

Modulo.
Une introduction à l'informatique

Groupe de travail DGEP, EPFL, HEP-VD, UNIL

13 septembre 2024

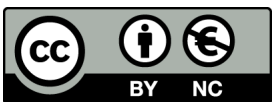


Table des matières

1 Architecture des ordinateurs **1**

1.0 Introduction . . . 1

1.0.1 De quoi sont faits les nombres binaires ? . . . 1

1.0.2 Électricité et nombres binaires . . . 3

1.0.3 Le transistor . . . 4

1.0.4 Des transistors aux systèmes logiques . . . 6

1.1 Portes logiques . . . 7

1.1.1 Exemple suivi : addition de deux nombres . . . 7

1.1.2 Portes logiques . . . 8

1.1.3 Porte ET . . . 8

1.1.4 Porte OU . . . 10

1.1.5 Porte NON . . . 10

1.1.6 Combinaisons de portes . . . 11

1.2 Additionneur . . . 21

1.2.1 Additionneur complet . . . 22

1.2.2 Chaînage d'additionneurs . . . 23

1.3 ALU . . . 29

1.3.1 Sélection de l'opération . . . 29

1.3.2 Une ALU à 4 bits . . . 35

1.4 Mémoire . . . 41

1.4.1 Le verrou SR . . . 41

1.4.2 La bascule D . . . 43

1.4.3 Addition en plusieurs étapes . . . 46

1.4.4 Récapitulatif . . . 49

1.5 CPU . . . 51

1.5.1 Horloge et accès mémoire . . . 53

1.5.2 L'unité arithmétique et logique . . . 55

1.5.3 Processeur à noyau unique . . . 57

1.5.4 Processeur à double cœur . . . 57

1.5.5 Les processeurs quadricœur et autres processeurs à cœurs multiples . . . 58

1.5.6 Le pipeline . . . 60
1.6 Architecture générale . . . 63
1.6.1 La mémoire . . . 63
1.6.2 Le CPU (Central Processing Unit) . . . 67
1.6.3 Les entrées-sorties . . . 69
1.6.4 Les bus . . . 70
1.6.5 Autres composants matériels . . . 72
1.7 TP Portes logiques . . . 75
1.7.1 Transmission d'un signal . . . 75
1.7.2 Commutateur/poussoir . . . 75
1.7.3 Feu de circulation . . . 76
1.7.4 Lampe clignotante . . . 76
1.7.5 Affichage à 7 segments . . . 77
1.7.6 Affichage à 2 chiffres . . . 78
1.7.7 Affichage à 16 segments . . . 78
1.7.8 Porte NON . . . 79
1.7.9 Afficher 0 et 1 . . . 79
1.7.10 Alterner 0 et 1 . . . 80
1.7.11 Porte OU . . . 80
1.7.12 Décodeur de clavier . . . 80
1.7.13 Clavier numérique . . . 81
1.7.14 Porte ET . . . 82
1.7.15 Décodeur binaire . . . 82
1.7.16 Décodeur de dé . . . 83
1.7.17 Décoder 0 à 3 . . . 83
1.8 TP Additionneur . . . 85
1.8.1 Clock et fréquence . . . 85
1.8.2 Porte OU-X . . . 86
1.8.3 Construire un OU-X . . . 86
1.8.4 Détecteur de parité . . . 87
1.8.5 Multiples commutateurs . . . 88
1.8.6 Addition binaire . . . 88
1.8.7 Additionneur complet . . . 89
1.8.8 Additionneur 4 bits . . . 90
1.8.9 Incrémenter (`i++`) . . . 91
1.8.10 Décrémenter (`i--`) . . . 92
1.8.11 Changer de signe (`-i`) . . . 92
1.8.12 Soustraction (`a-b`) . . . 93
1.8.13 Inversion commutée . . . 94
1.8.14 Négation commutée . . . 95
1.8.15 Soustraction commutée . . . 95
1.8.16 Les fanions (flags) . . . 96
1.9 TP ALU . . . 97
1.9.1 Sélectionneur . . . 97
1.9.2 Multiplexeur . . . 97
1.9.3 Sélection d'opérations . . . 98
1.9.4 ALU . . . 99

1.9.5 Addition signée . . . 100
1.9.6 Addition 8 bits . . . 100
1.9.7 Carry et Overflow (V) . . . 101
1.9.8 Multiplier 1x1 bit . . . 102
1.9.9 Multiplier par 2, 4, 8 . . . 103
1.9.10 Multiplier 1x4 bit . . . 103
1.9.11 Multiplier 4x4 bits . . . 104
1.9.12 Diviser par 2, 4 et 8 . . . 105
1.9.13 Registre . . . 106
1.9.14 Accumulateur . . . 107
1.9.15 Incrémenter/décrémenter . . . 108
1.9.16 Comparer . . . 108
1.10 TP Mémoire . . . 111
1.10.1 Cellule de mémoire . . . 111
1.10.2 Verrou SR . . . 112
1.10.3 Bascule D . . . 112
1.10.4 Registre 4 bits . . . 113
1.10.5 Décodeur de touche . . . 113
1.10.6 Décodeur pour 8 touches . . . 114
1.10.7 Compteur 4 bits . . . 115
1.10.8 Compteur 8 bits . . . 115
1.10.9 Compteur avec remise . . . 116
1.10.10 Rouler un dé . . . 117
1.10.11 Registre à décalage . . . 118
1.10.12 RAM (mémoire vive) . . . 118
1.10.13 RAM avec bits aléatoires . . . 119
1.10.14 RAM avec binaire . . . 120
1.10.15 RAM avec image . . . 120
1.10.16 RAM avec ASCII . . . 121
1.11 TP CPU . . . 123
1.11.1 Intel 4004 . . . 123
1.11.2 Langage assembleur . . . 124
1.11.3 Le langage machine . . . 125
1.11.4 Mémoire de programme . . . 125
1.11.5 Le jeu d'instructions . . . 126
1.11.6 Décoder une instruction . . . 127
1.11.7 Bus 4 bits . . . 127
1.11.8 Charger imméd. (LDM) . . . 128
1.11.9 Charger depuis reg (LDR) . . . 129
1.11.10 Choix entre LDR/LDM . . . 130
1.11.11 Choix entre ADD/SUB . . . 131
1.11.12 Program counter (PC) . . . 132
1.11.13 Le saut (jump) . . . 133
1.11.14 La pile (stack) . . . 134
1.11.15 Projet final . . . 135
1.11.16 Nand Game . . . 136
1.12 Objectifs . . . 137

1.13 Personnages-clés 139

CHAPITRE **1**

Architecture des ordinateurs

1.0 Introduction

Attention

Ce document doit être retravaillé …

Dans ce chapitre, nous aborderons la question de l'architecture des ordinateurs, c'est-à-dire les multiples couches physiques qui rendent possibles des opérations numériques aussi complexes que celles qu'effectuent à chaque instant nos smartphones.

Comme vous avez pu le voir dans le chapitre lié à la représentation de l'information, tout ce qui apparaît sur votre écran est représenté par l'ordinateur par suite de 0 et de 1. Pour comprendre comment ces *0 et 1* sont traités par l'ordinateur, il faut avoir en tête que les ordinateurs sont construits à partir d'une couche et de multiples niveaux, comme un mille-feuille, dont chacun possède ses propres règles.

Dans ce chapitre, nous nous concentrerons sur les couches de bas niveau, et tenterons de remonter progressivement jusqu'aux couches logicielles.

1.0.1 De quoi sont faits les nombres binaires ?

Les ordinateurs ne comprennent que les nombres binaires. La lettre «A», par exemple, est pour ces derniers une suite de 0 et de 1. Même chose pour une image, une vidéo, une chanson et ainsi de suite. Mais alors comment ces 0 et ces 1 sont-ils *stockés* et manipulés physiquement par les ordinateurs ? De quelle matière sont-ils faits ? Un indice : que mettez-vous dans votre smartphone pour le faire fonctionner : de l'essence ? Du gaz ? De l'énergie solaire ?

De l'électricité !

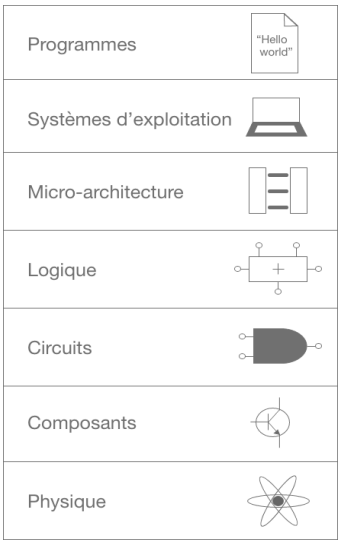


Fig. 1.1 – Les différents niveaux d'abstraction de l'informatique, en partant des électrons, jusqu'aux «programmes»

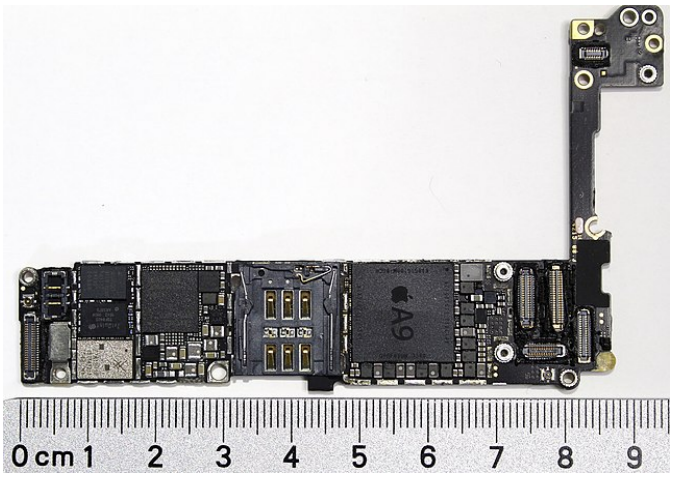


Fig. 1.2 – Vos photos, vos vidéos, vos messages, tout ce que vous consultez sur votre téléphone portable, sont traitées par un processeur similaire au modèle A9 de Apple, commercialisé dans les iPhone SE.



Fig. 1.3 – Vos likes, vos partages, vos vidéos transmises via des applications telles que WhatsApp, Instagram, TikTok, Snapchat, YouTube, sont stockées dans des centres de données aux quatre coins de la planète.

1.0.2 Électricité et nombres binaires

Les nombres binaires, au niveau le plus élémentaire, sont matérialisés par des courants électriques, qui traversent les circuits des ordinateurs. Mais pourquoi avoir choisi des 0 et des 1 comme alphabet ? Quel rapport avec l'électricité ?

En informatique, si nous avons choisi d'utiliser un code binaire, ça n'est pas par hasard. Ce sont les deux signaux les plus élémentaires que l'on puisse transmettre avec l'électricité. Soit le courant passe, soit il ne passe pas. Ouvert ou fermé ; allumé ou éteint ; 1 ou 0.

Le saviez-vous ?

On aurait pu choisir un code possédant plus de deux signaux différents. Par exemple, avec trois signaux, on pourrait coder trois valeurs avec un courant faible, un courant moyen, un courant fort, ou encore mieux : une tension négative, une tension nulle et une tension positive. On appelle cette dernière proposition le ternaire balancé. En fait, cela s'est déjà fait : les soviétiques ont développé en 1958 un ordinateur nommé Setun [2] basé sur ce principe, réputé très fiable et extrêmement peformant dans le développement d'applications dans certains domaines. Mais ce projet, pour des raisons politiques, n'a pas reçu le soutient qu'il aurait mérité. D'autre part, il est plus simple de concevoir des circuits électroniques qui ne doivent traiter que deux valeurs.

2. https://en.wikipedia.org/wiki/Setun

La grande idée derrière la conception des ordinateurs et de leurs circuits électroniques repose sur l'utilisation de sortes d'«interrupteurs automatiques». Ce composant fonctionne donc comme un interrupteur (en laissant ou non passer le courant sur un fil donné), mais de façon automatique : ce n'est pas un humain qui doit venir commuter l'interrupteur, mais l'interrupteur commute automatiquement en fonction de si oui ou non du courant passe sur un *autre* fil du système. Historiquement, on a réalisé que

si l'on disposait d'un tel composant, on pouvait en assembler plusieurs (en fait, plusieurs milliers) et ainsi construire des systèmes à même de manipuler des données représentées par des 0 et des 1. Nous allons voir comment dans les prochaines sections.

Dans les premiers ordinateurs entre les années 1950 et 1960, ce sont les tubes à vide [3] qui ont rempli cette fonction. Mais les tubes à vides étaient gros, consommaient beaucoup d'électricité, et avaient une durée de vie limitée : il fallait souvent les changer, un peu comme de vieilles ampoules à incandescence. En utilisant des tubes à vide, on pouvait certes construire des ordinateurs, mais certainement pas ceux que l'on connaît aujourd'hui.



Fig. 1.4 – Différents modèles de tubes à vide. Photographie de Stefan Riepl, 2008, CC BY-SA.

Il a fallu attendre une invention majeure pour permettre aux ordinateurs de se miniaturiser, de consommer moins, d'être plus efficaces et fiables, pour finalement arriver à ce qu'on a appelé dès la fin des années 1970 des **PC** : des *personal computers*, des ordinateurs personnels. L'invention qui a permis cette évolution est le **transistor**.

1.0.3 Le transistor

Le *transistor* est aujourd'hui la brique de base de construction des systèmes informatiques. Il a été développé dans les années 1940 dans les laboratoires Bell [4], aux Etats-Unis. Ce n'est que vers la fin des années 1950 que l'on commence à construire des ordinateurs commerciaux qui utilisent des transistors plutôt que des tubes à vide. Le transistor est à l'origine d'une révolution dans la taille, la fiabilité, et les performances générales des ordinateurs de l'époque.

Le transistor, comme le tube à vide qu'il remplace, fonctionne comme un interrupteur automatique. Il laissera ou non passer du courant entre deux de ses pattes en fonction de ce qui se passe sur sa troisième. On peut aussi le comparer à un robinet d'eau qui peut être ouvert ou fermé, et qu'on peut ouvrir ou fermer automatiquement sans devoir l'activer manuellement.

3. https://fr.wikipedia.org/wiki/Tube_%C3%A9lectronique
4. https://fr.wikipedia.org/wiki/Laboratoires_Bell

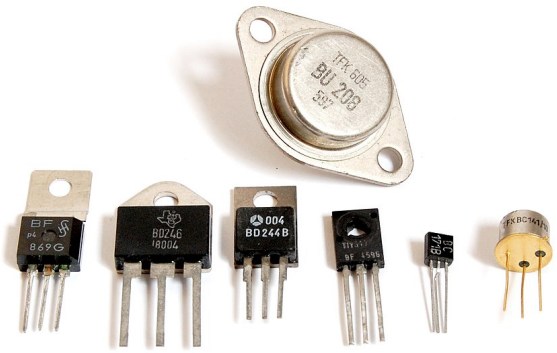


Fig. 1.5 – Différents modèles de transistor. On les reconnaît à leurs trois «pattes» aussi appelées : émetteur, base, collecteur.

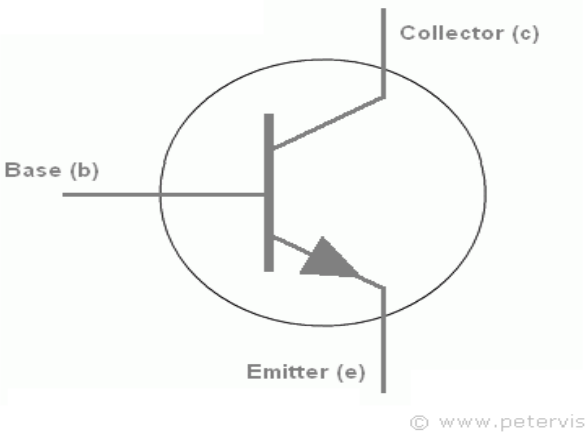


Fig. 1.6 – En appliquant un courant qui va de la base à l'émetteur (en rose pâle), on permet au courant de circuler entre le collecteur et l'émetteur (appelés ainsi parce que l'émetteur *émet* des électrons, et le collecteur les *collecte*). Envoyer du courant dans la base, c'est donc *ouvrir* le transistor ; ne plus en envoyer a, inversement, l'effet de *fermer* le transistor.

En appliquant un courant qui va de la base à l'émetteur (en rose pâle), on permet au courant de circuler entre le collecteur et l'émetteur (appelés ainsi parce que l'émetteur *émet* des électrons, et le collecteur les *collecte*). Envoyer du courant dans la base, c'est donc *ouvrir* le transistor ; ne plus en envoyer a, inversement, l'effet de *fermer* le transistor

De par sa capacité à être ouvert ou fermé, le transistor fonctionne comme une brique fondamentale dans la construction de systèmes informatiques permettant de *transmettre*, stocker et *traiter* des nombres binaires.

Pour aller plus loin

Les transistors sont faits avec des matériaux dit «semi-conducteurs». Voici une vidéo qui explique en détail ce qui se passe dans ces semi-conducteurs et qui permet de faire fonctionner un transistor.

Vidéo 1: https://www.youtube-nocookie.com/embed/33vbFFFn04k

Des transistors presque invisibles

Chercher à se représenter la taille des transistors utilisés dans les microprocesseurs actuels n'a pas d'intérêt tellement ils sont petits. À titre d'exemple, disons simplement que le microprocesseur Apple A9 en possède six milliards.

Zoom sur un transistor

Vidéo 2: https://www.youtube-nocookie.com/embed/Fxv3JoS1uY8

1.0.4 Des transistors aux systèmes logiques

Il reste difficile de concevoir des circuits d'ordinateurs en réfléchissant en termes de transistors. Un transistor seul ne peut représenter ou traiter qu'un bit d'information. Oui ou non, ouvert ou fermé, 1 ou 0.

Dans le chapitre suivant, où nous commençons à voir comment sont conçus les circuits électroniques des ordinateurs, nous parlerons tout d'abord de ***portes logiques***. Ce sont des composants qui sont eux-mêmes constitués de plusieurs transistors. Réfléchir en termes de portes logiques permet de véritablement concevoir les circuits des ordinateurs qui vont manipuler les bits d'informations formant nos données.

1.1 Portes logiques

En informatique, les *systèmes logiques* décrivent comment sont connectés les circuits électroniques des ordinateurs afin de leur permettre, d'une part, d'effectuer des calculs et de traiter des données et, d'autre part, d'utiliser leur mémoire de travail, où sont stockées les données qu'ils traitent.

Même si on a l'impression que les ordinateurs peuvent faire toutes sortes de choses, il y a un ensemble limité d'opérations de base que l'électronique d'une machine peut faire. Parmi ces quelques opérations de base, on trouve l'addition, la soustraction, la multiplication ou la division de nombres. La plupart des tâches que l'ordinateur exécute reposent sur ces quelques opérations (ainsi que sur quelques opérations dites *logiques*, qui vont être explicitées).

C'est assez fascinant de se dire que des tâches a priori non mathématiques, comme corriger l'orthographe ou la grammaire d'un texte automatiquement, sont réalisées avec ces opérations de base.

En parallèle à ce qui leur permet de faire des calculs, les ordinateurs disposent et utilisent de la mémoire. Il y en a au cœur des microprocesseurs, les *registres*, ce qu'on appelle la *mémoire vive* — appelée aussi RAM (*Random-Access Memory*). La mémoire servant au stockage de longue durée comme disques durs et autres SSD n'est pas discutée dans cette section. L'étude des systèmes logiques permet de comprendre les principes derrière la gestion de cette mémoire et de voir comment les ordinateurs peuvent y lire et écrire des données entre deux calculs.

1.1.1 Exemple suivi : addition de deux nombres

On s'intéresse à une des opérations arithmétiques les plus simples : l'**addition**. Comment l'ordinateur additionne-t-il deux nombres ? On va définir le cadre de travail et s'intéresser aux circuits électroniques qui vont être à même de réaliser une addition.

Que se passe-t-il pour l'addition de deux nombres entiers ? On va utiliser leur représentation binaire (avec uniquement des 1 et des 0). Pour faire simple, on va chercher à additionner simplement deux bits, disons A et B, où chacun peut valoir soit 0 soit 1. Posons que la somme $S = A + B$. En énumérant tous les cas de figure, on a :

A	B	S
0	0	0
1	0	1
0	1	1
1	1	10

La dernière ligne est intéressante : on sait que $1 + 1 = 2$, mais en *binaire*, on sait aussi que n'existent que des 0 et des 1, et 2 s'écrit ainsi 10 (voir le chapitre représentation de l'information). Cela veut dire que, pour traiter tous les cas d'une addition de deux *bits*, on a besoin aussi de deux bits de sortie, et qu'un seul ne suffit pas. En explicitant chaque fois le deuxième bit de sortie, notre tableau devient :

A	B	S
0	0	00
1	0	01
0	1	01
1	1	10

La question est de déterminer comment faire calculer les deux bits de la somme S à partir de A et B à un circuit électronique. Pour cela, on a besoin du concept de *portes logiques*. Ces portes logiques sont elles-mêmes constituées de *transistors*, dont on a parlé en début de chapitre.

Dans un premier temps, on détaille les portes logiques et on s'intéresse à la réalisation des circuits logiques.

Ensuite, on regarde comment, fort de cette connaissance des portes logiques, il est possible de concevoir un circuit qui effectuera l'addition en question.

Finalement, on comprendra comment un ordinateur est capable, avec un circuit logique, de stocker le résultat d'un tel calcul afin qu'il soit réutilisable plus tard.

Les opérations arithmétiques et logiques et l'accès à la mémoire ne suffisent pas à constituer un ordinateur complet. C'est dans le chapitre suivant que sera traité la problématique de l'agencement de ces sous-systèmes afin de constituer une machine capable d'exécuter une suite d'instructions, c'est à dire un programme.

1.1.2 Portes logiques

Les circuits électroniques qui composent un ordinateur sont constitués de composants électroniques comme des *résistances*, des condensateurs, des *transistors*, etc., qui déterminent où va passer le courant électrique et sur quelles parties du circuit règnera quelle *tension*.

Quand on parle de portes et de circuits logiques, on simplifie tout cela. On considèrera simplement qu'un segment de circuit électronique où la tension est nulle (0 volt) représente la valeur binaire 0, alors qu'une tension non nulle (par exemple, 3 volts) représente la valeur binaire 1. Ainsi, pour véhiculer deux bits comme A et B dans un circuit, on a besoin de deux «fils».

Les portes logiques sont des composants électroniques (eux-mêmes constitués en général de transistors et résistances) qui ont une ou plusieurs entrées et qui combinent ces entrées pour produire une sortie donnée. La manière dont la sortie est calculée dépend du type de la porte. On se propose à présent d'étudier en détails l'ensemble de ces portes.

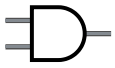
1.1.3 Porte ET

Une de ces portes est la porte **ET**. Elle a deux entrées, qu'on appellera X et Y, et une sortie Z. Z sera 1 si et seulement si aussi bien X que Y valent 1. D'où son nom : il faut que X ***et*** Y soient à 1 pour obtenir un 1 sur la sortie.

En énumérant les quatre possibilités pour les entrées, on peut écrire ce qu'on appelle *table de vérité* pour la porte **ET** :

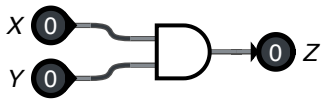
X	Y	Z
0	0	0
1	0	0
0	1	0
1	1	1

On peut dessiner des diagrammes avec des portes logiques. Ce ne sont pas des diagrammes électroniques, ils cachent une partie de la complexité réelle des circuits. Dans un tel diagramme logique, la porte **ET** est représentée ainsi :



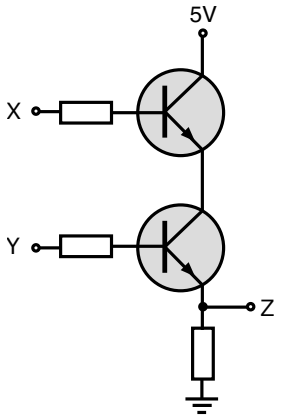
Sur ce schéma logique, les entrées sont à gauche, la sortie à droite et la porte est connectée au milieu. Les circuits sont représentés en noir s'ils véhiculent un «0» et avec une couleur s'ils véhiculent un «1».

Cliquez sur l'entrée X ou l'entrée Y pour changer leurs valeurs et observez le comportement de la sortie Z. Est-ce que cela correspond à la table de vérité ci-dessus ?



Pour aller plus loin

Comment une porte **ET** est-elle elle-même construite ? Cela a déjà été mentionné : avec d'autres composants électroniques plus simples. En simplifiant un peu, on peut considérer qu'une porte **ET** est constituée de deux transistors (avec quelques résistances en plus) :



Ici, les deux transistors sont les composants symbolisés par un cercle. Rappelons qu'ils laissent passer du courant de haut en bas lorsqu'ils détectent un courant sur l'entrée qui vient de la gauche. Ici, comme on a en haut une tension de 5 volts, on aura une tension similaire sur la sortie Z que si à la fois les entrées X et Y sont «actives» — donc lorsque les deux transistors sont «ouverts». Sinon, on aura une tension de 0 volt sur la sortie Z.

1.1.4 Porte OU

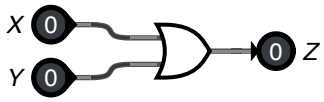
Pour que la sortie de la porte **OU** vaille 1, il suffit que l'une des deux entrées X ou Y vaille 1.

Voici sa table de vérité :

X	Y	Z
0	0	0
1	0	1
0	1	1
1	1	1

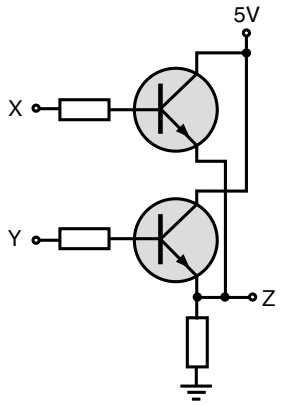
On notera que le **OU** logique est un peu différent du «ou» que l'on utilise en général à l'oral : on voit à la dernière ligne de la table de vérité que la sortie Z vaut également 1 si les deux entrées X et Y valent 1. À l'oral, le «ou» est en général interprété comme *exclusif* : si l'on propose à un ami un bonbon *ou* une glace, on exclut la possibilité qu'il choisisse les deux. Ce n'est pas le cas pour le **OU** logique.

Essayez la porte **OU** :



Pour aller plus loin

Voici comment une porte **OU** peut être construite avec deux transistors :



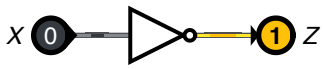
1.1.5 Porte NON

Cette porte est plus simple : elle n'a qu'une entrée, et sa sortie se contente d'inverser la valeur en entrée. On l'appelle d'ailleurs aussi un *inverseur*.

Voici sa table de vérité :

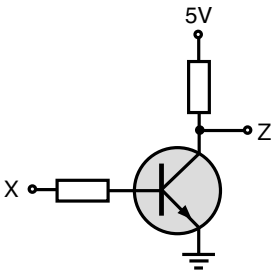
X	Z
0	1
1	0

Essayez l'inverseur :



Pour aller plus loin

Voici comment un inverseur peut être construit avec un transistor :



Ensemble, les portes **ET**, **OU** et **NON** représentent les relations logiques de la *conjonction*, la *disjonction* et la *négation*. Même si on ne les appelle pas ainsi, on utilise tous les jours des relations logiques de conjonction, de disjonction et de négation.

— La **conjonction** est ainsi l'«intersection logique» de deux propositions. Si on dit «je vais à la piscine s'*il fait beau* **ET** *que mes amis m'accompagnent*», on utilise la conjonction.

— Au contraire, si on dit «je vais à la piscine s'*il fait beau* **OU** *que mes amis m'accompagnent*», on utilise la **disjonction**, qui est comme une sorte de «somme logique» de deux propositions (même si, comme noté plus haut, le «ou», dans le langage courant, est généralement exclusif, contrairement au **OU** logique, qui est inclusif).

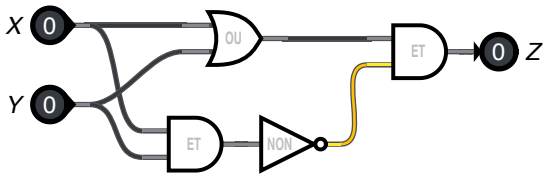
— La **négation** est encore plus évidente, puisque la proposition «je ne vais pas à la piscine» est simplement la négation, ou l'inverse, de la proposition «je vais à la piscine».

Ressource complémentaire

Une application pour s'exercer à l'interprétation des conjonctions, disjonctions et négations logiques : The Boolean Game [5]

1.1.6 Combinaisons de portes

Les portes peuvent être connectées les unes aux autres. Voici par exemple un diagramme logique réalisant la fonction appelée **OU-X**, qui est un «ou exclusif» et dont la sortie Z vaut 1 lorsque soit X, soit Y vaut 1, mais pas les deux en même temps :



Ce circuit contient une porte **OU**, deux portes **ET** et un inverseur, tous interconnectés.

Ce diagramme n'est pas forcément facile à lire — discutons d'abord comment l'interpréter avec papier et crayon pour vérifier s'il effectue bien un **OU-X**.

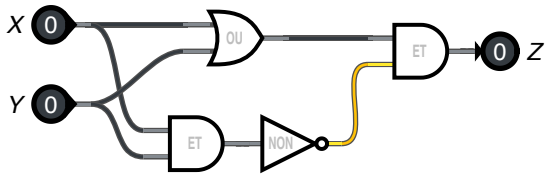
5. https://booleangame.com/

Analyse d'un circuit

Pour analyser un circuit logique comme celui présenté ci-dessus, on cherchera à établir sa table de vérité. En l'occurrence, comme pour les portes précédentes, ce circuit a deux entrées : si chaque entrée peut valoir 1 ou 0, on a en tout, de nouveau, quatre configurations possibles à examiner dans le but de remplir la dernière colonne :

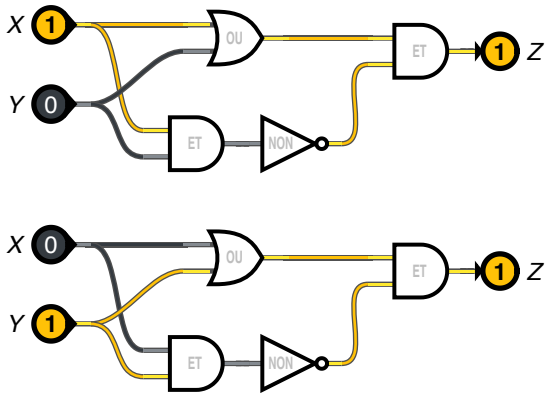
X	Y	Z
0	0	?
1	0	?
0	1	?
1	1	?

Pour remplir chaque ligne, on va changer les entrées selon les valeurs de X et Y et observer l'effet des portes et ainsi voir comment le circuit se comporte. Prenons $X = Y = 0$: c'est le cas représenté par le diagramme fixe ci-dessous. Rappelons qu'un segment noir véhicule un «0», alors qu'un segment coloré véhicule un «1».



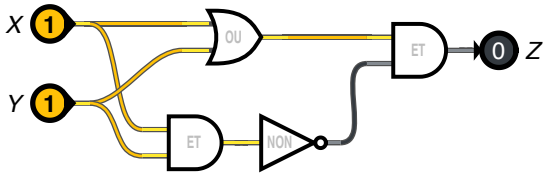
Le résultat intermédiaire des deux portes de gauche sera 0. L'inverseur transforme en 1 la sortie de la porte **ET**, mais la porte finale, qui est aussi une porte **ET**, n'obtient qu'un seul 1 en entrée et donc livre une sortie de 0.

Le cas est différent si l'une des deux entrées vaut 1. Voici deux diagrammes fixes, une fois pour $X = 1, Y = 0$ et une fois pour $Y = 1, X = 0$:



Ici, dans les deux cas, la porte **OU**, en haut, livrera un 1, dont a besoin la porte **ET** finale de droite pour donner une sortie de 1. La porte **ET** du bas, elle, continue de livrer un 0.

Mais dans le cas $X = Y = 1$, représenté ici, la situation est différente :

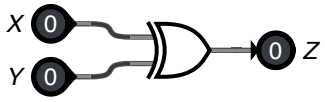


La porte **ET** du bas livre un 1, qui est inversé en 0 avant d'atteindre la porte finale, qui ne peut dès lors elle-même que livrer un 0 comme sortie.

La table de vérité complétée de ce circuit est ainsi :

X	Y	Z
0	0	0
1	0	1
0	1	1
1	1	0

Cette fonction s'appelle «ou exclusif», car pour avoir un 1 de sortie, elle exclut le cas où les deux entrées sont 1 en même temps. Elle est souvent utilisée, au point qu'on la représente en fait dans les diagrammes simplement par le dessin de cette porte, appelée **OU-X**, comme simplification du diagramme ci-dessus :



Exercice 1 – Vérification d'une porte

Vérifiez que la porte **OU-X** se comporte bien comme le circuit ci-dessous réalisé avec des portes **ET**, **OU** et **NON**.

Création d'un circuit

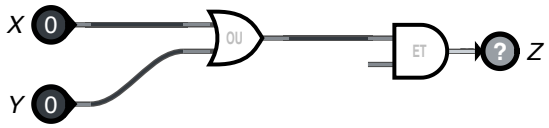
On s'intéresse à présent à la création de ce diagramme réalisant un **OU-X** avec les portes à disposition à partir de sa table de vérité. Plusieurs approches sont possibles, et on constatera que, suivant l'approche, on aurait très bien pu créer un circuit logique différent réalisant la même fonction.

Approche ad hoc

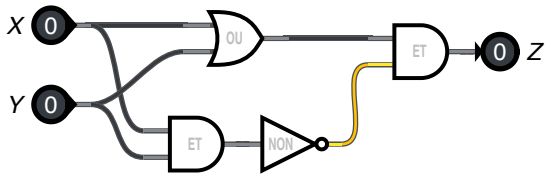
On se dit donc, selon la table de vérité, que la sortie de notre circuit «ou exclusif» doit être 1, donc l'une ou l'autre des entrées X ou Y est à 1, mais pas les deux. On peut ainsi commencer par insérer une porte **OU** dans le diagramme, qui fait une partie du travail. Mais il faut modifier sa sortie, pour ne pas avoir la valeur 1 lorsque les deux entrées sont à 1 : cela contredirait la quatrième ligne de la table de vérité. Comment effectuer cela ? En connectant la sortie de cette porte **OU** à une nouvelle porte **ET** à droite (dont on n'a pas encore déterminé la seconde entrée).

Pourquoi rajouter une porte **ET** ? On utilise ici le fait que connecter une porte **ET** à un signal peut *restreindre* les conditions sous lesquelles la nouvelle sortie Z sera 1 (alors qu'au contraire, on aurait pu *étendre* ces conditions si on avait connecté une nouvelle porte **OU**). Comme si, pour être d'accord de finalement livrer 1 sur la sortie, la porte **ET** voulait la «confirmation» d'un autre signal avant de livrer 1…

À ce moment, on a ce diagramme partiel, qui peut être lu comme : «la sortie Z sera 1 lorsque ces deux conditions sont vraies en même temps : (1) le **OU** de X et Y vaut 1, et (2) quelque chose qui reste ici à définir, qui sera connecté à la seconde entrée de la porte **ET**».



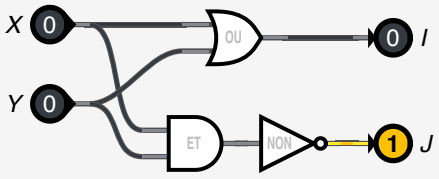
Ce qui reste à définir en complétant avant la porte **ET**, c'est l'exclusion du cas où X et Y valent les deux 1, de manière à ce que la condition (2) puisse être lue comme «X et Y ne sont pas en même temps les deux à 1». Avec une porte **ET** connectée directement aux deux entrées X et Y, on obtient une partie de ceci en créant le signal «X et Y sont les deux à 1». Mais c'est en fait la condition inverse de celle que l'on cherche. Pour l'inverser, on insère un inverseur à la sortie de cette nouvelle porte **ET**, ce qui complète le circuit :



La lecture finale du circuit est donc «la sortie Z sera 1 lorsque ces deux conditions sont vraies en même temps (selon la porte **ET** de droite) : (1) le **OU** de X et Y vaut 1, et (2) X et Y ne sont pas les deux en même temps à 1».

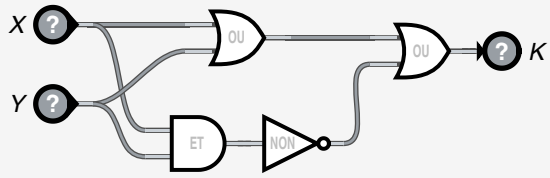
Exercice 2 – Analyse d'un circuit

Ceci est le même circuit que ci-dessus, mais sans la porte **ET** finale. À la place, on a inséré deux sorties intermédiaires, I et J, qui sont les deux signaux qui allaient précédemment à la porte **ET** :



1. Combien de lignes a une table de vérité pour I et J en fonction des deux entrées X et Y ? Écrivez cette table de vérité.
2. Quelle différence y a-t-il entre J et ce qu'on obtient en connectant directement une porte **ET** aux entrées X et Y : quel élément du schéma réalise cette différence ?
3. Dans votre table de vérité, ajoutez une colonne et remplissez-là : elle doit représenter une nouvelle sortie K, qui serait produite si on connectait une porte **OU** en lui donnant I et J comme entrées, comme montré ci-dessous. Le schéma représente ici le circuit dans un état

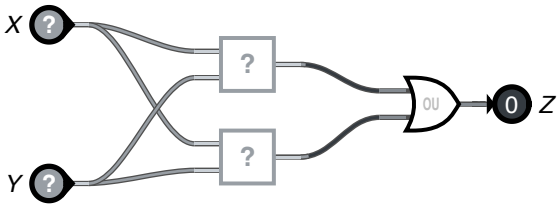
indéterminé, mais les types des portes ont été ajoutés pour vous aider. La sortie K est-elle ici toujours la même que la sortie Z plus haut ? Quelles sont les éventuelles différences ? Finalement, la sortie K a-t-elle un intérêt ?



Approche systématique

Il est parfois difficile d'avoir l'«intuition» nécessaire pour suivre une telle approche ad hoc. Voici donc une autre technique, illustrée avec le même exemple.

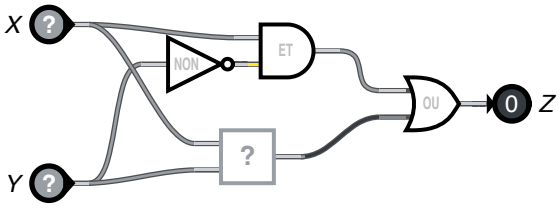
La table de vérité montre qu'il y a deux lignes où la sortie doit valoir 1 : (a) la ligne où $X = 1$ et $Y = 0$, et (b) la ligne où $X = 0$ et $Y = 1$. Si l'on pouvait créer un premier sous-circuit qui livre un 1 lorsque qu'on se trouve dans la circonstance (a) et un autre sous-circuit qui livre un 1 lorsqu'on se trouve dans la circonstance (b), on pourrait ensuite les combiner avec une porte **OU** et ainsi construire notre sortie Z ainsi :



Ici, les deux sous-circuits notés avec « ?» et encadrés donc encore à définir — potentiellement avec plus d'une seule porte. Essayons de les créer.

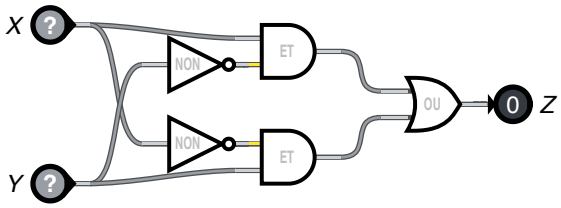
Disons que le sous-circuit du haut correspond à la deuxième ligne de la table de vérité, le cas de figure (a). Pour cette ligne, nous voulons un 1 de sortie lorsque $X = 1$ et $Y = 0$. En lisant littéralement cette dernière phrase, on y détecte un **ET** de deux conditions qui doivent être remplies : $X = 1$ et $Y = 0$. Mais ajouter une porte **ET** directement avec les signaux X et Y ne fera pas l'affaire, parce que cela livrerait un 1 lors que les *deux* entrées X et Y sont à 1. La solution ici, c'est d'*inverser* Y avant l'entrée dans la porte **ET** — ce qui donne bel et bien la condition (a).

On avance ainsi à ceci :



Pour la condition (b), qui correspond à la troisième ligne de la table de vérité, un raisonnement similaire s'applique. À la place d'inverser Y, on inversera cette fois X afin d'obtenir, à la sortie de la nouvelle porte **ET** du bas, un signal qui vaut 1 lorsque $X = 0$ et $Y = 1$.

Voici le circuit final ainsi réalisé :



(Ce schéma ne peut être simulé que dans l'indice de l'exercice suivant.)

Ce que cette approche systématique apprend, c'est qu'un circuit peut toujours être pensé comme un **OU** de toutes les conditions sous lesquelles la sortie doit être à 1. Ces conditions sont elles-mêmes réalisables avec les entrées du circuit avec des portes **ET** et des inverseurs directement connectés aux entrées.

On fait également les constats suivants :

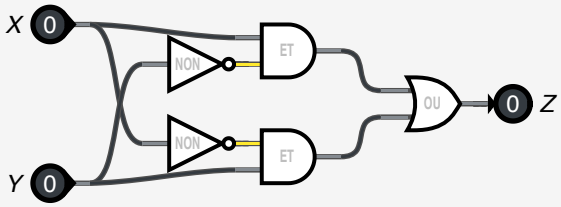
— plusieurs circuits logiques différents peuvent réaliser la même fonction de sortie,

— l'approche systématique décrite ici ne livre pas forcément le circuit le plus compact : on a obtenu un circuit avec cinq portes pour réaliser un **OU-X** alors que l'approche ad hoc a conduit à la construction d'un circuit à quatre portes.

Exercice 3 – Analyse d'un circuit

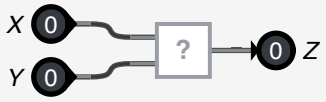
En annotant le schéma logique avec les quatre cas de figure possibles pour les entrées X et Y, faites l'analyse du circuit **OU-X** ci-dessus construit avec l'approche systématique et montrez que la table de vérité ainsi reconstituée est la même que celle de la porte **OU-X**.

Indice



Exercice 4 – Porte cachée

Quelle est la porte cachée de ce circuit ?

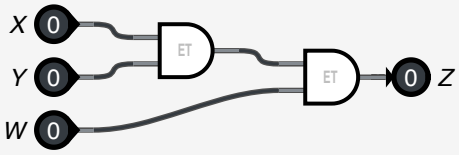


Corrigé

C'est une porte **OU**.

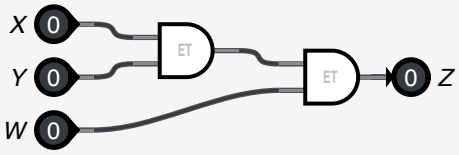
Exercice 5 – Circuit défectueux

Analysez ce circuit. De quel type de portes est-il constitué ? Fonctionne-t-il correctement ? Déterminez ce qui pose problème. Dites ce que fait ce circuit une fois corrigé et écrivez sa table de vérité.



Indice

Voici le circuit corrigé (il a la même apparence que le circuit de la question, mais toutes les portes fonctionnent ici correctement).



Corrigé

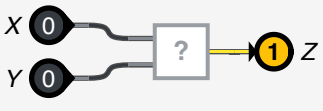
Ce circuit est constitué de deux portes **ET**. Mais la porte **ET** de droite semble poser problème, parce qu'elle se comporte comme une porte **OU** ! Le circuit montré dans l'indice se comporte correctement.

Ce circuit, une fois corrigé, implémente en fait un **ET** à trois entrée X, Y et W, où la sortie Z ne vaut 1 que si les trois entrées valent 1. Sa table de vérité, à huit lignes dues aux trois entrées, est ainsi la suivante :

X	Y	W	Z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	0
1	1	0	0
1	1	1	1

Exercice 6 – Conception d'un circuit

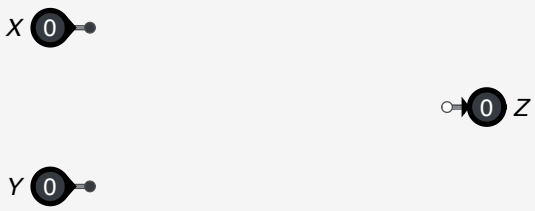
Écrivez la table de vérité de ce circuit, dont une partie est masquée :



Corrigé

X	Y	Z
0	0	1
0	1	0
1	0	1
1	1	1

Réalisez ensuite un circuit logique avec les mêmes deux entrées X et Y et la même sortie Z qui implémente cette table de vérité. On peut utiliser des portes **ET** et **OU** et des inverseurs. Glissez les portes depuis la gauche pour en ajouter, et glissez entre les connecteurs rond pour les connecter.



Indice 1

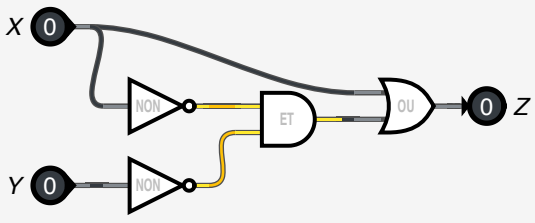
On peut lire cette fonction comme «Z vaut 1 lorsque X et Y sont les deux à 0 (la première ligne de la table de vérité) ou lorsque X est à 1 (les deux dernières lignes)».

Indice 2

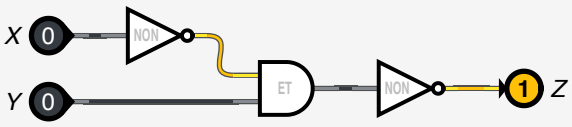
Z est donc le **OU** de X et du **ET** de l'inverse de X et de Y.

Corrigé

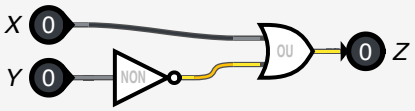
Il y plusieurs solutions possibles. Celle qui correspond aux indices est la suivante :



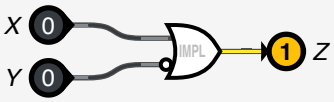
Une autre solution est la suivante, où on se dit qu'on construit d'abord une partie du circuit qui identifie le cas où $X = 0$ et $Y = 1$, et on l'inverse pour correspondre à la table de vérité.



Voici un circuit plus simple, qui fait la même chose mais qui est plus difficile à concevoir d'emblée :



En fait, il existe même une porte spéciale qui réalise exactement la fonction correspondant à la table de vérité, la porte **IMPLIQUE** :



1.2 Additionneur

On a découvert quelques *portes logiques* ainsi que la possibilité de les connecter pour en faire des circuits logiques plus complexes. Ces portes logiques vont maintenant permettre de réaliser l'additionneur annoncé en début de chapitre précédent.

On rappelle qu'on a deux *bits* de sortie à calculer pour la sortie $S = A + B$. S est donc constitué de S_0, le bit des unités, et de S_1, le bit représentant la valeur décimale 2. On rappelle ici la *table de vérité* pour S_0, tirée directement du chapitre précédent :

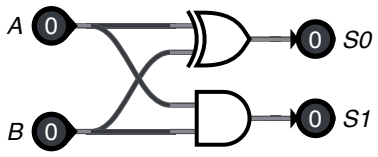
A	B	S_0
0	0	0
1	0	1
0	1	1
1	1	0

En comparant cette table de vérité avec celles des portes logiques, on se rend compte que S_0 n'est autre qu'un **OU-X** (OU exclusif) de A et B.

La table de vérité pour S_1 est :

A	B	S_1
0	0	0
1	0	0
0	1	0
1	1	1

Et on constate que S_1 n'est autre qu'un **ET** logique de A et B. On peut dessiner l'additionneur de deux bits ainsi :



Exercice 1

Vérifiez que ce circuit livre bien les bonnes valeurs de sortie qui correspondent aux tables de vérité ci-dessous. Combien de combinaisons différentes devrez-vous tester ?

Corrigé

Le circuit fonctionne correctement. Il faut tester les quatre combinaisons qui apparaissent dans les tables de vérité.

1.2.1 Additionneur complet

Le circuit précédent est particulièrement intéressant, car il montre qu'il est possible d'utiliser des opérateurs logiques pour réaliser l'opération arithmétique de l'addition. L'additionneur est limité : en fait, on l'appelle un *demi-additionneur*. Il n'est capable d'additionner que deux bits — c'est très limité. En fait, il serait intéressant d'avoir un additionneur de *trois* bits. Pourquoi ? À cause de la manière dont on pose les additions en colonnes.

Lorsqu'on additionne deux nombres à plusieurs chiffres, que ce soit en base 10 ou en base 2, on commence par la colonne de droite, les unités. On connait le concept de *retenue* : en base 10, si l'addition des unités dépasse 9, on retient 1 dans la colonne des dizaines. En base 2, de façon similaire, si l'addition des unités dépasse… 1, on retient 1 dans la colonne suivante à gauche. C'est ce qu'on a fait avec le demi-additionneur : on peut considérer que la sortie S_0 représente la colonne des unités dans la somme, et la sortie S_1 représente la retenue à prendre en compte dans la colonne suivante.

C'est ici que ça se complique : pour additionner les chiffres de la deuxième colonne, on doit potentiellement additionner *trois* chiffres, et plus seulement deux. On a donc, en entrée, les deux bits A et B qui viennent des nombres à additionner, et aussi potentiellement cette retenue qui vient de la colonne des unités, qu'on appellera C_{in} (pour_carry_, «retenue» en anglais). Ceci est vrai en base 2 comme en base 10. Il faut donc un additionneur plus puissant, à trois entrées, pour prendre en compte cette retenue. Il s'appelle_additionneur complet_et livrera deux sorties : le bit de somme, appelé simplement S, et la retenue à reporter pour la colonne suivante, appelée C_{out}.

Exercice 2 – Bases de l'additionneur complet

- Déterminez combien de combinaisons différentes sont possibles pour trois signaux d'entrée A, B et C_{in} qui chacun peuvent valoir soit 1 soit 0.
- Listez toutes ces combinaisons.
- Pour chaque combinaison, déterminez la valeur binaire qui est la somme des trois signaux d'entrée.
- Finalement, avec les informations ainsi obtenues, complétez la table de vérité d'un additionneur complet qui a deux sorties S et C_{out}

Corrigé

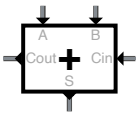
Il y a $2 \cdot 2 \cdot 2 = 2^3 = 8$ combinaisons différentes. Avec la notation $A + B + C =$ valeur en décimal $=$ valeur en binaire, les voici :

- $0 + 0 + 0 = 0_{(10)} = 00_{(2)}$
- $0 + 0 + 1 = 1_{(10)} = 01_{(2)}$
- $0 + 1 + 0 = 1_{(10)} = 01_{(2)}$
- $0 + 1 + 1 = 2_{(10)} = 10_{(2)}$
- $1 + 0 + 0 = 1_{(10)} = 01_{(2)}$
- $1 + 0 + 1 = 2_{(10)} = 10_{(2)}$
- $1 + 1 + 0 = 2_{(10)} = 10_{(2)}$
- $1 + 1 + 1 = 3_{(10)} = 11_{(2)}$

La table de vérité est ainsi :

A	B	C_{in}	C_{out}	S
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

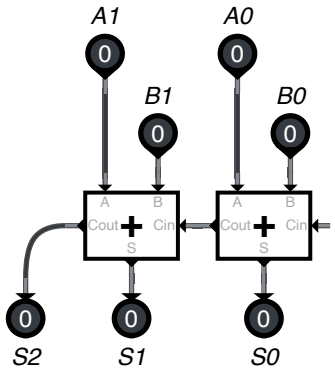
En faisant pour l'instant abstraction des détails d'un additionneur complet, on peut se dire qu'on le dessine simplement ainsi :



1.2.2 Chaînage d'additionneurs

La flexibilité de ce composant fait qu'on peut maintenant facilement l'utiliser pour construire un circuit qui additionne deux nombres A et B à 2 bits chacun (donc de $0 + 0 = 0$ à $3 + 3 = 6$).

Si A est formé de deux bits A_0 et A_1 et que B est formé des deux bits B_0 et B_1 et avec une sortie S sur trois bits S_0, S_1 et S_2, on a :



L'additionneur de droite, comme précédemment, additionne les deux bits des unités : A_0 et B_0. Son entrée C_{in}, qui représente l'éventuel troisième chiffre à additionner issu d'une retenue, n'est pas connectée et est toujours 0, vu qu'il n'y a aucune colonne précédente dans l'addition qui aurait pu en livrer une. Il livre comme première sortie S_0, le chiffre des unités, et sa seconde sortie C_{out} est la retenue à utiliser pour l'addition des chiffres suivants. C'est pourquoi elle est connectée à l'entrée de la retenue du second additionneur C_{in}, qui va lui ajouter également les deux bits de la colonne suivante, A_1 et B_1. Les sorties du second additionneur livrent le deuxième bit S_1 de la valeur de sortie, ainsi que la retenue pour la troisième colonne. Comme il n'y a plus de bits d'entrée pour la troisième colonne, cette retenue peut directement être considérée comme le troisième bit de sortie S_2.

Exercice 3 – Limite de cet additionneur à 2 bits

Avec l'additionneur ci-dessus, est-il possible d'obtenir des 1 sur toutes les sorties, donc d'avoir $S_2 = S_1 = S_0 = 1$?

Indice

Déterminez quel est le nombre décimal qui serait représenté par $S_2 = S_1 = S_0 = 1 : 111_{(2)} = ???_{(10)}$ Ensuite, déterminez les nombres les plus grands représentables sur les deux fois 2 bits d'entrée et tirez-en une conclusion.

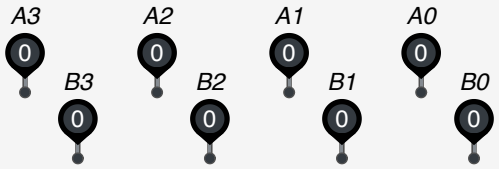
Corrigé

La configuration $S_2 = S_1 = S_0 = 1$ représente le nombre décimal 7. Ce serait le résultat de l'addition. Il faudrait ainsi chercher une configuration des bits d'entrées qui, une fois additionnés, donnent 7. Mais ceci n'est pas possible, car sur chacune des entrées (A_1, A_0) et (B_1, B_0), la plus grande valeur représentable est $11_{(2)}$, autrement dit $3_{(10)}$ — et c'est impossible d'atteindre 7 en évaluant au maximum $3 + 3$.

Exercice 4 – Additionneur de demi-octets

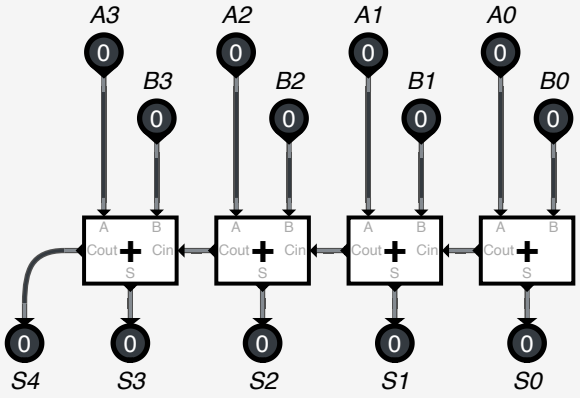
En connectant des additionneurs complets, réalisez un circuit qui additionne deux nombres A et B de quatre bits, numérotés A_0 à A_3 et B_0 à B_3, respectivement. Combien de bits de sortie doit-il y avoir pour traiter toutes les valeurs possibles ?

Les entrées sont déjà disposées. Glissez autant d'additionneurs et de bits de sortie que nécessaire et connectez les composants du circuit.



Corrigé

On a besoin de cinq bits de sortie. Le schéma, représenté horizontalement et de droite à gauche pour être proche de la représentation selon laquelle les additions se résolvent en colonne, est :



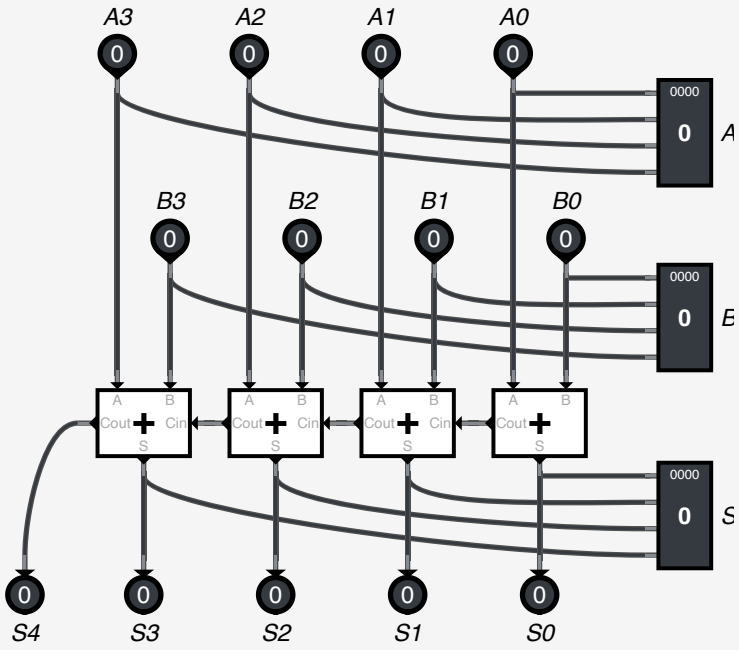
Cet exercice démontre l'opportunité de penser en termes modulaires, ce qui revient souvent en informatique. Ici, on a réalisé qu'un additionneur complet résout un sous-problème bien défini d'une addition générale d'un nombre à n bits, et qu'une fois qu'on a créé un tel additionneur, il suffit d'en connecter plusieurs les uns derrière les autres de manière structurée pour additionner des nombres plus grands.

Exercice 5 – Dépassement de capacité

Le schéma ci-dessous montre le même additionneur de demi-octets de l'exercice précédent, mais, de plus, la valeur en base 10 de ses 4 bits d'entrée pour A et pour B est affichée avec un module d'affichage spécial à droite. La même chose est faite pour représenter la valeur $S = A + B$ (mais seulement sur les quatre premiers bits de S). Actuellement, le circuit effectue le calcul $0 + 0 = 0$.

Réglez les entrées du circuit de manière à lui faire effectuer les additions suivantes, et vérifiez le résultat. Dans quelles circonstances est-il correct et pourquoi est-il de temps en temps incorrect ? Comment, en regard de ceci, interpréter le bit de sortie S_4, qui est la retenue de l'additionneur de gauche ?

1. $1 + 0$
2. $3 + 1$
3. $3 + 3$
4. $10 + 5$
5. $14 + 1$
6. $14 + 2$
7. $15 + 15$



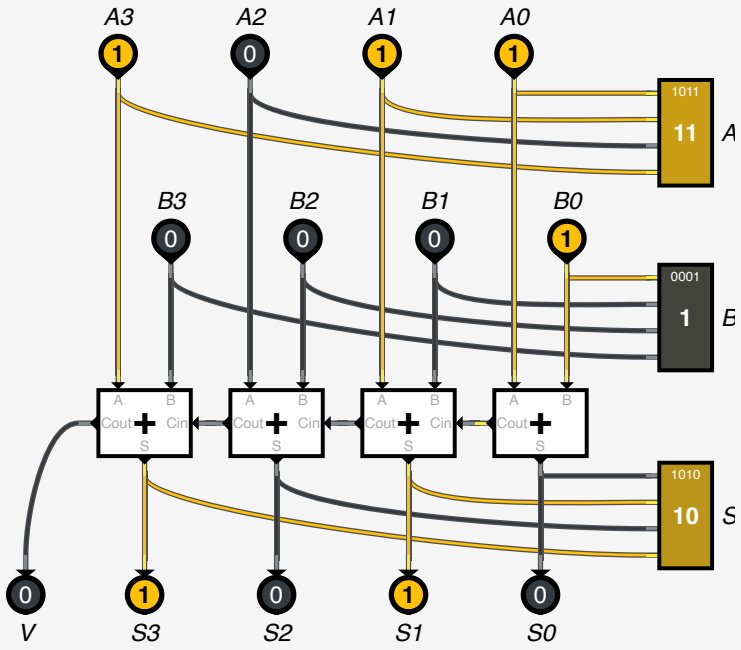
Corrigé

Dès que la somme dépasse 15, elle n'est plus représentable sur les 4 bits qui sont affichés sur la sortie. La plupart des ordinateurs et smartphones actuels représentent les nombres non pas sur 4 bits, mais sur 64. Mais même avec 64 bits, il y a un nombre maximal que l'on peut représenter (en l'occurrence, $2^{64} - 1 = 18\,446\,744\,073\,709\,551\,615$.) La retenue du dernier additionneur indique si le résultat est valable : il vaut 1 lorsque le résultat de l'addition n'est pas correctement représenté avec les 4 (ou 64) bits de sortie. Dans les processeurs, il porte souvent simplement le nom de C (pour *carry*, retenue). On utilisera dorénavant aussi ce nom.

Exercice 6 – Circuit défectueux

L'additionneur de demi-octets ci-dessous a été endommagé et ne fonctionne plus correctement. Par exemple, lorsqu'on lui demande d'effectuer le calcul $11 + 1$, il livre comme réponse 8.

Déterminez quel composant est défectueux dans ce circuit et comment il faudrait le réparer. Vous pouvez changer les entrées pour vérifier ce qui ne fonctionne pas.



Corrigé

La retenue sortant du deuxième additionneur depuis la droite est bloquée à 0 à la place de correctement changer de valeur suivant ses entrées.

Exercice 7 – Design d'un additionneur complet

Note : *exercice difficile et actuellement peu guidé ici; prochainement complété par davantage d'indications.*

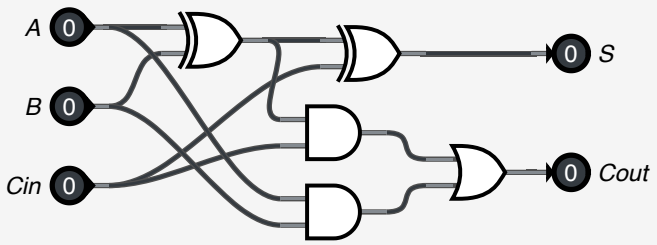
En s'aidant de la table de vérité d'un seul additionneur complet, créer un circuit logique qui calcule ses sorties S et C_{out} en fonction des entrées A, B et C_{in}.

Indice

— La sortie S doit être 1 soit lorsque les trois entrées valent 1, soit lorsqu'une seule des trois entrée vaut 1.

— La sortie C_{out}, qui est la retenue, doit être 1 lorsque deux ou trois des trois entrées valent 1.

Corrigé



1.3 ALU

Dans cette section, nous continuons notre exploration de comment les portes logiques, selon leur assemblage, fournissent les fonctionnalités de base des ordinateurs. En particulier, nous nous intéressons à comment faire effectuer plusieurs opérations à un ordinateur via ce qui s'appelle une **unité arithmétique et logique**, ou simplement une ALU.

Dans la section précédente, nous avons vu comment créer, via un assemblage de portes logiques, un circuit qui réalise l'addition de deux nombres de 4 bits. Ce circuit était fixe : avec les deux nombres d'entrées, il réalisait toujours une addition et ne servait ainsi qu'à ça.

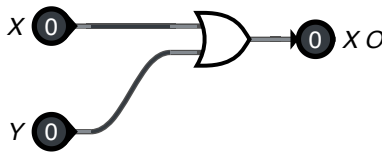
Les ordinateurs ont la propriété d'être programmables : ils savent effectuer plusieurs opérations, et c'est la manière dont ils sont programmés qui va déterminer l'opération qu'ils effectuent. C'est aussi vrai pour des machines plus simples ; une calculatrice de poche, par exemple, pourra effectuer au moins les quatre opérations de base : addition, soustraction, multiplication et division.

Le composant qui nous permettra de sélectionner une opération ou une autre s'appelle «unité arithmétique et logique», communément appelé simplement «ALU» (de l'anglais *arithmetic logic unit*). Avant d'inspecter une ALU, nous avons besoin de comprendre comment on peut sélectionner une opération ou une autre avec des circuits logiques.

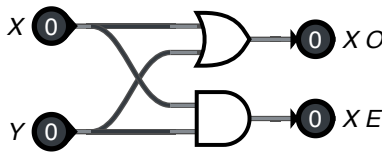
1.3.1 Sélection de l'opération

Comment créer un circuit qui permet de sélectionner une opération à faire, et comment indiquer l'opération à sélectionner ? Essayons d'abord de créer un circuit à deux entrées qui va calculer soit le **OU** soit le **ET** de ces deux entrées.

Nous savons faire un circuit simple qui calcule le **OU** de deux entrées X et Y, avec une seule porte logique :

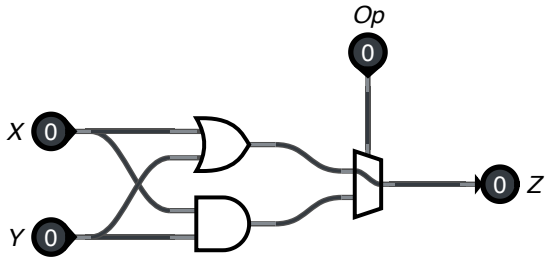


Nous pouvons sans autre y ajouter une porte **ET** pour calculer une autre sortie en parallèle, à partir des mêmes entrées X et Y :

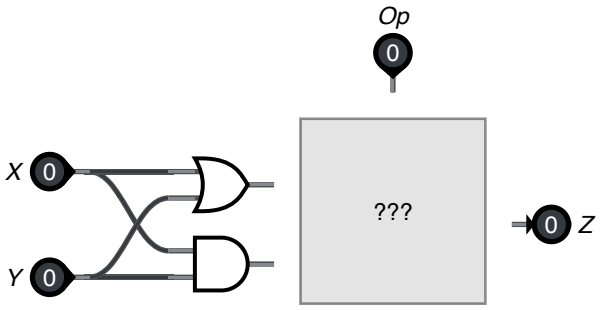


L'idée est maintenant de combiner ces sorties intermédiaires pour n'en avoir plus qu'une, qui sera soit le **OU**, soit le **ET**. Mais comment faire pour indiquer si l'on désire le **OU** ou le **ET** ? Nous pouvons rajouter une entrée pour choisir cette opération. Appelons-la Op, pour «opération». Choisissons une convention : lorsque Op vaut 0, nous souhaitons effectuer l'opération **OU**, et lorsque Op vaut 1, nous souhaitons effectuer l'opération **ET**.

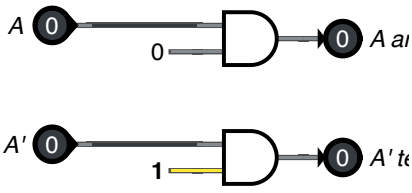
Ce que nous cherchons à créer est conceptuellement ce que fait ce composant tout prêt ci-dessous. Il s'appelle *multiplexeur* et connecte à sa sortie soit le signal du haut, soit le signal du bas, en fonction de la valeur de Op. Essayez de changer Op :



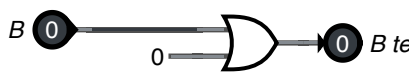
Mais comment construire ce multiplexeur avec les portes logiques que nous connaissons ? Sans le multiplexeur, voici ce que nous devons compléter :



Pour savoir quelles portes utiliser pour recréer le multiplexeur, nous avons besoin de nous rappeler ceci. Lorsqu'un signal, disons A, passe à travers une porte logique **ET** dont la seconde entrée vaut 0 (ici affichée sans qu'on puisse la changer en cliquant dessus), la sortie de cette porte-là sera toujours identique à 0. Cela nous permet ainsi d'*annuler* le signal A — de le mettre à zéro. De manière similaire, si cette seconde entrée vaut 1, le signal A passera tel quel. On peut ainsi voir la porte **ET** comme un annulateur de signal. Vérifiez ceci ici :



Ensuite, lorsqu'un signal, disons B cette fois, passe à travers une porte logique **OU** dont la seconde entrée est annulée et vaut 0, la sortie de cette porte sera toujours identique à B. Vérifiez ceci ici :



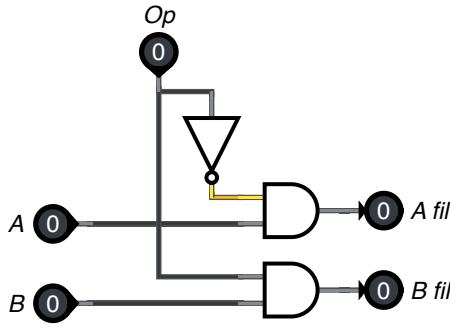
La porte **OU** peut ainsi servir à combiner deux signaux, pour autant que l'un soit annulé.

Avec tout ceci, on peut ainsi construire un multiplexeur (un sélecteur de signal). Supposons qu'on ait les deux signaux A et B : nous pouvons construire un circuit qui combine soit A tel quel et B annulé, soit A annulé et B tel quel. Cela nous aidera pour notre projet initial ! Il faut cependant s'assurer que A soit chaque fois annulé quand B passe tel quel, et inversement. Ceci peut se faire en réutilisant l'idée d'une entrée de contrôle Op ainsi. Nous avons ainsi deux cas :

— Lorsque $Op = 0$, on va sélectionner A et annuler B. On va donc faire passer A à travers une porte **ET** à laquelle on donne 1 à l'autre entrée, et faire passer B à travers une porte **ET** à laquelle on donne 0 à la seconde entrée.

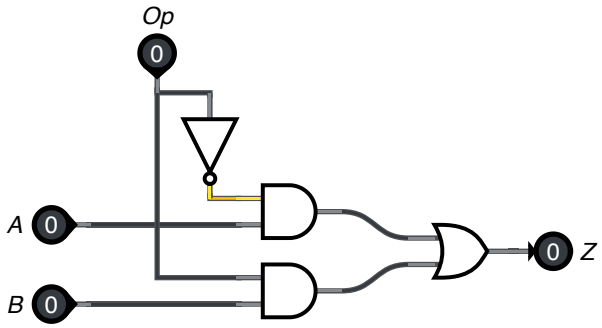
— Lorsque $Op = 1$, on va faire exactement l'inverse : sélectionner B et annuler A. On donnera donc un 0 à la porte **ET** qui filtre A, et 1 à la porte **ET** qui filtre B.

En relisant ces lignes, on voit que ce qu'on donne à la seconde entrée de la porte qui filtre B est toujours la même chose que Op, et que ce qu'on donne à la seconde entrée de la porte qui filtre A est toujours l'inverse de Op. On peut donc construire ce circuit avec un inverseur en plus :

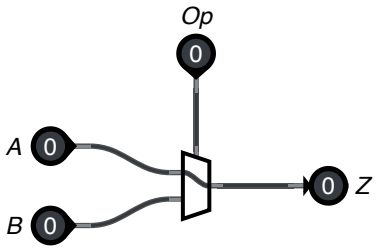


Essayez ce circuit. Quand $Op = 0$, B filtré sera toujours 0 mais A passera, et inversement.

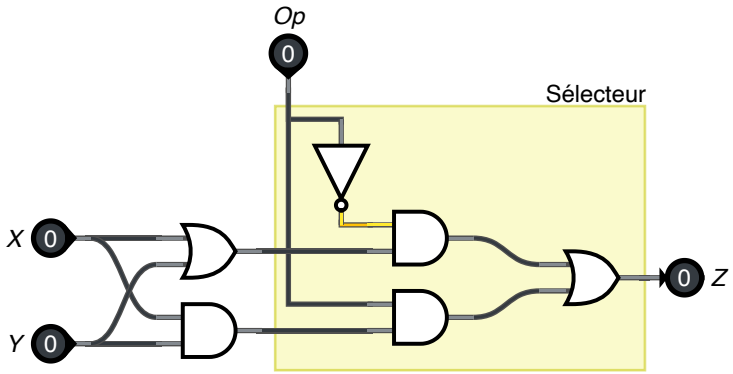
Pour recombiner ces sorties filtrées, il ne nous reste plus qu'à les connecter par une porte **OU** :



Essayez ce circuit pour confirmer qu'il agit comme un sélecteur : lorsque $Op = 0$, on aura sur la sortie $Z = A$, et lorsque $Op = 1$, on aura $Z = B$. Comparez-le avec le multiplexeur tout seul ci-dessous : est-ce que notre circuit fait bien la même chose ?



Ceci nous permet de compléter le circuit lacunaire de début de chapitre pour sélectionner avec le même mécanisme soit le **OU** soit le **ET** de nos deux entrées X et Y. On ajoute les portes de notre sélecteur en connectant à l'entrée A le signal représentant X **OU** Y, et à l'entrée B le signal représentant X **ET** Y :



Exercice 1 – Test du sélecteur OU/ET

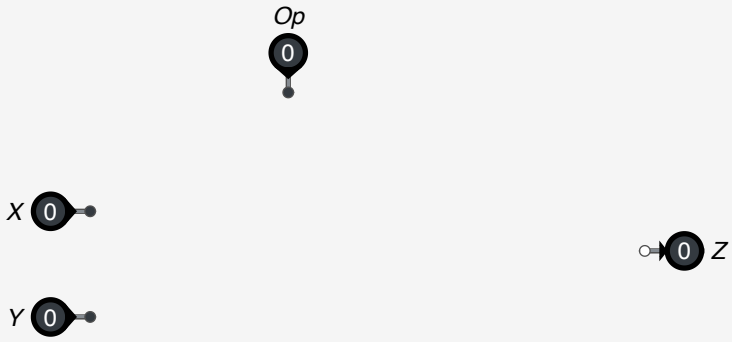
Testez le circuit ci-dessus. Établissez la table de vérité de Z en fonction de X, Y et Op. À l'aide de la table de vérité, montrez que, lorsque $Op = 0$, Z représente bien X **OU** Y, et que, lorsque $Op = 1$, Z représente bien X **ET** Y.

Nous avons ici construit un circuit qui, grâce à un bit de contrôle Op, sélectionne une opération ou une autre à appliquer à ses deux bits d'entrées X et Y.

Exercice 2 – Construction d'un sélecteur

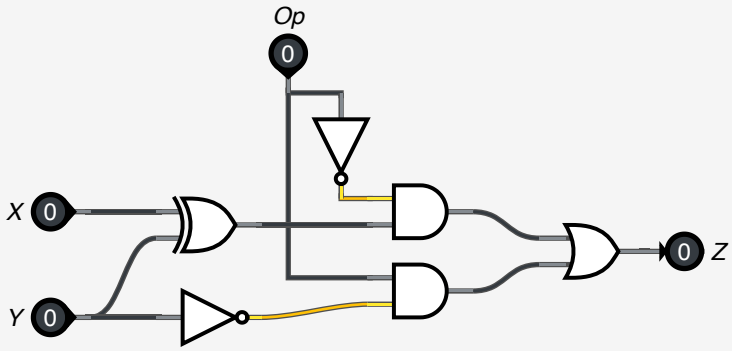
En réutilisant les principes appliqués ci-dessus, construisez un circuit à deux bits d'entrées X et Y et un bit de contrôle Op qui donnera sur sa sortie Z :

— Le **OU** exclusif de X et Y, lorsque $Op = 0$;
— L'inverse du bit Y, lorsque $Op = 1$.



Corrigé

Voici un circuit qui réutilise le sélecteur de signal et qui fournit à ce sélecteur les deux nouvelles entrées décrites, à savoir, en haut, le **OU** exclusif de X et Y tel que fourni par une porte **OU-X**, et en bas, Y une fois inversé par une porte **NON** :

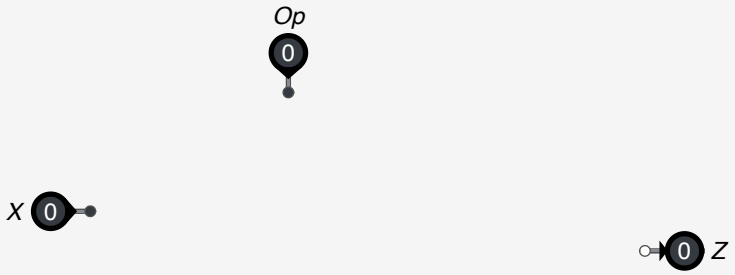


Exercice 3 – Inverseur conditionnel

En réutilisant les principes appliqués ci-dessus, construisez un circuit à une entrée X avec un bit de contrôle Op qui donnera sur sa sortie Z :

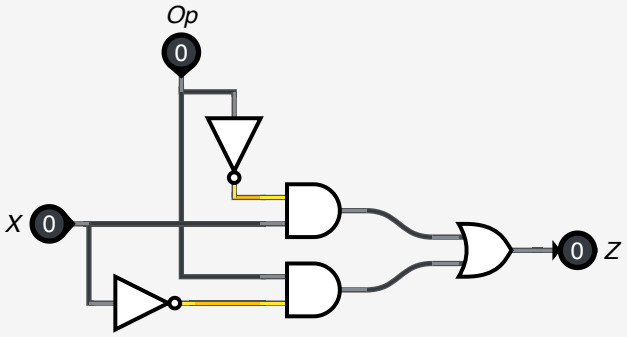
— X tel quel, lorsque $Op = 0$;
— X inversé, lorsque $Op = 1$.

Écrivez la table de vérité de ce circuit. Correspond-elle par hasard à une porte déjà connue ? Serait-ce dès lors possible de simplifier votre circuit ?

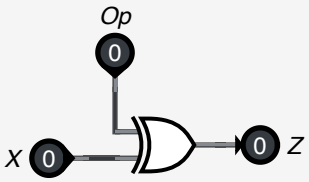


Corrigé

Voici une proposition qui réutilise le sélecteur de signal et qui fournit à ce sélecteur X en haut et X inversé en bas :



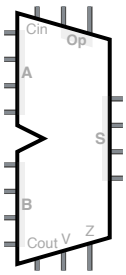
La table de vérité est identique à celle d'une porte **OU-X**. On peut donc simplement remplacer tout le circuit par cette unique porte :



1.3.2 Une ALU à 4 bits

Une unité arithmétique et logique, ou ALU, est un circuit qui ressemble dans ses principes de base à ce que nous venons de faire. L'ALU réalise plusieurs opérations et permet de sélectionner, via un ou plusieurs bits de contrôle, l'opération qui est réalisée. Les opérations proposées sont, comme le nom de l'ALU indique, des opérations arithmétiques (typiquement, l'addition et la soustraction) et des opérations logiques (par exemple, un **ET** et un **OU** logiques).

Nous présentons ici une ALU simple à 4 bits :



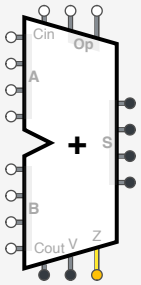
Cette ALU sait effectuer l'addition ou la soustraction de deux nombres entiers représentés sur 4 bits. Elle a ainsi 8 bits d'entrée pour les données et 4 bits de sorties, à gauche et à droite. En plus de l'addition et de la soustraction, elle sait aussi faire les opérations logiques **ET** et **OU** — en tout donc, quatre opérations. Pour sélectionner l'une des quatre opérations, on ne peut plus se contenter d'un seul bit de contrôle, mais nous allons en utiliser deux pour avoir quatre combinaisons possibles. Ce sont les deux entrées supérieures gauches de l'ALU (on ignore ici l'entrée C_{in}).

La convention utilisée pour la sélection de l'opération est la suivante :

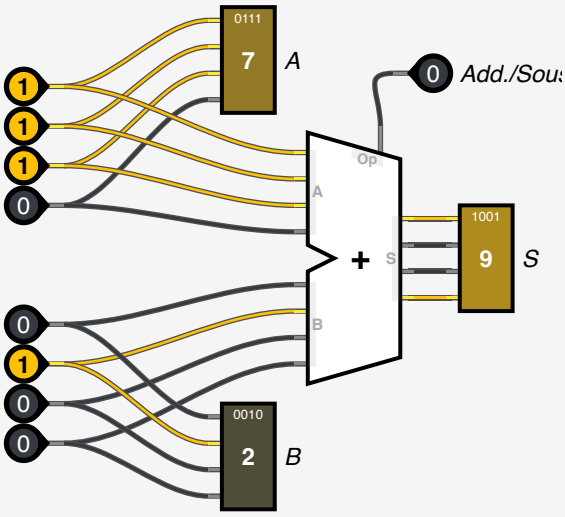
Op	Opération effectuée
00	Addition
01	Soustraction
10	**OU**
11	**ET**

Exercice 4 – Test de l'ALU

Connectez cette ALU à 8 entrées et à 4 sorties de manière à lui faire effectuer l'opération $7 + 2 = 9$. Connectez les 4 bits des entrées et de la sortie à des afficheurs de demi-octet pour vérifier leur fonctionnement. Connectez ensuite une entrée pour le bit de contrôle qui permettra d'effectuer la soustraction avec les mêmes données d'entrée, donc $7 - 2 = 5$.



Corrigé



L'ALU a trois sorties en plus, en bas du composant :

— la sortie C_{out} (pour *carry out*) vaut 1 lors d'un dépassement de capacité (si le résultat de l'opération arithmétique représenté sur la sortie n'est pas valable parce qu'il vaudrait davantage de bits pour le représenter ; par exemple, le résultat de $8 + 8 = 16$ n'est pas représentable sur 4 bits, qui suffisent à représenter les valeurs entières jusqu'à 15 seulement) ;

— la sortie V (pour *oVerflow*) vaut 1 lors d'un dépassement de capacité si on considère les entrées et les sorties comme des nombres signés. Nous ne faisons pas cela ici et ignorons cette sortie ;

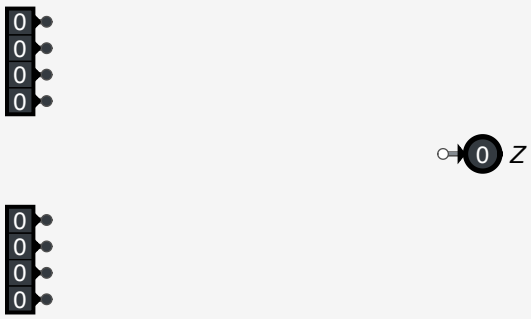
— finalement, la sortie Z (pour *Zero*) vaut 1 lorsque tous les bits de sortie valent 0.

Exercice 5 – Une ALU comme comparateur

En programmation, c'est fréquent de tester, par exemple dans une condition avec un `if`, si deux valeurs sont égales. Par exemple, ce fragment de code affichera «Ces valeurs sont égales !» uniquement si les deux nombres entiers donnés lors de l'exécution du code sont les mêmes :

```
A = int(input("Quel est le premier nombre? "))
B = int(input("Quel est le second nombre? "))
if A == B:
    print("Ces valeurs sont égales!")
```

Ce qui nous intéresse spécialement, c'est la comparaison à la ligne 3. Cette comparaison peut être réalisée avec une ALU. Pour cet exercice, créez un circuit avec une ALU qui compare deux nombres de quatre bits et indique sur la sortie Z un 1 si les deux nombres sont égaux et un 0 s'ils sont différents.

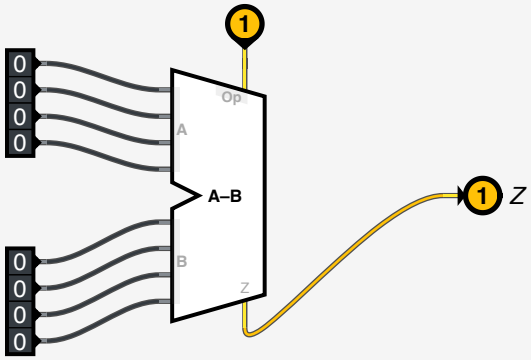


Indice

Deux nombres A et B sont égaux si leur différence est nulle — donc si tous les bits de sortie de la soustraction $A - B$ valent 0.

Corrigé avec ALU — approche arithmétique

On connecte les 8 entrées, on règle l'opération de l'ALU sur soustraction et on utilise la sortie de l'ALU qui indique si tous les bits de sortie sont à zéro. En effet, cela ne se produit que lorsque la différence entre les deux nombres d'entrée est 0 — c'est-à-dire, s'ils sont égaux. On constate qu'on peut ignorer les 4 bits de sorties ici !



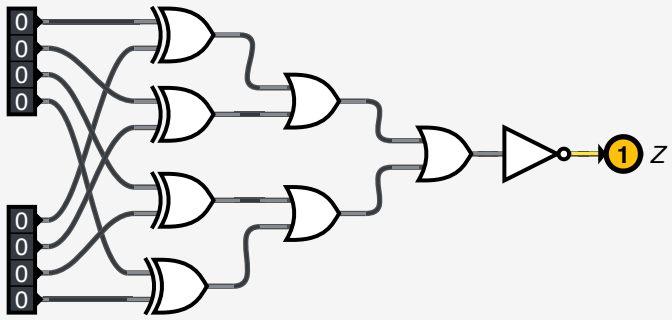
Plus difficile : essayez de réaliser un circuit qui calcule la même valeur de sortie, mais sans utiliser d'ALU.

Indice

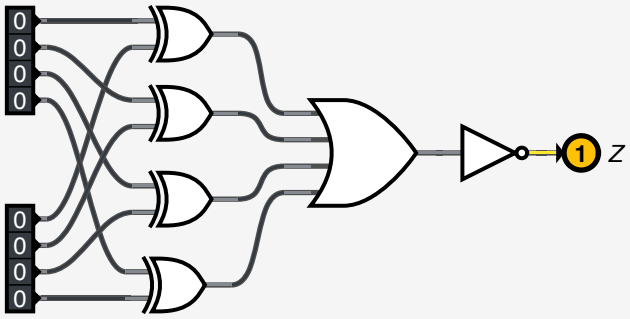
Une porte **OU-X** peut être vue comme un comparateur de deux bits : sa sortie vaudra 1 si et seulement si ses deux entrées sont différentes.

Corrigé sans ALU — approche logique

Cette solution utilise des portes **OU-X** comme comparateurs. On voit ici que 4 portes **OU-X** comparent deux à deux les 8 bits d'entrée. Leurs sorties sont ensuite combinées avec des portes **OU**, afin d'obtenir un signal qui vaudra 1 si au moins une différence est détectée, donc si les deux nombres d'entrées ne sont pas égaux. Il ne reste plus qu'à inverser ce signal pour obtenir la sortie demandée qui, selon la donnée, doit valoir 1 lorsque les nombres sont égaux.



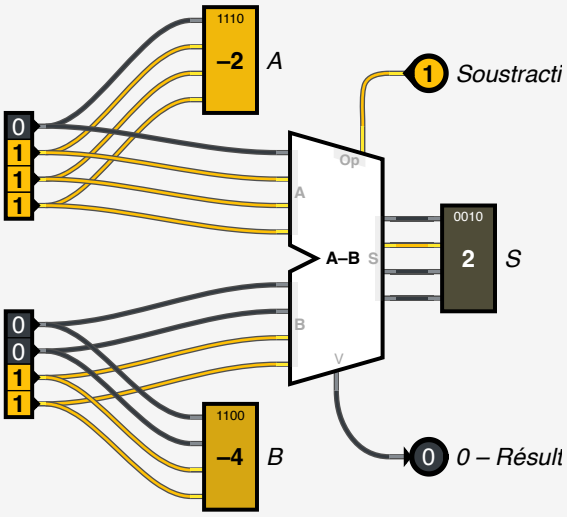
Alernativement, à la place d'utiliser trois portes **OU** dans le schéma ci-dessus, on aurait pu utiliser une grande porte **OU** à quatre entrées :



En résumé, nous avons appris ici ce qu'est une unité arithmétique et logique et avons examiné de plus près comment construire un multiplexeur, un circuit qui est à même de «choisir» parmi plusieurs signaux d'entrées lequel il va propager sur sa sortie. L'ALU est spécialement intéressante, car c'est le premier composant que nous rencontrons qui incarne une des propriétés de base d'un ordinateur, à savoir d'être programmable, en faisant en sorte que l'opération qu'elle effectue dépende d'un signal externe.

Pour aller plus loin

Notre petite ALU peut aussi faire des calculs en utilisant une représentation signée des nombres entiers. Sur 4 bits, une représentation en complément à deux peut représenter les nombres de -8 à $+7$. Il est possible d'utiliser les mêmes afficheurs de demi-octets en mode signé pour effectuer des opérations arithmétiques avec des valeurs négatives, par exemple, ici, $-2-(-4)=2$:



Notez que grâce à la représentation en complément à deux, la circuiterie interne de l'ALU peut se permettre de complètement ignorer si ses entrées sont signées ou pas et livrera le bon résultat tant que la convention d'entrée et de sortie reste la même.

Attention, lorsqu'on interprète les entrées et la sortie comme des nombres signés, ce n'est plus la sortie C_{out} de l'ALU qui indique un dépassement de capacité, mais la sortie V, qui est calculée différemment.

Essayez de régler les entrées pour que cette ALU calcule le résultat de $-8-4$. Vérifiez qu'un dépassement de capacité est signalé et expliquez pourquoi.

1.4 Mémoire

Les *transistors*, les *portes logiques* et leur représentation en *tables de vérités*, permettent de manipuler des 0 et des 1 au niveau physique… Tant qu'un courant électrique se déplace dans les *circuits*, on est capable de le transformer, de le laisser passer ou de l'arrêter, dans le but d'exprimer des portes «ouvertes» ou des portes «fermées» et donc des nombres binaires. L'ALU, explorée au chapitre précédent, va une étape plus loin et permet de choisir une opération à effectuer en fonction de bits de contrôle supplémentaire, et livre le résultat de l'opération arithmétique ou logique choisie.

Mais comment faire pour *stocker* cette information ? Comment faire pour que l'on se rappelle le résultat d'une addition effectuée par une ALU afin de pouvoir réutiliser cette valeur plus tard ? C'est là que nous avons besoin de *mémoire*.

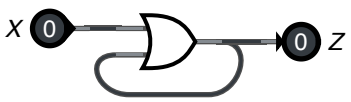
Dans les ordinateurs, il y a en fait plusieurs types de *mémoires*, qu'on peut classer en deux grandes catégories. La *mémoire volatile*, et la mémoire non volatile. La mémoire volatile s'efface quand la machine et éteinte. C'est le cas de la RAM (*random-access memory*), par exemple. La *mémoire non volatile*, elle, persiste. C'est le cas d'un disque dur ou d'un SSD (*solid-state drive*). Si un smartphone s'éteint inopinément alors qu'on est en train de retoucher une photo sans avoir validé les modifications, ces retouches disparaissent. Elles étaient stockées sur la mémoire volatile. Par contre, au moment où ces retouches sont sauvegardées, elles s'inscrivent dans la mémoire non volatile.

On peut se demander pourquoi on n'utiliserait pas que de la mémoire non volatile, vu les «risques» posés par la mémoire volatile. La réponse est que la mémoire non volatile va probablement être entre 100 et 100000 fois moins rapide que la mémoire volatile. On privilégie donc la mémoire volatile comme mémoire de travail rapide d'un ordinateur.

Dans les sections qui suivent, on propose de s'intéresser au cas le plus simple : la construction d'une cellule de mémoire volatile qui sera à même de stocker un bit. Par la suite, nous discuterons de la manière dont ce genre de mémoire est utilisée au cœur des microprocesseurs.

1.4.1 Le verrou SR

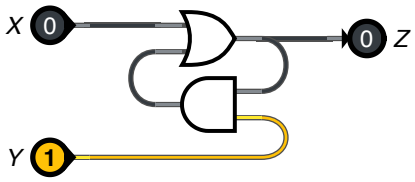
L'idée principale derrière la conception d'un circuit logique qui est capable de stocker un signal est que l'on va utiliser la ou les sorties du circuit en les reconnectant à certaines de ses entrées. Essayons par exemple ce circuit simple avec une seule porte **OU** :



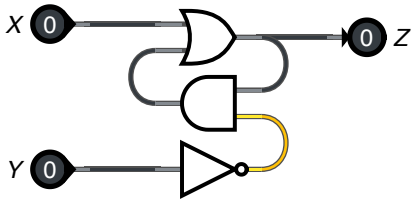
Au début, les deux entrées de la porte valent 0, comme sa sortie. Si l'on essaie de faire passer l'entrée X à 1, on voit que la sortie Z passera à 1 elle aussi, comme il s'agit d'une porte **OU**. Mais comme Z est aussi relié à l'autre entrée de la porte, on a maintenant un circuit dont on ne peut plus modifier la sortie : même si X passe de nouveau à 0, l'autre entrée reste à 1 et suffit donc pour que Z vaille maintenant 1 indéfiniment. On est obligé de remettre le circuit complètement à zéro (l'équivalent de débrancher la prise de courant et de la rebrancher) pour obtenir à nouveau un 0 sur la sortie Z.

Assurément, ce circuit n'est pas très intéressant : il se bloque dans un état sans retour possible. Il faudrait pouvoir faire repasser la valeur de sortie à 0. Pour ce faire, une idée est d'ajouter une porte **ET** avant de faire repasser la sortie de la porte **OU** dans sa propre entrée. Cela nous permet d'annuler le signal de retour si la seconde entrée du **ET** (appelons-la Y) vaut 0.

Essayez ce circuit : quand Y vaut 1, il se comporte comme le circuit précédent, mais se remettra à 0 dès qu'Y passera à 0.



Le circuit est plus facile à utiliser lorsqu'on inverse l'entrée Y, comme dans le circuit suivant :

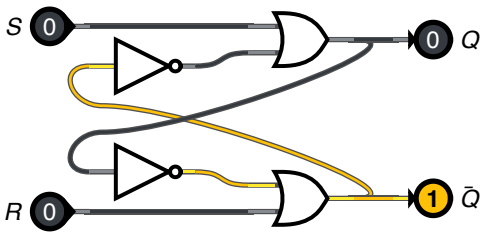


Dans ce circuit-ci, on peut considérer que :

- Tant que Y est inactif (donc vaut 0), le passage de X de 0 à 1 fait passer la sortie Z à 1. Ceci fait donc en sorte que la sortie «verrouille» sur la valeur 1, car elle garde sa valeur même si X repasse à 0.
- Tant que X est inactif, le passage de Y de 0 à 1 fait passer la sortie Z à 0, et la sortie se «verrouille» ainsi à 0 même si Y repasse à 0.

On appelle effectivement ce genre de circuits des *verrous*. Nous avons ici construit un des plus simples. D'habitude, son entrée X est appelée S, pour *set* en anglais, qui dénote le stockage d'un 1, et son entrée Y est appelée R, pour *reset* en anglais, qui dénote sa remise à zéro. C'est le verrou dit «SR», pour *set/reset*. Un tel verrou est donc une minicellule mémoire ! La sortie de ces cellules mémoires est fréquemment appelée Q plutôt que Z, nous allons donc faire pareil dans la suite du texte.

Voici une représentation équivalente du même circuit[6], donc du même verrou SR :

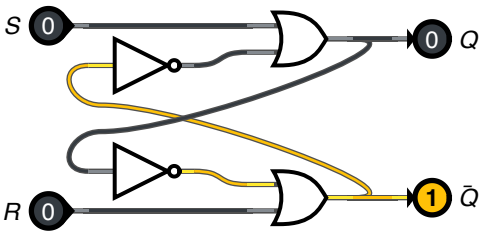


6. On peut montrer l'équivalence de ce circuit et du précédent à l'aide des lois de De Morgan[7], qui ne sont volontairement pas abordées dans ce chapitre.
7. https://fr.wikipedia.org/wiki/Lois_de_De_Morgan

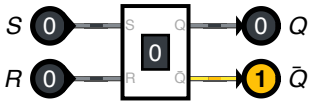
Ce circuit stocke ainsi un bit de donnée — un 0 ou un 1 — qu'on va pouvoir lire via la sortie Q et modifier avec les deux entrées R et S. (La seconde sortie $\bar{Q}$ est ici toujours l'inverse de Q.)

Testez le circuit ci-dessus et observez l'effet de R et S pour vérifier qu'il correspond bien à notre circuit précédent.

On essaie en général d'éviter d'avoir un 1 sur R et sur S en même temps, car cela place le verrou dans un état où $\bar{Q}$ n'est plus l'inverse de Q. Pour cette raison, nous allons plutôt créer le circuit comme dans le schéma suivant. Les connexions sont exactement les mêmes, mais les entrées S et R ne restent pas à 1 lorsqu'on clique dessus, elles retombent à 0 dès que le clic se termine — elles se comportent comme des boutons poussoirs.



Ces verrous sont communs, et pour le reste du chapitre, on simplifiera la notation pour les représenter ainsi, sans changement de fonctionnalité, mais en faisant abstraction des détails internes :



Pour aller plus loin

Voici une vidéo qui illustre ce principe de verrou SR.

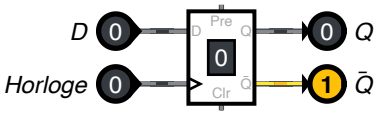
Vidéo 3: https://www.youtube-nocookie.com/embed/KM0DdEaY5sY?start=298

1.4.2 La bascule D

Un souci avec le verrou SR est qu'on a rarement un signal d'entrée qui soit facilement exploitable pour être «converti» en cette logique *set/reset*. La plupart du temps, on a simplement un signal donné, disons D, pour «donnée» (ou *data* en anglais), et c'est ce signal-ci qu'on aimerait stocker. Avec ce système, il serait impossible de connecter D à ce verrou ; on ne peut le brancher directement ni à l'entrée S, ni à l'entrée R.

On va utiliser pour cela un circuit similaire, mais qui fonctionne un peu différemment, qui s'appelle une **bascule D**[8] :

8. Il y a une différence conceptuelle fondamentale entre les verrous et les bascules : les verrous sont des composants dits *asynchrones*, dont l'état peut changer dès qu'une des entrées change, alors que les bascules sont des composants dits *synchrones*, qui ont une entrée appelée Horloge, et dont l'état ne changera qu'au moment où le signal d'horloge effectuera une transition (dans notre cas, passera de 0 à 1). Une discussion plus poussée de ces différences dépasse le cadre de ce cours.



Cette bascule va stocker son entrée D et la propager sur sa sortie Q uniquement lorsque l'entrée spéciale $Horloge$ passe de 0 à 1. Le reste du temps, Q et $\overline{Q}$ garderont leur valeur précédente. Notez que cette bascule a aussi deux entrées Pre et Clr, ici déconnectées, qui servent à forcer l'état interne à valoir 1 ou 0, respectivement, indépendamment du signal D et de l'horloge.

Testez cette bascule. Réglez l'entrée de données D à 1 ou 0 et observez comme la bascule ne réagit pas : sa sortie Q reste telle quelle. Donnez ensuite une impulsion en cliquant sur l'entrée $Horloge$ et voyez comme la valeur de D est maintenant stockée sur la bascule.

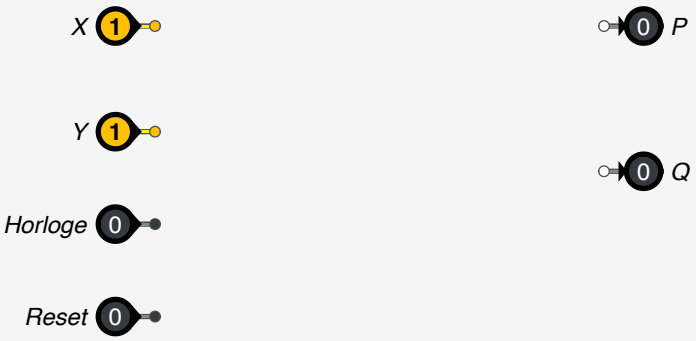
Pour aller plus loin

Pour aller plus loin, une vidéo de résumé qui parle aussi des bascules et des registres :

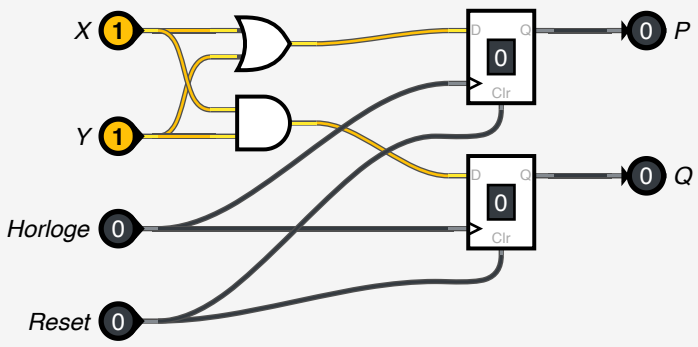
Vidéo 4: https://www.youtube-nocookie.com/embed/I0-izyq6q5s

Exercice 1 – Stocker deux bits

Créez un circuit qui calcule, d'une part, le **OU** de deux entrées X et Y, et, d'autre part, le **ET** de ces deux mêmes entrées. À l'aide de bascules D, complétez le circuit de manière à ce qu'il stocke ces deux valeurs calculées lors d'un coup d'horloge et les sorte sur les sorties P et Q, respectivement. Faites finalement en sorte que le signal $Reset$, si activé, réinitialise les bascules à 0. Vérifiez qu'une fois les valeurs stockées par les bascules, des changements sur les entrées X et Y n'aient pas d'effet direct sur P et Q.

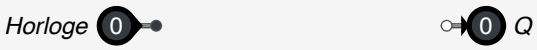


Corrigé

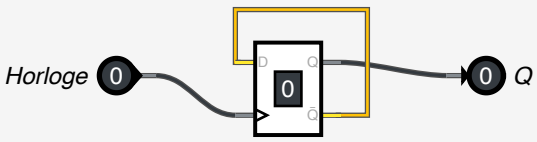


Exercice 2 – Signal alternatif

À l'aide d'une bascule, créez un circuit avec une sortie Q qui s'inverse à chaque coup d'horloge.



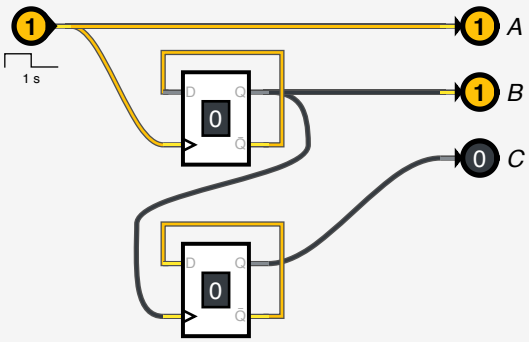
Corrigé



Exercice 3 – Jeu de fréquences

Observez le circuit ci-dessous. L'horloge principale A fonctionne ici toute seule et produit un coup d'horloge par seconde (elle a donc une fréquence d'un hertz — 1 Hz). Que pouvez-vous dire des signaux B et C par rapport au signal A ? Comment expliquer cela avec ce que vous savez des bascules ? (Pour simplifier, le délai de propagation est ici presque nul.)

Vous pouvez mettre l'animation en pause et exécuter chaque transition pas à pas pour mieux comprendre ce qui se passe.



Corrigé

Le signal B a une fréquence deux fois plus petite que le signal A, et le signal C, de façon similaire, a une fréquence deux fois plus petite que le signal B. Ainsi, B «bat» à 0.5 Hz et C à 0.25 Hz.

TODO ajouter explication

Si ce petit circuit fonctionne à 1 Hz, les appareils que nous utilisons aujourd'hui ont des horloges qui fonctionnent à plusieurs gigahertz (GHz), c'est-à-dire plusieurs milliards de fois plus vite. On attend ainsi moins d'une nanoseconde entre deux coups d'horloge.

1.4.3 Addition en plusieurs étapes

Dans cet exemple final, nous allons construire un circuit capable d'effectuer l'addition de plusieurs nombres ; par exemple, d'évaluer la somme $1 + 4 + 5 + 3$ pour trouver 13.

Si ce calcul a l'air simple, il s'y cache une subtilité : nous n'avons aucun circuit auquel nous pourrions donner quatre nombres et qui en ferait la somme. Nous ne savons additionner que deux nombres à la fois. Mais nous pouvons additionner progressivement les nombres un à un à une sorte d'«accumulateur» qui stockerait les résultats intermédiaires. Au début, avant d'avoir additionné quoi que ce soit, cet accumulateur représenterait un 0. Ensuite, on y additionnerait, l'un après l'autre, chacun des nombres du calcul ainsi :

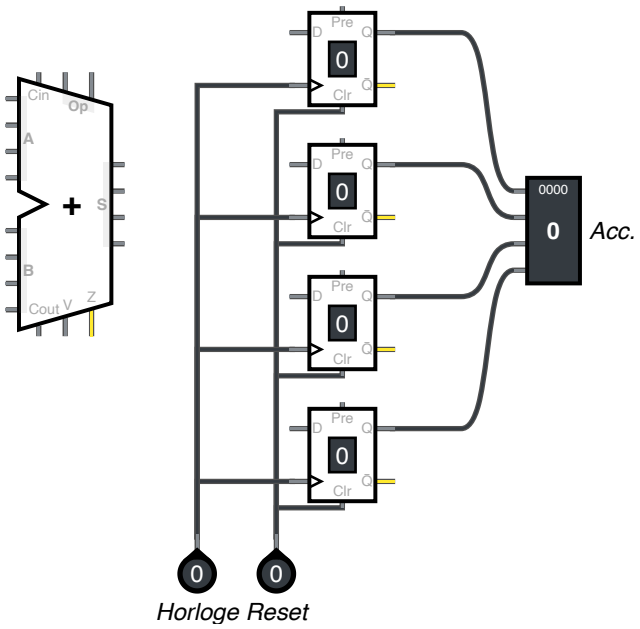
$$0 + 1 = 1$$
$$1 + 4 = 5$$
$$5 + 5 = 10$$
$$10 + 3 = 13$$

Chacune de ces lignes a la forme «accumulateur + nombre à additionner = nouvel accumulateur».

L'avantage de procéder ainsi, en décomposant à l'extrême, est que chaque étape est une addition de précisément deux nombres — et nous savons faire de telles additions avec une ALU.

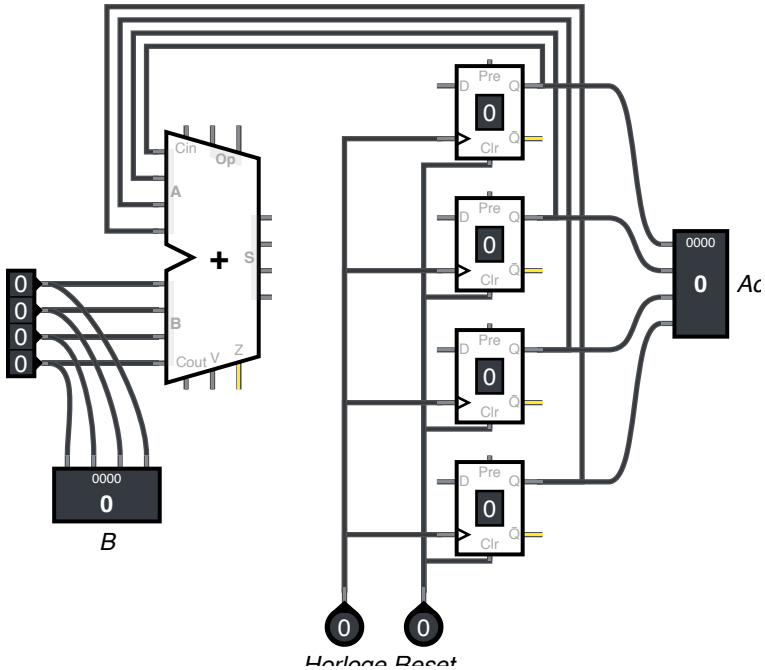
Commençons à créer un circuit capable de faire ceci. Notre ALU opérant sur des nombres de 4 bits, prenons le parti de représenter notre accumulateur via également 4 bits — 4 cellules mémoire, et donc 4 bascules. Pour remettre l'accumulateur à zéro, nous allons connecter un signal unique au *reset* de chacune de ces bascules. Nous allons aussi, comme chaque fois, connecter un signal d'horloge aux bascules, pour leur indiquer leur moment où elles doivent stocker les valeurs qui sont sur leurs entrées respectives. Ajoutons aussi une ALU pour effectuer l'addition et un afficheur décimal pour les 4 bits stockés dans les bascules.

Cela nous donne ce début de circuit, qui pour l'instant n'est pas fonctionnel :



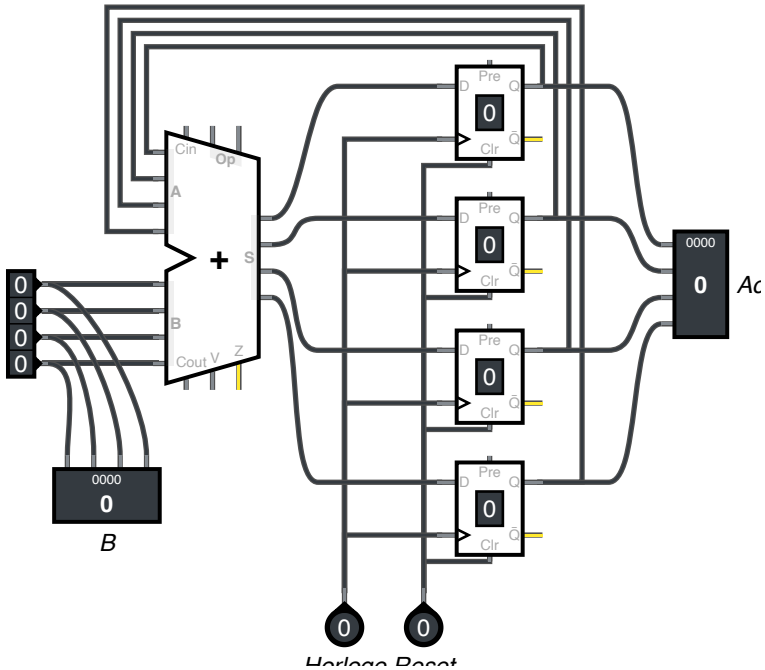
Connectons maintenant les entrées de l'ALU. On se rappelle qu'à chaque étape, l'ALU calculera une addition de la forme «accumulateur + nombre à additionner = nouvel accumulateur». L'entrée A de l'ALU est la valeur de l'accumulateur, donc ce qui est stocké par nos bascules. On connecte donc la sortie Q de chaque bascule vers le bit d'entrée A correspondant de l'ALU.

L'entrée B de l'ALU est le nouveau nombre à additionner. Pour cela, nous ajoutons simplement quatre entrées normales, ainsi qu'un afficheur décimal pour nous simplifier la lecture du nombre représenté par ces entrées :



Il reste à connecter la sortie S de l'ALU. Cette sortie nous livre la prochaine valeur à stocker dans l'accumulateur, et nous pouvons ainsi la connecter aux quatre entrées D des bascules.

Voici le circuit final :



Ce circuit fonctionne ainsi : au début du calcul, on réinitialise les bascules à zéro avec le signal $Reset$. Ensuite, on compose le prochain nombre à additionner sur l'entrée B. L'ALU va calculer immédiatement la somme $A + B$, mais ce n'est qu'au prochain coup d'horloge que cette somme sera stockée dans les bascules et apparaîtra ainsi à droite. Après avoir donné ce coup d'horloge, donc, on pourra à nouveau composer sur l'entrée B le prochain nombre à additionner, et ainsi de suite.

On réalise ici l'importance du coup d'horloge : si les bascules stockaient immédiatement la valeur livrée par l'ALU sans attendre le coup d'horloge, on retrouverait presque sans délai cette valeur sur la sortie des bascules et donc… à l'entrée A de l'ALU, qui recalculerait immédiatement la somme de cette valeur et de l'entrée B, livrerait le résultat sur la sortie vers les bascules, qui feraient à nouveau la propagation immédiate de ceci sur leurs sorties et sur l'entrée A de l'ALU, etc. — le système s'emballerait. Le signal d'horloge veille à ce que l'opération de stockage et de propagation soit coordonnée et se passe au bon moment.

Exercice 4 – Additions avec bascules

Suivez la procédure décrite ci-dessus pour effectuer l'addition $1 + 4 + 5 + 3 = 13$.

1.4.4 Récapitulatif

Au cours des quatre chapitres précédents, nous avons vu comment les portes logiques sont utilisées comme composants de base des ordinateurs. Nous avons d'abord exploré des portes simples comme **OU** et **ET**, puis montré comment ces portes peuvent être combinées en systèmes logiques plus complexes.

Avec des portes, nous avons construit un additionneur de deux bits. Nous avons ensuite été à même, en enchaînant plusieurs additionneurs, de créer un système qui peut additionner non pas simplement deux bits, mais deux nombres entiers codés sur 4 bits chacun.

Nous avons ensuite découvert l'unité arithmétique et logique, capable de réaliser plusieurs opérations différentes avec ses entrées en fonction de bits supplémentaires qui permettent de sélectionner l'opération à effectuer.

Notre dernière étape d'exploration des systèmes logiques nous a menés aux verrous et aux bascules, des composants pensés pour stocker des bits de données et ainsi constituer des cellules de mémoire pour l'ordinateur. Nous avons enfin été capables, avec une ALU et une série de bascules, d'additionner à la chaîne plusieurs nombres, en nous rappelant les résultats des additions intermédiaires.

Il existe bien d'autres éléments qui composent les ordinateurs et nous n'avons pas l'occasion de tous les explorer en détail. Dans la section qui suit, faisons un saut conceptuel et parlons de l'architecture générale des ordinateurs et de la manière dont les grands composants sont interconnectés pour permettre à un ordinateur de remplir les fonctions que nous lui connaissons.

Jeu pour aller plus loin

Dans le jeu en ligne «Nandgame» (https://nandgame.com), on construit petit à petit un ordinateur complet juste avec, à la base, des portes **NON-ET**. Elles ont la particularité (avec les portes **NON-OU**, d'ailleurs) de pouvoir simuler toutes les autres portes — y compris un inverseur.

1.5 CPU

Le processeur, en anglais central processing unit (CPU), est un composant qui exécute les instructions machine des programmes informatiques.

On a précédemment détaillé les différents composants et systèmes logiques à partir desquels on peut construire un processeur. On va à présent évoquer l'architecture de von Neumann qui décrit la façon dont le processeur s'insère dans son environnement. Les différents éléments qui constituent le processeur et qui en assurent le bon fonctionnement seront ensuite détaillés.



Gordon Moore

— **Naissance** 1929 / San Francisco

Bio

Gordon Earle Moore est le cofondateur d'Intel en 1968. Intel est le premier fabricant mondial de microprocesseurs. Gordon Moore est célèbre pour avoir formulé en 1965 une loi empirique portant son nom : **loi de Moore**. Cette loi prédit un doublement de la complexité, et donc du nombre de transistors présents dans les microprocesseurs tous les deux ans. Bien que nous ayons atteint certaines limites physiques au niveau atomique et des effets de bruits parasites liés aux effets quantiques et à la désintégration alpha, la loi est toujours vérifiée aujourd'hui malgré un ralentissement de la progression pour certaines caractéristiques. Ces limites sont aujourd'hui compensées par des puces intégrant de plus en plus de composants de plus en plus complexes.

(micro)-processeur

Le processeur est l'unité de traitement central de l'ordinateur. Il est construit avec des circuits regroupés en systèmes qui produisent des fonctions logiques et arithmétiques en suivant un programme et en utilisant des éléments de mémoire appelés registres. Un microprocesseur est un processeur construit avec un circuit intégré, c'est-à-dire un dispositif qui tient sur quelques cm2. Il n'y a donc que la taille qui fasse la différence.

1.5.1 Horloge et accès mémoire

Un processeur est un dispositif synchrone, ce qui signifie que les opérations à l'intérieur du processeur se déroulent de manière synchrone à un temps donné. Pour assurer cette simultanéité, il faut comme pour un orchestre, donner le tempo. Cette fonction de métronome est assurée par une horloge, ou un signal d'horloge. Cette horloge est constituée d'un simple signal carré dont la fréquence atteint aujourd'hui plusieurs gigahertz, c'est-à-dire plusieurs milliards de cycles par seconde.

La notion de *synchronisation*

La notion de synchronisation est centrale. Les systèmes numériques synchrones sont ceux dont les opérations (instructions, calculs, logique, etc.) sont coordonnées par un ou plusieurs signaux d'horloge centralisés, par opposition, aux systèmes asynchrones qui n'ont pas d'horloge globale. Les systèmes asynchrones ne dépendent pas d'heures d'arrivée strictes des signaux ou des messages pour un fonctionnement fiable. La coordination est obtenue via des tests sur l'arrivée des évènements.

Sans entrer dans les détails ici, on notera que dans un système synchrone, il est possible d'assurer une coordination et une cohérence des opérations, ce qui est impossible autrement. Cet aspect devient crucial dans les systèmes distribués qui ne disposent plus de la garantie de synchronisation.

Les circuits asynchrones ont été envisagés comme une alternative possible aux circuits synchrones, plus répandus, particulièrement pour diminuer la consommation d'énergie, augmenter la vitesse, faciliter la conception et augmenter la fiabilité. Il semblerait qu'après avoir été un peu délaissée par le monde de la recherche et du développement depuis les années 1990 et 2000, la thématique des circuits asynchrones suscite à nouveau un regain d'intérêt, en particulier relativement aux impératifs de faible consommation énergétique en relation avec le changement climatique.

L'accès à la mémoire

Rappel

Comme on l'a vu dans l'architecture de von Neumman, la mémoire contient le programme et les données du programme. Un programme peut donc théoriquement se modifier lui-même en se modifiant dans la mémoire, même si, en pratique, toutes les architectures modernes l'interdisent (c'est rarement un effet souhaité !).

L'UCT doit accéder à la mémoire RAM en lecture ou en écriture. Les deux mécanismes sont très similaires, mais avant de regarder plus en détail comment cela fonctionne, il faut d'abord définir comment la mémoire est structurée. La mémoire RAM permet, comme on l'a vu au premier chapitre, d'accéder à tout moment à n'importe quel emplacement.

Pour y accéder, le processeur envoie d'abord l'adresse au module mémoire, puis lit ou écrit la valeur via le bus d'adressage.

Anecdote

Le processeur Intel 80286 (ancêtre des processeurs Pentium), sorti en 1982, présentait un bus de données de 16 bits et un bus d'adresses de 24 bits. De plus l'adressage par segments (relativement compliqué) réduisait l'adressage physique à un adressage sur 20 bits.

Il manque encore un élément : lorsque la mémoire voit une adresse apparaître elle doit pouvoir déterminer s'il s'agit d'une lecture ou d'une écriture. Pour cela deux connexions supplémentaires relient le processeur à la mémoire : une ligne *enable* et une ligne *set*. Lorsque la ligne *enable* est à 1, alors le processeur accède à la mémoire en lecture et sur le bus de données doit apparaître les données qui sont stockées dans la mémoire à l'adresse indiquée sur le bus d'adressage. Lorsque c'est la ligne *set* qui est à 1, alors la mémoire doit enregistrer les données à l'adresse indiquée.

Le contenu de la mémoire

Les instructions : la mémoire contient le programme sous forme de codes qui représentent des instructions à exécuter par le processeur. Ces codes correspondent à un jeu d'instructions propre à chaque modèle de processeur. On parle de langage machine. Pour écrire de tels programmes, on utilise un programme et un langage assembleur, proche de la machine : c'est une représentation exacte du langage machine, mais qui est une interface plus lisible dans la communication machine. C'est le langage de plus bas niveau qui représente le langage machine sous une forme lisible par un humain.

Les données : les données stockées dans la mémoire peuvent être des nombres, des lettres, des chaînes de caractères ainsi que des adresses d'autres emplacements en mémoire. On trouvera plus de détails à ce sujet dans le chapitre représentation de l'information.

Question 1

Avec un bus d'adressage de 24 bits, quelle est la taille maximum de la mémoire ?

1. 32ko
2. 16Mo
3. 16Go

Réponse: 2

Question 2

Quelle est la taille maximale de la mémoire pour un processeur 80286, sachant que l'adressage physique est finalement réduit à 20 bits ?

1. 32ko
2. 16Mo
3. 1Mo

Réponse: 3

L'unité de contrôle

L'unité de contrôle reçoit les instructions en provenance de la RAM. Elle s'occupe d'activer les composants qui doivent l'être dans le microprocesseur.

Les registres

Les registres permettent de stocker des valeurs, comme la RAM, mais directement à l'intérieur du processeur. Ils fonctionnent aussi en mode lecture ou écriture. C'est l'unité de contrôle qui détermine si un registre est utilisé en lecture ou en écriture avec deux fils de connexion : *enable* et *set*. En principe ces registres stockent les informations en provenance de la mémoire ou le résultat d'un calcul. Il existe trois registres plus spécifiques :

Le registre d'état

Le registre d'état regroupe les drapeaux (en anglais *flags*). Ils servent à renseigner l'état d'exécution du processeur. Par exemple le drapeau *dépassement* s'il est mis à 1 signale qu'un dépassement de capacité est survenu, ou encore le drapeau *division par zéro* signale une division par zéro.

Le compteur de programme

Le compteur de programme (registre **PC** pour *Program Counter*) contient l'adresse mémoire de la prochaine instruction devant être exécutée. En principe l'unité de contrôle l'incrémente de un après chaque instruction, mais certaines instructions qui permettent de se *brancher* ailleurs dans le programme modifient différemment ce registre.

Le compteur de pile

Le compteur de pile (registre **SP** pour *Stack Pointer*) contient la position sur une pile. Cette dernière est une zone mémoire à laquelle on ne peut pas accéder aléatoirement, mais uniquement en empilant ou dépilant des éléments.

1.5.2 L'unité arithmétique et logique

L'unité arithmétique et logique (UAL plus communément appelée ALU en abréviation anglaise) effectue tous les calculs arithmétiques et logiques. Quelques-uns de ces composants comme l'additionneur ont été abordés dans le chapitre *De la logique à l'arithmétique*.

Exemple : le 6502

Le 6502, conçu en 1975, est le premier microprocesseur grand public avec un prix de 25$ (bien en-dessous des concurrents de cette époque). Une de ses premières utilisations pour le grand public fut la console de jeux vidéo Atari 2600. A partir de 1985, Nintendo équipe la NES d'une version modifiée du 6502. Il a équipé également le célèbre Apple II. Il a donné lieu à de nombreuses versions, jusqu'aux processeurs 16 bits actuels de dernière génération.

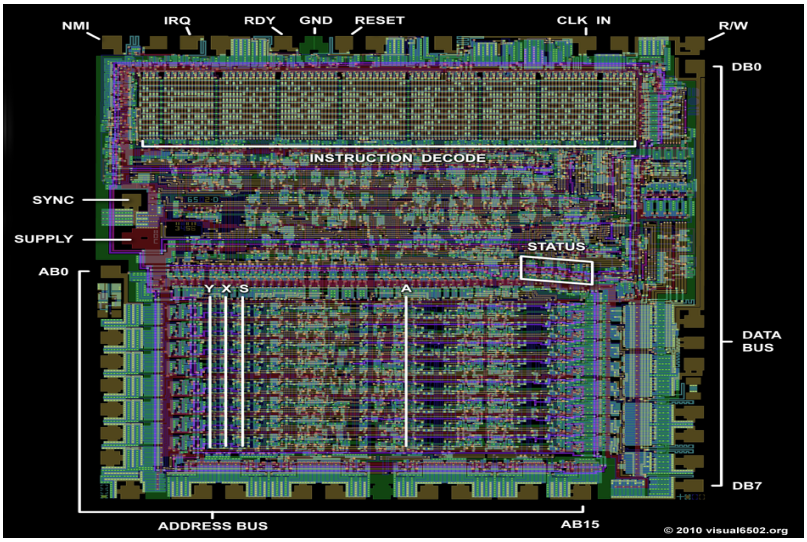


Fig. 1.7 – Ce schéma détaille l'ensemble des transistors du 6502. On voit également quelques-uns des éléments principaux (horloge, registres, etc)

Activité

Simulateur visuel du 6502 [9]

Ce simulateur reproduit le fonctionnement complet du 6502 jusque dans l'activité de chaque transistor. On peut clairement visualiser la façon dont la complexité du fonctionnement de ce qu'on appelle communément le *cerveau* de l'ordinateur émerge de la quantité de dispositifs triviaux pris individuellement.

1. Observer le déroulement du programme proposé et tenter d'en déduire le fonctionnement. On pourra s'aider du désassembleur proposé sur la même page.

Question

Que fait le programme en exemple sur le site visual6502 ? Il parcourt la mémoire et recopie la valeur 40 à des adresses successives. Il effectue une boucle et incrémente une valeur en mémoire à l'adresse FF. Il additionne deux registres et stocke le résultat dans un autre registre.

Réponse : ?

2. Modifier ou écrire un nouveau programme en allant sur la page *Avanced*.

---

9. http://visual6502.org/JSSim/index.html

Aller plus loin

La partie qui suit présente de manière plus approfondie certaines spécificités des processeurs modernes.

1.5.3 Processeur à noyau unique

C'est le processeur standard : un processeur à noyau unique ou CPU utilise un seul noyau à l'intérieur du processeur.

Avantages :

Un processeur à un seul cœur consomme moins d'énergie que les processeurs à plusieurs cœurs. Ceci est surtout problématique pour les appareils mobiles, oû le problème de l'autonomie de la batterie est essentiel. Comme les processeurs à cœur unique consomment moins d'énergie, l'ensemble du système qu'ils font fonctionner chauffe moins. Un processeur à un seul cœur est toujours adapté pour la plupart des applications : vérification du courrier, navigation sur Internet, téléchargement de données, etc. peuvent toujours être traitées par un processeur à noyau unique.

Inconvénients :

C'est un processeur relativement lent. Il n'a pas une grande puissance de calcul pour traiter de grandes opérations complexes, ou plusieurs opérations à la fois. Comme les applications modernes nécessitent une grande puissance de traitement, un processeur monocœur qui les fait fonctionner peut se bloquer, paralysant ainsi l'ensemble du système alors «planté».

1.5.4 Processeur à double cœur

Un processeur à double cœur possède deux cœurs pour exécuter les opérations, intégrés dans un circuit unique pour se comporter comme une seule unité - un seul processeur -, à la différence d'un système multiprocesseur ; toutefois, ces cœurs possèdent leurs propres contrôleurs et caches, ce qui leur permet de travailler plus rapidement que les processeurs à cœur unique.

Avantages :

Un processeur double cœur exécute l'ensemble des tâches beaucoup plus rapidement. Si un processeur à noyau unique est chargé de deux tâches différentes, il ne peut pas les effectuer simultanément. Il passe à toutes les tâches une par une, en série, alors qu'un processeur à double cœur peut effectuer les deux opérations en même temps, en parallèle. Un processeur double cœur «équivaut» à deux ordinateurs en un… mais à un tarif moindre.

Inconvénients :

Peu d'opérations nécessitent réellement la puissance des processeurs double cœur. Une grande partie de la puissance est ainsi gaspillée et vide rapidement la batterie. Un appareil mobile utilisé à des fins informatiques générales, telles que la vérification du courrier électronique, la navigation sur Internet, la saisie de documents et le partage de données, ne nécessite pas réellement la puissance d'un processeur

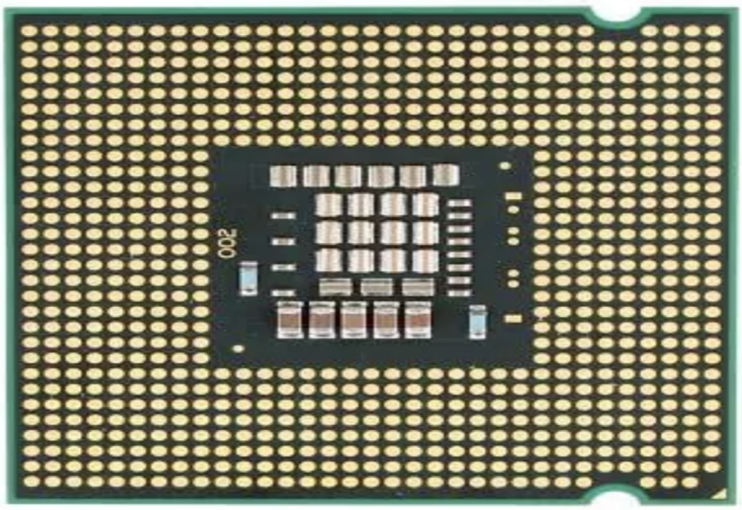


Fig. 1.8 – Microprocesseur bicœur

double cœur. Pour ces raisons, de nombreux développeurs d'applications mobiles ne programment pas leurs applications pour qu'elles fonctionnent avec des processeurs à multiple cœur, les rendant ainsi incompatibles avec les mobiles qui fonctionnent toujours avec des processeurs à double ou multiple cœur.

1.5.5 Les processeurs quadricœur et autres processeurs à cœurs multiples

En termes simples, un processeur quadricœur possède quatre cœurs et il en va de même pour un processeur hexacœur (six cœurs), octocœur (huit cœurs), etc… Ces cœurs peuvent être soit sur le même circuit intégré, soit sur le même boîtier de puce.



Fig. 1.9 – Microprocesseur quadricœur

Avantages :

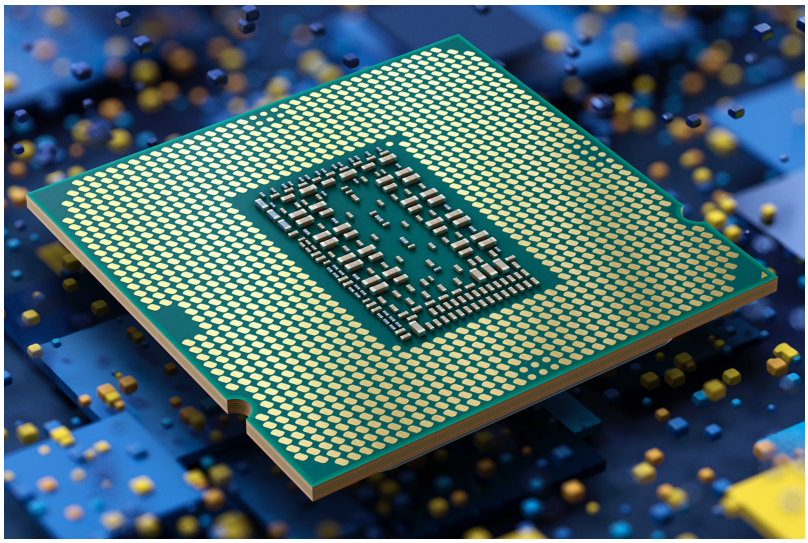


Fig. 1.10 – Microprocesseur octocœur

Le multitâche est le principal avantage des processeurs quadri ou octocœurs. Un plus grand nombre de cœurs offre évidemment une plus grande capacité à effectuer plusieurs tâches en parallèle. Ces processeurs sont utiles pour exécuter des applications qui sont plutôt intensives et nécessitent beaucoup de ressources. Ces applications comprennent les éditeurs vidéo, les antivirus, les programmes graphiques, etc. Les nouveaux processeurs quadricœurs consomment moins d'énergie, dégagent moins de chaleur, et sont donc très efficaces. Ces processeurs sont en fait très en avance sur la technologie actuelle de développement d'applications mobiles, car tous les développeurs ne sont pas capables de programmer des applications fonctionnant sur ces processeurs multiples. De nombreux programmes sont encore écrits pour des processeurs à double ou simple cœur.

Inconvénients :

… encore et toujours la consommation énergétique, vidant très rapidement la batterie.

Le nombre de cœurs de processeur est important dans certaines activités comme le *gaming* : il est de plus en plus courant de trouver des processeurs hexa-cœurs, ou octo-cœurs ; les dernières générations de multiprocesseurs possèdent jusqu'à 12 ou 16 cœurs [10] !

On doit également mentionner les cœurs logiques, c'est-à-dire les *threads*, comme on les appelle plus communément (tâches en français). La performance d'un monoprocesseur est jugée sur sa capacité à gérer plusieurs «fils» d'instructions. Du point de vue de l'utilisateur, ces exécutions semblent se dérouler en parallèle. Toutefois, là où chaque processus possède sa propre mémoire virtuelle, les threads d'un même processus se partagent sa mémoire virtuelle. Par contre, tous les threads possèdent leur propre pile d'exécution.

Les technologies d'hyperthreading d'Intel et de multithreading d'AMD permettent à un seul cœur physique de gérer deux tâches simultanément, fonctionnant ainsi comme deux cœurs logiques distincts. Cette technologie est aujourd'hui très performante. La plupart de la gamme Ryzen d'AMD propose le multithreading, y compris les modèles de milieu et de haut de gamme, tandis que l'hyperthreading est pour l'instant réservé aux processeurs i7 et i9 haut de gamme d'Intel.

10. https://www.futura-sciences.com/tech/comparatifs/meilleur-processeur-comparatif

1.5.6 Le pipeline

On l'a vu, l'exécution d'une instruction par le microprocesseur implique plusieurs opérations : accès à la mémoire en lecture et en écriture, accès aux registres en lecture et en écriture, opération logique. Pour optimiser la vitesse d'exécution, les processeurs modernes effectuent en série ces opérations. Ainsi, alors que les opérations logiques d'une instruction sont effectuées, l'instruction précédente est déjà chargée en mémoire. La difficulté de ce type d'optimisation réside dans le fait que des branchements conditionnels provoquent l'annulation des instructions déjà chargées. Pour optimiser encore ce genre de procédé, les processeurs font de la prédiction dans l'exécution. Ces optimisations sont extrêmement compliquées à gérer.

Anecdote

La vulnérabilité Spectre (ainsi que d'autres vulnérabilités similaires) exploite justement cette fonction de prédiction dans l'exécution de branchements conditionnels pour accéder à des emplacements mémoire auxquels le programme ne devrait en principe pas accéder.

Matière à réfléchir. Vite… très vite

Nous avons démontré que finalement nos ordinateurs ont un cerveau très simple dans le fonctionnement de ses éléments de base : des portes logiques qui traitent des **0** ou des **1**. Il est cependant très difficile de se représenter à quel point ces traitements vont vite. Imaginons pour cela que le processeur écrive toutes les opérations qu'il effectue sur un ruban de papier et calculons la vitesse de défilement de ce papier.

Pour cela, nous faisons les hypothèses suivantes :

- Les processeurs actuels ont une cadence d'horloge de 3 GHz, c'est à dire $3{\cdot}10^9\ [s^{-1}]$. Pour simplifier, nous allons supposer qu'ils effectuent une opération par cycle [1].
- Nous transcrivons un mot de 64 bits (taille standard pour les processeurs actuels) sur une longueur de 15 cm, ce qui correspond à $15 \cdot 10^{-2}\ [m]$.

Le calcul devient alors :

$$3 \cdot 10^9\ [s^{-1}] \times 15 \cdot 10^{-2}\ [m]$$
$$45 \cdot 10^7\ [m/s]$$

Que nous convertissons en km :

$$45 \cdot 10^5\ [km/s] \text{ ou encore : } 450'000\ [km/s]$$

Rappelons que la vitesse de la lumière est :

$$c \cong 300'000\ [km/s]$$

Ce qui veut dire que si un microprocesseur, tel que ceux que l'on trouve dans notre ordinateur ou notre smartphone, écrivait sur un ruban de papier tout ce qu'il fait, ce ruban de papier devrait se déplacer à une fois et demi la vitesse de la lumière. Ou encore, ce ruban ferait chaque seconde plus de 11 fois le tour de la terre.

1. En fait le opérations d'un processeur prennent plus d'un cycle pour être réalisées, mais comme les processeurs ont plusieurs cœurs et un pipeline dont nous n'abordons pas ici le fonctionnement, la simplification proposée n'est pas aberrante.

Si les éléments de base sont simples, la complexité et la richesse des expériences numériques comme l'immersion dans un jeu vidéo proviennent de la quantité extraordinaire d'opérations effectuées.

1.6 Architecture générale

Il est commun d'entendre parler du microprocesseur comme du «cœur de l'ordinateur». On se propose de dégager les caractéristiques essentielles de ce qui constitue cette machine «intelligente» appelée ordinateur, tout en explicitant les composants informatiques spécifiques (le matériel ou «hardware»).

Si l'on suit l'évolution de l'ordinateur, depuis les années 50 jusqu'à aujourd'hui, on peut distinguer les éléments caractéristiques illustrés sur la figure suivante.

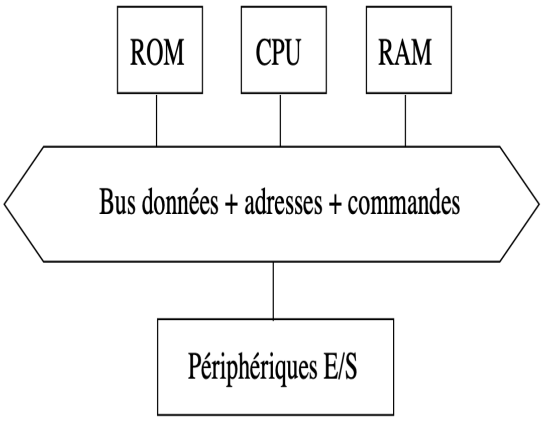


Fig. 1.11 – Schéma simplifié d'un ordinateur

D'un point de vue matériel, on distingue :

- l'alimentation,
- la carte mère,
- le processeur,
- la mémoire vive RAM,
- le disque dur / SSD / eMMC,
- le lecteur-graveur,
- la carte graphique.

On peut également citer les cartes sons, réseau, sorties USB etc. Ce type de matériel n'étant pas indispensable au bon fonctionnement du PC et souvent directement intégré à la carte mère, on ne s'y intéressera pas ici. On distingue ce matériel, partie intégrante de la machine, avec les périphériques externes qui lui sont reliés par des câbles ou des moyens de communication sans fil.

1.6.1 La mémoire

ROM (Read-Only Memory) : ce que l'on nomme ROM constitue une mémoire «fixe», statique de la machine, dont la taille est définie à la conception. On parle de mémoire morte, ou mémoire en lecture seule. Ce qu'elle stocke ne «part pas» lors de la mise hors tension de la machine.

Cette mémoire fixe va intégrer tous les éléments nécessaires en particulier au démarrage de la machine, au lancement du système d'exploitation ; il en est de même en ce qui concerne les facteurs de conversion, tables de constantes, instructions propres de la machine. On distingue différents types de ROM :

— ROM standard,
— EPROM : ce type de mémoire rend programmation et effacement accessibles à l'utilisateur,
— PROM : une programmation unique est possible sur ce type de mémoire,
— EEPROM : c'est une mémoire programmable dont les données peuvent être effacées électriquement (succède à l'UVPROM dont les données pouvaient être supprimées dans une «chambre à UV»).

Il s'agit d'une mémoire à long terme.

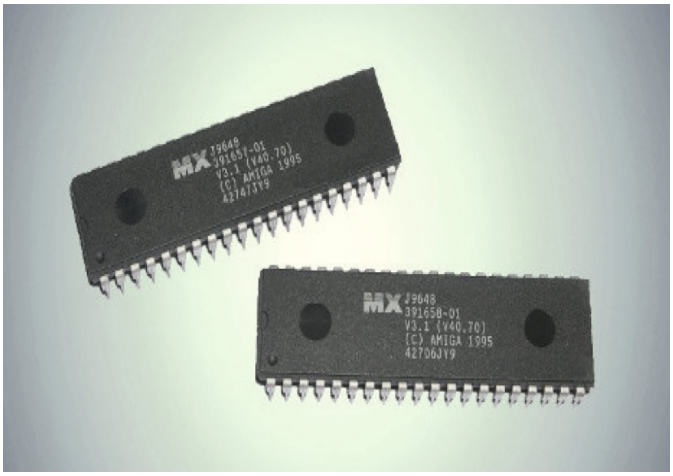


Fig. 1.12 – Barrettes ROM

RAM (Random Access Memory) : cette mémoire est une mémoire volatile, c'est à dire que son contenu va «disparaître» lorsque l'ordinateur est hors tension. On parle aussi de mémoire tampon. L'information étant stockée sous forme électrique dans les transistors, elle disparaît quand l'alimentation est coupée.

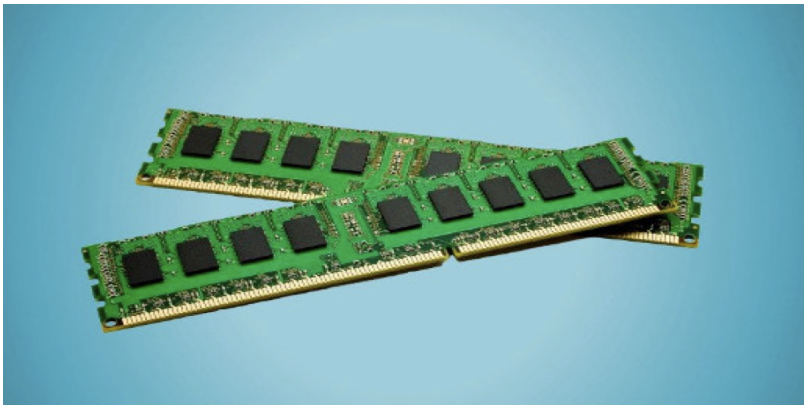


Fig. 1.13 – Barettes RAM

Cette mémoire stocke temporairement et aléatoirement les fichiers sur lequel on travaille : on parle de mémoire vive en français. Elle est accessible en lecture / écriture. Ce type de composants se présente sous la forme de «barrettes» amovibles faciles à remplacer. Les temps d'accès sont très courts (de l'ordre de la nanoseconde), et la capacité de stockage élevée en comparaison avec celle de la ROM. Elle est la plus coûteuse des deux. On distingue deux types de RAM :

— la RAM statique nécessite un flux constant d'énergie pour conserver les données qu'elle contient,
— la RAM dynamique (DRAM, SDRAM) : elle doit être actualisée pour conserver les données qu'elle contient ; elle est plus lente et moins chère que la RAM statique.

Les composants de mémoire RAM existent en général en «barrettes» allant de 1 à 8 Go par unité (les configurations les plus courantes actuellement proposent 4 à 8 Go de RAM), elles sont à choisir en fonction du processeur et de l'utilisation que l'on fait du PC d'une part et des possibilités de la carte mère (capacité totale, nombre d'emplacements disponibles…) d'autre part.

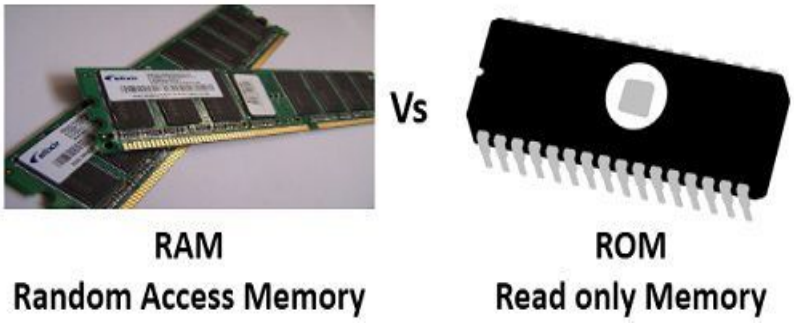


Fig. 1.14 – RAM versus ROM

Une analogie intéressante pour comprendre les particularités et missions respectives de la ROM et de la RAM est celle du commerçant et de la gestion quotidienne de sa caisse. Dans sa journée de travail, le commerçant encaisse, rend de la monnaie, gère donc des flux monétaires fluides et dimensionnés : dans le cas d'une boulangerie par exemple, les flux moyens vont être de l'ordre de quelques francs. En fin de journée, le commerçant compte sa caisse et part déposer la recette du jour à la banque : cela peut représenter alors plusieurs milliers de francs. La caisse représente en quelque sorte la RAM, accessible facilement et rapidement, avec des flux modérés et relativement réguliers ; en revanche sa taille est limitée. Elle est vide le matin, se charge et décharge dans la journée, puis se vide en fin de journée à la clôture. La banque représente la ROM : le stockage de l'argent prend plus de temps, mais l'espace de stockage est beaucoup plus vaste et est sécurisé.

Mémoire externe ou mémoire de masse

Ce terme désigne les supports externes à la machine permettant du stockage supplémentaire et donc venant en complément du stockage initial fixe de la machine (ROM). Par exemple : disque dur interne ou externe, bande magnétique, SSD, disque optique, clé usb, carte SD, … Cette mémoire permet de stocker une grande quantité d'informations, mais à une vitesse limitée. C'est un peu l'intermédiaire entre la RAM et la ROM.

Sur le disque dur peuvent être enregistrées les données à conserver : les fichiers du système d'exploitation, les logiciels et surtout les données personnelles (photo, vidéo, musique, emails etc…).

Le disque dur se présente sous la forme d'un boîtier rectangulaire, vissé au boiter du PC, qui intègre toute la mécanique (plateau, bras, tête de lecture…). Plus la vitesse de rotation des plateaux est importante, plus les performances sont élevées : on trouve actuellement des disques durs tournant à 5400, 7200, 10000 ou 15000 RPM (Round Per Minute : tours par minute), les vitesses de 7200 et 10000 RPM étant les plus répandues.

Il est relié à la carte mère grâce à une nappe (câble plat) de type IDE ou grâce aux interfaces SATA (Serial ATA) ou SCSI. Un cavalier à positionner à l'arrière du boîtier permet généralement de le désigner comme disque «Maître», le disque dur principal (Master) ou comme «Esclave», un disque auxiliaire (Slave).

Les disques durs aujourd'hui peuvent contenir des centaines de gigaoctets, voire plusieurs téraoctets de données.



Fig. 1.15 – Disque dur mécanique

Les ordinateurs récents sont de plus en plus équipés de SSD (Solid-State Drive) qui permettent de stocker des données tout comme les disques durs, mais leur conception est purement électronique et non plus mécanique. Ils sont donc plus résistant aux chocs et plus légers - et donc particulièrement adaptés aux ordinateurs portables - et beaucoup plus rapides. Ils ont une taille de plus en plus réduite et des gains de performance importants : temps d'accès réduits, meilleure bande passante que les disques durs traditionnels.

La fiabilité et les capacités des disques durs classiques pérennisent cependant leur utilisation.



Fig. 1.16 – Disque dur SSD

Sur certains ordinateurs portables d'entrée de gamme, le disque dur ou le SSD sont parfois remplacés par un stockage sous forme d'eMMC, solution peu coûteuse comme les cartes SD ou Multimedia Card.

L'avantage, en plus du coût, réside au niveau de l'encombrement et du poids, et convient donc aux ordinateurs portables de petite taille. Compte-tenu des performances, à prix et configuration équivalents, on privilégiera le SSD, puis l'eMMC et enfin le disque dur.

Le lecteur/graveur CD/DVD Un ordinateur peut encore aujourd'hui être équipé d'un graveur, vissé au boîtier, glissé dans un emplacement ouvert sur l'avant du PC, permettant ainsi l'ouverture du tiroir qui recevra le disque optique que l'on appelle plus communément CD (Compact Disc) ou DVD (Digital Versatile Disc). Il est connecté à la carte mère par un câble plat (nappe) IDE ou SATA.



Fig. 1.17 – Lecteur CD

1.6.2 Le CPU (Central Processing Unit)

Il s'agit du processeur de l'ordinateur. C'est le cœur de l'ordinateur, c'est à dire l'espace où va se dérouler l'ensemble des opérations et instructions de la machine. C'est un peu le «cerveau» de la machine. Le CPU va aller chercher les informations dans la ROM en passant par la RAM qui est donc essentielle pour le traitement du processeur. On parle d'Unité Centrale de Traitement en français. Le processeur sert à l'échange de données entre composants informatiques : disques durs – carte graphique – ROM – RAM. Il coordonne, interprète, calcule, exécute.

La puissance du CPU est caractérisée par son nombre de bits, 32 ou 64 bits aujourd'hui, et la fréquence de traitement de l'information qu'il assure caractérise la rapidité avec laquelle il traite les informations. Cette puissance de traitement des cycles CPU, qui est donc la puissance de l'ordinateur, représente la capacité d'un ordinateur à manipuler des données. La puissance de calcul et la rapidité de traitement se trouvent multipliées par le nombres de cœurs éventuellement présents sur la puce. Nombre de bits et fréquence de traitement sont donc deux paramètres essentiels, mais également le nombre de cœurs que le processeur comporte.

Le cœur du processeur est en fait une unité de traitement qui permet de lire des instructions pour effectuer des actions spécifiques. Ainsi, quelle que soit l'action que l'on souhaite effectuer sur la machine, elle est exécutée par le cœur, et s'il y a plusieurs cœurs, qui sont en fait des unités de traitement, on peut effectuer toutes les actions rapidement et en même temps.

Les principaux acteurs du marché sont Intel et AMD.



Fig. 1.18 – Différents types de microprocesseurs simple cœur et multicœurs

La carte mère

Une carte mère est le composant central de l'ordinateur. Elle est vissée au boîtier du PC, et possède les connecteurs (slots) pour accueillir des dizaines de composants et périphériques en plus des éléments indispensables décrits ici, et gérer les flux logiciels, chaque information envoyée ou reçue par le matériel ou un programme transitant par elle.

Elle intègre également la ROM sur laquelle est enregistrée le BIOS, petit programme gérant la configuration «de base» du matériel et se chargeant de faire le lien avec le système d'exploitation. Ces réglages sont conservés en mémoire même en l'absence de courant grâce au CMOS [11], alimenté par la pile de carte mère.

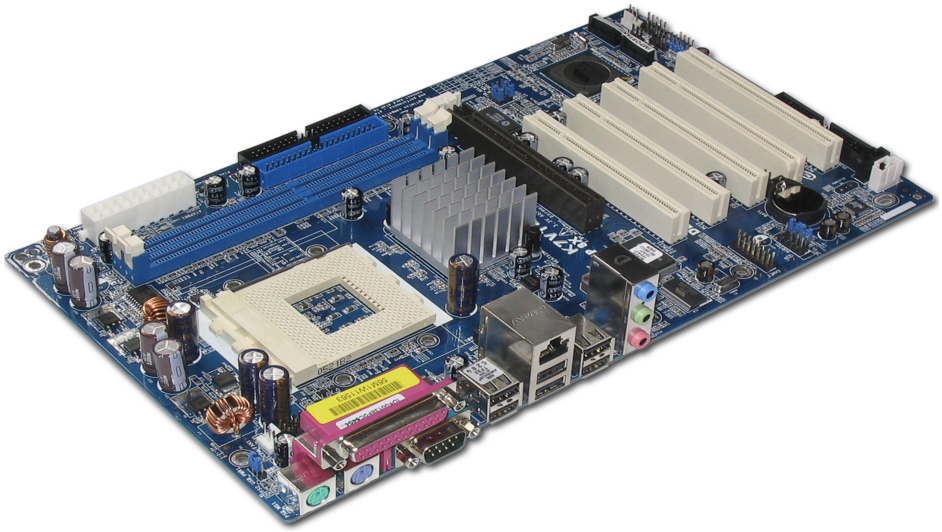


Fig. 1.19 – Carte mère

11. https://fr.wikipedia.org/wiki/Complementary_metal_oxide_semi-conductor

1.6.3 Les entrées-sorties

Un ordinateur traite de l'information au niveau de sa mémoire et de son processeur. Il récupère donc cette information via des ports d'entrée et redistribue une information après traitement via des ports de sortie. L'ensemble de cet environnement d'entrées-sorties constitue ce que l'on nomme les périphériques : clavier, écran, enceintes audio ou casque, imprimante, souris ou pad, disques externes, microphone, réseau ethernet ou wifi, etc. Certains périphériques sont par nature destinés uniquement à l'entrée de données (claviers et souris, microphones), tandis que d'autres s'occupent avant tout de la sortie (imprimantes, écrans non-tactiles) ; d'autres encore permettent à la fois l'entrée et la sortie de données (disques durs, disquettes, CD-ROM inscriptibles, clés usb).

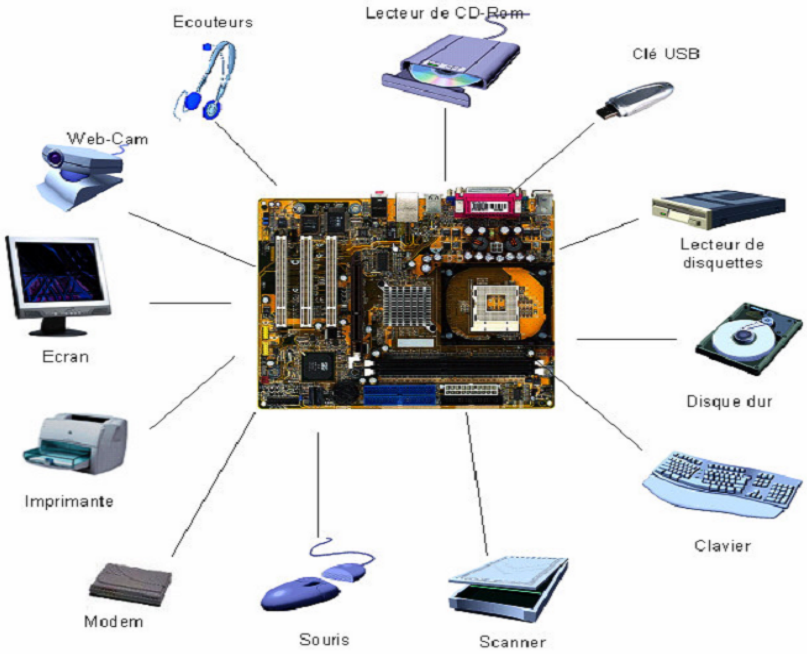


Fig. 1.20 – Unité centrale et périphériques

Interfaçage

Dans une description idéale, le processeur se connecte au bus et envoie sur le bus adresses, données et commandes au périphérique. Ensuite, le processeur va devoir attendre et rester connecté au bus tant que le périphérique n'a pas traité sa demande intégralement en lecture ou en écriture. Mais les périphériques sont lents et le processeur attend le périphérique… Pour résoudre ce problème, on intercale des registres d'interfaçage entre le processeur et les entrées-sorties. Une fois que le processeur a écrit les informations à transmettre dans ces registres, il peut faire autre chose, le registre se chargeant de maintenir et mémoriser les informations à transmettre. Les registres d'interfaçage sont surveillés régulièrement par le processeur pour voir si un périphérique lui a envoyé une information, mais le processeur peut utiliser quelques cycles pour faire son travail en attendant que le périphérique traite intégralement sa commande. Ces registres peuvent contenir des données ou des commandes, des valeurs numériques auxquelles le périphérique répond en effectuant un ensemble d'actions préprogrammées.

Les commandes sont traitées par un contrôleur de périphérique, qui va lire les commandes envoyées par le processeur, les interpréter, et piloter le périphérique de façon à faire ce qui est demandé. Le contrôleur de périphérique génère des signaux de commande qui déclencheront une action effectuée par le périphérique. Certains contrôleurs de périphérique peuvent permettre au processeur de communiquer avec plusieurs périphériques en même temps. C'est notamment le cas pour tout ce qui est des contrôleurs PCI, USB et autres. Certains périphériques, comme les disques IDE intègrent en leur sein ce contrôleur. Certains de ces contrôleurs intègrent un registre d'état, lisible par le processeur, qui contient des informations sur l'état du périphérique. Ils servent à signaler des erreurs de configuration ou des pannes touchant un périphérique.

Le système d'exploitation d'un ordinateur ne connait pas toujours le fonctionnement d'un périphérique ou de son contrôleur : il faut alors installer un programme qui va permettre la communication avec le périphérique, et qui va gérer transfert des données, adressage du périphérique, etc. Ce petit programme est appelé un *driver* ou pilote de périphérique.

1.6.4 Les bus

Un bus informatique est un dispositif de transmission de données partagé entre plusieurs composants d'un système informatique. Le bus informatique est la réunion des parties matérielles et immatérielles qui permet la transmission de données entre les composants de la machine. On distingue deux types de bus : le FSB (Front Side Bus), ou *bus système*, et le bus d'extension. Le premier permet au processeur de communiquer avec la mémoire vive, le second est une voie de liaison entre le processeur et les cartes d'extension. Des connecteurs d'extension présents sur la carte mère permettent d'y ajouter de nouveaux composants : cartes d'extension tels que carte son, carte d'acquisition vidéo, carte réseau, etc. Il existe différents types de bus d'extension : ISA, EISA, PCI, PCMCIA, VESA. [12] On se propose ici de décrire exclusivement les différents types de bus système : bus de données, d'adressage et de commande.

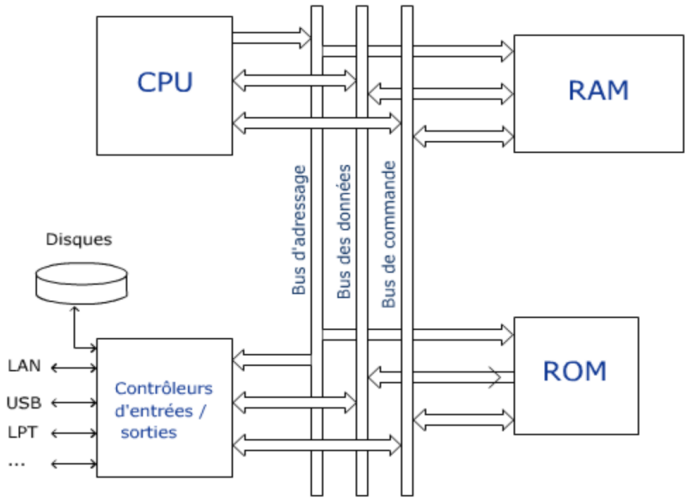


Fig. 1.21 – Schéma général d'un ordinateur

12. http://www.dicofr.com/cgi-bin/n.pl/dicofr/definition/20010101000612

Bus de données (Data Bus)

Le bus de données interconnecte le processeur, la mémoire centrale et les contrôleurs de périphériques et véhicule les données entre ces composants. Il est bidirectionnel (contrairement au bus d'adressage) puisque le processeur l'utilise pour lire et pour écrire en mémoire ou dans les entrées-sorties.

Le bus de données est commandé par le CPU, les autres composants y sont connectés à tour de rôle pour répondre aux commandes de lecture ou d'écriture du processeur.

Le débit des données véhiculées par ce bus dépend d'une part des vitesses de transmission ou plus exactement de la capacité des composants à saisir rapidement les signaux des bus et à y répondre aussi vite. La cadence de ces signaux est liée à fréquence de la carte mère.

La largeur du bus est le second critère qui va influencer le débit des transmissions des données. Plus le bus est large et plus important sera le nombre de données qui pourront être véhiculées simultanément. La largeur du bus de donnée peut être comparée au nombre de voies de circulation d'une autoroute. Elle dépend directement de la puissance du processeur.

NB : Les premiers microprocesseurs qui ne pouvaient traiter que 8 bits simultanément avaient un bus de données de 8 bits. Actuellement, les microprocesseurs traitent en général les données par mots de 32 bits mais le bus de donnée est plus large encore (64 bits) ce qui lui permet de véhiculer plus de données en parallèle.

Le bus d'adressage (Address Bus)

Le bus d'adressage (ou bus d'adresse, ou bus mémoire) reçoit du processeur les adresses des cellules mémoire et des entrées/sorties auxquelles il veut accéder. Chacun des conducteurs du bus d'adressage peut prendre deux états, 0 ou 1. L'adresse est donc le nombre binaire qui est véhiculé par ces lignes. La quantité d'adresses qui peuvent ainsi être formées est égale à deux exposant le nombre de bits d'adresse. Ce bus est physiquement constitué de câbles parallèles qui relient le processeur à la mémoire. La taille de ce bus ou sa largeur définissent le nombre de connexions parallèles et dépendent des caractéristiques du processeur et de la RAM. Chaque connexion transporte un bit, un bus de largeur 32 bits transporte 32 bits, ce qui permet de répertorier 232 adresses mémoire différentes (env. 4 Go). Les deux bus, d'adressage et de données, ne sont pas forcément de largeur identique.

NB : Le processeur 8088 qui équipait des premiers PC n'avait que 20 lignes d'adresse. Il pouvait donc accéder à 2 exposant 20 adresses différentes, soit 1 Mo. C'est pour cette raison que le système d'exploitation DOS qui date de cette époque ne peut pas adresser la totalité de la mémoire des systèmes actuels. Le nombre de lignes du bus d'adresse a ensuite évolué avec les différentes générations de processeurs.

Le bus de commande (Control Bus)

Le bus de commande ou bus de contrôle transporte les ordres et les signaux de synchronisation en provenance du CPU et à destination de l'ensemble des composants matériels via un ensemble de connexions physiques telles que des câbles ou des circuits imprimés. Il s'agit d'un bus bidirectionnel qui transmet également les signaux de réponse des éléments matériels. Le CPU utilise un des signaux pour indiquer le sens des transferts sur le bus de données (lecture ou écriture). Le bus de contrôle est constitué de lignes de contrôle qui envoient chacune un signal spécifique, tel que lecture, écriture et interruption. La plupart des microprocesseurs incluent des lignes d'horloge système, des lignes d'état et des lignes d'activation d'octets.

1.6.5 Autres composants matériels

L'alimentation

L'alimentation branchée sur le secteur transforme et fournit l'énergie nécessaire à la carte mère, mais l'alimentation est aussi directement reliée à certains composants tel que le lecteur/graveur de DVD ou le disque dur par exemple.

La transformation du courant cause une déperdition d'énergie thermique, un système de ventilation est donc installé dans le coffret de l'alimentation et expulse l'air via l'arrière du boîtier de l'ordinateur.

Une puissance de 400 watts est généralement suffisante pour les ordinateurs en «configuration bureautique» même si certaines alimentations peuvent atteindre les 1000 watts pour des configurations gourmandes en énergie (gaming par exemple).



Fig. 1.22 – Alimentation

La carte graphique

La carte graphique, bien que très importante pour certains usages, est placée en dernière position de cette liste car elle peut-être remplacée par un chipset intégré (jeu de circuit) directement à la carte mère. Toutefois, pour certaines applications et notamment les jeux, gros consommateurs de ressources graphiques, elle est indispensable. En prenant à sa charge la gestion de l'affichage, elle libère le processeur de cette fonction, traite elle-même les informations et utilise sa propre mémoire (voir accélération matérielle).

La carte graphique s'insère dans un connecteur de la carte mère : le port AGP ou le port PCI Express pour les plus récentes. Une fois connectée, les entrées et sorties de la carte sont accessibles par l'arrière du boîtier afin de fournir une image au système de visualisation (écran, TV, projecteur).



Fig. 1.23 – Carte graphique

1.7 TP Portes logiques

Dans ce chapitre, nous allons explorer les portes logiques NON, OU et ET.

- Créez chaque circuit comme demandé
- Quand le circuit est terminé, cliquez sur **Screenshot** (dans le menu à droite)
- La capture d'écran contient le code pour réouvrir le circuit dans l'éditeur Logic [13]
- Déposez vos captures de circuit sur Moodle

1.7.1 Transmission d'un signal

Dans ce premier exemple se trouvent une entrée (in) et une sortie (out). Les deux sont liées par un fil de transmission qui transmet un signal binaire identifié par une couleur :

- 0 (noir)
- 1 (jaune)

Avec le menu contextuel (clic droit sur le fil), vous pouvez changer la couleur du fil ainsi que son délai de propagation.

- Ajoutez un deuxième fil avec un délai de propagation de 100 ms (rapide)
- Ajoutez un troisième fil avec un délai de propagation de 10 ms (instantané)
- Ajoutez un point intermédiaire au milieu du fil et déplacez-le un peu
- Mettez la couleur du fil en rouge



1.7.2 Commutateur/poussoir

Les entrées ont deux modes que vous pouvez changer avec le menu contextuel :

- commutateur : bascule entre l'état 0 et 1
- poussoir : garde la valeur 1 seulement pendant qu'il est appuyé

Changez la deuxième entrée en mode **poussoir** et augmentez le délai de propagation du fil à 1000 ms. Vous verrez alors des paquets d'informations se propager le long du fil.

Vous pouvez ajouter des noms aux entrées sorties.

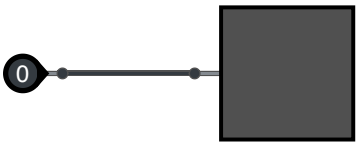
- Ajoutez une entrée *lumière*, configurez en commutateur, et reliez-la à une sortie *lampe*
- Ajoutez une entrée *sonnerie*, configurez en poussoir, et reliez-la à une sortie *alarme*

13. https://logic.modulo-info.ch/



1.7.3 Feu de circulation

— Cliquez sur l'entrée pour basculer l'état entre 0 et 1
— Ajoutez deux autres segments carrés pour compléter un feu de circulation.
— Changez l'affichage en grand carré
— Changez les couleurs en jaune et vert
— Ajoutez les noms *rouge, jaune, vert*.

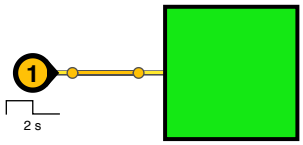


1.7.4 Lampe clignotante

L'entrée horloge (clock) produit un signal qui alterne entre 0 et 1.

— Ajoutez une deuxième horloge
— Ajoutez une deuxième lampe (grand segment carré)
— Configurez l'horloge pour une période de 1 seconde

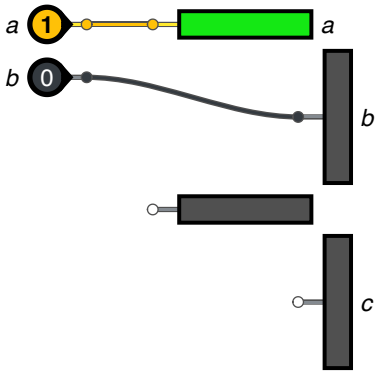
Avec le bouton **Pause** vous pouvez arrêter l'horloge.



1.7.5 Affichage à 7 segments

Les affichages à 7 segments permettent d'afficher des chiffres à l'aide de 7 diodes lumineuses (LED).

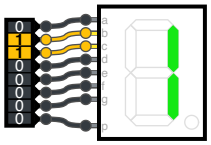
- Ajoutez les entrées et les lampes qui manquent pour compléter cet affichage à 7 segments.
- Tournez la barre avec *Affichage* > *Barre verticale*
- Ajoutez les étiquettes a-g
- Ajoutez 7 entrées et ajoutez les étiquettes a-g
- Affichez un nombre entre 0 et 9



1.7.6 Affichage à 2 chiffres

Les 7 diodes lumineuses (LED) permettent d'afficher les chiffres de 0 à 9. L'état des lampes (a-g) ainsi que du point décimal § est déterminé par 8 bits.

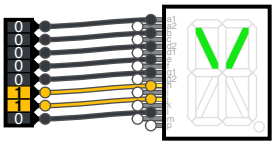
- Ajoutez un deuxième bloc affichage à 7 segments
- Ajoutez une entrée octet
- Connectez les deux automatiquement en alignant les broches
- Configurez les entrées pour afficher le nombre 42



1.7.7 Affichage à 16 segments

Les affichages à 16 segments permettent d'afficher aussi les lettres de l'alphabet, ainsi que des symboles de ponctuation.

- Configurez l'affichage pour afficher la lettre Y
- Ajoutez un deuxième affichage à 16 segments
- Ajoutez deux entrées 8-bits et connectez-les.
- Affichez la première lettre de votre nom



1.7.8 Porte NON

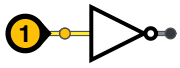
La porte NON inverse un signal. Montrez sa table de vérité.

- Mettez la première entrée à 0 et ajoutez une sortie
- Ajoutez une deuxième porte NON avec l'entrée à 1

Montrez l'effet de multiples portes NON.

- Mettez deux portes NON en série, ajoutez une entrée et une sortie
- Mettez trois portes NON en série, ajoutez une entrée et une sortie

table de vérité

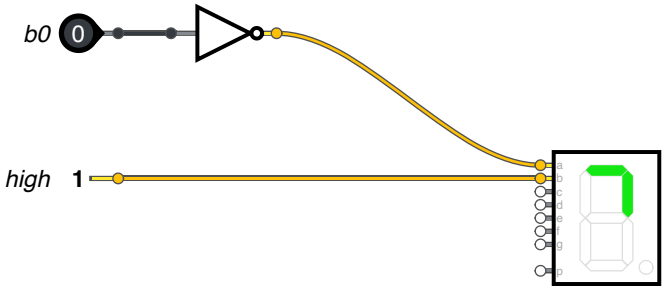


multiples portes NON

1.7.9 Afficher 0 et 1

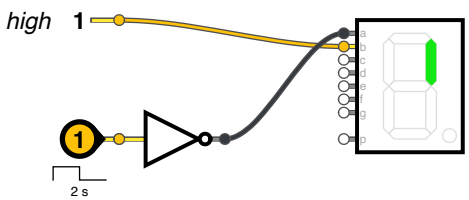
La porte NON inverse un signal.

- Ajoutez les connexions pour afficher 0 ou 1 selon le signal sur l'entrée **b0**.



1.7.10 Alterner 0 et 1

Utilisez une horloge et une porte NON pour afficher les chiffres 0 et 1 en alternance.



1.7.11 Porte OU

Une porte OU donne une sortie 1 si **au moins une** des entrées est à 1.

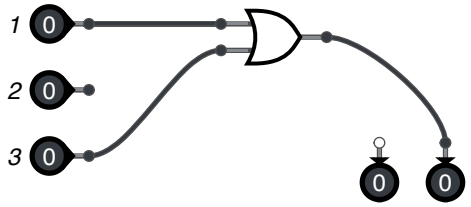
- Montrez la table de vérité pour la porte OU.
- Ajoutez 3 portes OU et mettez les entrées à 01, 10, et 11
- Créez une porte OU avec 3 entrées



1.7.12 Décodeur de clavier

Complétez le circuit pour un décodeur qui a le comportement suivant :

- bouton 1 appuyé produit la sortie binaire 01
- bouton 2 appuyé produit la sortie binaire 10
- bouton 3 appuyé produit la sortie binaire 11

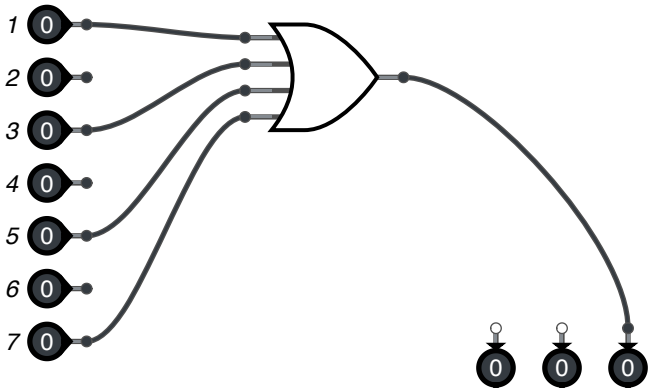


1.7.13 Clavier numérique

Quand on appuie sur une touche d'une calculette électronique, en interne l'action d'appuyer le bouton est transformée en une représentation binaire de la touche appuyée.

Complétez le circuit pour un décodeur qui a le comportement suivant :

- bouton 1 appuyé produit la sortie binaire 001
- bouton 2 appuyé produit la sortie binaire 010
- bouton 3 appuyé produit la sortie binaire 011
- bouton 4 appuyé produit la sortie binaire 100
- bouton 5 appuyé produit la sortie binaire 101
- bouton 6 appuyé produit la sortie binaire 110
- bouton 7 appuyé produit la sortie binaire 111



1.7.14 Porte ET

Une porte ET donne une sortie 1 seulement si **toutes** les entrées sont à 1.

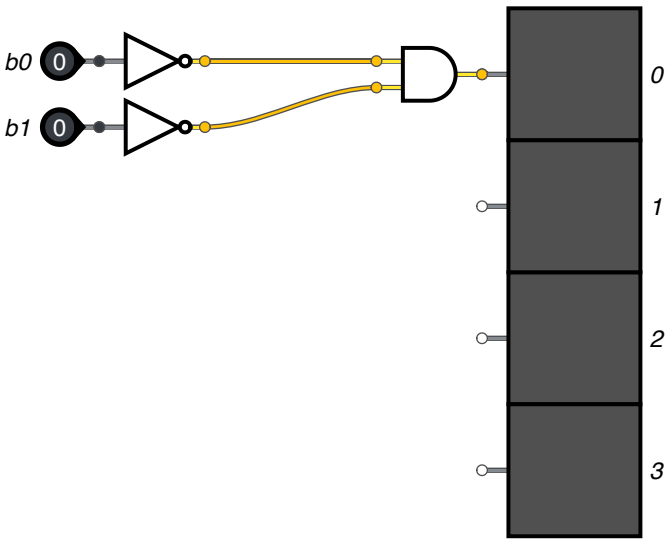
- Montrez la table de vérité pour la porte ET.
- Ajoutez 3 portes ET et mettez les entrées à 01, 10, et 11
- Créez une porte ET avec 3 entrées



1.7.15 Décodeur binaire

Complétez le circuit pour créer un décodeur binaire. Chaque combinaison des deux entrées binaires allume une et une seule des 4 lampes.

- 00 allume seulement lampe 0
- 01 allume seulement lampe 1
- 10 allume seulement lampe 2
- 11 allume seulement lampe 3



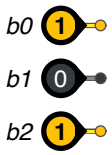
1.7.16 Décodeur de dé

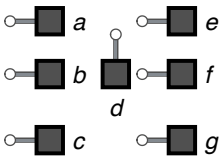
Un dé de jeu peut afficher les nombres 1 à 6 à l'aide de 7 lampes.Plusieurs lampes s'allument par paire. Voici la table de vérité.

b2	b1	b0	valeur	a,g	b,f	c,e	d
0	0	0		0	0	0	0
0	0	1	1	0	0	0	1
0	1	0	2	1	0	0	0
0	1	1	3	1	0	0	1
1	0	0	4	1	0	1	0
1	0	1	5	1	0	1	1
1	1	0	6	1	1	1	0
1	1	1		1	1	1	1

Utilisez des portes logiques OU et ET pour créer le circuit de décodage pour afficher les lampes qui correspondent aux nombres 1 à 6.

Le nombre binaire $b_2 b_1 b_0$ doit allumer les lampes a-g pour afficher ce nombre dans la façon d'un dé à jeu standard.





1.7.17 Décoder 0 à 3

Le tableau ci-dessous montre les segments à allumer pour afficher les nombres 0 à 3 d'un affichage à 7 segments.

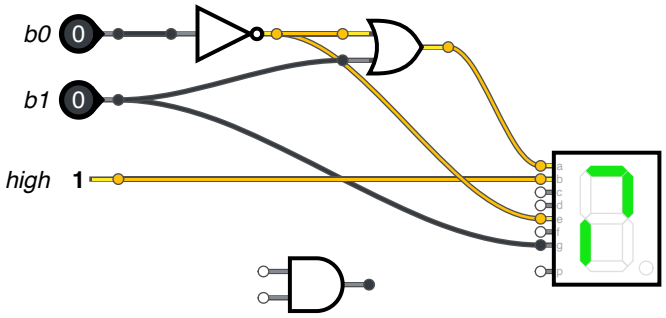
dec	b1	b0	a	b	c	d	e	f	g
0	0	0	1	1	1	1	1	1	0
1	0	1	0	1	1	0	0	0	0
2	1	0	1	1	0	1	1	0	1
3	1	1	1	1	1	1	0	0	1

Ajoutez des portes NON, OR, et AND pour compléter le circuit

Astuce - Essayez de trouver le circuit logique pour chaque colonne. C'est-à-dire il faut trouver le circuit pour allumer le segment. Par exemple pour le segment a vous avez la table de vérité suivante.

b1	b0	a
0	0	1
0	1	0
1	0	1
1	1	1

Certaines colonnes sont identiques.Donc vous pouvez utiliser le même signal.



Pour ce dernier circuit, faites une capture d'écran pour chacune des 4 conditions.

1.8 TP Additionneur

Dans cette section, nous allons explorer d'abord la porte **ou exclusif** (OU-X), qui nous sert à construire un **additionneur simple**. Nous modifions l'additionneur simple pour en créer un **additionneur complet** qui prend en compte la retenue. Cet additionneur 1 bit nous sert à construire des circuits pour créer d'autres opérations telles que :

— addition
— soustraction
— incrémentation
— décrémentation

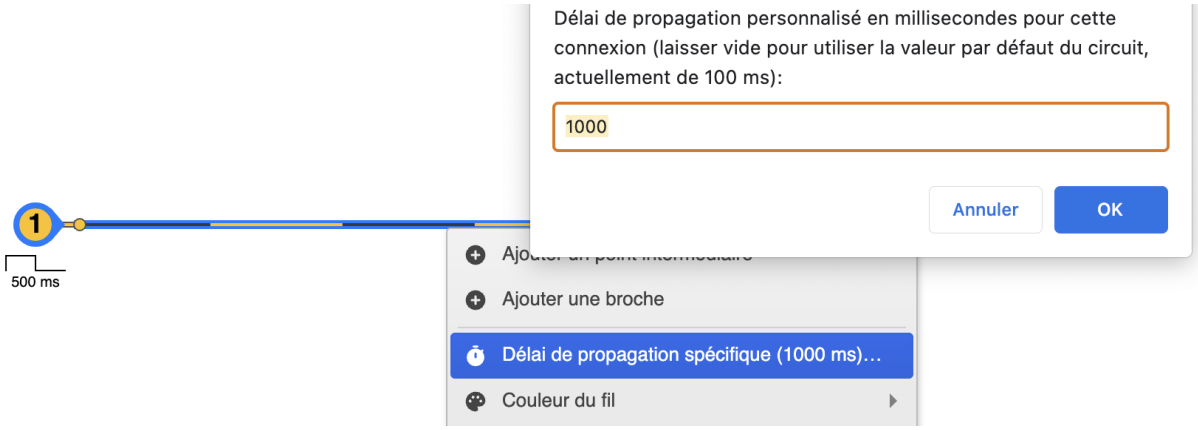
1.8.1 Clock et fréquence

L'entrée horloge (clock) produit un signal qui alterne entre 0 et 1. L'horloge ci-dessous a une période de 500 ms. Le délai de transmission a été mis à 1000 ms. Ceci fait apparaitre 2 cycles du signal (en jaune). On peut donc visualiser la fréquence du signal qui est de 2 Hz.

— Ajoutez une deuxième horloge avec une période de 250 ms.
— Visualisez cette fréquence avec un délai de propagation de 1000 ms.
— Quelle est sa fréquence ? Mettez-la comme étiquette.
— Faites la même chose avec une troisième horloge qui a une période de 100 ms.



Rappel : pour choisir le délai de propagation, cliquez avec le bouton droit sur le fil et sélectionnez le menu contextuel *Délai de propagation spécifique…*. Mettez-le à 1000.



1.8.2 Porte OU-X

Une porte OU-X (ou exclusif) avec 2 entrées donne une sortie 1 si **exactement une** des entrées est 1.

- Montrez la table de vérité pour la porte OU-X.
- Ajoutez 3 portes OU-X et mettez les entrées à 01, 10, et 11
- Créez une porte OU-X avec 3 entrées et observez son comportement

Comment se comporte une porte OU-X avec plus que 2 entrées ?



1.8.3 Construire un OU-X

Comment peut-on construire un circuit OU-X avec des portes de base (NON, OU, ET) ? Regardons d'abord la table de vérité.

a	b	OU-X
0	0	0
0	1	1
1	0	1
1	1	0

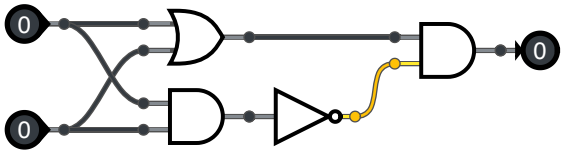
Le circuit ci-dessous représente une porte OU exclusive (OU-X). Mais il y a multiples façons de créer un circuit logique spécifique à partir des éléments de base.

Créez une deuxième façon pour obtenir une porte OU exclusive en partant de l'observation :

```
(a and not b) or (b and not a)
```

Utilisez donc :

- 2 portes NON
- 2 portes ET
- 1 porte OU

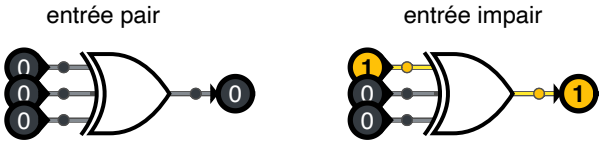


1.8.4 Détecteur de parité

Une porte ou exclusif est un détecteur de parité (pair/impair). La sortie d'une porte ou exclusif est 1 si le nombre des entrées actives est impair.

Ajoutez encore 6 portes OU-X et complétez la table de vérité pour les 8 combinaisons possibles :

— pair : 000, 011, 101, 110
— impair : 001, 010, 100, 111



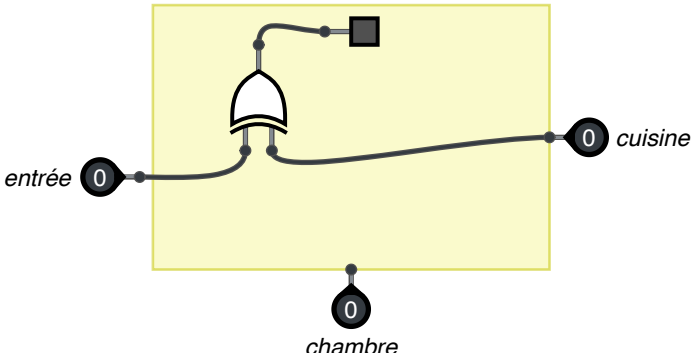
Pour détecter si un nombre d'entrées est pair, il suffit d'ajouter un NON à la sortie du OU-X. On appelle ce circuit un NON-OU-X.

1.8.5 Multiples commutateurs

La porte OU-X permet d'allumer et éteindre une lampe avec des commutateurs multiples.

Dans le schéma ci-dessous, on peut allumer la lumière dans une pièce à partir de la porte d'entrée et de la cuisine.

Ajoutez un circuit pour qu'on puisse également l'allumer depuis la chambre.



1.8.6 Addition binaire

Nous avons maintenant tous les éléments pour construire un additionneur binaire. Rappelons-nous que l'addition binaire est très simple.

A	B	A+B	C	S
0	0	0	0	0
0	1	1	0	1
1	0	1	0	1
1	1	2	1	0

Le résultat `A+B` peut être 0, 1 ou 2. Nous avons besoin de deux bits pour représenter le résultat :

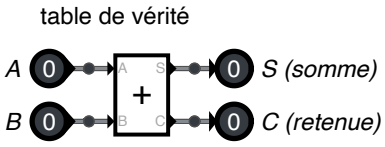
- le bit de somme `S`
- le bit de retenue `C` (*carry* en anglais)

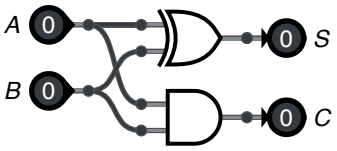
En regardant la table de vérité, on constate que :

- la somme `S` est exprimée par la fonction OU-X
- la retenue `C` est exprimée par la fonction ET

Vous trouvez le circuit ci-dessous à droite. Vérifiez sa fonction en cliquant sur ses entrées.

Ajoutez encore 3 demi-additionneurs et montrez la table de vérité pour les 4 conditions d'entrée : 00, 01, 10, 11.





1.8.7 Additionneur complet

Dans le cas général de l'addition, nous n'additionnons pas deux bits, mais deux nombres à plusieurs bits. Voici l'addition en colonne de deux nombres 4 bits (3+11=14).

```
 0011
+1011
-----
 1110
```

Pour être explicite, nous introduisons une ligne supplémentaire qui représente la retenue (C = carry).

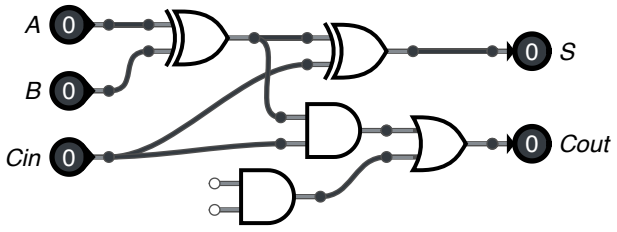
```
 0110 (retenue)
 0011
+1011
-----
 1110
```

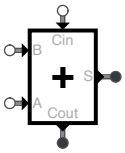
L'additionneur de 2 bits de la section précédente n'est plus suffisant. Pour le cas général, nous avons besoin d'un additionneur qui additionne 3 bits. Il faut tenir compte de la retenue (`Cin`), qu'il faut inclure dans l'addition. Voici donc la table de vérité pour un additionneur complet.

Cin	A	B	Cin+A+B	Cout	S
0	0	0	0	0	0
0	0	1	1	0	1
0	1	0	1	0	1
0	1	1	2	1	0
1	0	0	1	0	1
1	0	1	2	1	0
1	1	0	2	1	0
1	1	1	3	1	1

Regardez les colonnes et essayez de comprendre avec quelles portes on pourrait le construire. Vous constatez que la colonne `S` représente la parité. On pourra donc la construire avec des portes OU-X.

— Ajoutez les deux fils qui manquent à l'entrée de la porte ET pour que le circuit produise le signal `Cout` et se comporte comme un additionneur complet.
— Ajoutez des entrées et sorties au bloc de l'additionneur complet et vérifiez son fonctionnement.



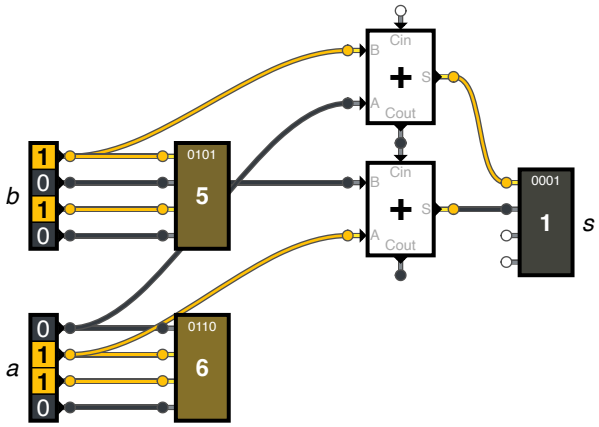


1.8.8 Additionneur 4 bits

Pour additionner deux nombres 4-bits (quartets) nous avons besoin de 4 additionneurs complets. Chaque sortie `Cout` est liée à la l'entrée `Cin` de l'additionneur suivant.

Pour additionner **a** et **b** vous devez additionner les bits correspondants : a0+b0, a1+b1, etc.

— Ajoutez les circuits manquants pour additionner deux nombres 4-bits.
— Montrez l'addition de 7+5 dont le résultat devrait être 12.

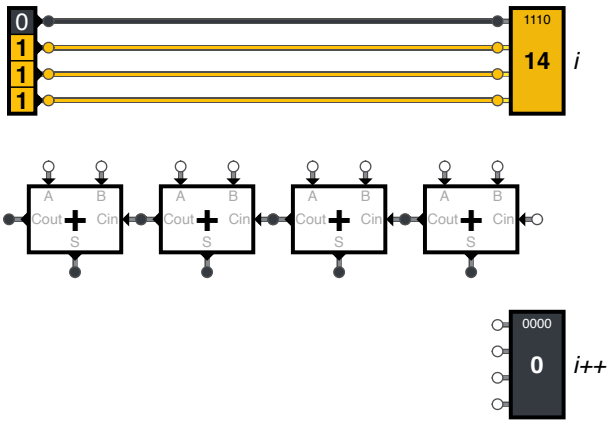


1.8.9 Incrémenter (`i++`)

Additionner 1 à un nombre binaire est une opération très fréquente. Elle est utilisée pour incrémenter le compteur de programme `pc` (program counter), pour pointer à la prochaine instruction.

Complétez le circuit pour incrémenter la variable `i`. Dans beaucoup de langages de programmation, une variable incrémentée est désignée par `i++`.En Python nous écrivons `i = i + 1`.

D'ailleurs le nom du langage de programmation C++ est une référence à cet opérateur d'incrémentation.

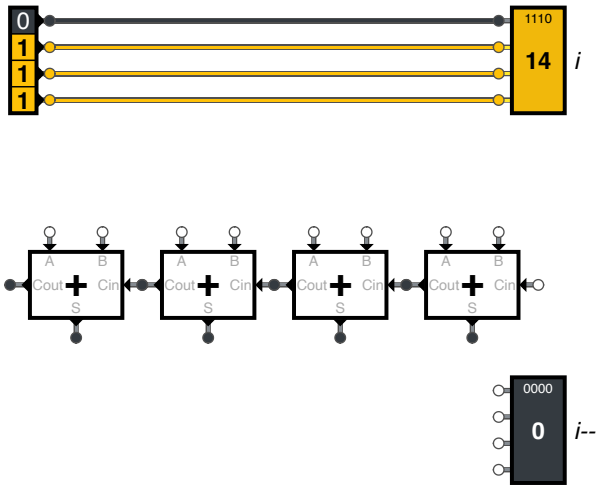


1.8.10 Décrémenter (`i--`)

Soustraire 1 à un nombre binaire est une opération très fréquente. Elle est utilisée pour décrémenter un compteur de boucle `i`, un pointeur de pile `sp` (stack pointer), ou un pointeur `p` vers les adresses de la mémoire.

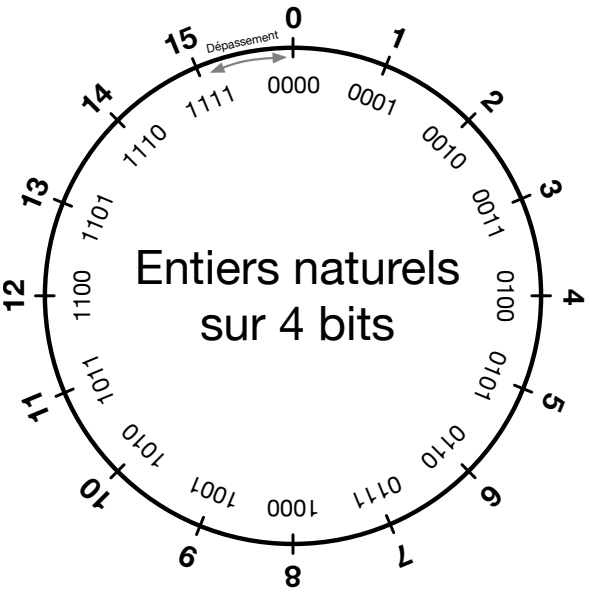
Complétez le circuit pour décrémenter la variable `i`. Dans beaucoup de langages de programmation, une variable incrémentée est désignée par `i--`.En Python nous écrivons `i = i - 1`.

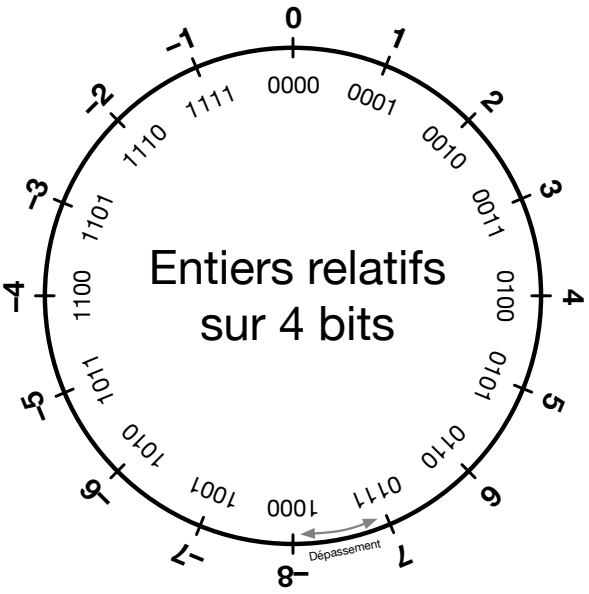
Astuce : pour décrémenter la valeur `i` il suffit d'additionner `1111` qui représente la valeur -1 en format signé.



1.8.11 Changer de signe (`-i`)

Les nombres signés sont représentés avec le format *complément à deux*. Pour un nombre 4-bits, ceci nous donne une plage de -8 à +7 pour des entiers relatifs, et une plage de 0 à 15 pour des entiers naturels. Nous constatons que la plage signée n'est pas symétrique : le côté négatif compte un nombre en plus.



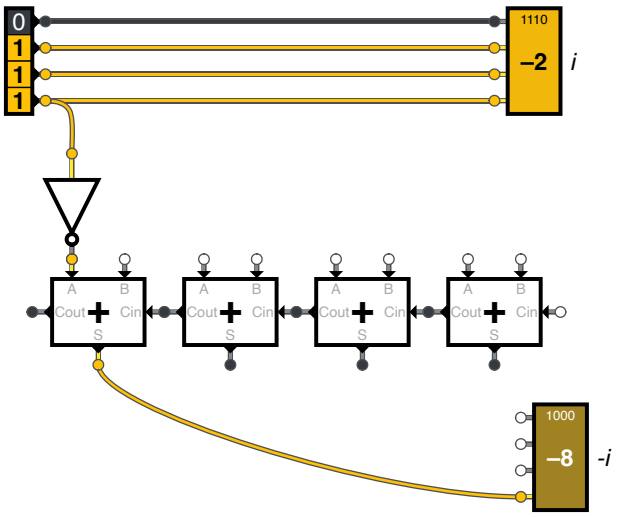


L'opération pour trouver le nombre négatif est : inverser tous les bits (symbolisé par ~) et additionner 1.Mathématiquement nous pouvons exprimer cette opération comme :

`-i = ~i + 1`

Par exemple, pour obtenir la représentation binaire de -1 nous inversons `0001`, ce qui donne `1110` et nous additionnons 1, ce qui donne `1111`.

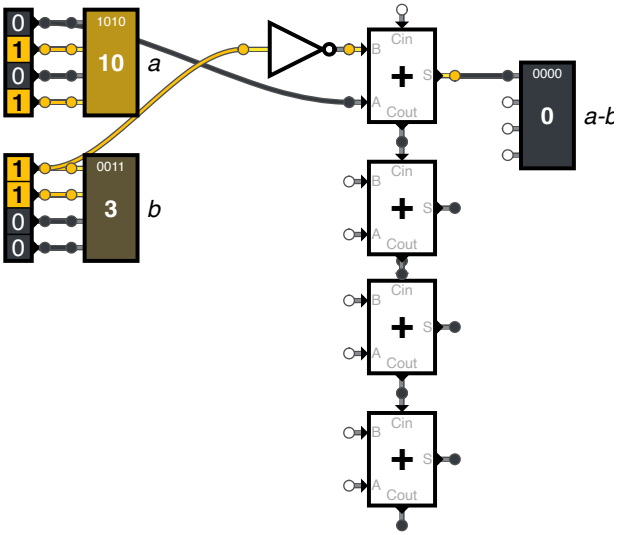
Complétez le circuit pour inverser le signe de la variable `i` et obtenir son négatif `-i`



1.8.12 Soustraction (a-b)

Pour soustraire deux nombres `a-b` il suffit d'additionner le nombre négatif du deuxième (`-b`). Ce nombre négatif peut être obtenu en inversant tous les bits et additionner 1.Donc `-b = ~b + 1`.

Complétez le circuit pour soustraire `a-b`. Le résultat de 10-3 devrait être 7.

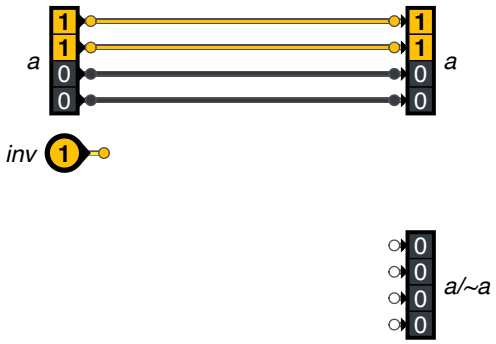


1.8.13 Inversion commutée

L'inverseur commuté permet d'inverser tous les 4 bits d'un nombre avec un signal de contrôle `inv` :

- pour `inv = 0` la sortie est inchangée (`a`)
- pour `inv = 1` la sortie est inversée (`~a`)

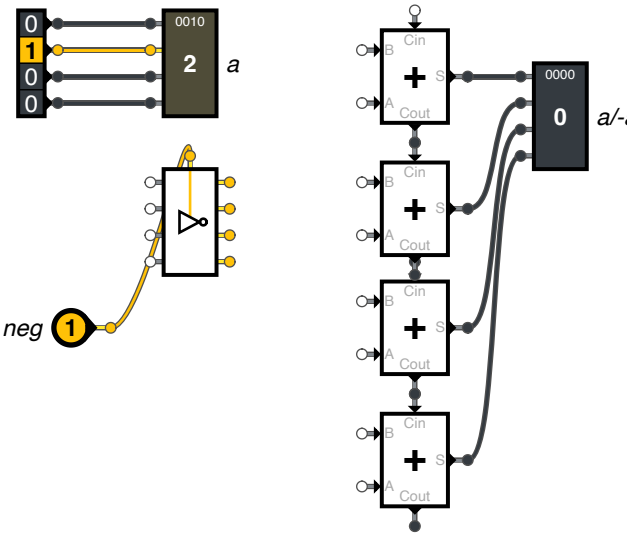
Ajoutez un inverseur commuté pour obtenir `~a` ou `a` selon l'état du sélecteur.



1.8.14 Négation commutée

Complétez le circuit pour pouvoir obtenir `-a` ou `a` selon l'état du sélecteur **neg** :

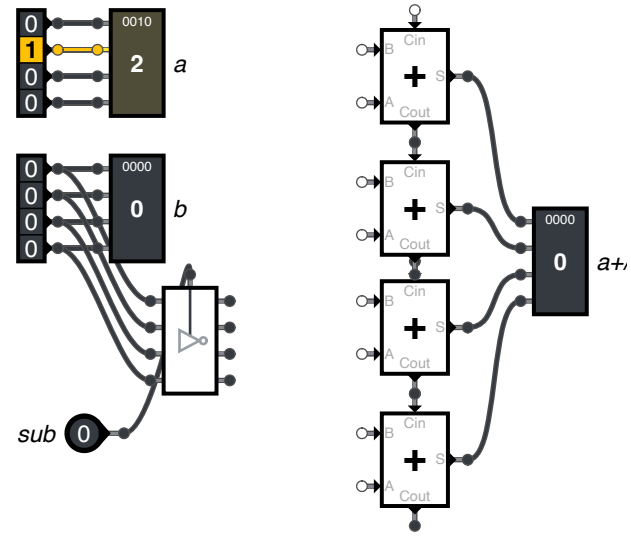
— pour `neg` = `0` la sortie est inchangée (`a`)
— pour `neg` = 1 la sortie change de signe (`-a`)



1.8.15 Soustraction commutée

Complétez le circuit pour pouvoir obtenir une opération différente selon l'état du sélecteur **sub** :

— pour `sub` = 1 les opérandes sont soustraits (`a-b`)
— pour `sub` = `0` les opérandes sont additionnés (`a+b`)



1.8.16 Les fanions (flags)

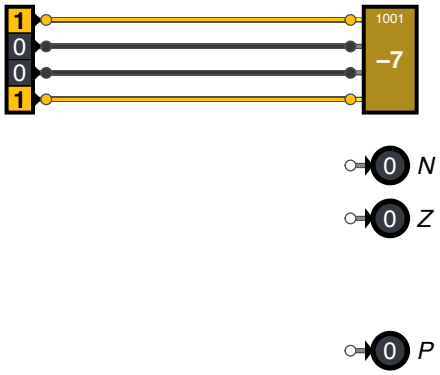
Les fanions (flag) sont des signaux qui caractérisent un nombre.

- N pour indiquer que le nombre est négatif
- Z pour indiquer que le nombre est zéro
- P pour indiquer que le nombre de bits à 1 est pair

Par exemple pour le nombre `1001` (-7) on aura `N=1`, `Z=0` et `P=1`.

> **Prudence :** Faites attention à la différence entre la **parité du nombre** et la **parité des bits**.
>
> - la parité du nombre est exprimée par le bit de poids faible (b0),
> - la parité du nombre des bits est obtenue avec une opération OU-X (ou exclusif).

Complétez le circuit pour correctement afficher les fanions N, Z et P.



1.9 TP ALU

L'unité arithmétique et logique (ALU) permet de choisir parmi un certain nombre d'opérations. Nous allons voir comment une ALU peut choisir entre différentes opérations (ET, OU, addition, soustraction) à l'aide d'un **multiplexeur**.

Nous allons découvrir comment des décalages et additions successives peuvent constituer une **multiplication** ou une **division**.

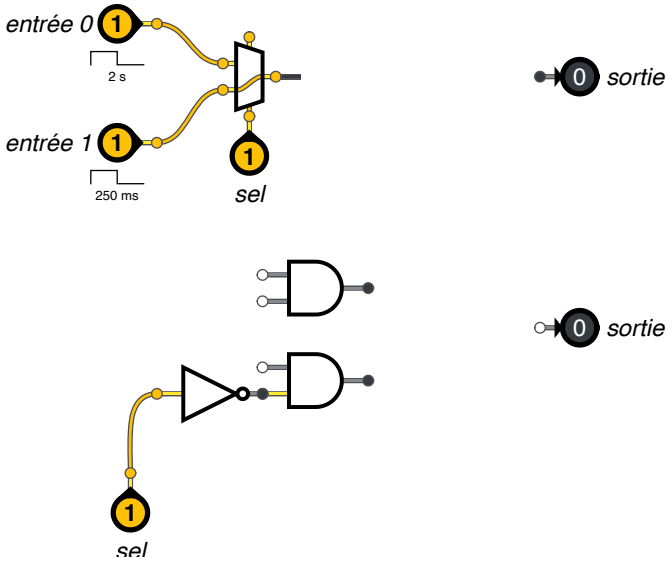
Finalement, un **registre** permet de mémoriser les opérandes du calcul. Un registre particulier appelé **accumulateur** permet de faire des additions successives et *accumuler* une somme courante.

1.9.1 Sélectionneur

L'entrée **sel** du sélectionneur permet de choisir entre deux signaux d'entrée :

- entrée 0 : signal lent (période de 2 s)
- entrée 1 : signal rapide (période de 250 ms)

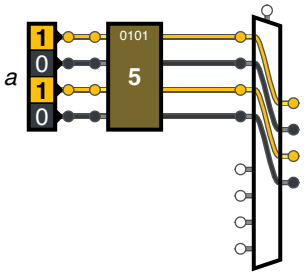
Recréez un tel sélecteur avec des portes NON, ET, OU.



1.9.2 Multiplexeur

Le multiplexeur (MUX) permet de choisir entre deux signaux 4-bits nommés a et b. Ajoutez les éléments qui manquent.

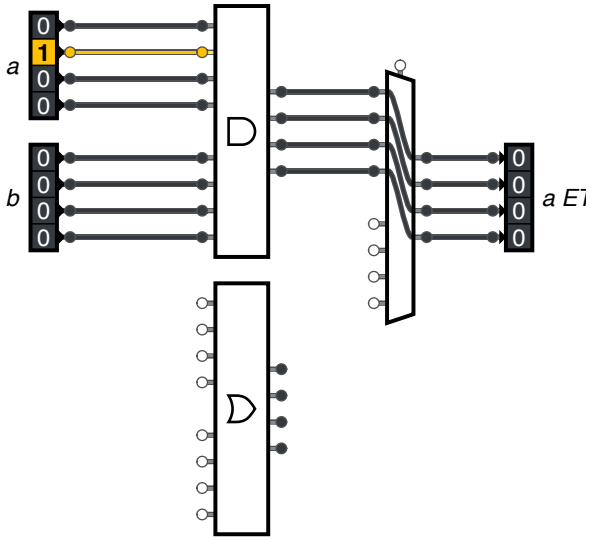
- Ajoutez une deuxième entrée 4-bits avec un affichage.
- Ajoutez un décodeur et affichage à 7 segments.



1.9.3 Sélection d'opérations

Complétez le circuit qui permet de sélectionner entre les deux opérations `a ET b` et `a OU b`.

- Connectez a et b aux entrées des 4 portes OU,
- Ajoutez une sorie 4-bits pour afficher le résultat des opérations logiques
- Ajoutez une entrée de sélection pour le multiplexeur.

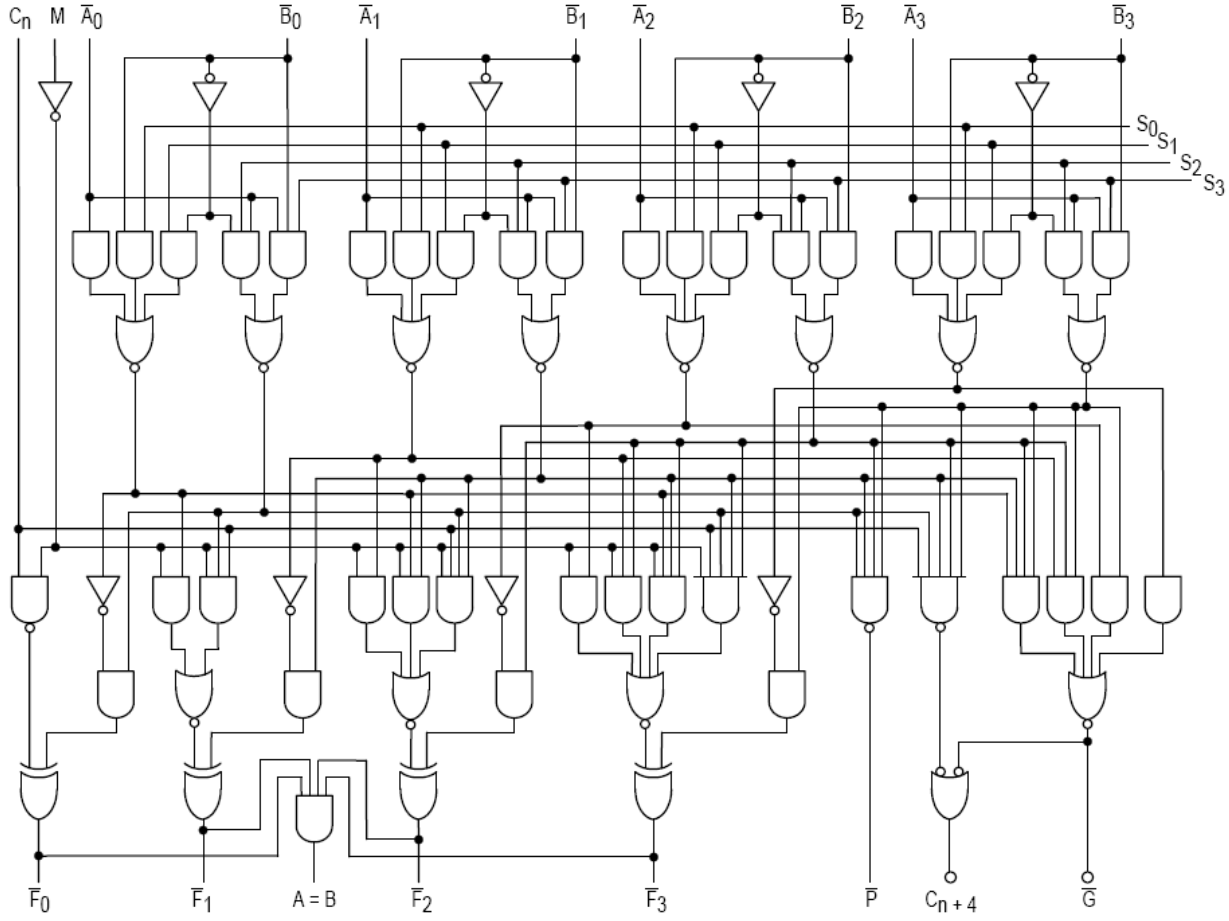


Un clic droit sur la porte quadruple permet de choisir son type (AND, OR, XOR, NAND, NOR, XNOR).

1.9.4 ALU

L'unité arithmétique et logique, ALU (arithmetic and logic unit), est la partie de l'ordinateur qui effectue les différents calculs arithmétiques et logiques.

Ci-dessous vous pouvez voir les circuits logiques d'une ALU 4-bits très utilisée dans les années 60 et 70, le modèle 74181.



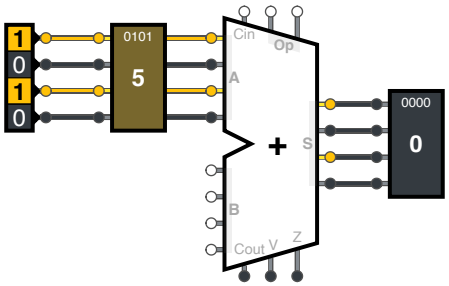
L'ALU dont nous disposons peut effectuer 4 opérations :

- addition (00)
- soustraction (01)
- OU logique (10)
- ET logique (11)

Avec l'ALU ci-dessous, ajoutez

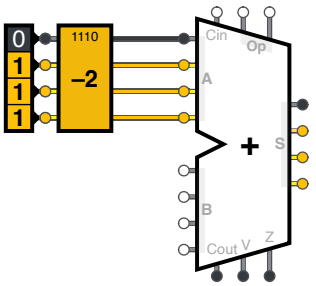
- la deuxième entrée (B) avec un bloc d'affichage 4 bits
- un bloc d'affichage 4 bits pour la sortie (S)
- les 3 entrées (Cin, Op0, Op1)
- les 3 sorties (Cout, V, Z)

Ensuite, testez les 4 opérations. Montrez une soustraction.



1.9.5 Addition signée

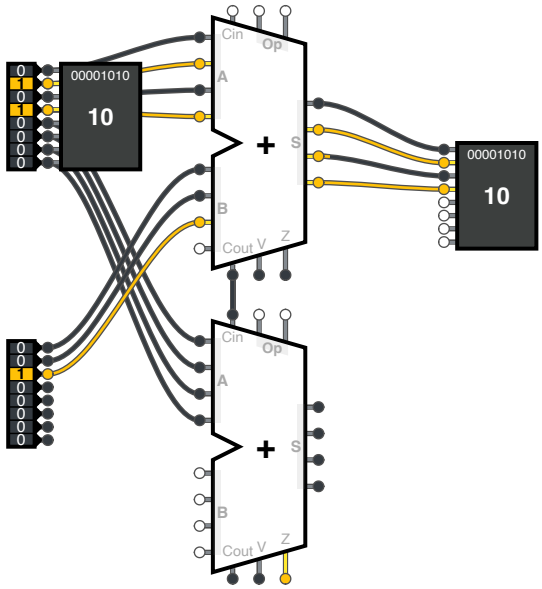
Interprétez les nombres binaires comme des nombres signés. Vous pouvez configurer l'afficheur 4 bits avec son menu contextuel. Complétez l'additionneur 4-bit et montrez que l'addition de -2 et -3 donne bien -5.



1.9.6 Addition 8 bits

Pour additionner un nombre à 8-bits, il faut combiner deux ALU 4-bits. Dans ce cas il faut connecter Cout de la première ALU avec Cin de la deuxième.

— Complétez le circuit pour afficher l'addition de deux nombres binaires 8-bits.
— Ajoutez une entrée pour soustraire deux nombres.
— Montrez une soustraction correcte, dont le résultat est plus grand que 30



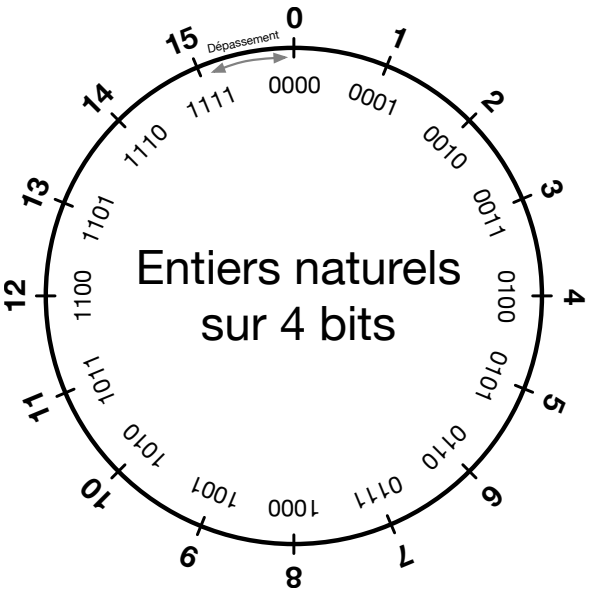
1.9.7 Carry et Overflow (V)

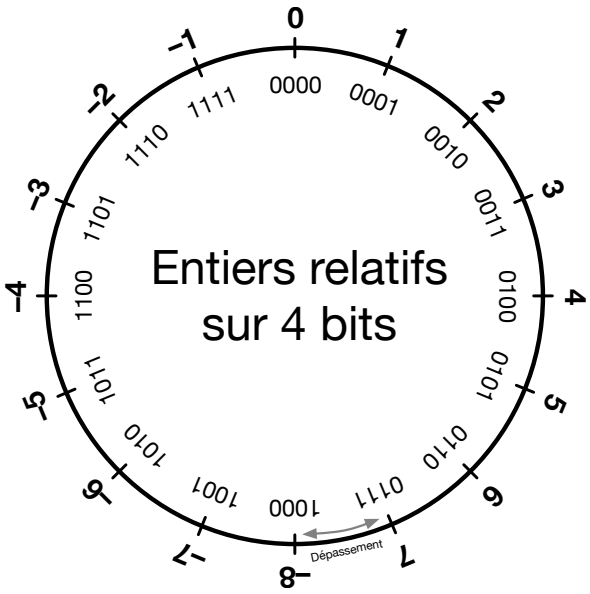
L'ALU possède deux sorties pour indiquer un dépassement de plage de résultat. Le résultat affiché est alors décalé de 16. Ce cas est signalé par l'ALU à l'aide de deux signaux de sortie spéciaux.

— C (carry) signale un dépassement pour des nombres non signés,

— V (overflow) signale un dépassement pour des nombres signés.

L'addition de deux nombres naturels (0 à 15) peut produire un résultat de 0 à 30.L'addition de deux nombres relatifs (-8 à 7) peut produire un résultat de -16 à 14.Dans certains cas on aura donc besoin de 5 bits pour représenter correctement le résultat.





Pour les nombres non signés (première ALU)

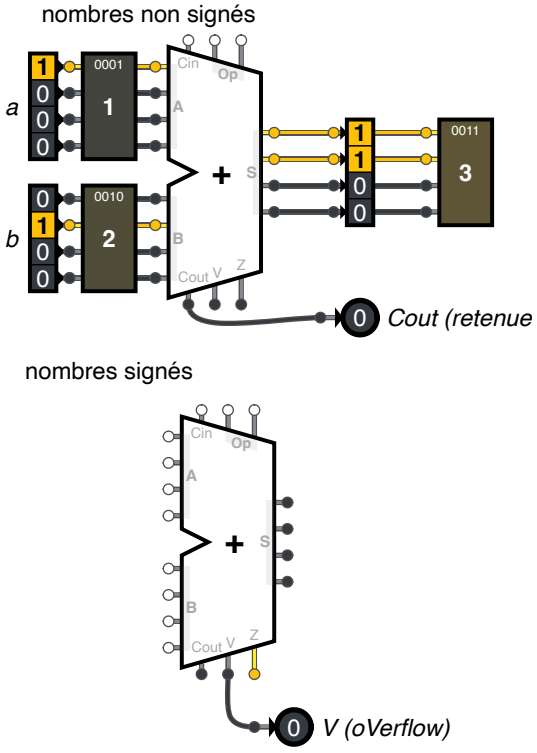
— Choisissez des valeurs a et b qui produisent un dépassement (C = 1), et donc un affichage incorrect sur une sortie de 4 bits.

— Ajoutez en plus un affichage 8 bits et connectez-y les 4 bits de S et C comme 5e bit, pour afficher le résultat correct.

Pour les nombres signés (deuxième ALU)

— Ajoutez 2 entrées 4 bits et 1 sortie 4 bits.
— Ajoutez 3 affichages 4 bits configurés (via menu contextuel) en nombres signés.
— Choisissez des valeurs a et b négatives qui produisent un dépassement (V = 1).
— Ajoutez un affichage 8 bits configuré (via le menu contextuel) en nombres signés, et connectez-y les 4 bits de S et C comme 5e bit.

Attention : Dans ce cas vous devez connecter C également avec les 3 autres bits (b5-b7) pour faire une propagation du bit de signe et traiter correctement le cas des nombres négatifs.



1.9.8 Multiplier 1x1 bit

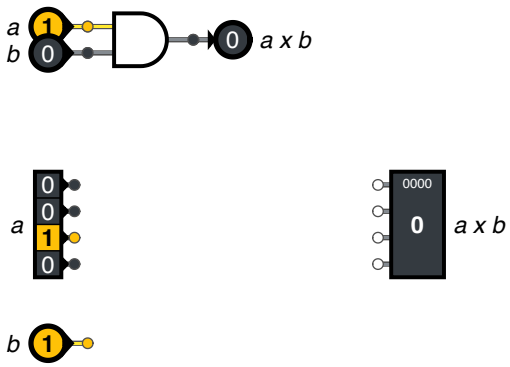
Les règles de la multiplication 1-bit sont très simples. Voici la table de vérité.

a	b	a x b
0	0	0
0	1	0
1	0	0
1	1	1

On voit tout de suite que ceci correspond à la porte ET. Dans l'exemple si dessous vous voyez une porte ET pour multiplier a et b, les deux ayant juste 1 bit.

— Vérifiez le bon fonctionnement du multiplicateur 1-bit
— Ensuite, utilisez 4 portes ET pour créer un multiplicateur **a** (4-bits) fois **b** (1-bit).

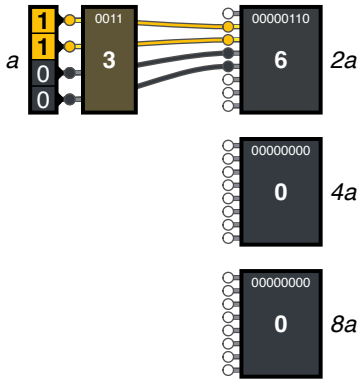
— Basculez b entre 0 et 1 pour vérifier si votre circuit fonctionne correctement.



1.9.9 Multiplier par 2, 4, 8

La multiplication par une puissance de 2 est facile. Il suffit de décaler les bits. Le circuit ci-dessous calcule 2a en décalant d'un bit en direction du poids fort.

— Complétez le circuit pour calculer et afficher 4a et 8a.
— Vérifiez avec a=5. Votre affichage devrait montrer 10, 20 et 40.



1.9.10 Multiplier 1x4 bit

Le circuit ci-dessous utilise un multiplexeur 8x4 pour faire la multiplication d'un nombre **a** (4 bits) avec un nombre **b** (1 bit), au lieu des 4 portes ET utilisées précédemment.

Nous pouvons représenter un nombre binaire **b** par une séquence de 4 chiffres binaires : $b_3b_2b_1b_0$. Chaque bit b_i contrôle la multiplication de son poids (2^i) avec a.

— $b_0 \cdot 2^0 a$
— $b_1 \cdot 2^1 a$

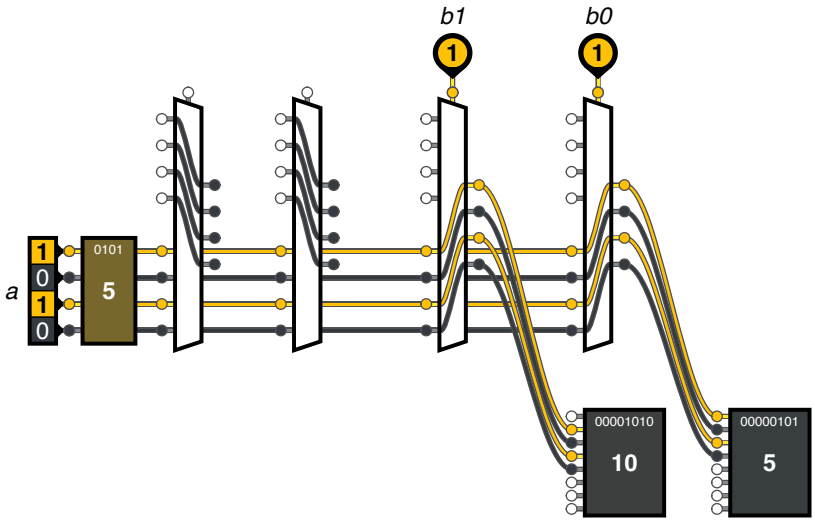
— $b_2 \cdot 2^2 a$
— $b_3 \cdot 2^3 a$

Pour compléter l'opération de multiplication 4x4 bits, la dernière étape sera d'additionner les 4 nombres.

Complétez le circuit avec :

— deux entrées que vous appelez **b2** et **b3**
— un affichage 8 bits qui affiche 4a sous contrôle de b2
— un affichage 8 bits qui affiche 8a sous contrôle de b3

Essayer de multiplier deux nombres, par exemple a=5 et b=5 (0101). Vous trouvez le résultat de la multiplication en additionnant les 4 nombres affichés.



1.9.11 Multiplier 4x4 bits

La multiplication 4 x 4 bits nécessite :

— 4 multiplexeurs pour la multiplication 4 x 1 bit
— 3 additionneurs pour additionner les 4 opérandes décalés

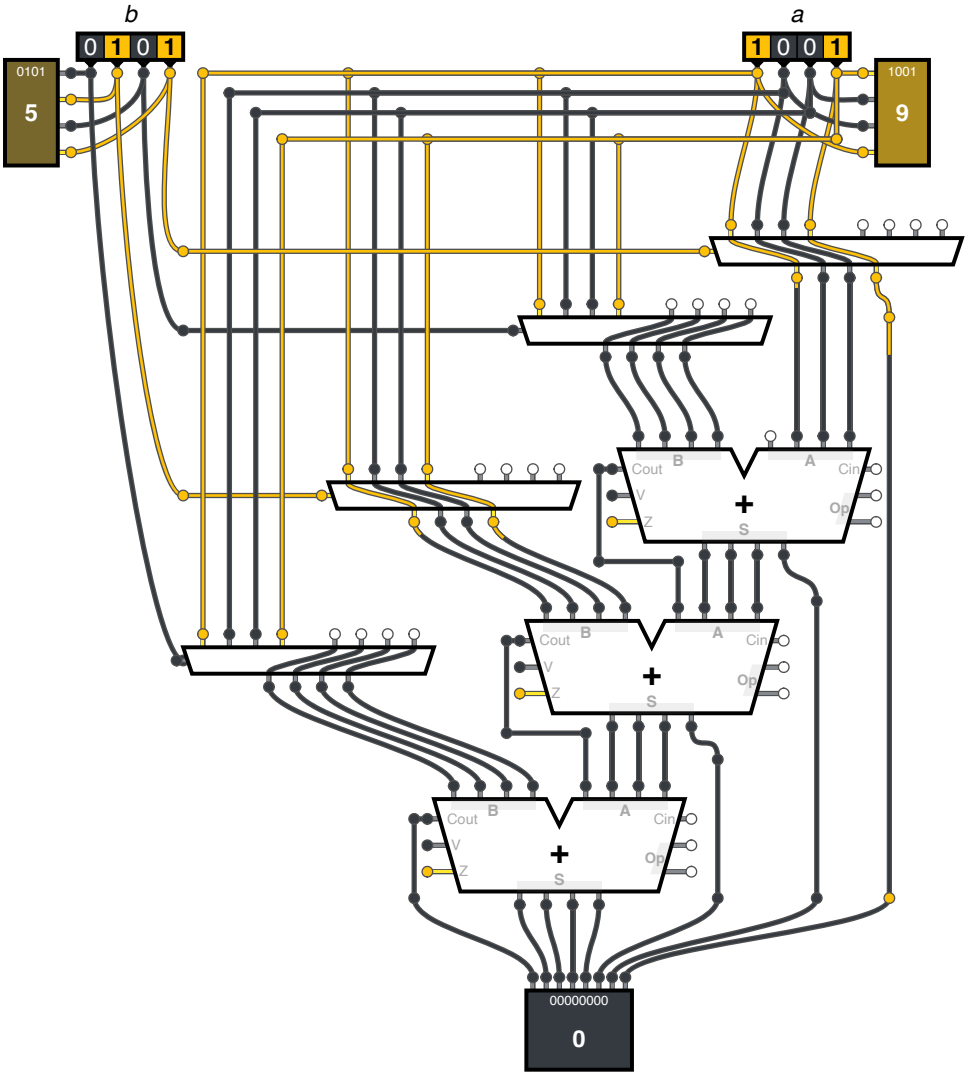
Pour multiplier `0101` x `1001` = `00101101` (5 x 9 = 45) nous écrivons en colonnes ceci :

```
1     1001
0    0000
1   1001
0 +0000
----------
  00101101
```

Cet algorithme peut être exprimé mathématiquement comme

$$produit = \sum_{i=0}^{4} (b_i \cdot a) \cdot 2^i$$

Modifiez a et b dans le circuit multiplicateur 4 x 4 bits ci-dessus vérifiez que vous obtenez bien le produit de a et b. Faites une capture d'écran avec la plus grande valeur possible.



1.9.12 Diviser par 2, 4 et 8

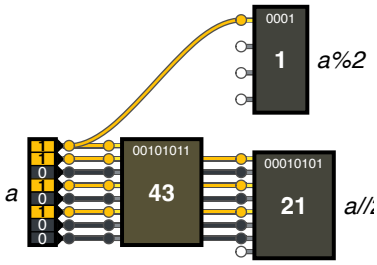
La division par une puissance de 2 est très simple. Il suffit de décaler le nombre binaire. Pour diviser par 2 nous décalons d'une unité, et nous obtenons :

— La division entière (`a // 2`)

— Le reste de la division, l'opération modulo (`a % 2`)

Ajoutez deux affichages 8 bits pour la division par 4 et 8 Ajoutez deux affichages 4 bits pour le modulo 4 et 8 Ajoutez les étiquettes (`a % 4`, `a // 4`, etc.)

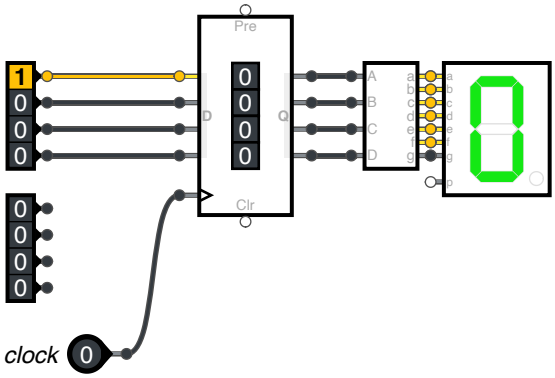
Par exemple pour `43 // 8` vous devriez obtenir 5,et pour `43 % 8` vous devriez obtenir 3.



1.9.13 Registre

Le registre que nous allons voir plus en détail dans le prochain chapitre permet de mémoriser une donnée. Avec un coup d'horloge (clock), les 4-bits de données sont mémorisés.

Ajoutez un deuxième registre, décodeur et affichage à 7 segments, pour permettre d'afficher un nombre décimal de 00 à 99 ou un nombre hexadécimal de 00 à FF.



1.9.14 Accumulateur

Un accumulateur est un registre spécial qui *accumule* une somme. La sortie de l'accumulateur est reliée avec l'entrée A de l'ALU. À chaque coup d'horloge du registre, le calcul `acc + b` est effectué et affiché.

Par exemple dans le circuit ci-dessous, l'accumulateur contient 3. Au prochain coup d'horloge, l'entrée b qui est 2 y sera additionnée. Ceci permet de calculer une somme courante.

Voici un exemple typique, calculer la somme 1+3+7. En Python ceci correspondrait à :

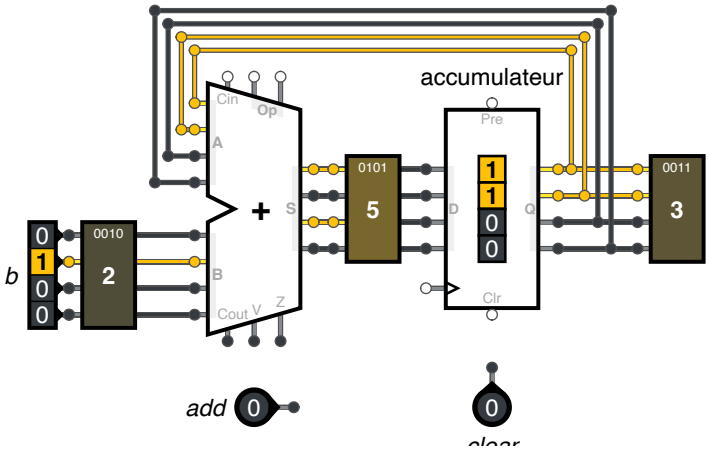
```
acc = 0     # clear
acc += 1    # add
acc += 3    # add
acc += 7    # add
print(acc)
```

Avec le circuit ci-dessous ceci correspond à 4 étapes :

1. **clear**
2. b=1 et **add**
3. b=3 et **add**
4. b=7 et **add**

Connectez les entrées **clear** et **add** au bon endroit et calculez 1+3+7.

Attention : tenez le bouton suffisamment longtemps pour laisser propager les signaux jusqu'au bout.



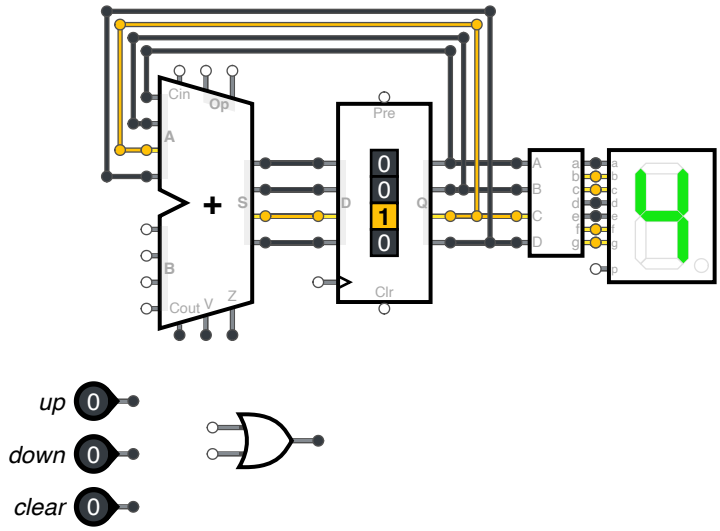
1.9.15 Incrémenter/décrémenter

Certains appareils électroniques ont très peu de touches et on doit utiliser juste deux boutons. C'est le cas pour régler la température ou le volume.

Compléter le circuit pour les boutons

- **up** pour incrémenter la valeur (clock)
- **down** pour décrémenter la valeur (clock + soustraction)
- **clear** pour mettre la valeur à zéro

Attention au délai de transmission par défaut de 100 ms. Il faut soit appuyer plus longtemps sur les boutons, ou diminuer ce délai.



1.9.16 Comparer

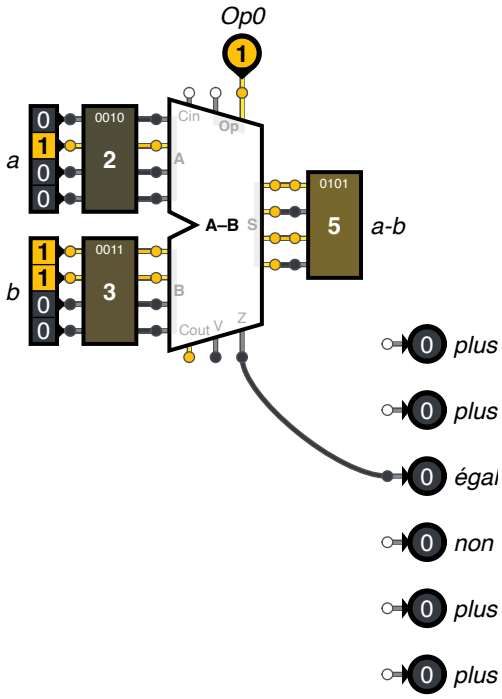
En Python nous disposons de 6 comparateurs pour comparer deux nombres a et b :

- > plus grand
- >= plus grand ou égal
- == égal
- != non égal
- <= plus petit ou égal
- < plus petit

Nous pouvons créer ces 6 comparaisons en utilisant une ALU qui soustrait deux nombres a et b et quelques portes logiques. Voici quelques astuces :

- quand a-b est zéro alors `Z=1`, donc **a égal b**
- quand a-b est négatif, alors `C=1`, donc **a plus petit que b**
- quand vous combinez les deux `Z=1` ou `C=1`, vous trouvez **a plus petit ou égal à b**

Utilisez les portes ET, OU et NON pour décoder les 6 types de comparaisons.


Op0
1
a
0
1
0
0
0010
2
b
1
1
0
0
0011
3
Cin
Op
A
A–B
S
B
Cout
V
Z
0101
5
a-b
0
plus
0
plus
0
égal
0
non
0
plus
0
plus

1.10 TP Mémoire

Les circuits que nous avons vus jusqu'à maintenant s'appellent **circuits combinatoires**. Leur sortie est le seul résultat de leurs entrées. Une même entrée produit toujours la même sortie. Le circuit n'a pas de mémoire.

La famille de circuits que nous allons découvrir s'appelle **circuit séquentiel**. Ces circuits permettent de mémoriser un état.

1.10.1 Cellule de mémoire

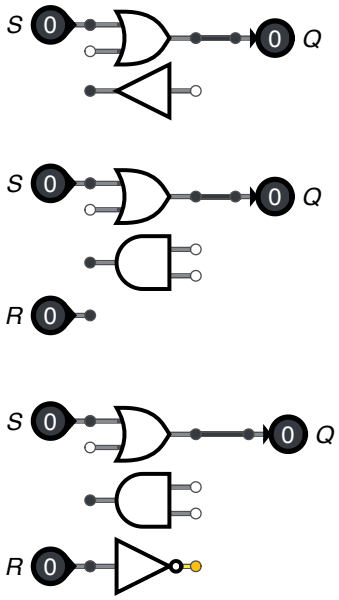
Une cellule de mémoire élémentaire peut être créée avec une porte OU, ou la sortie est connectée avec une de ses entrées.

Circuit 1
- Connectez la sortie de la porte OU via la porte OUI avec une de ses entrées
- Montrez que vous avez une cellule de mémoire qui peut mémoriser une seule fois (fusible)

Circuit 2
- Connectez la sortie de la porte OU via la porte ET avec une de ses entrées
- Montrez que vous avez une cellule de mémoire qui peut mémoriser S (set) et qui peut être remis avec R (reset)

Circuit 3
- Connectez la sortie de la porte OU via la porte ET avec une de ses entrées
- Contrôlez la porte ET via une porte NON
- Mettez l'entrée R en poussoir

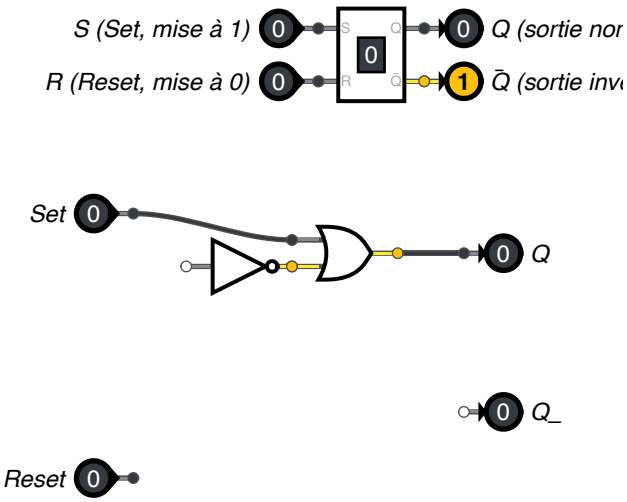
Attention : Pour éviter des effets d'oscillation, vous devez appuyer sur le bouton pressoir plus longtemps que 200 ms ou diminuer fortement le temps de propagation dans la boucle.



1.10.2 Verrou SR

Un verrou SR (set-reset) permet de “verrouiller” (mémoriser) une information.

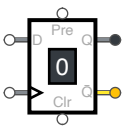
— Jouez avec le verrou SR pour comprendre son fonctionnement
— Recréez le verrou SR avec les composants NOT et OR



1.10.3 Bascule D

La bascule D (data) recopie la donnée sur l’entrée **D** vers sa sortie **Q** à chaque front montant de l’horloge **clock**.

— ajoutez les 6 entrées/sorties à la bascule D,
— lisez et comprenez les étiquettes,
— observez et comprenez son comportement.

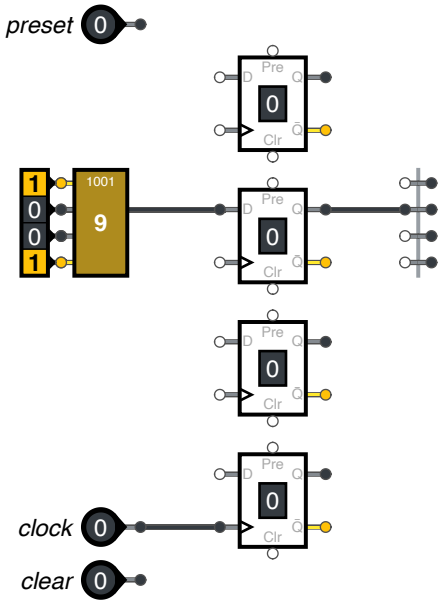


1.10.4 Registre 4 bits

Le **registre** est un circuit qui peut mémoriser multiples bits sur un coup d'horloge. À partir de 4 bascules D, nous pouvons construire un registre pour mémoriser 4 bits.

Ajoutez les connexions qui manquent :

- Connectez chacun des 4 bits d'entrée vers une entrée **D** de la bascule.
- Connectez chacun des 4 bits de sortie **Q** vers son bit de sortie correspondante.
- Ajoutez une sortie 4 bits et un affichage 4 bits
- Connectez les 4 entrées **preset**, **clock** et **clear**

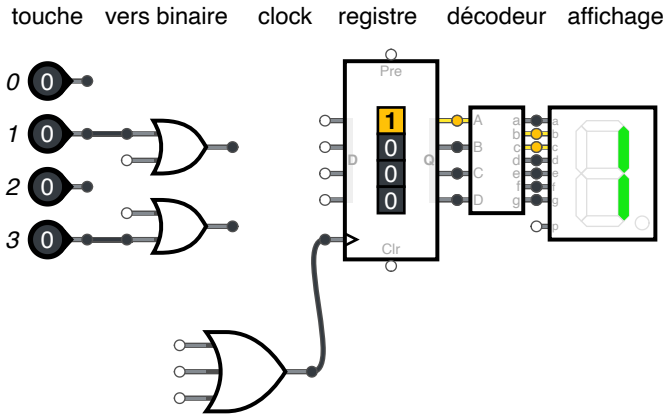


1.10.5 Décodeur de touche

Quand on appuie sur une touche d'une calculatrice électronique, telle que la TI-30, la valeur de la touche est transformée en binaire 4 bits, enregistré dans un **registre 4 bits**, et affiché avec un affichage à 7 segments. Les étapes sont :

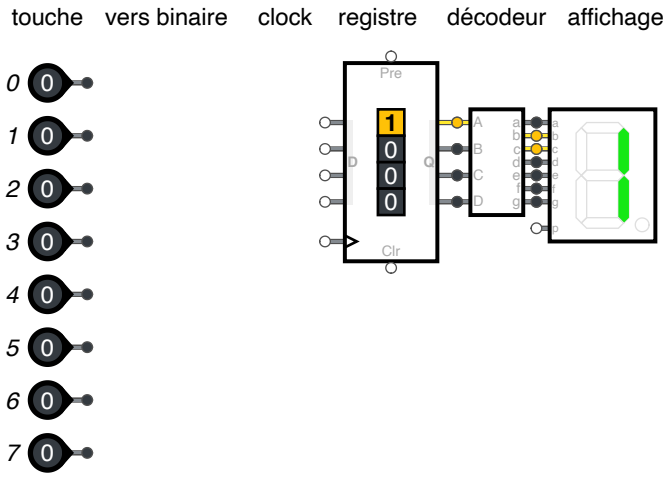
- une **touche** 0 - 9 est appuyé
- une conversion en **binaire** est faite
- une entrée **clock** est produite
- la valeur est mémorisée dans un **registre**
- la valeur est décodée vers les signaux a-g
- la valeur numérique est montrée sur l'**affichage** à 7 segments

Complétez le circuit pour traiter les touches 0 à 3



1.10.6 Décodeur pour 8 touches

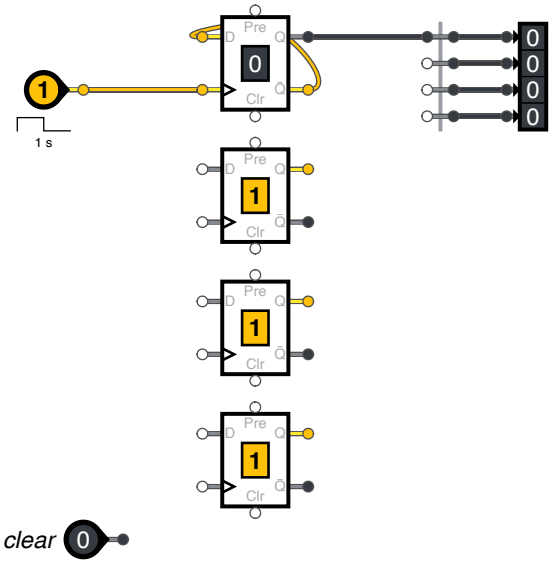
Complétez le circuit pour traiter les touches 0 à 7



1.10.7 Compteur 4 bits

Une bascule D avec une rétroaction de la sortie inversée $\bar{Q}$ vers son entrée D divise la fréquence de l'horloge par 2. Nous avons effectivement un compteur 1 bit.

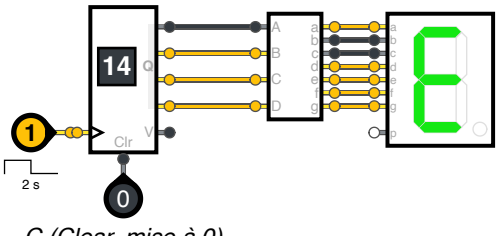
— Complétez le circuit pour construire un compteur 4 bits.
— Le circuit final doit compter de `0000` vers 1111.
— Ajoutez un affichage 4 bits
— Connectez aussi les 4 signaux **clear** pour la remise du compteur



1.10.8 Compteur 8 bits

Le compteur 4 bits utilise un signal d'horloge et incrémenté à chaque coup d'horloge. Un décodeur à 7 segments transforme les 4 signaux qui représentent un nombre binaire de 0 à 16 vers les sorties correspondant pour activer les bonnes lampes de l'affichage à 7 segments.

— Utilisez le signal de sortie V (overflow) pour faire fonctionner un deuxième compteur
— Ceci donnera un compteur 8 bit, permettant de compter de 00 à FF (255)
— Diminuez la période de l'horloge à 250 ms

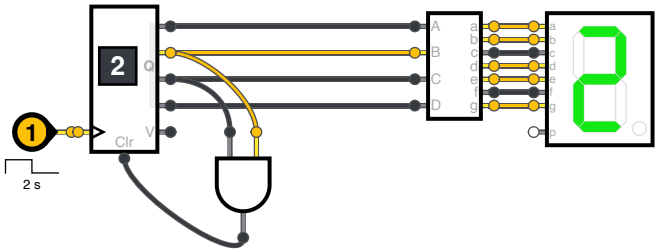


C (Clear, mise à 0)

1.10.9 Compteur avec remise

Pour créer une montre, un minuteur ou une alarme, nous devons compter à 60, 12 ou 24. L'entrée Reset peut être utilisée pour remettre le compteur à zéro. Une porte ET détecte le nombre 6 et réinitialise le compteur.

- Ajoutez un deuxième compteur
- Configurez-le pour qu'il compte de 0 à 9
- Ajoutez le décodeur et un affichage à 7 segments
- Utilisez les deux compteurs pour faire un compteur qui affiche les nombres 00 à 59
- Diminuez la période à 1 seconde



1.10.10 Rouler un dé

Un dé électronique utilise 7 LEDs pour afficher les points. Une horloge rapide, liée par une porte ET vers l'entrée d'un compteur compte rapidement de 0 à 5. À 6 le compteur est remis à 0 à l'aide d'une autre porte ET.

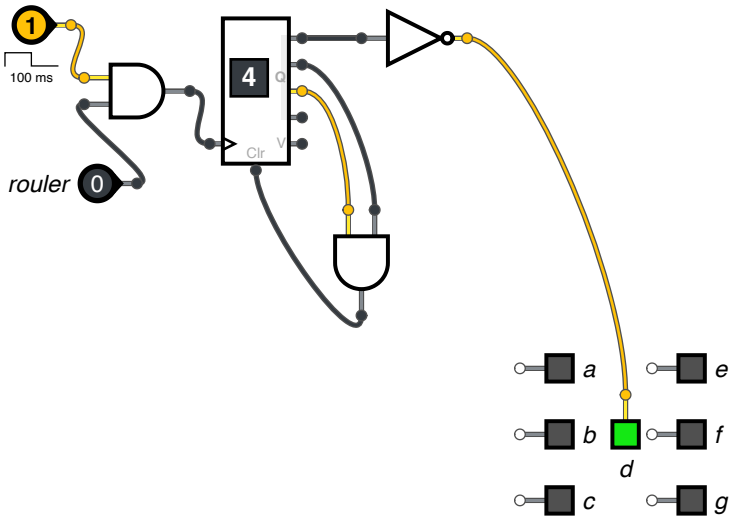
Voici la table de vérité pour les 7 LEDS.

dice	bin	a,g	b,f	c,e	d
1	000	0	0	0	1
2	001	1	0	0	0
3	010	1	0	0	1
4	011	1	0	1	0
5	100	1	0	1	1
6	101	1	1	1	0

Ajoutez les portes logiques appropriées pour implémenter ce dé électronique.Par exemple, le signal pour lampe a,g correspond à

```
b0 or b1 or b2
```

Trouvez les autres circuits et construsez ce dé électronique.

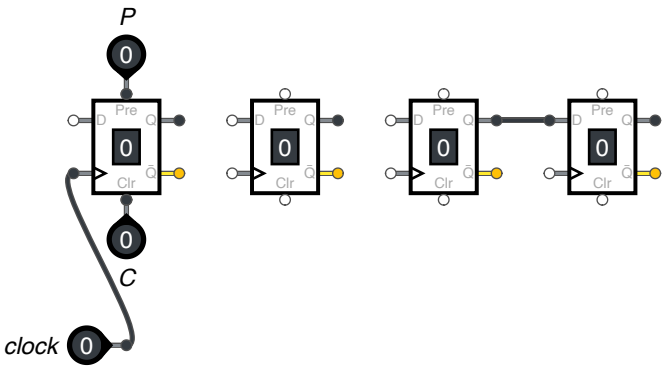


1.10.11 Registre à décalage

Un registre de décalage propage une information d'un registre vers l'autre.

- Placez tous les entrées P (preset) et C (clear)
- Connectez la sortie Q avec l'entrée D suivante
- Connectez tous les entrées clock
- Testez le circuit à décalage

Si ça fonctionne, connectez la sortie du registre à décalage avec son entrée et connectez une entrée horloge à 1 second à l'entrée clock. Vous avez un **registre à décalage circulaire**.

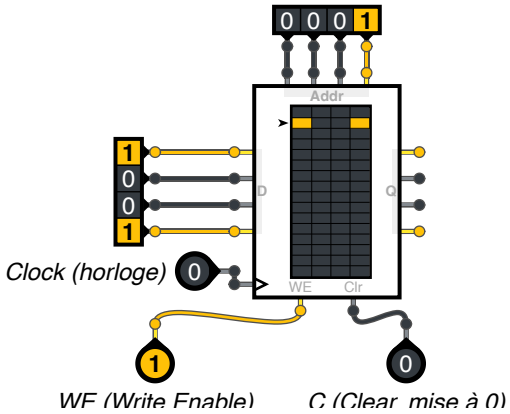


1.10.12 RAM (mémoire vive)

La RAM (Random Access Memory) de 16 x 4 bits permet de stocker 16 mots de 4 bits. Les quatre bits de l'adresse **Addr** déterminent l'endroit où seront écrites les données **D**.

- **Addr** détermine l'endroit des données
- **D** signifie les données à écrire
- **Q** représente les 4 bits des données lus
- **WE** (Write Enable) permet d'écrire si 1
- **clock** transmet les données sur D en mémoire
- **Clr** (clear) remet toute la mémoire à zéro

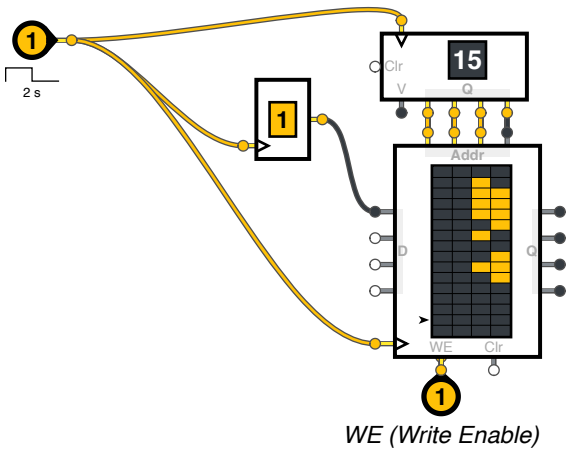
À l'adresse `0001` (1) se trouvent déjà les données `1001`. Ceci ressemble à des yeux. Ajoutez d'autres valeurs pour en faire un smiley.



1.10.13 RAM avec bits aléatoires

Le circuit suivant utilise un compteur pour créer les 16 adresses et écrire un bit aléatoire dans la RAM.

Complétez le circuit pour écrié 16 x 4 bits aléatoires dans la mémoire.

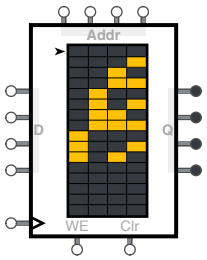


1.10.14 RAM avec binaire

Utilisez un compteur 4-bit pour remplir automatiquement la RAM avec les valeurs de 0 à 15. Voici le contenu à écrire dans la RAM.

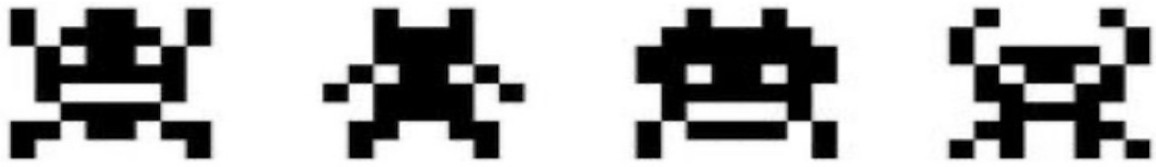
```
0000
0001
0010
...
1111
```

Utilisez la sortie overflow (V) du compteur pour faire une mise à zéro de la RAM, à chaque fois quand elle est pleine. Dans l'image ci-dessous, la RAM est déjà remplie jusqu'à `1010` (10).

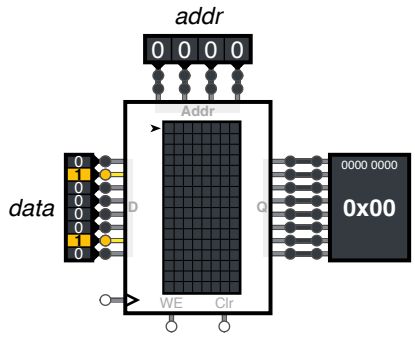


1.10.15 RAM avec image

La mémoire peut contenir des images. Chaque bit représente un pixel. Un des premiers jeux vidéo, **Space Invaders**, utilise des images monocolores pour présenter des envahisseurs extraterrestres.



Remplissez la mémoire avec l'image d'un *space invader* 8x8 bits. L'image sera alors visible dans la partie visualisation du bloc RAM 16x8 bits.



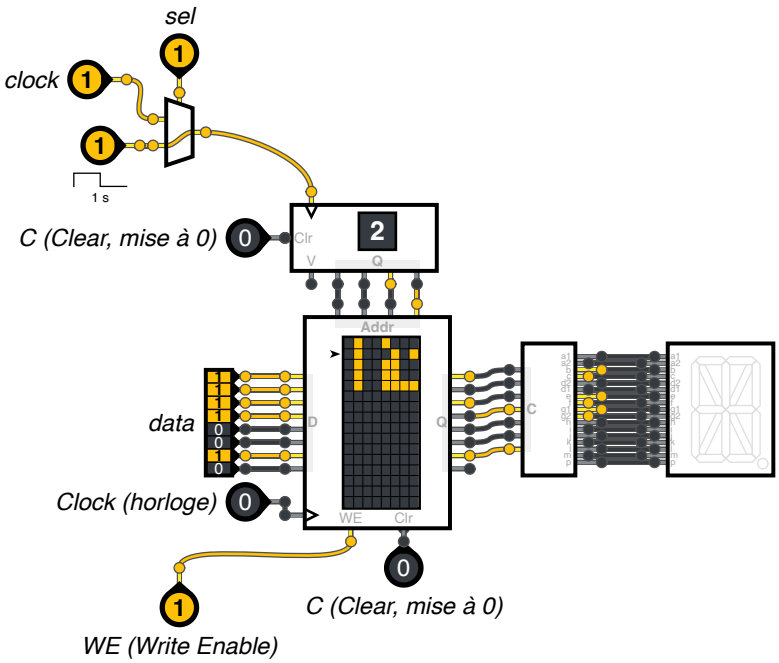
1.10.16 RAM avec ASCII

La mémoire peut contenir du code ASCII. Voici les codes ASCII des 6 lettres du mot `ON AIR`, exprimées en binaire et en hexadécimal.

```
O = 0b01001111 0x4f
N = 0b01001110 0x4e
  = 0b00100000 0x20
A = 0b01000001 0x41
I = 0b01001001 0x49
R = 0b01010010 0x52
```

Le circuit ci-dessous contient le mot `HELLO` en mémoire RAM. Un compteur avec une horloge 1 Hz affiche le contenu de la mémoire en boucle vers un affichage à 16 segments.

Remplacez le contenu de la mémoire pour afficher le mot `ON AIR`.


sel
clock
1
1
1 s
C (Clear, mise à 0)
0
Clr
V
2
Q
Addr
data
0
0
0
D
Q
C
Clock (horloge)
0
WE
Clr
0
1
C (Clear, mise à 0)
WE (Write Enable)

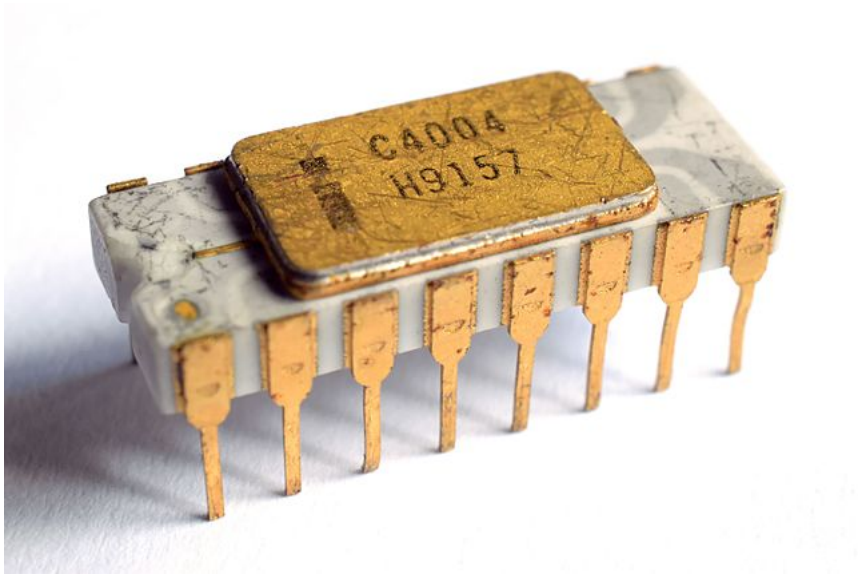
1.11 TP CPU

Le processeur, aussi appelé CPU (Central processing Unit) ou unité de traitement central, lit des instructions dans la mémoire programme et les exécute.

Tout ce que le processeur fait peut être décrit en 3 lignes :

- chercher une instruction dans la mémoire de programme (Fetch)
- exécuter cette instruction (Execute)
- incrémenter le pointeur vers la prochaine instruction (Increment)

Dans cette section nous allons étudier comment encoder des instructions en code binaire, et comment ensuite exécuter ce code dans le CPU. Nous allons nous inspirer du premier microprocesseur, la puce Intel 4004, sortie en 1971.

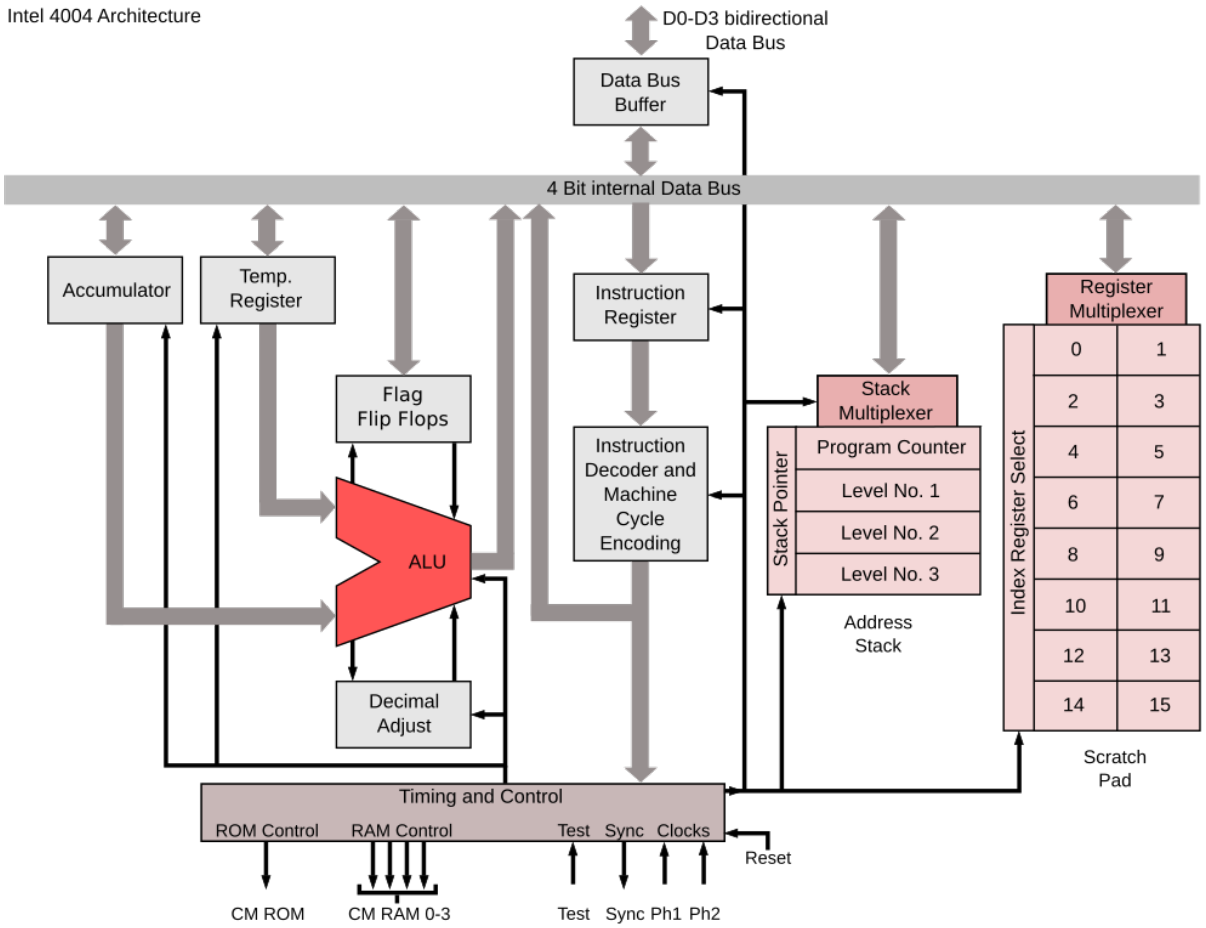


1.11.1 Intel 4004

Le premier CPU sur un seul circuit intégré fut le 4004 commercialisé par Intel en 1971. Il contenait les éléments suivants :

- une ALU (unité arithmétique et logique)
- 16 registres de travail
- un accumulateur (accumulator)
- un bus de données (data bus)
- un bus d'adresses (adress bus)
- des fanions (flags)
- un registre d'instruction (instruction register)
- un compteur de programme (program counter)
- un pointeur de pile (stack pointer)
- une pile (stack)
- une unité de contrôle (control unit)

Voici le schéma de l'architecture du 4004.



1.11.2 Langage assembleur

Nous allons commencer tout de suite avec un exemple de programme pour le 4004, en langage assembleur. Voici un bout de programme qui additionne deux nombres 4 bits.

```
ADD2
; add two 4bit numbers on the Intel 4004
;
    FIM P0, $A2 ; initialize: R0=2 R1=A
    LDR R0      ; load R0 into accumulator
    ADD R1      ; add R1 into accumulator
    XCH R1      ; and store in R1
```

Le point-virgule (`;`) sert comme symbole de commentaire. C'est l'équivalent du `#` en Python. Un programme en assembleur est typiquement structuré en 4 colonnes :

1. Une étiquette pour désigner une adresse de programme (`ADD2`)
2. Une mnémonique de l'opération (`FIM`, `LDR`, `ADD`, `XCH`)
3. Des données (P0, $A2, R0, R1)
4. Des commentaires en fin de ligne

1.11.3 Le langage machine

Le langage machine, ou code machine, est la suite de bits qui est interprétée par le processeur d'un ordinateur exécutant un programme informatique. C'est le langage natif d'un processeur, c'est-à-dire le seul qu'il puisse traiter. Il est composé d'instructions et de données à traiter codées en binaire.

- `1000RRRR` additionne (ADD) le registre `RRRR` à l'accumulateur
- `1001RRRR` soustrait (SUB) le registre `RRRR` de l'accumulateur
- `1101DDDD` charge (LDM = load) le nombre `DDDD` vers l'accumulateur

Le code machine est composée de :

- la partie instruction ou **opcode**, tel que `1000=ADD`, `1001=SUB`
- la partie donnée ou **data**, qui indique un registre `RRRR` où un nombre `DDDD`

Par exemple l'instruction en assembleur `ADD R3` se traduit en code machine comme `10000011`.

Trouvez les deux autres codes machine.

ADD R3



SUB R15



LDM 13



1.11.4 Mémoire de programme

Le CPU 4004 traite des données de 4 bits. Les données que ce processeur peut traiter avec une seule instruction sont limitées à une plage de 0 à 15 (de `0000` à `1111`). On dit que c'est un processeur 4 bits ou une **architecture 4 bits**.

Chaque instruction par contre est encodée sur 8 bits. Ce processeur pourrait donc avoir au maximum 256 instructions différentes. En réalité il a 46 instructions.

Le vrai CPU 4004 peut adresser un espace mémoire de programme avec une adresse 12 bits. Ceci lui permet d'adresser un maximum de 2^{12} instructions différentes dans sa mémoire programme. Ici nous simplifions beaucoup et utilisons des adresses de 4 bits. Notre mémoire programme a une taille de 16 x 8 bits. Notre programme peut avoir un maximum de 16 instructions.

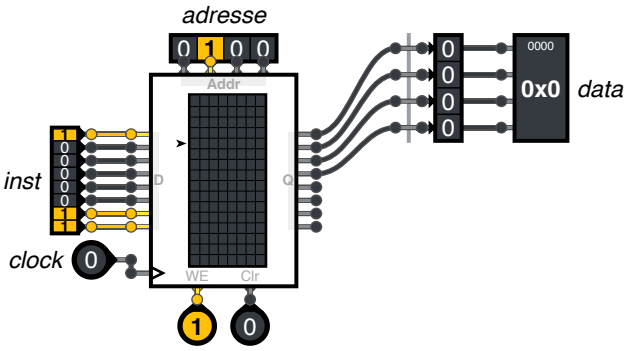
Beaucoup d'instructions sont composées d'une partie

- instruction (4 bits), appelée **opcode**
- données (4 bits), appelé **data**

Dans le circuit ci-dessous, ajoutez :

- une broche 4 bits
- une sortie 4 bits
- un affichage 4 bits (en hexadécimal)
- une étiquette **opcode**

— Mettez les instructions `ADD R3`, `SUB R15` et `LDM 13` dans les 3 premiers octets de la mémoire programme.



1.11.5 Le jeu d'instructions

On appelle **jeu d'instruction** (instruction set) la totalité des instructions qu'un processeur peut exécuter. Ces instructions sont représentées par une abréviation à 3 lettres (mnémonique). Les premières 14 instructions sont composées d'une partie :

— instruction (4 bits), appelée **opcode**, de `0000` à `1101`
— données (4 bits), appelé **data**

La partie `data' peu représenter 3 types de données :

— `AAAA` une adresse dans la mémoire programme
— `RRRR` un registre dans la banque des registres
— `DDDD` une donnée immédiate (un nombre de 0 à 15)

```
NOP 00000000  No Operation
JCN 0001AAAA  Jump Conditional
FIM 0010RRR0  DDDDDDDD  Fetch Immediate
FIN 0011RRR0  Fetch Indirect
JUN 0100AAAA  Jump Unconditional
JMS 0101AAAA  Jump to Subroutine
INC 0110RRRR  Increment
ISZ 0111RRRR  AAAAAAAA  Increment and Skip

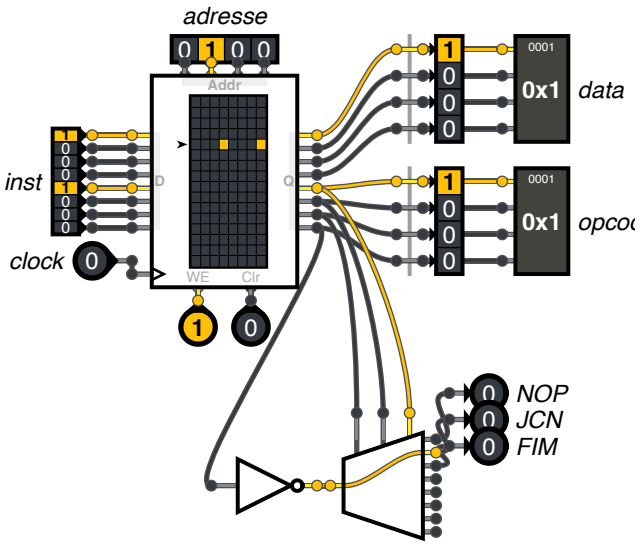
ADD 1000RRRR  Add register
SUB 1001RRRR  Subtract register
LDR 1010RRRR  Load register
XCH 1011RRRR  Exchange register
BBL 1100DDDD  Branch Back and Load
LDM 1101DDDD  Load Immediate data
```

1.11.6 Décoder une instruction

Pour décoder les 14 instructions, nous pouvons utiliser un démultiplexeur. Pour compléter le circuit de décodage d'instruction :

- Ajoutez un deuxième démultiplexeur
- Ajoutez les 11 sorties manquantes
- Ajoutez les étiquettes (`FIN` à `LDM`)
- Remplissez la mémoire programme avec les 14 instructions (`NOP` à `LDM`)
- Vérifiez que chaque instruction est décodée correctement

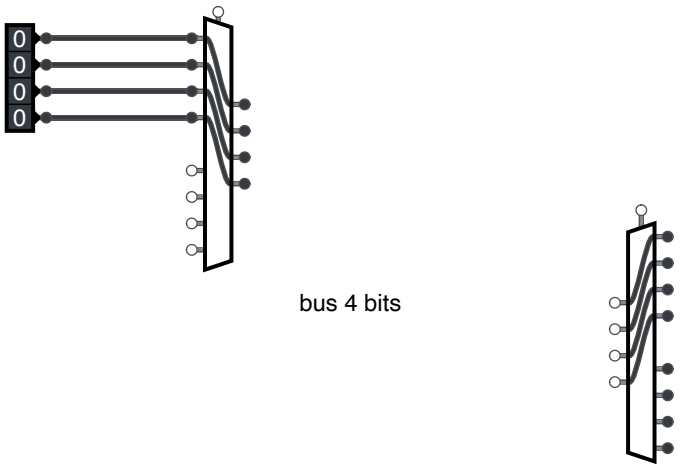
Par exemple à l'adresse `0100` (4) se trouve l'instruction `00010001` (`JCN 1`) et le décodeur active correctement la sortie `JCN`.



1.11.7 Bus 4 bits

Une ALU doit acheminer différents signaux sur une même ligne de transfert des données. On appelle un tel chemin un bus de données. Pour y connecter plusieurs sources, nous devons utiliser un multiplexeur.

- Ajoutez une deuxième entrée 4 bits
- Liez le multiplexeur et le démultiplexeur à travers une broche 4 bits
- Ajoutez les 2 entrées de sélection
- Ajoutez 4 affichages 4 bits pour montrer tous les signaux



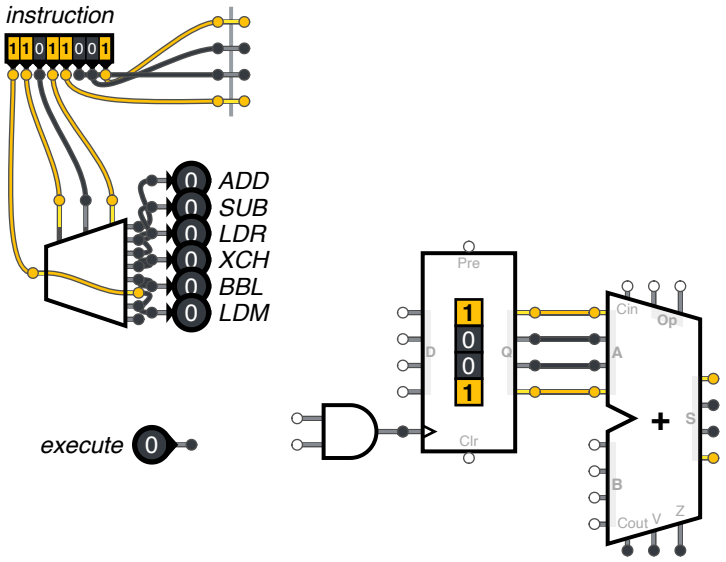
1.11.8 Charger imméd. (LDM)

La commande `LDM` (Load immediate) va charger une valeur directe (0-15), contenue dans le code de l'instruction, dans l'accumulateur.

`1101DDDD`

La partie `1101` est l'opcode (`LDM`) et la partie `DDDD` représente les 4 bits des données à charger dans l'accumulateur.

- Liez les bits b0-b3 avec l'accumulateur
- Utilisez la porte ET pour charger cette valeur dans l'accumulateur seulement si le signal **execute** est activé ET l'instruction `LDM` est décodée
- Placez un affichage à la sorte de l'accumulateur et à la sortie de l'ALU
- Mettez une valeur dans les bits b0-b3 et exécutez l'instruction avec **execute**



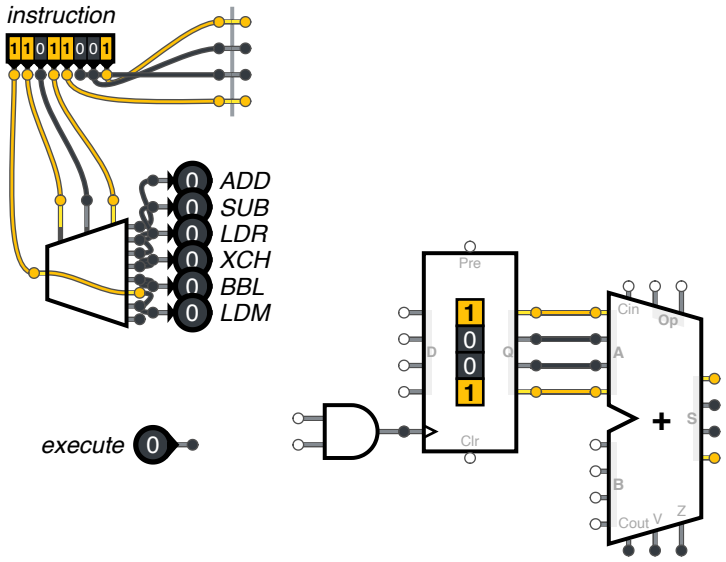
1.11.9 Charger depuis reg (LDR)

L'instruction `LDR` (load from register) charge l'accumulateur avec le contenu d'un des 16 registres.

`1010RRRR`

La parte `1010` est l'opcode (LD) et la parte `RRRR` représente un des 16 registres

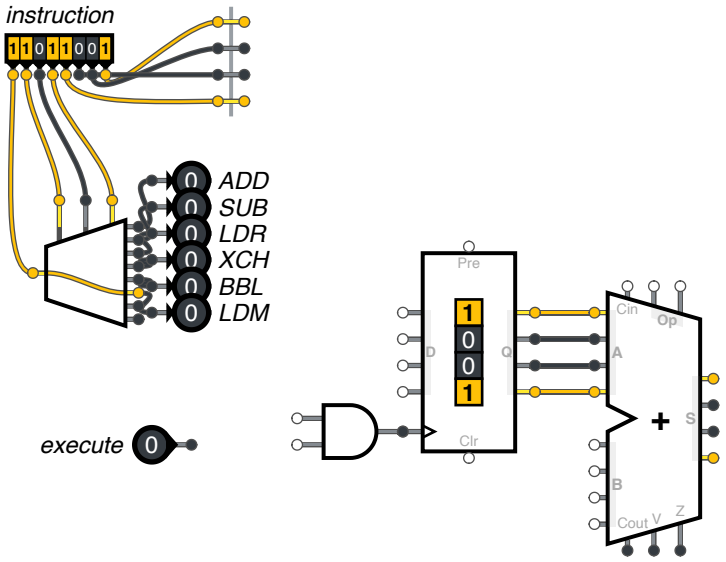
- Ajoutez la RAM avec les 16 registres
- Créez le circuit de décodage pour charger registre `RRRR` dans l'accumulateur
- Placez un affichage à la sortie de l'accumulateur et à la sortie de l'ALU
- Mettez une valeur dans `R5`
- Mettez l'instruction `LDR R5` et exécutez l'instruction avec **execute**



1.11.10 Choix entre LDR/LDM

Avec les deux opcode différents, le circuit de décodage du CPU choisit un registre ou une donnée immédiate comme valeur à charger dans l'accumulateur.

- Ajoutez la RAM avec les 16 registres
- Ajoutez un multiplexeur 8x4
- Créez le circuit de décodage pour choisir entre un registre `RRRR` ou une donnée immédiate `DDDD`
- Placez un affichage à la sortie de l'accumulateur et à la sortie de l'ALU
- Mettez une valeur dans `R5`
- Mettez l'instruction `LDR R5` et exécutez l'instruction avec **execute**
- Mettez l'instruction `LDM 13` et exécutez l'instruction avec **execute**



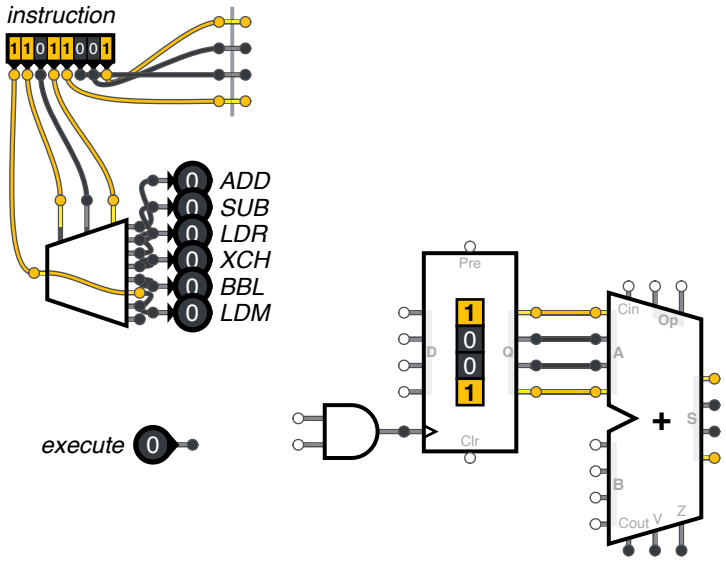
1.11.11 Choix entre ADD/SUB

L'addition et la soustraction se distinguent dans l'opcode d'un seul bit.

- `1000`RRRR ADD additionner registre RRRR à l'accumulateur
- `1001`RRRR SUB soustraire registre RRRR de l'accumulateur

Créez le circuit pour décoder et exécuter ces deux instructions.

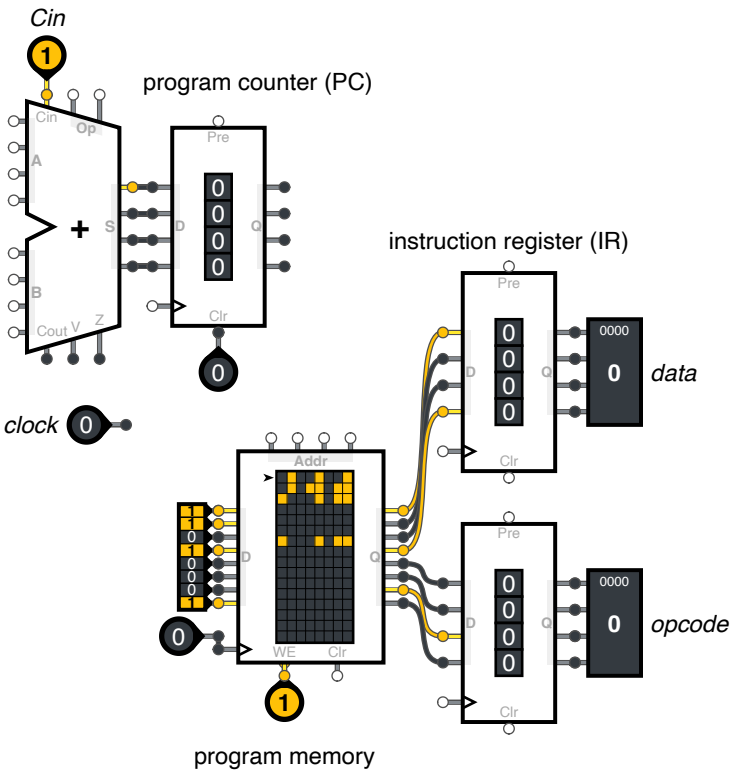
- Ajoutez la RAM avec les 16 registres
- Créez le circuit de décodage pour choisir entre `ADD` et `SUB`
- Placez un affichage à la sortie de l'accumulateur et nommez le **acc** (y mettre 9)
- Placez un affichage à la sortie de la RAM et nommez le **reg**
- Placez un affichage à la sortie de l'ALU et nommez le **result**
- Mettez la valeur 3 dans `R5`
- Mettez l'instruction `ADD R5` et vous devriez avoir **result = 12** (9 + 3)
- Mettez l'instruction `SUB R5` et vous devriez avoir **result = 6** (9 - 3)



1.11.12 Program counter (PC)

Le pointeur de programme, PC (program counter), pointe toujours à la prochaine instruction dans la mémoire de programme. Le contenu à l'adresse pointé par le PC est celui qui est chargé dans le registre d'instruction (IR) et exécuté au prochain pas.

— Liez la sortie du PC avec l'entrée A du l'ALU pour incrémenter de 1 à chaque pas
— Placez un affichage à la sortie du registre et nommez le **PC**
— Liez le PC avec l'entrée adresse de la mémoire de programme
— Liez l'entrée **clock** avec l'horloge de l'ALU et l'horloge des deux registres IR
— Faites avancer le PC et chargez des instructions successives dans le registre IR



1.11.13 Le saut (jump)

Le saut est une instruction qui permet de changer l'avancement linéaire du compteur de programme. L'instruction de saut a la forme :

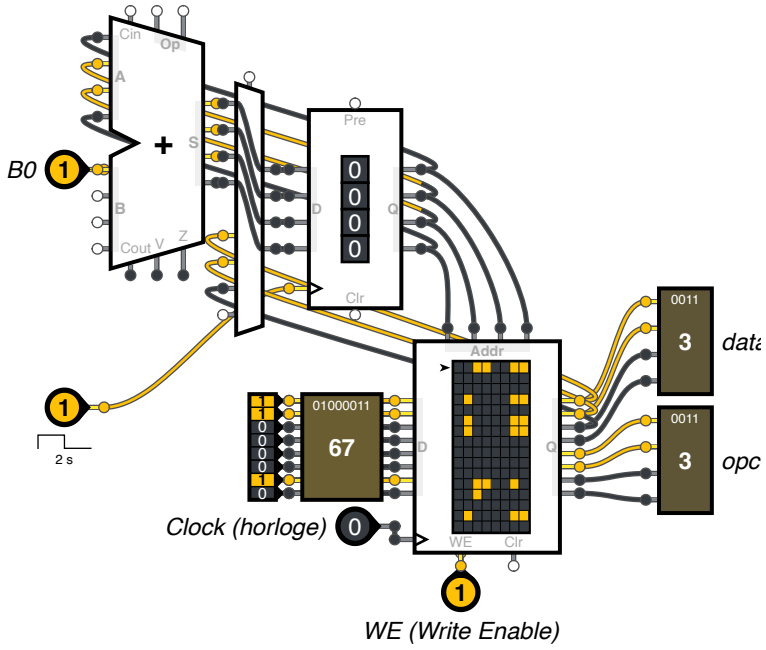
`Jump Uncoditional JUN 0100AAAA`

Pour l'exécuter, le CPU doit d'abord détecter l'opcode `0100`. Ceci peut être fait avec une porte ET à 4 entrées et des inverseurs.

Ensuite la valeur actuelle du compteur de programme doit être remplacée par la nouvelle adresse de destination du saut `AAAA`. Pour ceci nous utilisons un multiplexeur 8 vers 4.

Utilisez des portes NON et ET pour décoder l'opcode `0100` et l'utiliser pour sélectionner entre incrémentation normale et destination de saut.

A l'adresse 14 se trouve l'instruction `01000011` (`JUN 3`). Si le décodeur fonctionne correctement le programme va faire une boucle entre les addresses 3 et 14.



1.11.14 La pile (stack)

La pile est un espace de sauvegarde temporaire. Elle est utilisée pour sauvegarder les adresses de retour lors d'un saut vers une sous-routine.

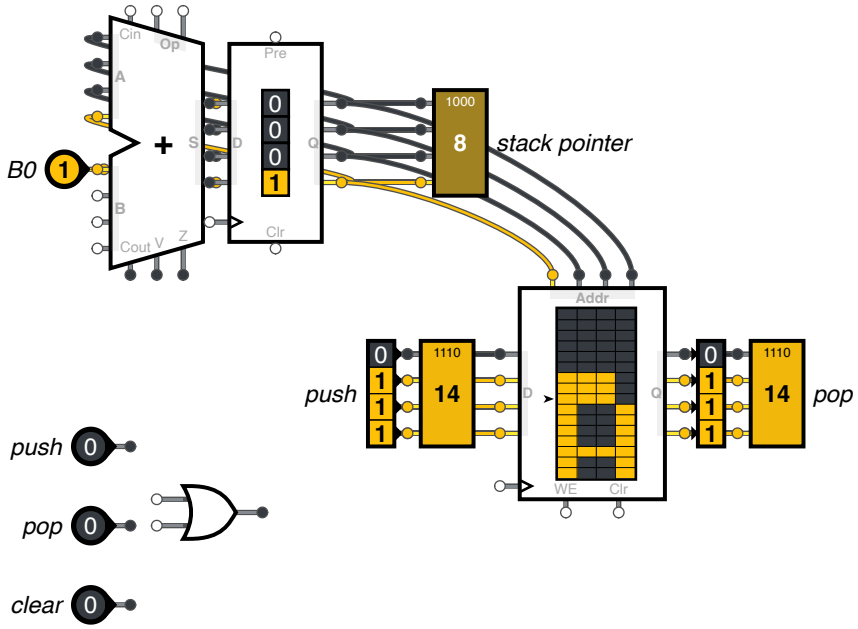
Ici nous créons une pile de 16 mots à 4 bits. D'habitude on commence la pile en bas de l'espace mémoire (adresse 15) et on *empile* les valeurs.

Le pointeur de pile (stack pointer) est un registre 4 bit, qui utilise une ALU pour être incrémenté ou décrémenté. Si l'entrée `Op = 0` il incrémente. Si l'entrée `Op = 1` il décrémente.

Ajoutez les circuits de contrôle.

- Le signal **clear** efface la pile et met le pointeur de pile à `1111` (tout en bas)
- Le signal **push** choisit la décrémentation (sp–) et envoie un coup d'horloge vers le registre du pointeur et la pile
- Le signal **pop** choisit l'incrémentation (sp++) et envoie un coup d'horloge vers le registre du pointeur et la pile

Mettez dans la pile `3141592` (les 7 premiers chiffres du nombre pi) avec l'instruction **push**.Ensuite, lisez ces chiffres dans l'ordre inverse avec l'instruction **pop**



1.11.15 Projet final

Ouvrez l'éditer logique [14] dans une page entière du navigateur et choisissez un des projets suivants :

1. Complétez l'architecture du CPU 4004 pour en faire un processeur fonctionnel.Voici le jeu d'instructions [15] complet.
2. Créez une calculatrice avec les touches 0 à 10, les opérations + et -, et un affichage à 7 segments avec 4 chiffres ou plus. Le point décimal est optionnel.
3. Créez une horloge avec un affichage à 7 segments du style `HH:MM:SS`, avec des boutons **up/down** pour mettre le temps.
4. Créez un minuteur avec un affichage à 7 segments du style `MM:SS` qui décompte, avec des boutons **up/down** pour mettre le temps, et des boutons **start/stop/clear**.
5. Créez un chronomètre avec un affichage à 7 segments du style `MM:SS.S` qui affiche des dixièmes de seconde. Ajoutez des boutons **start/stop/clear**.

14. https://logic.modulo-info.ch/
15. http://e4004.szyc.org/iset.html

1.11.16 Nand Game

Dans le jeu Nand Game [16] vous allez construire un ordinateur à partir de composants de base.

Le jeu se compose d'une série de niveaux. Dans chaque niveau, vous êtes chargé de construire un composant qui se comporte selon une spécification. Ce composant peut ensuite être utilisé comme bloc de construction dans le niveau suivant.

Le jeu ne nécessite aucune connaissance préalable de l'architecture informatique ou des logiciels, et ne nécessite aucune compétence en mathématiques au-delà de l'addition et de la soustraction. (Cela demande un peu de patience - certaines tâches peuvent prendre un certain temps à résoudre !)

Votre première tâche est de créer un composant nand (Non-Et).

Bonne chance !

L'ENIAC, l'un des tout premiers ordinateurs opérationnels, conçu en 1945, à la fin de la Seconde Guerre mondiale, pour calculer des trajectoires de missiles, était constitué de 17468 tubes électroniques de la taille d'une main, qui cassaient en moyenne une fois par semaine. Il s'étendait sur 170 mètres carrés et pesait plus de 25 tonnes. Il était capable d'exécuter environ 5000 opérations par seconde.

Pour comparaison, les microprocesseurs des smartphones modernes exécutent de l'ordre de plusieurs centaines de *milliards* d'opérations par seconde.

En 1991, 1 Go (un milliard d'octets) de mémoire non volatile coûtait environ 45000 dollars. Aujourd'hui, un smartphone moderne dispose de l'ordre de 256 Go d'espace de stockage, ce qui aurait coûté à l'époque 11.5 millions de dollars [1].

À quoi servaient précisément 17468 tubes électroniques de l'ENIAC dans le rôle que l'ordinateur avait ? Et par quoi ont-ils été remplacés dans nos machines modernes ? Et comment se fait-il qu'un objet qui tient dans la poche puisse contenir 256 fois plus d'espace disque qu'un ordinateur des années 1990 ?

Dans ce chapitre, vous découvrirez comment sont construits les ordinateurs, comment sont organisés leurs différents composants pour leur permettre d'effectuer des milliards de calculs à la seconde alors qu'ils ne comprennent que la distinction entre 0 et 1, allumé ou éteint.

16. https://nandgame.com/

1. https ://www.aei.org/technology-and-innovation/the-a12-chip-estimating-innovation-with-iphone-prices/

1.12 Objectifs

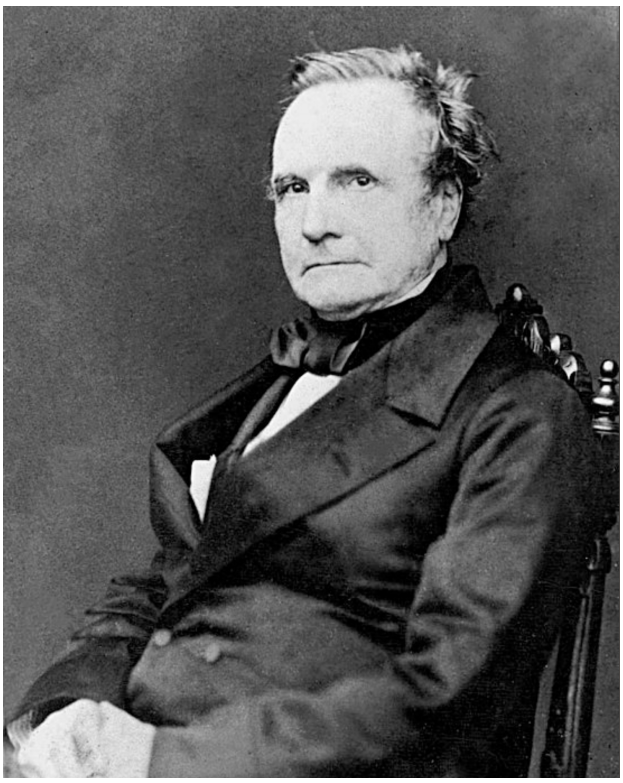
— Découvrir les **éléments de base** qui composent l'ordinateur.
— Comprendre les notions de **système logique** et de **microprocesseur**.
— Appréhender l'importance de **l'architecture des ordinateurs** pour optimiser les performances et effectuer des tâches informatiques spécifiques.

1.13 Personnages-clés



Anita Borg
1949-2003
Anita Borg [17] est une informaticienne américaine. Elle a notamment travaillé pour Digital Equipment Corporation où elle a développé une méthode permettant de concevoir des systèmes mémoriels à haute vitesse.

17. https://fr.wikipedia.org/wiki/Anita_Borg



Charles Babbage
1791-1871
Charles Babbage [18] fut le premier inventeur à énoncer le principe d'un ordinateur. C'est en 1834, pendant le développement d'une machine à calculer destinée au calcul et à l'impression de tables mathématiques, qu'il eut l'idée d'y incorporer des cartes du métier Jacquard, dont la lecture séquentielle donnerait des instructions et des données à sa machine.

18. https://fr.wikipedia.org/wiki/Charles_Babbage