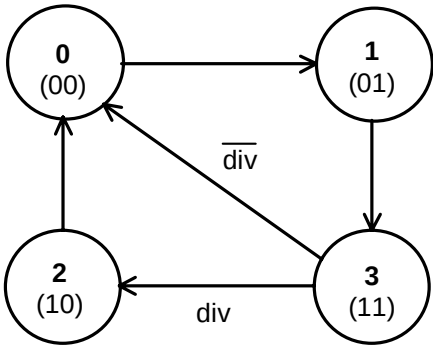


Figure V-5

Dans l'exemple précédent, nous sommes partis des équations d'un système pour aboutir au diagramme de transitions qui en décrit le fonctionnement. Il s'agit là d'un travail d'analyse. Le concepteur se trouve généralement confronté au problème inverse : du cahier des charges il déduit un diagramme de transitions, et de ce diagramme il souhaite tirer les équations de commandes des bascules du registre d'état, ou une description en VHDL[12].

[11]Cet exemple est une version simple de ce que l'on appelle les diviseurs par N/(N+1). Ces circuits sont utilisés dans les synthétiseurs de fréquences, à boucle à verrouillage de phase, avec des valeurs plus grandes de N (255, par exemple).

[12]Il existe des logiciels de traduction des diagrammes de transitions en code source, dans un langage. Le plus souvent le programme obtenu s'apparente plus à une description du type « flot de

Du diagramme aux équations

Rajoutons à l'exemple précédent une commande supplémentaire, en, telle que :

– si en = '0' , la machine ne quitte pas l'état 1, quand elle y arrive,
– si en = '1' , la machine fonctionne comme précédemment.

Le diagramme de transitions devient (figure V-6) :

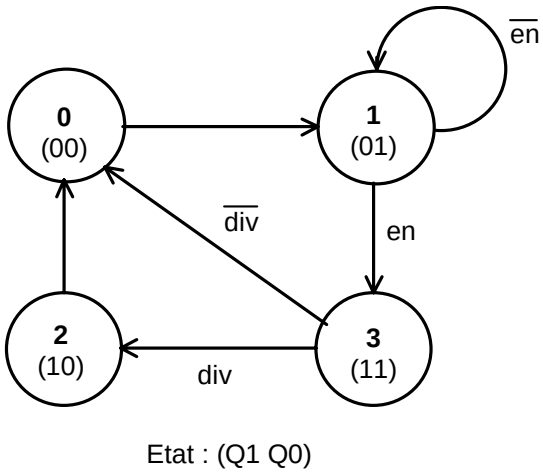


Figure V-6

Quand le nombre de bascules du registre d'état et le codage des états sont connus, le passage du diagramme de transitions aux équations de commandes des bascules est immédiat. Les détails des calculs dépendent du type de bascules utilisées pour réaliser le registre d'état.

Avec des bascules D :

Il suffit de recenser, pour chaque bascule, toutes les transitions, *et les maintiens*, qui conduisent à la mise à '1' de la bascule. Dans notre exemple il y a deux bascules :

– Q1 est à '1' dans les états 2 et 3, obtenus par les transitions $1 \rightarrow 3$ et $3 \rightarrow 2$.
– Q0 est à '1' dans les états 1 et 3, obtenus par les transitions $0 \rightarrow 1$, $1 \rightarrow 1$ (maintien) et $1 \rightarrow 3$.

---

données », avec des bascules décrites au niveau structurel, qu'à une réelle description de haut niveau, dans un langage comportemental.

D'où, en notant les états de façon symbolique par leur numéro souligné, et en simplifiant, éventuellement les expressions obtenues :

$$D1 = \underline{1} * en + \underline{3} * div = \overline{Q1} * Q0 * en + Q1 * Q0 * div$$
$$D0 = \underline{0} + \underline{1} * \overline{en} + \underline{1} * en = \underline{0} + \underline{1} = \overline{Q1}$$

Avec des bascules T :

Il suffit de recenser, pour chaque bascule, toutes les transitions qui conduisent à un changement d'état de la bascule. Dans notre exemple il y a deux bascules :
- Q1 change dans les transitions $1 \rightarrow 3$, $3 \rightarrow 0$ et $2 \rightarrow 0$.
- Q0 change dans les transitions $0 \rightarrow 1$, $3 \rightarrow 0$ et $3 \rightarrow 2$.

D'où, en notant les états de façon symbolique par leur numéro souligné et en simplifiant, éventuellement les expressions obtenues :

$$T1 = \underline{1} * en + \underline{3} * \overline{div} + \underline{2} = \overline{Q1} * Q0 * en + Q1 * \overline{div} + Q1 * \overline{Q0}$$
$$T0 = \underline{0} + \underline{3} * \overline{div} + \underline{3} * div = \underline{0} + \underline{3} = \overline{Q1 \oplus Q0}$$

La comparaison entre les deux solutions, bascules D ou bascules T, montre que dans l'exemple considéré, la première solution conduit à des équations plus simples (ce n'est pas toujours le cas). Certains circuits programmables offrent à l'utilisateur la possibilité de choisir le type de bascules, ce qui permet d'adopter la solution la plus simple[13].

Table de transitions

Pour passer d'un diagramme de transitions aux équations de commandes des bascules, le concepteur débutant peut toujours recourir à une table de vérité qui récapitule toutes les transitions possibles.

Si cette méthode est systématique, elle présente évidemment l'inconvénient d'être fort lourde. Le nombre de variables d'entrées de la table devient vite, même pour des problèmes simples, très élevé.

Le tableau qui suit correspond à la dernière version de notre diviseur par trois ou quatre. Quand, pour une transition, la valeur d'une commande est indifférente, elle apparaît par la valeur 'x'.

[13]Soyons clairs, c'est en général l'optimiseur du compilateur qui fait ce choix, mais l'utilisateur a un droit de regard, et d'action, sur ce que fait le logiciel.

Etat initial		Entrées		Etat final	
Q1	Q0	en	div	D1	D0
0	0	x	x	0	1
0	1	0	x	0	1
0	1	1	x	1	1
1	1	x	0	0	0
1	1	x	1	1	0
1	0	x	x	0	0

Table de transitions du diviseur par 3/4.

Cette table n'apporte rien de plus que le diagramme de transitions, son utilité est d'autant plus discutable que l'on effectue rarement les calculs à la main, et nous verrons qu'il est très simple de passer directement d'un diagramme de transitions au programme VHDL correspondant.

L'état futur est unique

Pour représenter, de façon non ambiguë le fonctionnement d'un système, un diagramme de transitions doit respecter certaines règles qui concernent les états, d'une part, et les transitions, d'autre part. Leur non respect constitue une erreur :

- Un état, représenté par un code unique, ne peut apparaître qu'une seule fois dans un diagramme. Cette règle, somme toute fort naturelle, n'est généralement pas source d'erreurs, ou, au pire, provoque par son non respect, des erreurs faciles à identifier et à corriger.
- La machine est forcément « quelque part ». Cela impose que la condition de maintien dans un état soit le complément logique de la réunion de toutes les conditions de sortie de l'état. Cette règle est générée automatiquement par les logiciels – seules les transitions doivent être spécifiées, les compilateurs en déduisent la condition de maintien – mais peut être une source d'erreurs dans une synthèse manuelle.
- La machine ne peut pas être à deux endroits différents à la fois. Les transitions qui partent d'un état, pour arriver à des états *différents*, doivent être assorties de *conditions mutuellement exclusives.*

Ces deux derniers points méritent des éclaircissements ; leur non respect, qui ne saute pas toujours aux yeux, est l'une des principales sources d'erreur dans les diagrammes de transitions. Précisons cela en modifiant quelque peu le diviseur étudié précédemment.

On souhaite remplacer, dans le diviseur par 3/4, les commandes en et div par deux commandes, div3 et div4, actives à '1', qui fournissent globalement les mêmes fonctionnalités, mais avec une répartition des rôles un peu différente :

- div3 commande le fonctionnement en diviseur par 3,
- div4 commande le fonctionnement en diviseur par 4.

– Si les deux commandes sont inactives, la machine s'arrête dans l'état 1.

Deux versions du diagramme de transitions de la nouvelle variante du diviseur sont représentées figure V-7.

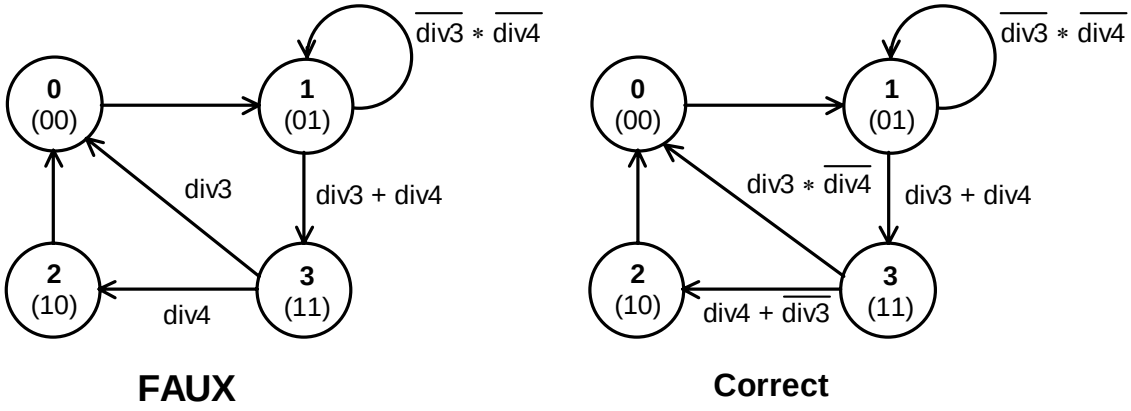


Figure V-7

Sur les deux versions on a indiqué que le maintien dans l'état 1 est bien obtenu par complémentation de la condition de cet état, ce qui est correct.

La première version, marquée comme fausse sur la figure, contient deux erreurs qui concernent l'évolution à partir de l'état 3 :

1. Une erreur de *syntaxe*, si div3 et div4 sont tous les deux actifs, le diagramme indique deux destinations différentes, ce qui est absurde.
2. Une erreur de *sens*, par rapport au cahier des charges, si div3 et div4 sont tous les deux inactifs, le diagramme indique un maintien dans l'état 3, par absence de transition.

La deuxième version présente une solution correcte au problème, l'erreur de syntaxe a disparu, et on a établi une priorité de la division par quatre. Si les deux commandes sont actives la machine prend le chemin de la division par quatre ; elle réagit de même si les deux commandes deviennent inactives alors que l'état 3 est actif, elle évolue vers l'état 1, pour s'y arrêter, en passant par les états 2 et 0.

Ce genre d'erreurs est vite arrivé. Lors de la traduction en VHDL d'un diagramme de transitions les erreurs de syntaxe disparaissent le plus souvent, grâce aux priorités qu'introduisent les algorithmes séquentiels :

– les instructions « `if ... elsif ... else ... end if` » décomposent un choix multiple en alternatives binaires, d'où les conflits de destinations ont disparu ;

– les instructions « `case ... when ... end case` » doivent obligatoirement traiter toutes les alternatives.

Mais la disparition des erreurs de syntaxe peut, malheureusement, s'accompagner d'une modification du sens qu'avait prévu un concepteur insuffisamment rigoureux.

Attention aux états pièges !

Un état piège est un état dans lequel la machine peut entrer, mais dont elle ne sort jamais, comme un piège à anguilles. Cela peut être volontaire, aboutissement d'une séquence d'initialisation qui suit la mise sous tension d'un système, par exemple ; mais c'est rare. De plus dans ce genre de situation, l'état piège est explicite, il est donc visible. Plus dangereux sont les pièges cachés, qui ne figurent pas sur le diagramme de transitions.

Prenons un exemple.

On souhaite réaliser, au moyen de deux bascules, deux signaux rigoureusement synchrones, issus de deux diviseurs de fréquence par deux couplés, tels que les sorties des bascules soient toujours complémentaires.

La première version du diagramme de transitions de la figure V-8 semble convenir, a tort.

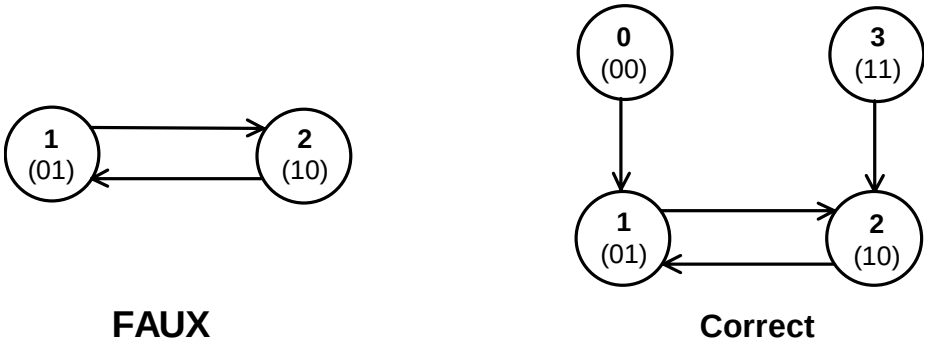


Figure V-8

Placé dans un circuit programmable de type 16V8, la machine d'états ainsi créée ne fonctionne pas du tout :

– Synthétisé à la main, à partir du diagramme, avec des bascules D, elle reste obstinément arrêtée dans l'état 0, après être passée par l'état 3 lors de la mise sous tension, par les vertus de l'initialisation automatique dont dispose le circuit. Par omission des termes correspondants dans les équations de commande, tous les états oubliés, dans une synthèse qui emploie des

bascules D, sont raccordés à l'état 0. Or celui-ci, toujours par omission, est un piège dans notre exemple.
– Synthétisé par un compilateur, qui génère par défaut les équations de maintien, le système reste obstinément figé dans l'état 3, autre piège.

La deuxième version fonctionne correctement, et a, de plus, la vertu d'être décrite par des équations plus simples, que l'on aurait d'ailleurs pu trouver directement par un simple raisonnement qualitatif[14].

Le problème des pièges cachés a conduit à l'introduction, dans les langages de description, de sortes de « méta états », qui regroupent tous les non-dits, pour pouvoir préciser ce qui doit leur arriver (`else` d'un `if`, `others` d'un `case`, en VHDL). Mais, comme l'exemple précédent le montre, le raccordement de tous les états inutilisés dans un même état du diagramme, ne conduit pas toujours à la solution la plus simple.

Une approche algorithmique : VHDL

VHDL offre de multiples possibilités pour traduire le fonctionnement d'une machine d'états. Seules nous intéressent ici les descriptions comportementales, dans lesquelles le coeur d'une machine d'états est associé à un processus.

Même avec cette restriction, qui exclut les représentations structurelles, toujours possibles, le langage offre des styles de programmation variés, qui permettent de traduire simplement les situations les plus diverses.

Nous tenterons, ci-dessous, de donner certaines indications générales, qui peuvent servir de guide pour les cas les plus courants. En conformité avec ce qui a été dit au début de ce chapitre, nous ne nous occuperons que de fonctions synchrones, dont le synoptique général correspond à celui de la figure V-1.

Le registre d'état

A tout seigneur tout honneur, nous commencerons par le registre d'état.

Il est matérialisé, dans un programme source en VHDL, par deux éléments indissociables :

1. un signal interne, de type `bit_vector`, énuméré ou `integer`, déclaré de façon à être codé sur n chiffres binaires,
2. un processus, activé par le seul signal d'horloge, qui est l'*unique* endroit où le signal d'état subit une affectation.

Le choix du type employé pour le signal d'état dépend de la nature des opérations les plus fréquemment rencontrées dans le diagramme de transitions, du lien entre le registre d'état et les sorties, nous reviendrons sur ce point important, et ... du goût du concepteur. Même s'il semble plus naturel d'adopter, par exemple, un type entier pour une machine dont le fonctionnement se modélise bien par des

[14]Nous ne pouvons que conseiller au lecteur de faire, à titre d'exercice, la synthèse des exemples dont nous ne donnons pas les équations.

opérations arithmétiques, il est bon de se souvenir que les opérateurs peuvent être surchargés, pour agir sur des vecteurs de bits. Les paquetages fournis avec un compilateur contiennent déjà la plupart de ces surcharges utiles.

Faut il créer un processus à part pour la fonction combinatoire f( ), qui calcule, dans le synoptique de la figure V-1, l'état futur ? Rien n'est moins sûr.

La séparation du registre d'état et de sa commande conduit à un premier processus, qui est trivial, pour le registre d'état, et à un second processus, qui l'est beaucoup moins, pour la commande. Notons, en particulier, que des combinaisons des entrées dont on ne précise pas l'effet sur la machine génèrent,par défaut, des maintiens[15] dans la version mono processus, et des mémorisations asynchrones des commandes dans la version à deux processus séparés !

Un exemple de prototype de machine d'états qui corresponde au synoptique de la figure V-1 peut être[16] :

```
entity proto_machine is
   generic (n , p , q : integer := 2 ) ;
   port(hor : in bit ;
        entrees : in bit_vector(0 to p - 1);
        sorties : out bit_vector(0 to q - 1) ) ;
end proto_machine ;

-- suivant le compilateur utilise :
use work.paquetage_arithmetique.all ;

architecture comporte of proto_machine is
signal etat : bit_vector(n - 1 downto 0) ;
begin

machine : process
   begin
      wait until hor = '1' ;
-- ci-dessous code du diagramme de transitions.

end process machine ;

actions : process
   begin
-- ci-dessous code du calcul des sorties.

end process actions ;

end comporte ;
```

[15]Dans le diagramme de transition. Ces maintiens peuvent être voulus, auquel cas tout va bien, ou involontaires, auquel cas le résultat est faux, mais pas scandaleux. Des oublis dans la description d'un processus combinatoire conduisent à des maintiens asynchrones, ce qui est scandaleux.

[16]La clause `generic`, présentée chapitre VI, permet de rendre modifiables certains paramètres.

L'activation d'une transition, dans un diagramme d'états, dépend de l'état initial et des entrées extérieures. On peut, quitte à caricaturer un peu une réalité toujours plus nuancée, situer une machine d'états quelque part entre deux extrêmes :

- Certains automates traitent beaucoup de variables d'entrées, une ou peu de fois chacune. L'état de la machine sert essentiellement à tester dans un ordre cohérent, ces différentes entrées, à attendre, à chaque étape, une condition sur l'une ou l'autre d'entre elles et à déclencher une action, avant de passer à la suite du programme. La valeur particulière de l'état de la machine, à chaque étape, est essentielle pour déterminer la grandeur testée et le trajet suivant. Un exemple typique de fonctionnement de ce genre est un programmateur de lave linge.
- D'autres machines répondent à des commandes globales, qui provoquent des parcours, dans l'espace des états accessibles, qui peuvent être décrits indépendamment des valeurs, à chaque instant, des états. L'exemple typique d'une telle machine est un compteur. Les commandes de comptage, de chargement parallèle, de remise à zéro, entraînent une évolution qui obéit à un algorithme général, dans lequel la valeur particulière de l'état actuel n'intervient pas pour prévoir celle de l'état futur : soit que l'état futur ne dépende pas de l'état actuel, soit que la valeur de l'état futur puisse être calculée à partir de celle de l'état actuel, de façon systématique, par exemple par une opération mathématique.

Les deux discussions qui suivent correspondent à ces deux situations.

Primauté à l'état de départ

Pour décrire un diagramme de transitions en VHDL, une méthode simple consiste à traiter toutes les valeurs possibles de l'état de la machine, et pour chaque cas, analyser les entrées pour en déduire l'état suivant. L'exemple ci-dessous est la transcription, avec cette démarche, du diviseur par trois ou quatre étudié précédemment (diagramme de la figure V-6).

```
entity div3_4 is
    port ( hor , div , en : in bit ;
           Q1 , Q0 : out bit ) ;
end div3_4 ;

architecture comporte of div3_4 is
signal etat : bit_vector(1 downto 0) ;
begin
Q1 <= etat(1) ;
Q0 <= etat(0) ;
process
begin
    wait until hor = '1' ;
```

```
    case etat is -- primauté a l'etat.
        when "00" => etat <= "01" ;
        when "01" => if en = '1' then
                       etat <= "11" ;
                   end if ;
        when "10" => etat <= "00" ;
        when "11" => if div = '1' then
                       etat <= "10" ;
                     else
                       etat <= "00" ;
                     end if ;
    end case ;
end process ;
end comporte ;
```

La même fonction de principe, mais avec un nombre plus important d'états possibles, conduirait vite à une énumération d'une lourdeur prohibitive. La sélection `others` de l'instruction `when` permet, quand un traitement collectif de certains états est possible, de résoudre le problème.

Le programme qui suit correspond à un diviseur par 255/256, pour lequel on a abandonné la contrainte d'obtenir la même fréquence pour toutes les sorties, contrainte irréalisable avec un registre d'état de largeur 8 bits :

```
entity dual_modulus is
   generic (n  : integer := 8 ) ;
      -- n est la taille du registre d'etat.
   port(hor : in bit ;
        en, div : in bit ;
        sortie : out integer range 0 to 1 ) ;
end dual_modulus ;

architecture comporte of dual_modulus is
signal etat : integer range 0 to 2**n - 1 ;
begin

machine : process
begin
   wait until hor = '1' ;
   case etat is
      when 1  => -- cas particulier.
    if en = '1' then
       etat <= etat + 1;
    end if ;
      when 2**n - 2 =>
    if div = '0' then
       etat <= 0 ;
         else
       etat <= etat + 1 ;
```

```
            end if ;
         when others => -- cas general.
           etat <= etat + 1 ;
      end case ;
   end process machine ;

   actions : process
   begin
      sortie <= etat / 2**(n-1);-- bit de poids fort.
   end process actions ;

   end comporte ;
```

L'exemple précédent illustre la limitation du dessin explicite d'un diagramme de transitions, dans des cas un peu complexes. Certains outils de CAO fournissent à l'utilisateur la possibilité de créer des « macro états », utiles quand une partie du diagramme peut être décrite par une formule. Donnons un exemple (figure V-9) qui correspond au diviseur par 255/256 précédent[17] :

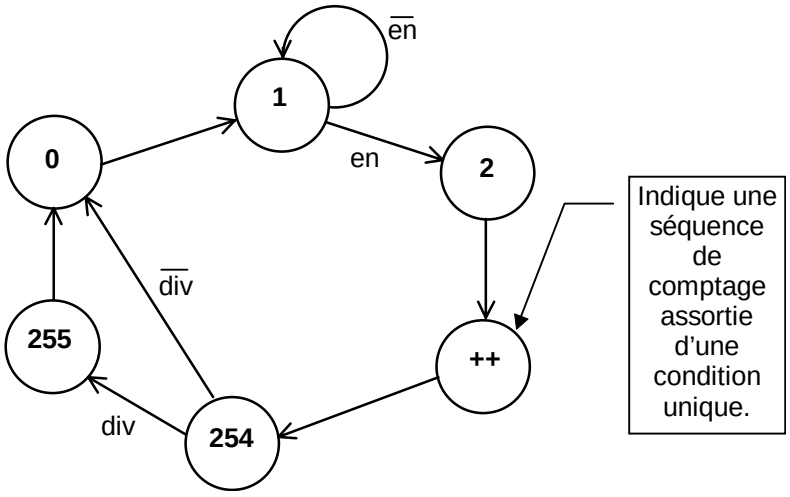


Figure V-9

Ces extensions, qui ne sont absolument pas standardisées, à la représentation traditionnelle des diagrammes de transitions permettent de représenter de façon visuelle des fonctionnement complexes, ce n'est pas à négliger.

Primauté à la commande

Les machines d'états qui disposent de commandes globales, dont les actions peuvent être décrites indépendamment de la valeur explicite de l'état, se prêtent

[17]Représentation inspirée du logiciel PLDDS, de la société Hewlett Packard.

fort mal à une description aussi détaillée que celle fournie par un diagramme de transitions. Leur description purement algorithmique peut, pourtant, être fort simple.

L'exemple ci-dessous illustre ce fait au moyen d'un compteur modulo dix, inspiré du circuit 74162, pourvu de trois commandes `clear`, `load` et `en`, dans l'ordre de priorités décroissantes :

- `clear = '0'` provoque la mise à zéro du compteur, quel que soit son état initial ;
- `load = '0'` provoque le chargement parallèle du compteur, avec des données extérieures, quel que soit son état initial ;
- `en = '1'` autorise le comptage ;
- quand toutes les commandes sont inactives, le compteur ne change pas d'état.

```
entity decade is
   port ( hor , clear, load, en : in bit ;
          donnee : in integer range 0 to 9 ;
          sortie : out integer range 0 to 9 ) ;
end decade ;

architecture comporte of decade is
signal etat : integer range 0 to 9 ;
begin
machine : process
begin
   wait until hor = '1' ;
   if clear = '0' then -- primauté aux commandes.
      etat <= 0 ;
   elsif load = '0' then
      etat <= donnee ;
   elsif en = '1' then
      case etat is -- un cas particulier.
         when 9 => etat <= 0 ;
         when others => etat <= etat + 1 ;
      end case ;
   end if ;
end process machine ;

sortie <= etat ;
end comporte ;
```

Il est clair qu'un diagramme de transitions complet d'un tel objet est pratiquement impossible à écrire : le chargement parallèle autorise des transitions entre toutes les paires d'états. L'approche algorithmique, par contre, ne pose aucune difficulté.

On notera également que la structure `if...elsif...else...end if` permet de traduire, de façon très lisible, la priorité qui existe entre les différentes commandes.

Résumons nous :

- Le processus qui décrit le fonctionnement d'une machine d'états comporte deux structures imbriquées : le traitement des commandes et le traitement de l'état de départ de chaque transition.
- Les commandes, compte tenu de leurs hiérarchies, se prêtent bien à une modélisation par des structures `if...elsif...else...end if`.
- Les états se prêtent bien à une modélisation en terme d'aiguillage, soit les structures `case...when...when others...end case`.
- Suivant le type de fonctionnement, primauté à l'état de départ ou primauté à la commande, on choisira l'ordre d'imbrication des deux structures correspondantes.

V.2.2 Des choix d'architecture décisifs

Les logiciels de synthèse libèrent le concepteur d'avoir à se préoccuper des détails des calculs qui conduisent, face à un problème posé, d'une idée de solution aux équations de commandes des circuits, déduites d'un diagramme de transitions ou d'un algorithme. Le travail de conception qui reste à sa charge réside principalement dans les choix généraux d'architectures : découpage du système en sous ensembles de taille humaine, choix de structures et de codages pour chaque sous ensemble. Ces derniers comprennent principalement le traitement des entrées – sorties et, en liaison avec les sorties, le type de codage des états.

Calculs des sorties : machines de Mealy et de Moore

Suivant la façon dont les sorties dépendent des états et des commandes, on distingue deux types de machines d'états : les machines de Moore et les machines de Mealy. Dans les premières les sorties ne dépendent que de l'état actuel de la machine, dans les secondes les sorties dépendent de l'état de la machine et des entrées.

Dans beaucoup de cas réels la séparation n'est pas aussi tranchée : certaines sorties sont traitées comme des sorties d'une machine de Moore, d'autres comme des sorties d'une machine de Mealy.

Machines de Moore

A l'image de M. Jourdain, nous avons, en réalité, fait des machines de Moore sans le savoir. Le synoptique général de la figure V-1, dans lequel les sorties sont fonctions uniquement de la valeur du registre d'état, est la définition même d'une telle machine.

Dans ce type d'architecture, le calcul des sorties et le codage des états sont évidemment intimement liés. Nous aurons l'occasion de revenir sur ce point ultéreurement.

Machines de Mealy

Dans une machine de Mealy les entrées du système ont une action directe sur les sorties, nous admettrons, dans un premier temps, que les sorties sont des fonctions purement combinatoires, symbolisées par une fonction g( ), des entrées et de l'état de la machine.

La structure générale d'une machine de Mealy est la suivante (figure V-10) :

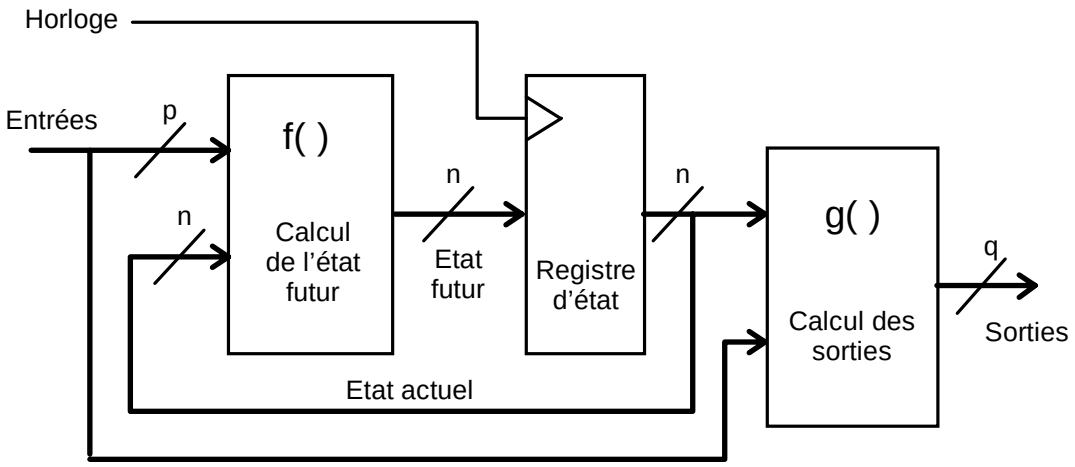


Figure V-10

Une première différence apparaît alors immédiatement entre les comportements de sorties de Moore et de Mealy : les premières évoluent suite à un changement d'état, donc à la période d'horloge qui *suit* celle où a varié l'entrée responsable de l'évolution, les secondes réagissent *immédiatement* à une variation d'une entrée, précédant en cela l'évolution du registre d'état.

Le chronogramme de la figure V-11 illustre ce point ; on y représente, en supposant le fonctionnement idéal, c'est à dire sans faire apparaître les temps de propagations :

- le changement d'une entrée com,
- un changement d'état qui en résulte, etat_i passe à 0,
- le changement associé d'une sortie de Moore, moore_i, qui passe à 1,
- le changement d'une sortie de Mealy, mealy_ou_i, calculée par

  mealy_ou_i = com + etat_arrivée,

  où etat_arrivée est l'état qui *suit* etat_i,
- le changement d'une autre sortie de Mealy, mealy_et_i, calculée par

mealy_et_i = com ∗ etat_i.

Un point intéressant, que souligne ce chronogramme, est la possibilité de générer, par une sortie de Mealy, une impulsion qui dure une période d'horloge, indiquant qu'un changement d'état *va* se produire au front d'horloge suivant (sortie mealy_et_i)[18].

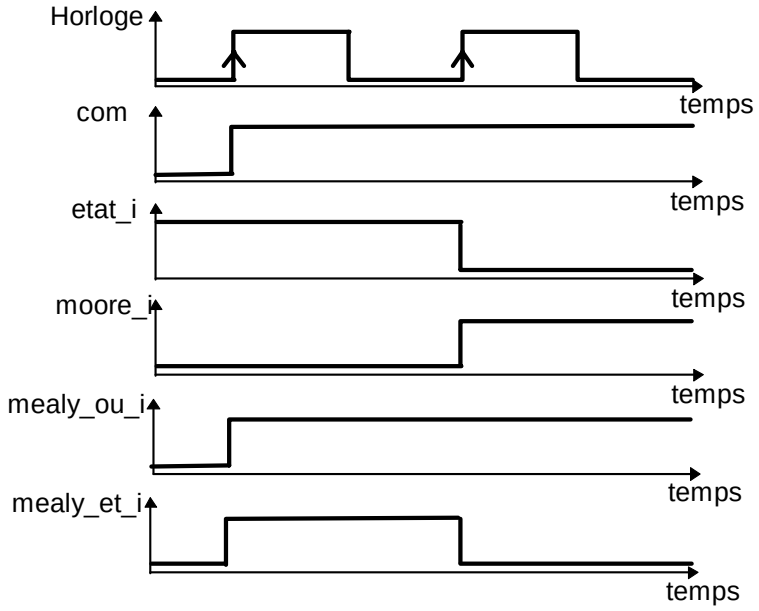


Figure V-11

Une application classique des machines de Mealy est la création d'opérateurs pourvus de sorties d'extension. Reprenons, à titre d'exemple, le compteur modulo 10 de l'exemple VHDL précédent. Il serait souhaitable de pouvoir associer simplement plusieurs de ces compteurs en cascade, de façon à réaliser un compteur sur plusieurs chiffres décimaux, un compteur kilométrique de voiture, par exemple, sans avoir à rajouter de circuiterie supplémentaire.

Pour cela il faut disposer d'une sortie, `rco` (pour *ripple carry out*), qui nous indique que la décade *va* passer à zéro, c'est à dire qu'elle est dans l'état 9 *et* qu'elle est autorisée à compter, car son entrée d'autorisation, `en`, est à un.

Mécanisme d'anticipation et influence directe d'une entrée, la sortie `rco` d'un compteur est bien une sortie de Mealy. Le programme ci-dessous contient la modification souhaitée :

[18]Il est clair que nous supposons ici que les commandes sont synchrones de la même horloge que la machine étudiée.

```
entity decade_rco is
   port ( hor , clear, load, en : in bit ;
          donnee : in integer range 0 to 9 ;
          sortie : out integer range 0 to 9 ;
     rco : out bit ) ;
end decade_rco ;
architecture comporte of decade_rco is
signal etat : integer range 0 to 9 ;
begin
machine : process
begin
   wait until hor = '1' ;
-- même code que précédemment.
end process machine ;
sortie <= etat ;
rco <= '1' when etat = 9 and en = '1' else '0';
end comporte ;
```

Pour créer un compteur à plusieurs chiffres décimaux, il suffit alors de connecter la sortie `rco` de chaque décade sur l'entrée `en` de la décade de poids *supérieur* ; il va sans dire que toutes les décades doivent être pilotées par la même horloge[19] !

Diagramme de transitions d'une machine de Mealy

Pour tenir compte de l'action immédiate des entrées sur les sorties, dans une machine de Mealy, on complète parfois le diagramme de transitions de la machine en faisant figurer, en plus de la condition de transition, la valeur associée des sorties du type Mealy.

Par exemple, pour une simple bascule R-S synchrone, mais qui « réagit » instantannément, nous obtenons (figure V-12) :

[19]Notons, au passage, un piège des sorties de Mealy : il est interdit de rajouter un rétrocouplage de la sortie `rco` sur l'entrée `en` de la même décade. Ce rétrocouplage créerait une réaction asynchrone, qui peut, par exemple, conduire à des oscillations du circuit. Dans les compteurs TTL de la famille 160, les constructeurs ont prévu deux entrées, `ent` et `enp`, d'autorisation de comptage, dont l'une, `enp`, n'a aucune action sur la sortie de mise en cascade. S'il est nécessaire, par exemple, d'inhiber le comptage en fin de cycle, c'est cette deuxième entrée de validation qui doit être employée.

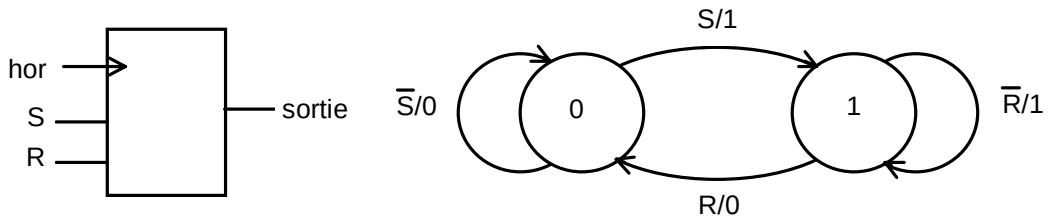


Figure V-12

Un tel diagramme se lit de la façon suivante :

Quand la bascule est à 0, la sortie est à 0 tant que l'entrée S est à 0, quand S passe à 1 la sortie passe à 1 et la bascule effectue la transition 0→1 au front d'horloge suivant ;

quand la bascule est à 1, la sortie est à 1 tant que l'entrée R est à 0, quand R passe à 1, la sortie passe à 0 et la bascule effectue la transition 1→0 au front d'horloge suivant.

De ce diagramme nous pouvons déduire l'équation de la commande, D, de la bascule, et l'équation de la sortie[20] :

$$D = \overline{Q} * S + \overline{R} * Q$$
$$\text{sortie} = \overline{Q} * S + \overline{R} * Q$$

Il se trouve que, dans cet exemple, l'équation de la sortie est identique, dans la forme mais pas dans le résultat), à celle de la commande de la bascule ; ce n'est évidemment pas toujours le cas.

Comparaison des machines de Moore et Mealy : un exemple

Afin d'illustrer les différences entre les deux types de machines d'états, nous allons donner un exemple d'application, traité par les deux méthodes.

Un décodeur Manchester différentiel.

Dans les communications séries entre ordinateurs on utilise généralement des techniques particulières de codage pour les signaux qui circulent sur le câble, par exemple, le codage *Manchester différentiel*. En codage Manchester différentiel, chaque intervalle de temps élémentaire, pendant lequel un signal binaire est placé sur le câble, nommé le plus souvent « temps bit », est divisé en deux parties de durées égales avec les conventions suivantes :

[20]On comparera utilement le fonctionnement de cette bascule R S synchrone avec celui d'une bascule J K.

– Un signal binaire 1 est représenté par une absence de transition au début du temps bit correspondant.
– Un signal binaire 0 est représenté par la présence d'une transition au début du temps bit considéré.
– Au milieu du temps bit il y a *toujours* une transition.

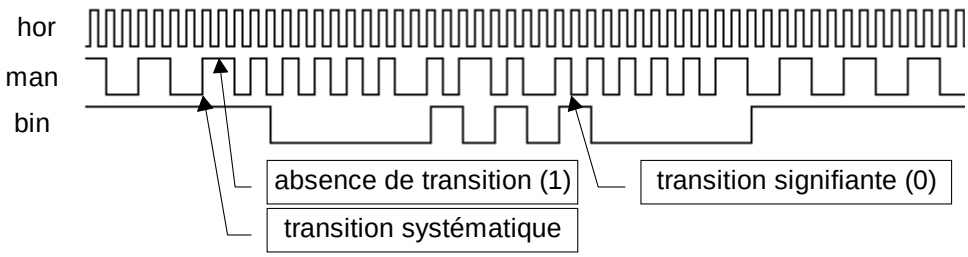


Figure V-13

Le chronogramme de la figure V-13 représente un exemple d'allure des signaux. Les données à décoder, le signal `man`, sont synchrones d'une horloge `hor`[21] ; on souhaite réaliser un décodeur qui fournit en sortie le code binaire correspondant, `bin`. La sortie du décodeur est retardée d'une période d'horloge par rapport à l'information d'entrée, pour une raison qui s'expliquera par la suite.

Analyse du problème :

L'idée est assez simple, nous allons construire une machine d'états qui, parcourt un premier cycle quand le signal d'entrée change à chaque période d'horloge, ce qui correspond à un '0' transmis, et change de cycle quand elle détecte une absence de changement du signal d'entrée, qui correspond à un '1' transmis.

Machine de Mealy :

L'absence de changement peut se produire tant pour un niveau haut que pour un niveau bas du signal d'entrée, d'où l'ébauche de diagramme de transitions de la figure V-14 (page suivante).

[21]La reconstruction par un récepteur du signal d'horloge, `hor`, n'est pas abordé ici. Les techniques employées relèvent généralement de l'analogique (boucle à verrouillage de phase).

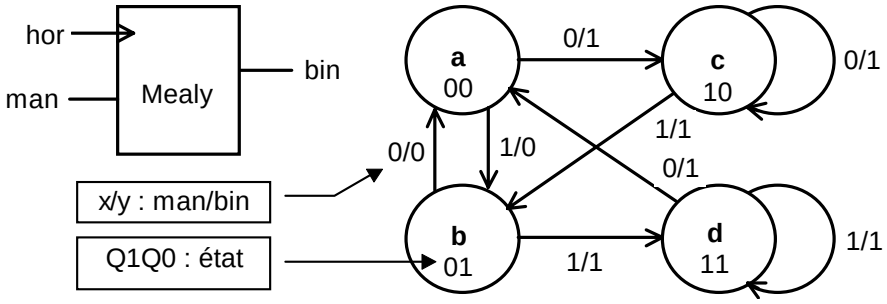


Figure V-14

On se convaincra facilement que les cycles parcourus sont :

- a→b→a→b→a→b→a... ou b→a→b→a→b→a→b... pour trois '0' consécutifs transmis,
- a→c→b→d→a→c→b... ou b→d→a→c→b→d→a... pour trois '1' consécutifs transmis.

En fonctionnement permanent, une fois le système synchronisé et sauf erreur dans le code d'entrée, les conditions de maintien dans les états c et d sont toujours fausses, elles servent à la synchronisation en début de réception.

Du diagramme précédent on déduit les équations de commandes des bascules, D1 et D0, et celle de la sortie ; après quelques simplifications on obtient :

$$D0 = \overline{man}$$
$$D1 = \overline{Q0 \oplus man}$$
$$bin = Q1 + \overline{Q0 \oplus man}$$

Machine de Moore (figure V-15) :

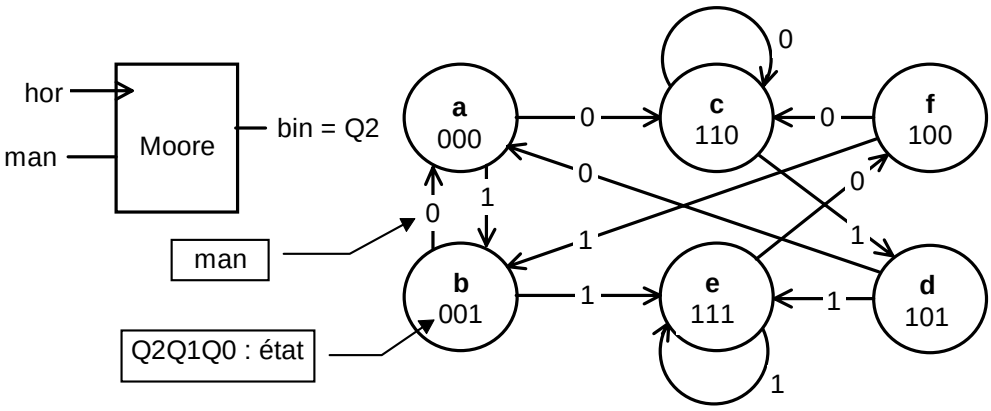


Figure V-15

Dans une machine de Moore, différentes valeurs des sorties correspondent à des états différents, le diagramme de transitions doit donc contenir plus d'états.

Le diagramme ne représente pas deux états inutilisés, 2 et 3, en décimal. Leur affectation se fait lors du calcul des équations de commandes, de façon à les simplifier au maximum (si man = '1', 3→7 et 2→5 ; si man = '0', 3→4 et 2→6). On obtient, après quelques manipulations :

$$D0 = man$$
$$D1 = \overline{Q0 \oplus man}$$
$$D2 = Q1 + \overline{Q0 \oplus man}$$

qui sont exactement les mêmes équations que celles obtenues dans le cas de la machine de Mealy[22], malgré l'apparente complexité du diagramme de transitions.

Comparaison :

Sur l'exemple que nous venons de traiter, il apparaît comme seule différence une bascule supplémentaire en sortie, pour générer le signal bin, dans le cas de la machine de Moore.

En regardant plus attentivement l'architecture des deux systèmes, on constate que la sortie combinatoire de la machine de Mealy risque de nous réserver quelques surprises : son équation fait intervenir des signaux logiques qui changent d'état simultanément, d'où des risques de création d'impulsions parasites étroites au moment des commutations. Une simulation confirme ce risque[23](figure V-16) :

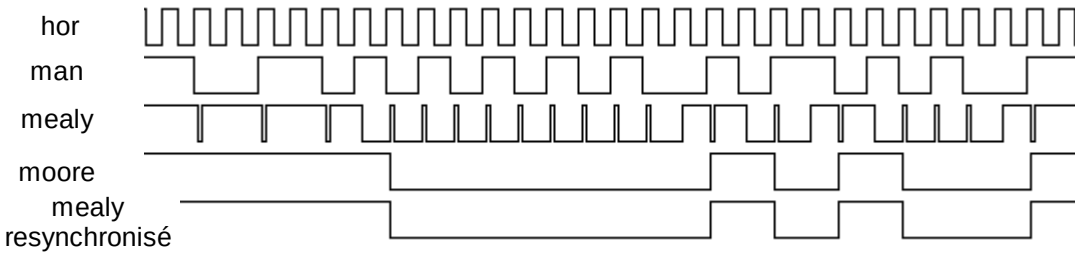


Figure V-16

[22]Soyons honnêtes, le codage des états a été choisi de façon à ce que les choses « tombent bien » ; mais, quel que soit le codage, les complexités des équations sont du même ordre de grandeur.

[23]Le simulateur utilisé est purement fonctionnel, mais causal. Chaque couche logique rajoute un temps de propagation virtuel égal à une unité (« tic » de simulation). L'allure des signaux n'a donc qu'une vertu qualitative, pour ce qui concerne les limites d'un fonctionnement.

Quand on compare les sorties des deux machines, elles diffèrent d'une période d'horloge, ce qui est normal, mais la sortie de la machine de Moore est exempte de tout parasite, contrairement à celle de la machine de Mealy, ce qui est un avantage non négligeable.

L'élimination des parasites en sortie a une solution simple : il suffit de resynchroniser la sortie incriminée, c'est ce que nous avons fait pour obtenir la dernière trace du chronogramme précédent, dans ce cas, les deux approches conduisent à des résultats strictement identiques !

Autocritique.

Pour familiariser le lecteur aux raisonnements sur les diagrammes de transitions, avec un exemple pas tout à fait trivial, nous n'avons pas respecté la règle d'or du concepteur : diviser pour régner.

En séparant le problème en deux :

1. Détection des absences de transition ;
2. génération du code binaire ;

l'élaboration des diagrammes de transitions des deux machines d'état devient un exercice extrêmement simple.

Le programme VHDL de l'étude précédente :

On trouvera ci-dessous le programme VHDL qui contient, sous forme de deux processus, les deux solutions présentées pour le décodeur Manchester différentiel.

La lecture de ce programme doit se faire en observant parallèlement les deux diagrammes d'états.

```
entity mandec is
   port ( hor, man : in bit ;
          mealy , mealysync : out bit ;
          moore : out bit );
end mandec ;

architecture comporte of mandec is
signal moore_state : bit_vector(2 downto 0);
signal mealy_state : bit_vector(1 downto 0);

begin
mealy <= mealy_state(1) or
           not(mealy_state(0) xor man) ;
moore <= moore_state(2) ;

mealy_mach : process
begin
   wait until hor = '1' ;
   mealysync <=  mealy_state(1) or
            not(mealy_state(0) xor man) ;
```

```
   case mealy_state is
when "00" => if man = '0' then
                mealy_state <= "10" ;
             else
                mealy_state <= "01" ;
             end if ;
when "01" => if man = '0' then
                mealy_state <= "00" ;
             else
                mealy_state <= "11" ;
             end if ;
when "10" => if man = '1' then
                mealy_state <= "01" ;
             end if ;
when "11" => if man = '0' then
                mealy_state <= "00" ;
             end if ;
   end case ;
end process mealy_mach ;

moore_mach : process
begin
   wait until hor = '1' ;
   case moore_state is
when O"0" => if man = '0'  -- etats en octal
              then
                moore_state <= O"6" ;
             else
                moore_state <= O"1" ;
             end if ;
when O"1" => if man = '0' then
                moore_state <= O"0" ;
             else
                moore_state <= O"7" ;
             end if ;
when O"6" => if man = '1' then
                moore_state <= O"5" ;
             end if ;
when O"7" => if man = '0' then
                moore_state <= O"4" ;
             end if ;
when O"4" => if man = '0' then
                moore_state <= O"6" ;
             else
                moore_state <= O"1" ;
             end if ;
when O"5" => if man = '0' then
                moore_state <= O"0" ;
             else
```

```
                moore_state <= O"7" ;
            end if ;
when O"2" => if man = '0' -- etat inutiles
             then
                moore_state <= O"6" ;
            else
                moore_state <= O"5" ;
            end if ;
when O"3" => if man = '0' then -- bis
                moore_state <= O"4" ;
            else
                moore_state <= O"7" ;
            end if ;
   end case ;
end process moore_mach ;
end comporte ;
```

Codage des états

Quand on utilise des circuits standard, des compteurs programmables, par exemple, pour réaliser une machine séquentielle, le codage des états du diagramme de transitions est, de fait, imposé par le circuit cible. Il en va tout autrement quand la dite machine doit être implantée dans un circuit programmable ou un ASIC. Libéré des contraintes liées à une quelconque fonction prédéfinie, le concepteur peut, à loisir, adapter le codage des états à l'application qu'il est en train de réaliser.

Le choix d'un code est particulièrement important quand on s'oriente vers la réalisation d'une machine de Moore. L'exemple du décodeur Manchester nous a appris que l'un des avantages de cette architecture réside dans la possibilité de générer les sorties directement à partir du registre d'état, donc dénuées de tout parasite lié à leur calcul. Mais, comme nous le verrons dans deux exemples, l'identification des sorties du système à celles des bascules du registre d'état ne suffit généralement pas pour définir le codage des états.

Ce choix du codage mérite une grande attention, il conditionne grandement la complexité de la réalisation, sa bonne adaptation au problème posé ; un choix judicieux conduira à un résultat simple et facilement testable, alors qu'aucun logiciel d'optimisation ne compensera des erreurs de décision à ce niveau.

Le nombre d'états nécessaires et le type de code adopté fixent, en premier lieu, la taille du registre d'état. Schématiquement, si n est la taille, en nombre de bits, du registre d'état, et N_e le nombre d'états nécessaires, ces deux nombres (entiers !) doivent vérifier la double inégalité :

$$n \leq N_e \leq 2^n$$

Si l'inégalité de gauche n'est pas vérifiée, certaines bascules sont probablement inutiles ; quand cette inégalité se transforme en égalité, on utilise un code très « dilué », une bascule par état, qui présente l'avantage de la lisibilité, mais le danger de générer en grand nombre des états accessibles inutilisés (rappelons ici qu'il y a toujours 2^n états accessibles).

Si l'inégalité de droite n'est pas vérifiée, la tentative est sans espoir ; si elle se transforme en égalité, on utilise un encodage « fort », auquel il faudra très probablement adjoindre des fonctions combinatoires de calcul des sorties ; on ne réalise pas que des compteurs binaires ou des codeurs de position absolue (code de Gray).

Les situations intermédiaires correspondent en général à des codes adaptés aux sorties.

Encodage « fort » ou code « dilué » ? En caricaturant un peu, on peut dire que les tenants de la première solution préfèrent les fonctions combinatoires, et que les seconds sont des adeptes des bascules. Il n'est pas évident, à priori, de prévoir la complexité des équations engendrées par tel ou tel code. On gagne souvent à suivre le fonctionnement « naturel » de la machine[24], et, surtout, on gagne à se souvenir que les ordinateurs, et leurs compilateurs, ne sont pas posés sur un bureau à titre de décoration ; ils permettent de voir très vite quelle est la complexité sous jacente d'un choix, sans pour celà tomber dans le BAO[25].

Codes adaptés aux sorties

L'idée qui vient naturellement à l'esprit est de choisir le codage en fonction des sorties à générer. C'est souvent la méthode la plus souple, celle qui conduit aux équations les plus faciles à interpréter, et pas forcément plus compliquées que celles que l'on obtiendrait avec d'autres codes.

Une commande de feux tricolores.

Pour satisfaire à une tradition bien établie, nous prendrons comme premier exemple une commande de feux de circulation routière.

Un passage pour piétons traverse une avenue ; il est protégé par un feu tricolore qui fonctionne à la demande des piétons : En l'absence de toute demande, les feux sont à l'orange clignotant (un nombre Tor de secondes allumés, Tor secondes éteints). Quand un piéton souhaite traverser l'avenue, il est invité à appuyer sur un bouton, ce qui provoque le déclenchement d'une séquence (vue des voitures) :

- orange fixe pendant $2*$Tor secondes,
- rouge pendant Tr secondes,
- vert pendant Tv secondes, pour laisser passer le flot de voitures pendant un minimum de temps,

[24]Mais qu'est-ce que ce fonctionnement naturel ? Sa recherche est, sans doute, l'une des parties les plus intéressantes, et donc souvent difficile, du travail.
[25]Bricolage Assisté par Ordinateur.

– retour à la situation par défaut.

Profitons de cet exemple pour subdiviser la solution du problème en sous ensembles. Trois blocs fonctionnels peuvent être identifiés :

1. La commande des feux proprement dite, les sorties de trois bascules du registre d'état commandent directement l'allumage, ou l'extinction, des lampes rouge, verte et orange.
2. Une temporisation qui, suite à une commande d'initialisation, fournit les trois durées Tor, Tr et Tv.
3. Une mémorisation de l'appel des piétons, qui évite de se poser des questions concernant la durée pendant laquelle le demandeur appuie sur le bouton ; une simple pression suffit, l'appel est alors enregistré, quel que soit l'état d'avancement de la séquence de gestion des feux.

Outre les commandes des feux proprement dites, le bloc principal fournit un signal d'initialisation (cpt) à la temporisation, qui doit durer une période d'horloge[26], et un signal d'annulation (raz) de la requête, mémorisée, d'un piéton.

Les signaux d'entrée de ce bloc sont la requête (piet) et les trois indications de durée Tor Tr et Tv ; nous supposerons que ces dernières passent à '1', pendant une période d'horloge, quand les durées correspondantes se sont écoulées.

D'où le synoptique de la figure V-17 :

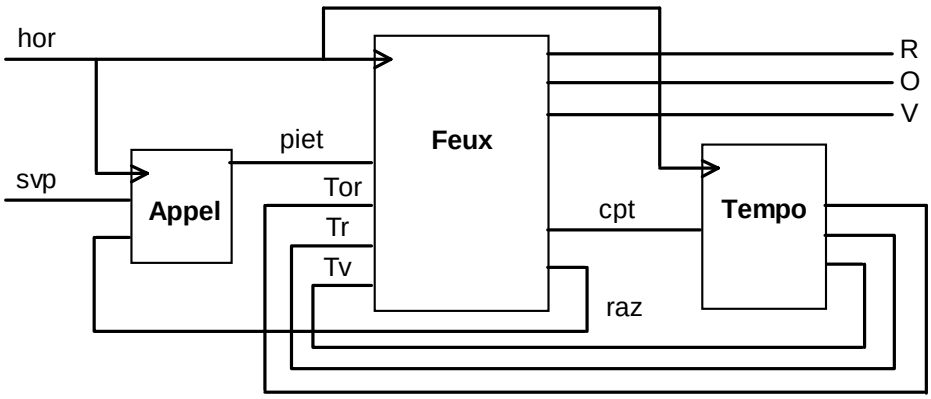


Figure V-17

[26]Nous sommes en train de définir trois processus qui se commandent et/ou s'attendent mutuellement. **Le** danger de ce type d'architecture, très fréquente, est de générer des interblocages : un processus en initialise un second et attend une réponse de ce dernier. Si le demandeur oublie de relâcher la commande d'initialisation, le système est bloqué. Ce type de situation porte, en informatique, le doux nom d'étreinte fatale (*deadly embrace*). La solution adoptée ici est d'envoyer des signaux fugaces (mais synchrones !), ce qui oblige le demandeur à attendre la réponse dans un état différent de celui où il a passé la commande d'initialisation.

Nous nous contenterons d'étudier, ici, le bloc principal, feux, laissant la synthèse des deux autres blocs à titre d'exercice.

Première ébauche :

Le fonctionnement général peut être celui illustré par la figure V-18 :

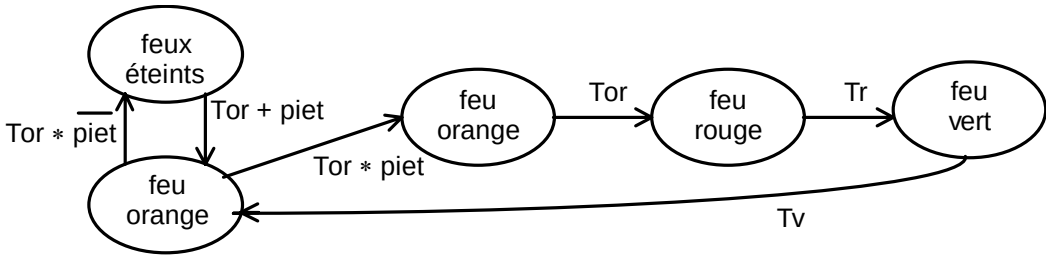


Figure V-18

Précisions :

A partir de l'ébauche précédente, il nous reste à préciser le mode de calcul des signaux gérés par le processus feux, et à en déduire le codage des états. Le signal cpt se prête bien à une réalisation sous forme de sortie de Mealy, les signaux de commande des feux à une réalisation sous forme de sorties de Moore. Les deux états où le feu orange est allumé doivent être distingués, une bascule supplémentaire, qui n'est attachée à aucune sortie, doit être rajoutée à cette fin. La sortie raz peut être identique à la sortie qui correspond au feu rouge ; il n'est pas utile de mémoriser une demande de piéton quand les voitures sont arrêtées au feu rouge. D'où une version plus élaborée du diagramme de transitions (figure V-19) :

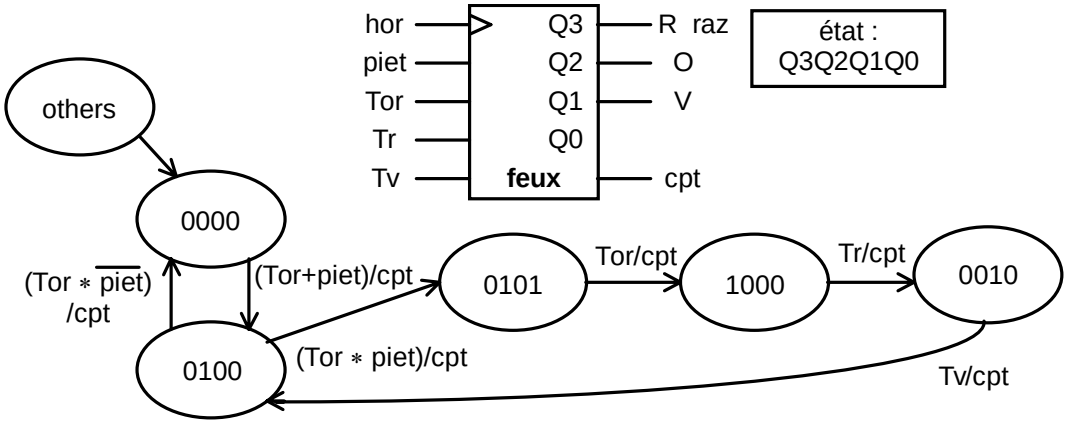


Figure V-19

Programme VHDL :

Un exemple de programme VHDL, qui correspond au module feux uniquement, est fourni ci-dessous ; il se déduit directement du diagramme de transitions précédent.

```
entity feux is
   port ( hor, piet, Tor, Tr, Tv : in bit ;
     R, O, V, cpt : out bit );
end feux ;

architecture comporte of feux is
signal etat : bit_vector(3 downto 0) ;
begin
R <= etat(3) ;
O <= etat(2) ;
V <= etat(1) ;

machine : process -- diagramme de transitions.
begin
   wait until hor = '1' ;
      case etat is
         when X"0" -- états en hexadécimal.
                  => if (Tor or piet) = '1' then
                        etat <= X"4" ;
                     end if ;
         when X"4" => if (Tor and piet) = '1' then
                        etat <= X"5" ;
            elsif (Tor and not piet) = '1' then
                        etat <= X"0" ;
                     end if ;
         when X"5" => if Tor = '1' then
                        etat <= X"8" ;
                     end if ;
         when X"8" => if Tr = '1' then
                        etat <= X"2" ;
                     end if ;
         when X"2" => if Tv = '1' then
                        etat <= X"4" ;
                     end if ;
         when others => etat <= X"0" ;
            -- pour les états inutilisés.
      end case ;
end process machine ;

mealy : process -- calcul de la sortie cpt.
begin
   wait on etat, piet, Tor, Tr, Tv ; -- liste de sensibilité
   cpt <= '0' ; -- assure un bloc combinatoire.
```

```
        case etat is
           when X"0" => if Tor = '1' or piet = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"4" => if Tor = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"5" => if Tor = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"8" => if Tr = '1' then
                           cpt <= '1' ;
                        end if ;
           when X"2" => if Tv = '1' then
                           cpt <= '1' ;
                        end if ;
           when others => null ; -- case complet.
        end case ;
    end process mealy ;
    end comporte ;
```

Le décodeur Manchester réexaminé.

Comme deuxième exemple, reprenons, en la complétant un peu, l'étude du décodeur Manchester différentiel. Nous avions omis, dans la version précédente, un deuxième signal de sortie, rx, qui indique aux utilisateurs la cadence de transmission. Comme on peut le voir sur la figure V-20 , ce signal a une fréquence moitié de celle de l'horloge, mais il ne peut pas s'agir d'un simple diviseur par deux : un diviseur par deux est incapable de distinguer les transitions systématiques des transitions signifiantes du signal d'entrée man, il est incapable de se synchroniser.

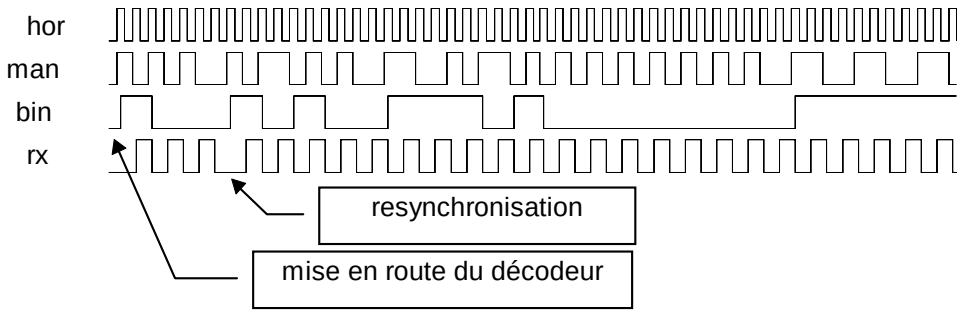


Figure V-20

Choisissons, comme précédemment, la sortie Q2 (poids fort) du registre d'état pour générer le signal bin. Pour le signal rx, il est pratique de prendre la sortie Q0 de ce registre ; une fois le décodeur synchronisé, les trajets parcourus dans le diagramme de transitions doivent être tels que la parité du code de l'état change à chaque transition : le successeur d'un nombre impair doit être pair, et réciproquement. Le diagramme de la figure V-7, étudié précédemment, ne respecte pas cette clause (transition a→c , par exemple), et ne peut pas la respecter, le nombre d'états n'étant pas suffisant (si on échangeait les codes des états a et b, par exemple, la transition e→a ne respecterait plus l'alternance de parité).

Partant du cycle c→d→e→f..., qui correspond aux transmissions de signaux binaires égaux à '1', on adjoint à ce cycle deux cycles équivalents, a→b→a...et g→h→g..., qui codent les '0' transmis, mais avec une parité inversée.

On obtient un diagramme à 8 états, qui peut, par exemple, être celui de la figure V-21.

Comme précédemment, les conditions de maintien sont, en régime établi, toujours fausses. Leur détection pourrait servir à indiquer une faute de synchronisation.

Nous laisserons au lecteur le soin de traduire ce diagramme de transitions en équations de commandes des bascules, et en programme VHDL, ce qui ne pose guère de difficulté.

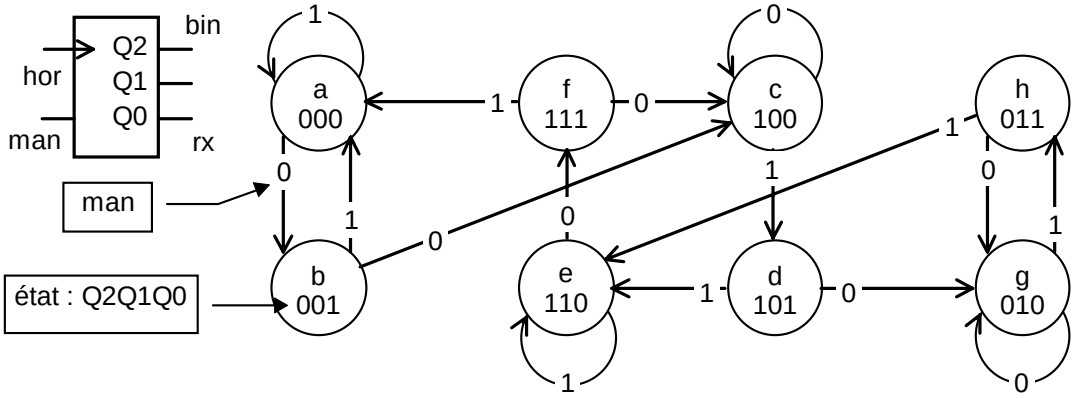


Figure V-21

Bascules enterrées.

Dans les deux exemples précédents, certaines bascules servent de sorties, d'autres ne servent qu'aux états internes. De telles bascules sont dites bascules *enterrées* (*buried flip flop*). De nombreux circuits programmables offrent la possibilité d'utiliser des bascules enterrées ; cela a l'avantage de diminuer, pour une complexité de circuit donnée, le nombre de broches d'accès nécessaires. Il est

clair, cependant, que ces circuits sont plus délicats à tester : les états ne sont pas tous visibles en sortie.

Codes « un seul actif »

Les codes dits un seul actif (*one hot*), sont les plus dilués : à chaque état on attribue une bascule ; la machine étant, par définition, dans un seul état à la fois, si l'une des bascules est active, toutes les autres sont inactives. La commande de feux, étudiée au paragraphe précédent, serait une commande de ce type si on n'avait pas eu la fantaisie d'y rajouter une bascule enterrée[27].

Code binaire

C'est le code classique des compteurs, nous l'avons rencontré, par exemple, à l'occasion du diviseur à double rapport de division 255/256.

C'est typiquement le code que l'on obtient quand on réalise des machines d'états avec des fonctions standard.

Codes adjacents

On dit qu'un code est adjacent, dans le cas d'une machine d'état, si pour toutes les transitions du diagramme d'états, le changement de valeur du registre d'état ne porte que sur un chiffre binaire.

Très en vogue quand on synthétisait des automates asynchrones, ces codes ont perdu de l'importance avec la généralisation des techniques synchrones. La contrainte que représente le respect de l'adjacence, pour tous les états successifs, devient rapidement très difficile à observer.

On peut, malgré tout noter que, si cela ne complique pas, par ailleurs, le problème, c'est souvent une bonne idée de respecter l'adjacence dans les transitions, au moins partiellement.

Les états inutilisés

Tous les codes qui n'occupent pas la totalité des états accessibles génèrent des états inutilisés. Notre feu rouge de tout à l'heure, par exemple, utilisait cinq des seize états disponibles. Le non raccordement des états inutilisés dans l'un des états du cycle relèverait, dans ce cas de la roulette russe, mais avec les deux tiers des logements du barillet du révolver chargés.

Si on n'est pas certain que les états inutilisés rejoignent naturellement l'un des états utiles du diagramme de transitions, il faut obligatoirement leur adjoindre une transition qui les ramène dans un territoire connu, faute de quoi on risque de créer une machine qui se « plante » à la première occasion.

---

[27]En l'occurence, un code *one hot zero*, car l'état où toutes les bascules sont à zéro fait partie du code. S'il y a toujours une bascule active, la combinaison « zéro » n'est pas dans le code. On parle parfois, dans ce cas, de code *one hot one*.

Les états équivalents

Lors de la première ébauche d'un diagramme de transitions, il peut arriver que l'on crée des états inutiles. Cela n'est pas, en soi, dramatique, mais il peut être intéressant de les rechercher quand, notamment, l'économie d'un ou deux états permet de réduire la taille du registre d'état.

Quand deux états sont ils équivalents ?
Quand ils génèrent les mêmes sorties et les mêmes valeurs futures des sorties, quelles que soient les séquences d'entrée.

Derrière cette définition, fort simple en apparence, se cache parfois une grande difficulté de mise en pratique de cette recherche.

Un cas particulier simple à identifier se rencontre assez souvent sur des diagrammes de transitions de dimension raisonnable : deux états fournissent les mêmes sorties et ont les mêmes états futurs, ils sont alors équivalents, on peut supprimer l'un d'entre eux.

Synchronisations des entrées et des sorties

Nous n'envisageons que la réalisation des machines d'états synchrones. Cela veut dire qu'au niveau local toutes les bascules qui interviennent sont pilotées par une horloge unique. Il est clair qu'au niveau d'un système cette règle du synchronisme absolu est rarement observée, elle nuirait à la modularité des sous ensembles.

Lors des échanges entre sous ensembles pilotés par des horloges différentes, ou pour des interfaces avec un monde extérieur qui ne possède pas d'horloge du tout, un piéton, par exemple, la question de la synchronisation des signaux d'entrée et de sortie se pose.

Synchronisations des entrées

Les signaux d'entrée qui proviennent d'un autre sous ensemble, piloté par une horloge différente, ou totalement asynchrone, doivent *toujours* être resynchronisés au moyen de bascules (voir paragraphe II-3 pour plus de précision).

Synchronisations des sorties

Les sorties calculées par des fonctions logiques combinatoires génèrent des impulsions parasites, nous en avons vu un exemple avec le décodeur Manchester. Totalement inoffensives si elles sont exploitées par un système piloté par la même horloge, ces impulsions peuvent être fort mal acceptées par un récepteur asynchrone de l'horloge locale.

Des bascules de synchronisation des sorties suppriment ce défaut, elles assurent des signaux stables entre deux fronts d'horloge.

L'adjonction pure et simple de bascules en sortie, à la place du bloc de calcul des sorties du synoptique de la figure V-1, retarde d'une période d'horloge les

valeurs des sorties par rapport à celles du registre d'état. Si ce retard présente un inconvénient, il est toujours possible, quitte à alourdir la partie combinatoire de la réalisation, d'anticiper le calcul des sorties en utilisant pour leur calcul l'état futur au lieu de l'état actuel. Le synoptique de la figure V-1 devient alors (figure V-22) :

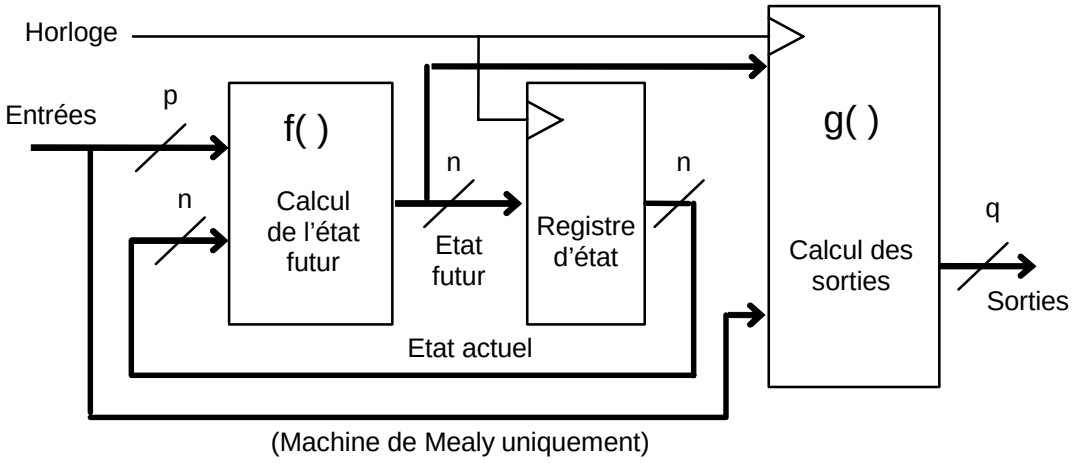


Figure V-22

V.3. Fonctions combinatoires

Dans toute réalisation d'un système numérique interviennent des fonctions combinatoires ; nous en avons déjà fait grand usage, sans avoir ressenti la nécessité de formaliser les choses au delà de quelques définitions élémentaires de l'algèbre de Boole. Dans les applications pratiques, ces fonctions combinatoires, bien que ne créant guère de difficulté de principe, sont la source principale de complexité du schéma électrique obtenu.

Sans y attacher une importance exagérée, le concepteur se doit de connaître quelques principes généraux, qui interviennent dans la manipulation de ces fonctions, ne serait-ce que pour comprendre les documents techniques qui accompagnent les circuits et les logiciels d'aide à la synthèse.

Après deux définitions classiques concernant les écritures standard des fonctions, nous aborderons rapidement les principes utilisés pour minimiser les équations correspondantes.

V.3.1 Des tables de vérité aux équations : les formes normales

Etant donnée une fonction $f(e_{n-1}, e_{n-2}, ...,e_0)$, elle est complètement spécifiée par la donnée des 2^n valeurs $f(0, 0, ...,0)$, $f(0, 0, ...,1)$ ...., $f(1, 1, ...,1)$, qui sont les

éléments de sa table de vérité. Mais cette forme de représentation est difficilement manipulable.

Aussi préfère-t-on généralement la décrire par une expression polynômiale qui fait intervenir les opérateurs fondamentaux de l'algèbre de Boole que sont la réunion (ou), l'intersection (et) et la complémentation (non). On distingue traditionnellement deux formes, équivalentes, d'écriture standard pour une fonction logique, nommées tout simplement première et deuxième formes normales (ou canoniques).

Première forme normale : une somme de produits

La première forme normale est la plus couramment utilisée : elle consiste à écrire une fonction combinatoire sous forme de somme de produits (réunion d'intersections) :

$$\begin{aligned} f(e_{n-1}, e_{n-2}, \ldots, e_0) = {} & f(0, 0, \ldots, 0) * \overline{e_{n-1}} * \overline{e_{n-2}} * \ldots * \overline{e_0} \\ & + f(0, 0, \ldots, 1) * \overline{e_{n-1}} * \overline{e_{n-2}} * \ldots * e_0 \\ & + \cdots\cdots\cdots \\ & + f(1, 1, \ldots, 1) * e_{n-1} * e_{n-2} * \ldots * e_0 \end{aligned}$$

Chaque terme de cette somme logique s'appelle un *minterme*.

Dans ce développement polynômial, seuls restent les termes qui correspondent à une valeur, dans la table de vérité de la fonction, égale à '1'.

Un opérateur ou exclusif, par exemple, s'écrit, dans cette représentation :

$$\begin{aligned} a \oplus b = {} & (0 \oplus 0) * \bar{a} * \bar{b} + (0 \oplus 1) * \bar{a} * b \\ & + (1 \oplus 0) * a * \bar{b} + (1 \oplus 1) * a * b \\ a \oplus b = {} & a * \bar{b} + \bar{a} * b \end{aligned}$$

Cette écriture correspond à une phrase du type :

« a ou exclusif b égale un si
a égale 1 *et* b égale 0,
ou si
a égale 0 *et* b égale 1. »

La grande majorité des circuits programmables ont une architecture interne qui reproduit, en trois couches logiques (non – et – ou), un développement en première forme normale.

Notation condensée

Pour alléger la notation, on désigne parfois chaque minterme par un symbole, m_i, où i représente le numéro de la ligne de la table de vérité correspondante.

Par exemple :

$$m_0 = \overline{e_{n-1}} * \overline{e_{n-2}} * \cdots * \overline{e_0} \quad , \quad m_1 = \overline{e_{n-1}} * \overline{e_{n-2}} * \cdots \overline{e_1} * e_0$$

Une fonction s'écrit alors :

$$f(e_{n-1}, e_{n-2}, \ldots e_0) = \sum_{f(ei)=1} m_i$$

ou, encore plus simplement :

$$f = \sum m(i_1, i_2 \ldots . i_p)$$

où les numéros i_k indiquent simplement les numéros des lignes de la table de vérité dans lesquelles la fonction vaut '1'.

Par exemple : $a \oplus b = m_1 + m_2 = \sum m(1,2)$

Deuxième forme normale : un produit de sommes

La même fonction peut être écrite sous forme de produit (logique) de sommes (logiques), on parle alors de deuxième forme normale, ou canonique :

$$\begin{aligned} f(e_{n-1}, e_{n-2}, \ldots, e_0) = {} & (f(0, 0, \ldots, 0) + e_{n-1} + e_{n-2} + \ldots + e_0) \\ & * (f(0, 0, \ldots, 1) + e_{n-1} + e_{n-2} + \ldots + \overline{e_0}) \\ & * \cdots\cdots\cdots \\ & * (f(1, 1, \ldots, 1) + \overline{e_{n-1}} + \overline{e_{n-2}} + \ldots + \overline{e_0}) \end{aligned}$$

Pour une raison dont nous lèverons le mystère un peu plus loin, chaque facteur de ce produit logique porte le nom de *maxterme*.

On notera que la règle d'association entre les valeurs des variables, dans l'écriture de la fonction, et la forme directe ou complémentée avec laquelle interviennent ces variables dans le développement diffère par rapport à la première forme normale : à un '0' on associe la variable elle même, et à un '1' son complément. Cette inversion de règle est une source d'erreurs fréquentes quand on utilise la deuxième forme normale. Aussi conseillerons nous vivement au lecteur la prudence :

1. Utiliser toujours la même forme normale, de préférence la première.
2. S'il s'avère nécessaire d'obtenir le développement en deuxième forme, la méthode la plus sûre consiste à écrire le complément de la fonction cherchée en première forme normale, puis d'appliquer les théorèmes de De Morgan pour obtenir le résultat souhaité.

Un opérateur ou exclusif, par exemple, s'écrit, dans cette représentation :

$$a \oplus b = ((0 \oplus 0) + a + b) * ((0 \oplus 1) + a + \bar{b}) * ((1 \oplus 0) + \bar{a} + b) * ((1 \oplus 1) + \bar{a} + \bar{b})$$
$$a \oplus b = (a + b) * (\bar{a} + \bar{b})$$

La deuxième forme normale correspond à une phrase du type :

a ou exclusif b égale un si
    a égale 1 *ou* b égale 1,
et si
    a égale 0 *ou* b égale 0.

Notation condensée

Pour alléger la notation, on désigne parfois chaque maxterme par un symbole, M_i, où i représente le numéro de la ligne de la table de vérité correspondante.
Par exemple :

$$M_0 = e_{n-1} + e_{n-2} + \cdots + e_0, \quad M_1 = e_{n-1} + e_{n-2} + \cdots + e_1 + \overline{e_0}$$

Une fonction s'écrit alors :

$$f(e_{n-1}, e_{n-2}, \ldots e_0) = \prod_{f(ei)=0} M_i$$

ou, encore plus simplement :

$$f = \prod M(i_1, i_2 \ldots . i_p)$$

où les numéros i_k indiquent simplement les numéros des lignes de la table de vérité dans lesquelles la fonction vaut '0'.

Par exemple : $a \oplus b = M_0 + M_3 = \prod M(0,3)$

V.3.2 L'élimination des redondances : les minimisations

Quand on exprime une fonction combinatoire, directement à partir du résultat à obtenir, il arrive presque toujours qu'il y ait des redondances dans les équations obtenues.

Prenons un exemple.

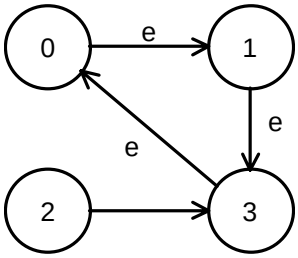


Figure V-23

Soit à réaliser un compteur modulo 3, qui, en fonction d'une entrée e, obéisse au diagramme de transitions de la figure V-23 :

- Quand e = '1', le compteur s'incrémente, sinon il reste dans l'état.
- L'état 2 a été raccordé dans le cycle, pour éviter tout risque de « piège ».

Les équations de ce système s'obtiennent immédiatement, avec deux bascules D nous obtenons :

$$D1 = \overline{Q1} * Q0 * e + Q1 * Q0 * \overline{e} + Q1 * \overline{Q0}$$
$$D0 = \overline{Q1} * \overline{Q0} * e + \overline{Q1} * Q0 * \overline{e} + \overline{Q1} * Q0 * e + Q1 * Q0 * \overline{e} + Q1 * \overline{Q0}$$

Avec un peu de réflexion on voit apparaître des simplifications, entre le deuxième et le troisième terme de l'expression de D0, par exemple, la variable e se simplifie. Le problème pratique qui se pose est de trouver une méthode systématique de recherche de telles simplifications. Ce point fait l'objet des paragraphes suivants.

Quelques remarques préalables s'imposent :

- Les simplifications algébriques, qui sont celles qui nous occupent ici, ne garantissent en aucun cas le nombre minimum d'opérateurs élémentaires pour réaliser une fonction. L'exemple présenté du contrôleur de parité, paragraphe III-2, en est un exemple flagrant. Ces simplifications fournissent une forme minimale des expressions dans une construction en trois couches, similaire à l'une des deux formes canoniques. Elles représentent donc un optimum dans lequel la vitesse de transfert du circuit réalisé (nombre de couches) intervient en premier, le nombre d'opérateurs, donc la surface de silicium occupée, intervenant en second. Le concepteur est souvent amené à accepter une certaine forme de compromis, dans les cas pratiques.
- Malgré les restrictions précédentes, il serait faux de croire que les simplifications sont inutiles, une optimisation passe de toute façon par des minimisations des sous-ensembles que les contraintes de surface occupée auront défini.
- Les « compilateurs de silicium » ont grandement changé les données du problème à plusieurs niveaux :

  1. Ils génèrent, à partir des langages de haut niveau des constructions extrêmement lourdes, où intervient, schématiquement, une couche de

multiplexeurs pour chaque niveau d'imbrication des instructions de tests (if, case, ... etc). Sans optimisation des équations générées un langage de haut niveau serait inutilisable[28].

2. Ils permettent de gérer des fonctions à grand nombre de variables d'entrée difficilement calculables à la main. La moindre machine d'états, pour laquelle on a adopté un codage dirigé par les sorties, donc relativement dilué, génère des fonctions qui découragent vite, par le nombre de variables mises en jeu, les calculs manuels.
3. Des deux points précédents résulte le fait que l'on ne fait plus jamais les calculs de simplifications entièrement à la main ; tous les compilateurs disposent de programmes de minimisation des expressions logiques. Par contre, il est essentiel que l'utilisateur de tels programmes domine le principe de ce qu'ils font, même si la conception de ces programmes n'est pas de son ressort.

Nous ne nous lancerons donc pas dans des calculs sur des fonctions de plus de quatre variables, l'essentiel étant de comprendre le principe de ces calculs, non de les étendre à des cas réels. Dans tout ce qui suit nous nous limiterons à des exemples qui font intervenir la première forme canonique d'une fonction.

Simplifications algébriques directes

Une variable d'entrée x est l'objet d'une simplification si, dans l'écriture de la fonction, apparaissent deux termes de la forme :

$$f(a, \ldots t, x, y, \ldots) = \ldots + x * f_x(a, \ldots t, y, \ldots) + \bar{x} * f_x(a, \ldots t, y, \ldots) + \ldots$$

Les deux mintermes concernés sont dits adjacents, on passe de l'un à l'autre en changeant la variable x uniquement.

La fonction se simplifie en :

$$f(a, \ldots t, x, y, \ldots) = \ldots + f_x(a, \ldots t, y, \ldots) + \ldots$$

Au delà de trois variables, et encore, la difficulté est de repérer les mintermes qui vont intervenir dans des simplifications. Cette difficulté est accentuée par le fait que le même minterme peut intervenir dans plusieurs simplifications différentes, auquel cas il faut le « dupliquer » avant de simplifier effectivement, c'est ce qui se produit dans l'expression suivante :

$$\begin{aligned} f(a, b, c) &= \bar{a} * \bar{b} * c + \bar{a} * b * c + a * \bar{b} * c \\ &= (\bar{a} * \bar{b} * c + \bar{a} * b * c) + (\bar{a} * \bar{b} * c + a * \bar{b} * c) \\ &= \bar{a} * c + \bar{b} * c \end{aligned}$$

[28]La même remarque peut être faite à propos des langages de programmation. Si certains pensent encore que la programmation en langage machine est plus efficace, c'est probablement en raison de la faible efficacité des premiers compilateurs disponibles. Les temps ont changé.

Cette difficulté de repérage limite grandement les calculs algébriques directs, d'où la méthode graphique des tableaux de Karnaugh.

Les tableaux de Karnaugh

Définition

Les tableaux de Karnaugh, une variante des tables de vérité, sont organisés de telle façon que les mintermes adjacents soient systématiquement regroupés dans deux cases voisines, donc faciles à identifier visuellement.

On représente les valeurs de la fonction dans une table à deux dimensions, organisée comme un damier, dans laquelle chaque ligne et chaque colonne correspond à une combinaison des variables d'entrée. Le codage des lignes et des colonnes est fait dans un code adjacent, ou code de Gray (figure V-24).

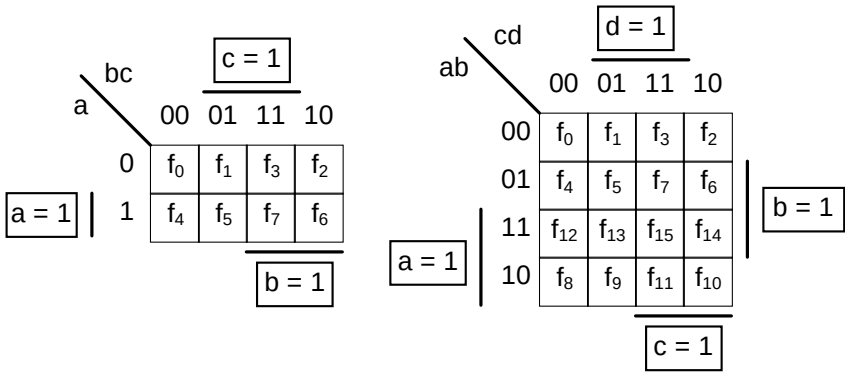


Figure V-24

Chaque minterme correspond à une cellule élémentaire de la table, qui constitue le plus petit groupement possible, d'où le nom. De même, un maxterme correspond à toutes les cellules sauf une égales à '1', soit le recouvrement de taille maximum qui ne soit pas trivial.

Cette disposition est telle que si le développement en première forme normale de la fonction contient deux mintermes adjacents, les deux '1' correspondant de la table de vérité se retrouvent dans des cases voisines.

On notera que les cases d'extrémité d'une même ligne; ou d'une même colonne, sont voisines.

Une simplification, qui est un regroupement de mintermes adjacents, apparaît alors comme une regroupement de plusieurs cases voisines. On notera qu'une simplification fait toujours intervenir un nombre de termes qui est une puissance de deux (2, 4, 8, ...).

Exemple

Reprenons les équations du compteur modulo 3 précédent. Le report des valeurs des entrées D1 et D0 des deux bascules, en fonction de Q1, Q0 et e, conduit aux tableaux de Karnaugh de la figure V-25 :

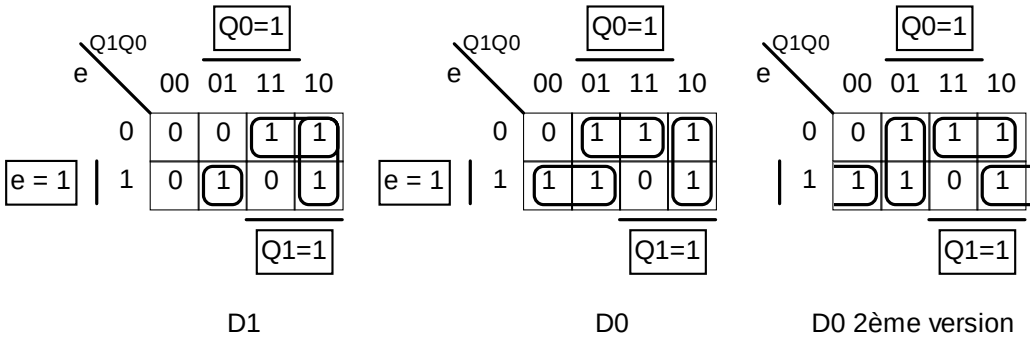


Figure V-25

La figure précédente met en évidence le fait que la solution n'est pas toujours unique, mais si deux solutions sont possibles, elles ont la même complexité.

Des diagrammes précédents nous déduisons les équations (1ère version de D0) :

$$D1 = \overline{Q1} * Q0 * e + Q1 * \overline{e} + Q1 * \overline{Q0}$$
$$D0 = Q0 * \overline{e} + Q1 * \overline{Q0} + \overline{Q1} * e$$

Nous ne pouvons que conseiller au lecteur de reprendre les exemples de machines d'états que nous avons étudiées, et d'en déduire les équations de commandes au moyen de tableaux de Karnaugh.

Fonctions incomplètement spécifiées

Il arrive, souvent, en fait, que les données du problème laissent non spécifiées les valeurs des sorties pour certaines combinaisons des variables d'entrées. Dans ce cas, le concepteur peut choisir des valeurs à sa convenance, avec prudence comme nous allons le voir, pour tenter de minimiser les équations qui en résultent. Une valeur non spécifiée, par le cahier des charges, est traditionnellement notée ϕ dans

un tableau de Karnaugh. Il est important de noter que, non spécifiée initialement, cette valeur sera bien déterminée une fois les équations choisies !

Reprenons, à titre d'illustration, les équations du décodeur Manchester, sous forme de machine de Moore, dont le diagramme de transitions à 6 états a été représenté à la figure V-15.

Les états 2 et 3 ne figurent pas sur le diagramme, nous pouvons donc les raccorder dans le cycle à notre guise, de façon à diminuer la complexité des équations générées pour les commandes des trois bascules.

Nous obtenons, en notant m pour l'entrée man, les tableaux de la figure V-26.

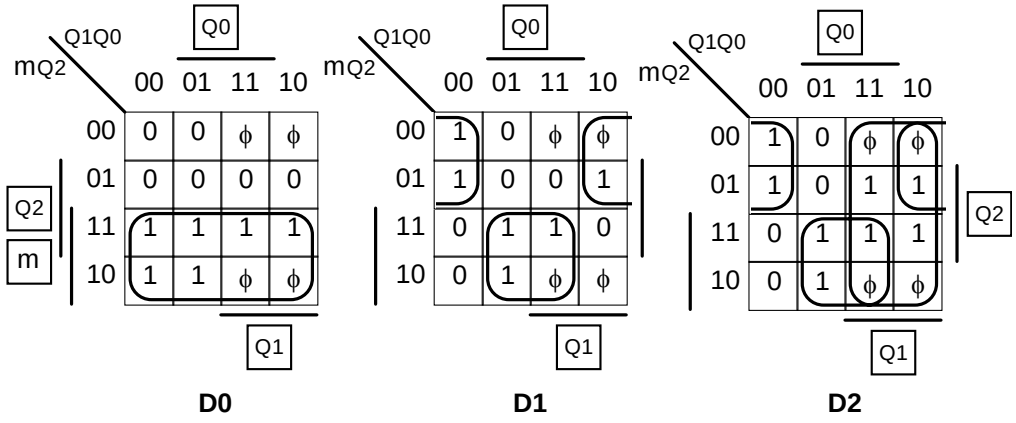


Figure V-26

D'où les équations précédemment données sans justification :

$$D0 = man$$
$$D1 = \overline{Q0 \oplus man}$$
$$D2 = Q1 + \overline{Q0 \oplus man}$$

On notera qu'il est essentiel, quand on a utilisé des valeurs non spécifiées, de contrôler à postériori, quand on connait les valeurs attribuées aux « ϕ », que l'on n'a pas généré de piège dans le diagramme de transitions.

L'élimination des parasites de commutation

Outre les simplifications, les tableaux de Karnaugh permettent de prévoir si, lors des changements des valeurs des entrées, on risque de créer des impulsions parasites, ce que l'on appelle des aléas statiques.

Un aléa statique se manifeste, par exemple, par la présence d'un '0', de durée brève, lié aux temps de propagation dans les circuits, lors du changement d'une entrée telle que la fonction vaut '1' avant et après le changement.

La règle est simple : pour assurer l'absence d'aléa statiques, il faut qu'il y ait des recouvrements partiels des groupements du tableau de Karnaugh, de sorte que quand on passe d'un groupement à un autre, on ne franchisse qu'une seule frontière.

Prenons comme exemple un multiplexeur (figure V-27) :

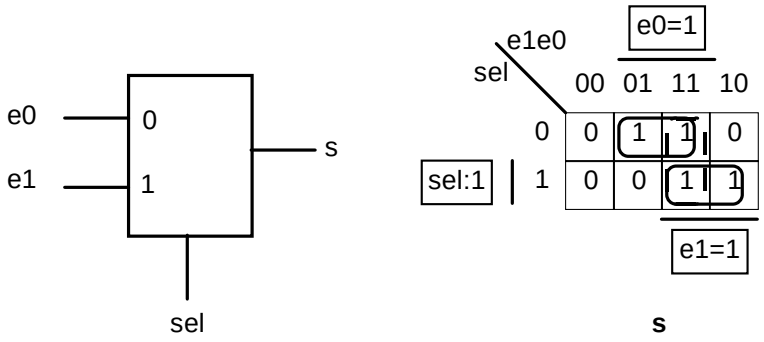


Figure V-27

Les deux groupements en traits pleins ne se recouvrent pas, quand les deux entrées sont à '1', et que l'on change la valeur de la commande de sélection, on risque d'obtenir un parasite à '0' en sortie. Ce parasite est éliminé par l'adjonction du groupement indiqué en pointillé, qui assure la continuité du pavage. D'où l'équation d'un multiplexeur « sans aléa » :

$$s = e0 * \overline{sel} + e1 * sel + e0 * e1$$

Il est essentiel d'utiliser un tel multiplexeur pour réaliser des bascules, si non la bascule mémorise justement l'aléa ! Dans les circuits programmables qui utilisent, comme cellules élémentaires des multiplexeurs, ceux-ci sont garantis sans aléa, de façon à pouvoir réaliser sans risque des bascules.

Les logiciels de minimisation

Comme nous l'avons mentionné, tous les compilateurs sont assortis d'un programme de minimisation des équations logiques générées. Il est hors de question, ici, de rentrer dans les détails de ces programmes, qui sont loin d'être triviaux. Nous nous contenterons de citer deux algorithmes, parmi les plus connus[29].

[29]Pour les lecteurs intéressés, voir : HILL F.J. et PETERSON G.R., *Computer aided logical design with emphasis on VLSI,* John WILEY, New York, 1993.

La méthode de Quine-McCluskey

L'algorithme de Quine-McCluskey, qui date de 1956, utilise, comme les tableaux de Karnaugh, mais d'une façon systématique, et non visuelle, des tables qui décrivent tous les mintermes possibles d'une fonction.

Comme toute méthode tabulaire, tableaux de Karnaugh compris, la taille des données à manipuler croît exponentiellement avec le nombre de variables d'entrées.

Cette croissance exponentielle rend vite impraticables, même sur ordinateur, ces méthodes dès qu'il s'agit de traiter des fonctions à grand nombre de variables d'entrées.

L'avènement des compilateurs de silicium a rendu nécessaire la création d'algorithmes qui, même s'ils ne garantissent pas un optimum absolu, laissent espérer une simplification importante, avec un algorithme qui ne soit pas exponentiel.

Espresso

Au lieu de partir d'une table des mintermes, l'algorithme Espresso (1984) manipule l'expression algébrique d'une fonction, en tentant de la transformer, de proche en proche, pour aboutir à une expression plus simple. La complexité de l'algorithme dépend de la complexité réelle de la fonction plus que du nombre de variables d'entrées.

Espresso ne garantit pas d'arriver à une expression minimale absolue ; l'algorithme peut s'achever dans des situations de minimums locaux, dont il n'arrive pas à sortir[30]. Des tests effectués ont montré, cependant, que pour des fonctions de plus de 25 variables d'entrées, il est arrivé à des résultats égaux, ou plus complexes d'un seul terme, qu'un algorithme systématique. Et ce, pour un temps de calcul plus faible dans un rapport 10 à 100 suivant les fonctions analysées.

La plupart des systèmes de CAO utilisent cet algorithme pour les calculs de minimisation d'expressions logiques[31].

---

[30]Schématiquement, le problème est que, même quand on a trouvé un recouvrement complet de groupements dans un tableau de Karnaugh, il peut correspondre à un minimum local. Le fait de réutiliser plusieurs fois des mintermes, qui ont peut être disparu lors de simplifications précédentes, permet parfois de construire un recouvrement complètement différent de celui dont on est parti, qui aboutit à un résultat plus simple. Autant ce genre de choses saute aux yeux d'un humain, qui a une vision globale de la situation, autant il est difficile de formaliser cette démarche. Espresso retire, au fur et à mesure de ses calculs les termes qu'il est sûr de devoir garder, et commence par « désimplifier » ce qui reste de la fonction pour chercher un autre recouvrement. Il applique cette démarche de façon récursive jusqu'à ce qu'il cesse de progresser.

[31]Il n'est jamais arrivé aux auteurs de rencontrer une fonction, humainement analysable, où le résultat manuel soit meilleur que celui d'espresso. Il est vrai que nous ne pratiquons plus guère les tableaux de Karnaugh de grandes dimensions.

V.4. Séquenceurs et fonctions standard

Pendant deux décennies la plupart des machines d'états furent réalisées au moyen de fonctions logiques standard, compteurs programmables, multiplexeurs et décodeurs principalement.

V.4.1 Séquenceurs câblés

Dans un séquenceur câblé les équations logiques du système sont matérialisées par un câblage figé entre les différents constituants.

Nous nous contenterons ici de donner l'architecture générale d'un tel système, et d'indiquer la méthode qui permet de passer d'un diagramme de transitions aux commandes des circuits.

Architecture

Le noyau d'une machine d'états, qui utilise des fonctions standard, est généralement un compteur à commandes de chargement parallèle et de remise à zéro, *synchrones*, cela va sans dire, mais disons le tout de même.

Les sorties du compteur pilotent les entrées d'adresses de multiplexeurs, par exemple, qui calculent, en fonction des entrées extérieures et de l'état actuel de la machine, les commandes à appliquer au compteur.

Ces sorties sont éventuellement décodées pour générer les sorties du séquenceur. Cela correspond au synoptique de la figure V-28.

Quand on utilise une telle architecture, il est clair que le codage du diagramme de transitions doit être adapté au circuit choisi : on tente de privilégier les séquences de comptage, en limitant au maximum les « sauts » par rapport au code binaire naturel du compteur. Les blocs logiques de calcul des commandes du compteur sont d'autant plus simples qu'il y a des séquences de comptage régulières, qu'il y a le moins possible d'adresses de rupture de séquence différentes.

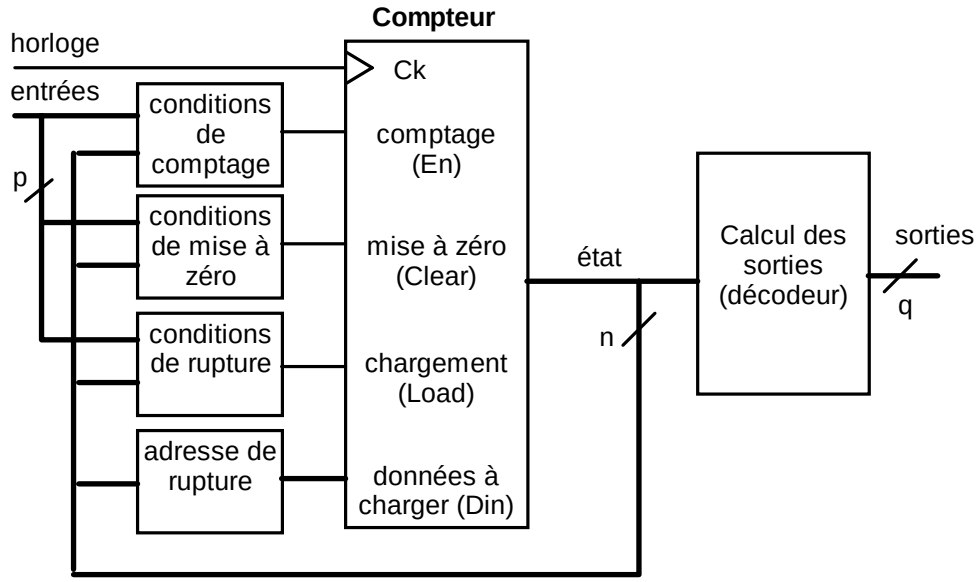


Figure V-28

Du diagramme de transitions aux commandes

Dans l'élaboration des équations, la priorité qui existe entre les commandes des compteurs classiques (74xx163, par exemple) simplifie les calculs.

Prenons comme exemple le diagramme de transitions de la figure V-21, notre dernière version du décodeur Manchester, que nous rappelons ici (figure V-29) :

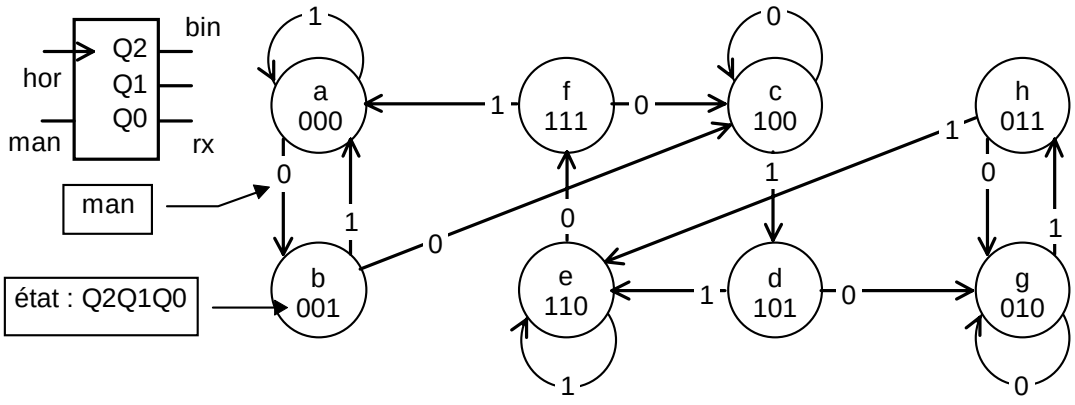


Figure V-29

De ce diagramme on peut déduire les commandes du compteur, en utilisant une notation symbolique où interviennent les noms des états[32] :

$$\begin{aligned}
\mathsf{Clear} &= \mathbf{b} * \overline{\mathsf{man}} + \mathbf{f} * \mathsf{man} \\
\mathsf{Load} &= \mathbf{b} * \overline{\mathsf{man}} + \mathbf{d} * \overline{\mathsf{man}} + \mathbf{f} * \overline{\mathsf{man}} + \mathbf{h} \\
\mathsf{En} &= \mathbf{a} * \overline{\mathsf{man}} + \mathbf{c} * \mathsf{man} + \mathbf{d} * \mathsf{man} + \mathbf{e} * \overline{\mathsf{man}} + \mathbf{g} * \mathsf{man}
\end{aligned}$$

Ces équations se prêtent bien à une réalisation avec des multiplexeurs ; si on les matérialise au moyen de portes élémentaires, la priorité entre les commandes du compteur permet de mettre des « ϕ » dans les tables de vérité qui déterminent Load (partout où Clear est vrai) et En (partout où Clear ou Load sont vrais), autorisant des simplifications supplémentaires.

Les adresses de rupture sont extrêmement simples, en raison du faible nombre de branchements (on peut mettre « ϕ » dans toutes les cases de la table de vérité qui correspondent à une condition où l'équation de Load est fausse) :

$$\begin{aligned}
\mathsf{D0} &= 0 \\
\mathsf{D1} &= \mathsf{Q1} \oplus \mathsf{Q2} \\
\mathsf{D2} &= \overline{\mathsf{Q0} \oplus \mathsf{Q2}} + \mathsf{man}
\end{aligned}$$

Nous ne détaillerons pas plus ce type d'application des compteurs programmables. Le lecteur intéressé en trouvera de nombreux exemples dans les références bibliographiques.

V.4.2 Séquenceurs micro-programmés

Dans l'architecture précédente, toutes les équations sont figées par les circuits utilisés et leur câblage ; pour obtenir des machines d'états universelles, capables de matérialiser n'importe quel diagramme de transitions sans modification du schéma, les concepteurs s'orientèrent vers les séquenceurs microprogrammés.

L'idée générale est simple, la partie la plus figée d'un séquenceur câblé traditionnel est le bloc de calcul de l'adresse, en cas de rupture de séquence. Si on introduit, dans le code binaire de l'état de la machine, un champ réservé à cette adresse, on gagne en souplesse.

Une architecture, simplifiée, d'une telle machine est donnée à la figure V-30 :

[32] Il faut, par exemple, remplacer l'état f par son équivalent binaire, soit Q2∗Q1∗Q0.

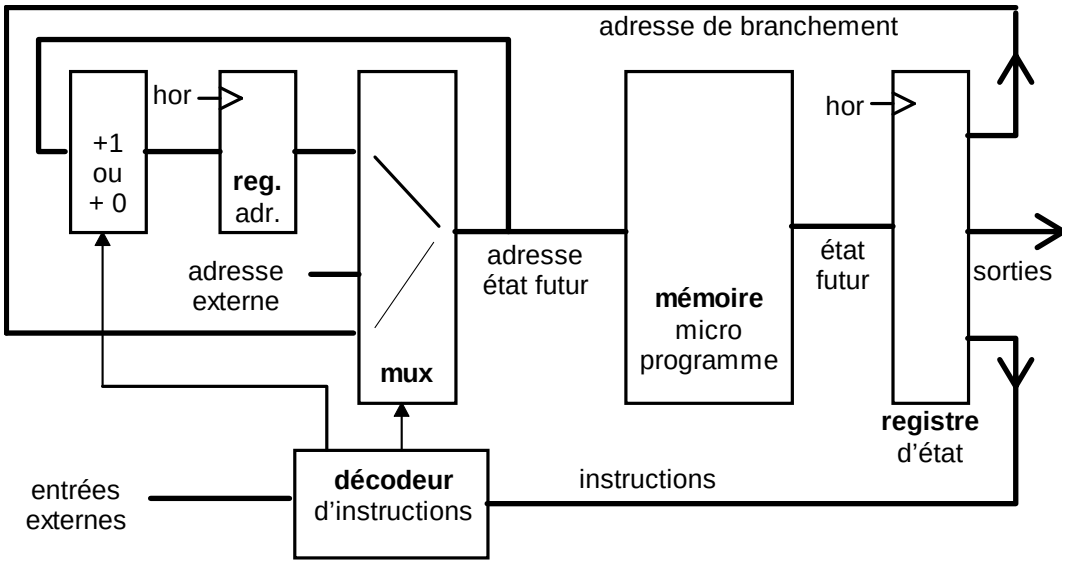


Figure V-30

Allant plus loin dans cette voie, on structure le code de l'état en champs qui représentent chacun une commande standard. On utilise une mémoire pour stocker les valeurs des états du diagramme de transitions, et deux registres pour stocker *la valeur* de l'état actuel et *l'adresse* de l'état futur.

Des commandes classiques sont[33] :

- Incrémenter l'adresse de l'état d'une unité si une condition est vraie, cela provoque un déroulement en séquence, avec une condition d'attente.
- Charger dans le registre d'état la valeur dont l'adresse est donnée dans le champ d'adresse, il s'agit d'un branchement.
- Charger dans le registre d'état la valeur dont l'adresse est fournie par une entrée externe, il s'agit d'un saut à un autre diagramme de transitions.

Le diagramme de transitions devient, en fait un véritable programme, d'où son nom de micro programme, dans lequel les états deviennent des micro instructions. Outre les commandes, les micro instructions contiennent des champs qui définissent les actions vers l'extérieur.

En enrichissant la machine d'une mémoire RAM organisée en pile et d'un compteur auxiliaire, il devient possible de créer des boucles et des sous programmes.

L'avènement des logiciels de synthèse en langage de haut niveau et des circuits programmables par l'utilisateur a considérablement restreint le champ d'application des machines micro programmées. Il est actuellement plus simple de créer sa propre machine d'états, décrite en langage évolué, programmée et modifiée

[33]On consultera, par exemple, une notice du circuit Am2910.

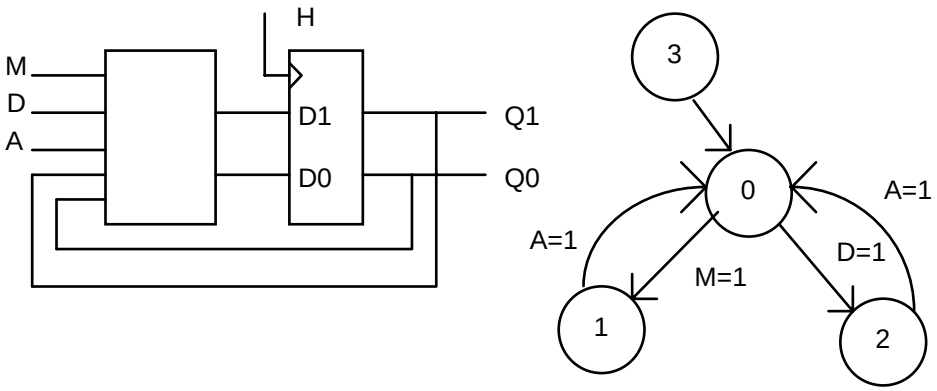
en quelques minutes dans un circuit en technologie *Flash*, que de créer des micro programmes que l'on inscrira dans une mémoire de même technologie.

La souplesse a changé de camp.

Exercices

Cohérence d'un diagramme de transition.

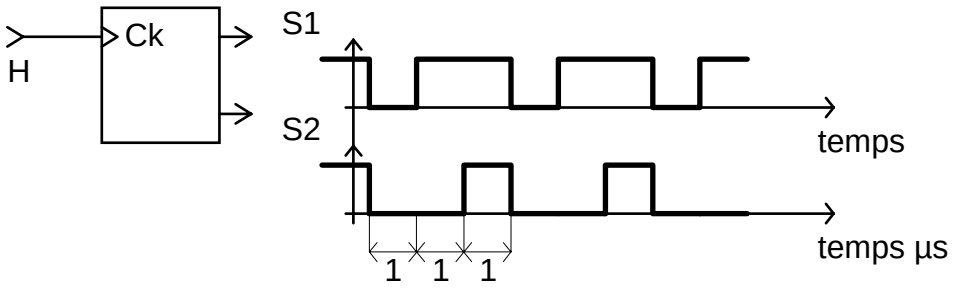
Une machine d'états synchrone dispose de trois entrées de commande (en plus de l'horloge), M, D et A (on peut imaginer qu'il s'agit, par exemple, d'une partie de commande d'ascenseur). Une ébauche de diagramme de transition est représentée ci-dessous. Cette ébauche de diagramme comporte une faute de principe, expliquer laquelle. Proposer une correction.



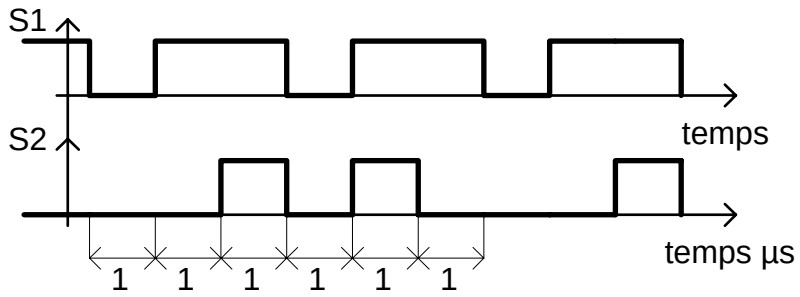
En rajoutant les conditions de maintien, qui ne sont pas représentées, donner les équations logiques de D1 et D0 induites par le diagramme corrigé.

Génération de signaux

On souhaite réaliser une fonction logique synchrone qui fournit en sortie les signaux suivants :



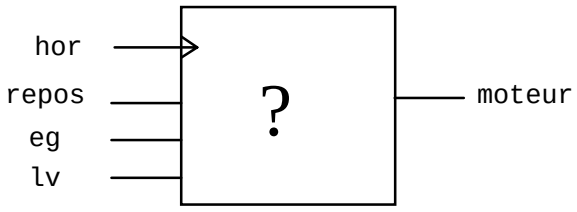
Etablir un diagramme de transition qui permet de répondre au problème.
Proposer une solution qui fait appel à des bascules D, puis à des bascules J-K.
Préciser quelle doit être la fréquence de l'entrée d'horloge.
Les chronogrammes précédents sont modifiés comme indiqué ci-dessous :



Montrer qu'il faut rajouter à la solution précédente une bascule.
Proposer une solution qui fait appel à des bascules D.

Commande d'un moteur d'essuie glaces

Le moteur d'essuie glace d'une voiture est mis en marche soit par une commande `eg`, soit par une commande `lv`, la seconde actionnant simultanément la pompe du lave glace. On se propose de faire une réalisation, en logique synchrone, de la commande du moteur :



Les signaux d'entrée sont supposés synchrones de l'horloge `hor`.

L'entrée `repos` provient d'un détecteur de fin de course, qui indique par un '1', que les balais sont en position horizontale, pare brise dégagé.

Dans tous les cas, le moteur ne doit être arrêté que quand les balais sont en position de repos.

La commande `eg` provoque, par un '1', la mise en route du moteur (`moteur = '1'`). Ce dernier ne doit s'arrêter que quand `eg` est désactivée, et que les balais sont en position de repos.

La commande `lv` provoque, outre la mise en marche de la pompe du lave glace, la mise en route du moteur. Quand cette commande redevient inactive, le moteur ne s'arrête qu'après que les balais d'essuie glace aient effectués quatre aller retours complets, pour assécher le pare brise.

1. Proposer une machine d'états qui réponde au problème. Décrire le fonctionnement de cette machine par un diagramme de transitions.
2. En déduire un programme VHDL. On veillera à ce que la sortie `moteur` corresponde à la sortie d'une bascule synchrone.