Circuits logiques programmables

Clément Foucher

▶ **To cite this version:**

Clément Foucher. Circuits logiques programmables. Licence. Circuits logiques programmables, IUT Paul Sabatier, Toulouse, France. 2023. hal-04301425

HAL Id: hal-04301425

https://hal.science/hal-04301425

Submitted on 23 Nov 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Circuits logiques programmables

NOTIONS DE BASE : LOGICIEL ET MATÉRIEL

BREF HISTORIQUE DU MATÉRIEL RECONFIGURABLE

COMMENT CONCEVOIR LE CIRCUIT LOGIQUE ?

BUS D'INTERCONNEXION

POINT DE SYNTAXE LANGAGE C

© 2023 Clément Foucher

Cours distribué sous licence libre

BUT GEII Toulouse S3 ESE

Notions de base : logiciel et matériel

BREF HISTORIQUE DU MATÉRIEL RECONFIGURABLE

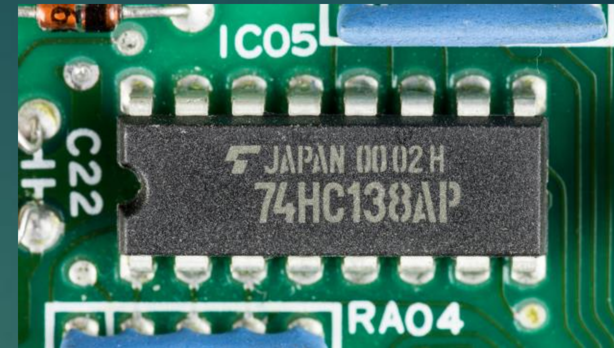
COMMENT CONCEVOIR LE CIRCUIT LOGIQUE ?

BUS D'INTERCONNEXION

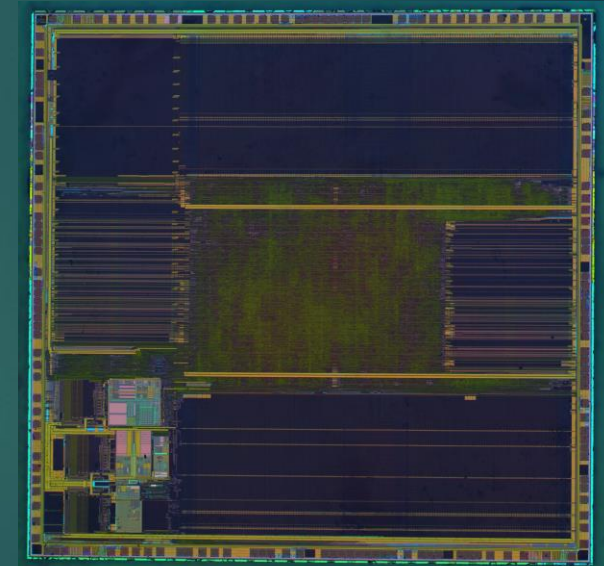
POINT DE SYNTAXE LANGAGE C

Logiciel et matériel : Le matériel

- Le matériel est un composant physique conçu pour réaliser une opération spécifique, par exemple :
  - Composant DIP (Dual In-line Packge)
  - ASIC (Application Specific Integrated Circuit)
  - Périphérique dans un microcontrôleur
- Un composant matériel est spécialisé dans une opération, et dispose d'un circuit logique conçu pour et dédié à cette tâche



74HC138 : composant DIP démultiplexeur



Vue au microscope électronique d'un STM32F103 : on remarque différentes zones pour le processeur, la mémoire et les différents périphériques

Note : on ne parlera dans ce cours que de composants numériques

Nedap ESD1 - printer controller - Toshiba 74HC138AP-91827.jpg © Raimond Spekking – CC BY-SA 4.0
STM32F103VGT6-HD.jpg © ZeptoBars – CC BY 3.0

Logiciel et matériel : Le logiciel

- Le logiciel est un programme s’exécutant sur un (ou plusieurs) cœur(s) de calcul
- Le logiciel peut être modifié et mis à jour sans modification physique du dispositif

```
void loop()
{
	static Etat_t etat = STOP;
	static uint32_t cc;

	switch (etat)
	{
	  case STOP:
	      if (Serial.available())
	      {
	          if (Serial.read() == 'G')
	          {
	              etat = GO;
	              cc = 0;
	              afficheMinuterie(cc);
	          }
	      }
	      break;
	  case GO:
	     cc++;
	     afficheMinuterie(cc);
	     if (Serial.available())
	      {
	          if (Serial.read() == 'H')
	          {
	              etat = PAUSE;
	          }
	      }
	      else if (cc == 6000)
	      {
	            etat = STOP;
	      }
	      break;
	  case PAUSE:
	      if (Serial.available())
	      {
	          if (Serial.read() == 'P')
	          {
	              etat = GO;
	          }
	      }
	      break;
	  }
	}
}
```

Une bribe de code logiciel

Matériel vs. logiciel

Logiciel

- Flexible
  - Peut être modifié sans intervention physique
  - La création d'une nouvelle fonctionnalité ne nécessite que le temps de développement du code
- Peu efficace
  - Un processeur doit pouvoir tout faire
    - Impossible d'optimiser le placement des éléments logiques pour une application
    - Exécution séquentielle du code
  - Très lent à consommation électrique équivalente, ou très consommateur à vitesse équivalente avec un ASIC

Matériel

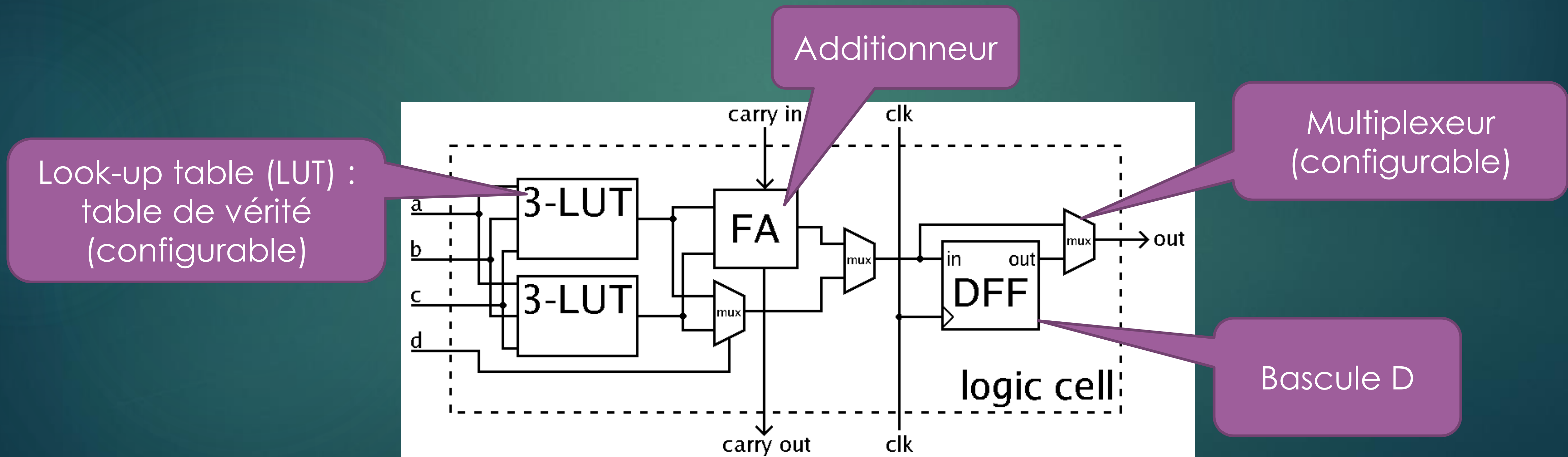
- Figé
  - La modification d'une fonction nécessite le remplacement d'un circuit
  - La création d'une nouvelle fonctionnalité nécessite la fabrication physique d'un nouveau composant (ASIC)
    - Extrêmement couteux, très long
- Très efficace
  - Le circuit étant conçu pour une tâche, il est optimisé pour elle
    - Placement des fonctions logiques réfléchi
    - Opérations réalisées en parallèle
  - Consommation électrique minimale et/ou vitesse maximale selon les choix de design

Une troisième voie : le matériel reconfigurable

- Et si… on imaginait un composant matériel, mais reprogrammable ?
- Les notions semblent antinomiques, et pourtant…

Une troisième voie : le matériel reconfigurable

- Imaginons une cellule capable de réaliser une petite fonction logique en remplissant une table de vérité configurable…



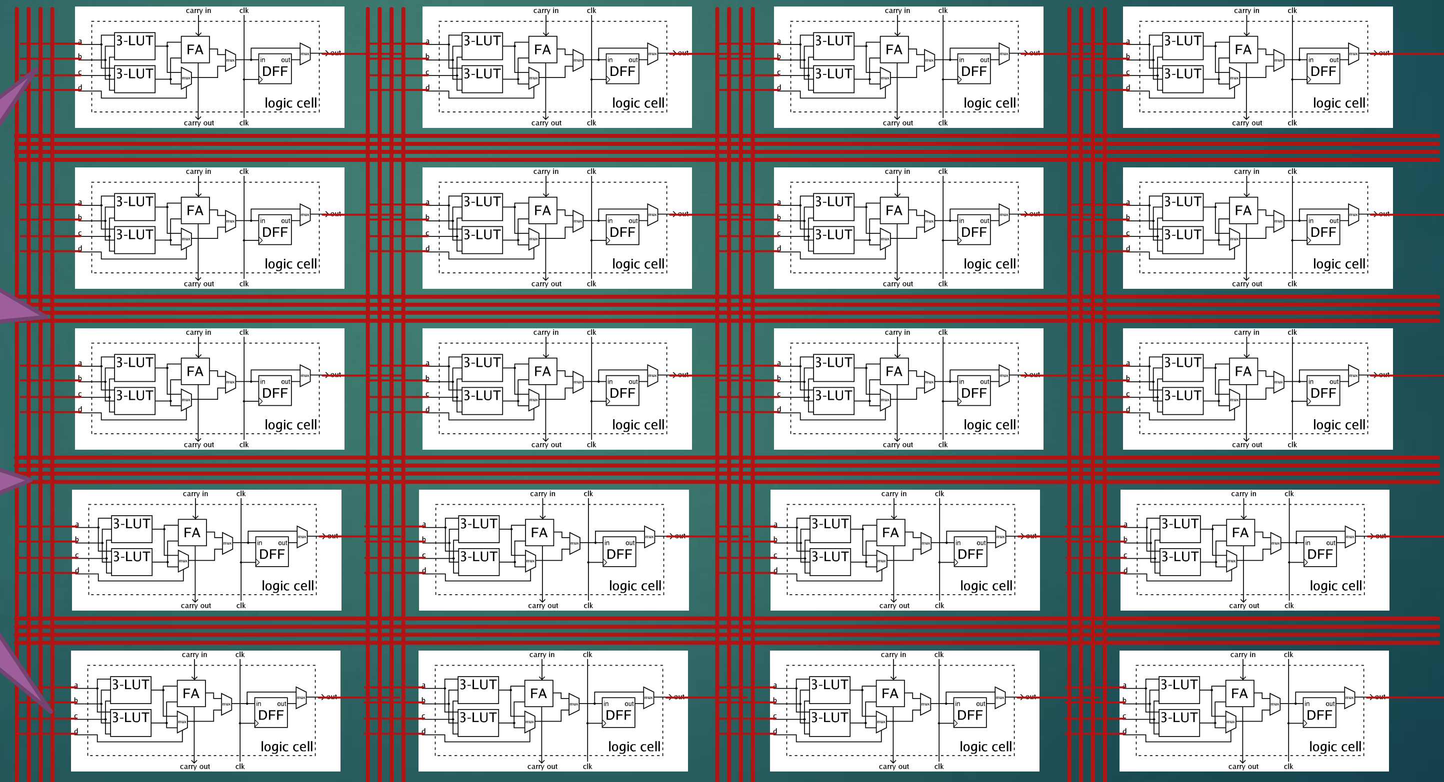
Exemple de cellule logique programmable

FPGA cell example.png © Petter.kallstrom – Public domain

Une troisième voie : le matériel reconfigurable

- … et mettons-en plein !

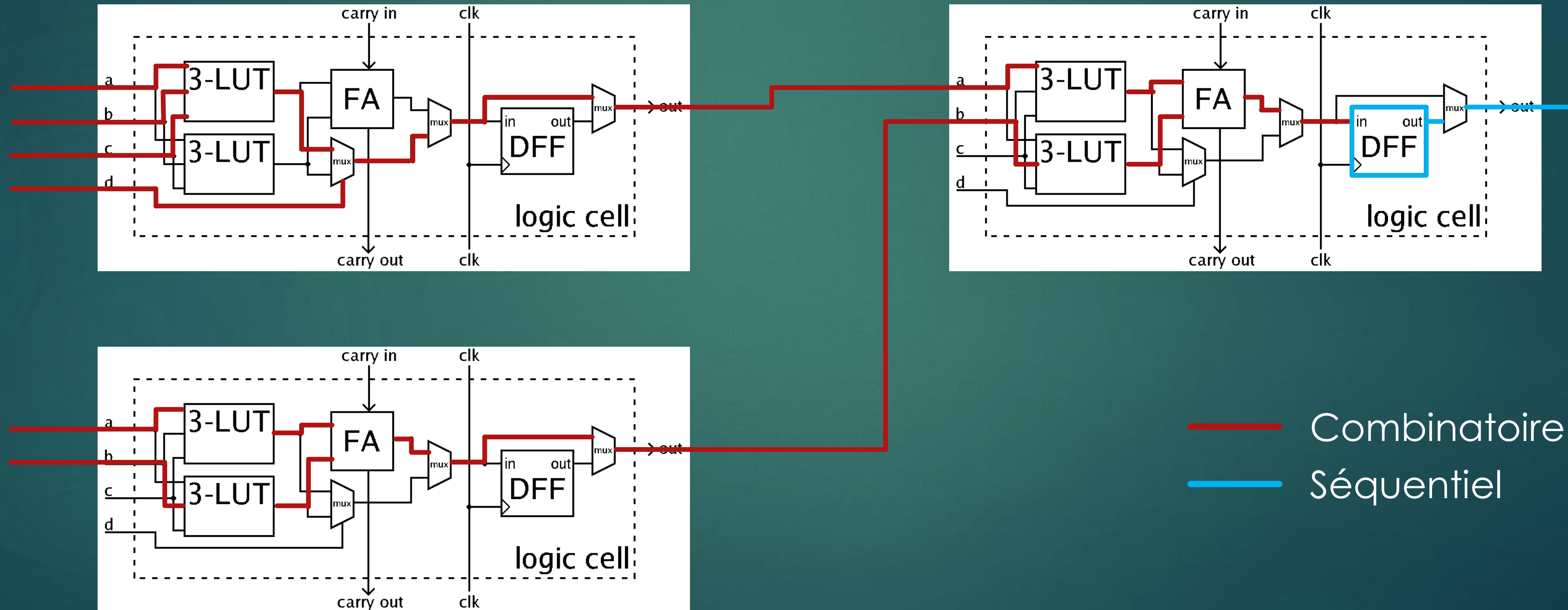
Routage (configurable) : Utilisation de transistors pour choisir quelles lignes sont connectées entre elles



FPGA cell example.png © Petter.kallstrom – Public domain

Une troisième voie : le matériel reconfigurable

- Exemple de placement / routage



FPGA cell example.png © Petter.kallstrom – Public domain

Matériel programmable : Avantages et inconvénients

- Par rapport au logiciel
  - Vitesse ++ (à consommation équivalente)
    - Mais pas autant qu'un ASIC
  - Consommation -- (à vitesse équivalente)
    - Mais pas autant qu'un ASIC
- Par rapport à un ASIC
  - Flexible
    - Reprogrammable sans fabrication dédiée couteuse

- Il s'agit donc d'une voie intermédiaire entre logiciel et matériel
- Épargne le cout et le temps de développement d'un ASIC, sans pour autant égaler les performances que l'on pourrait obtenir par la conception d'un circuit dédié
- Très utile pour :
  - Le prototypage
  - Les fabrications en petites quantités
  - Les dispositifs devant réagir au cycle d'horloge près et devant permettre une mise à jour

NOTIONS DE BASE : LOGICIEL ET MATÉRIEL

Bref historique du matériel reconfigurable

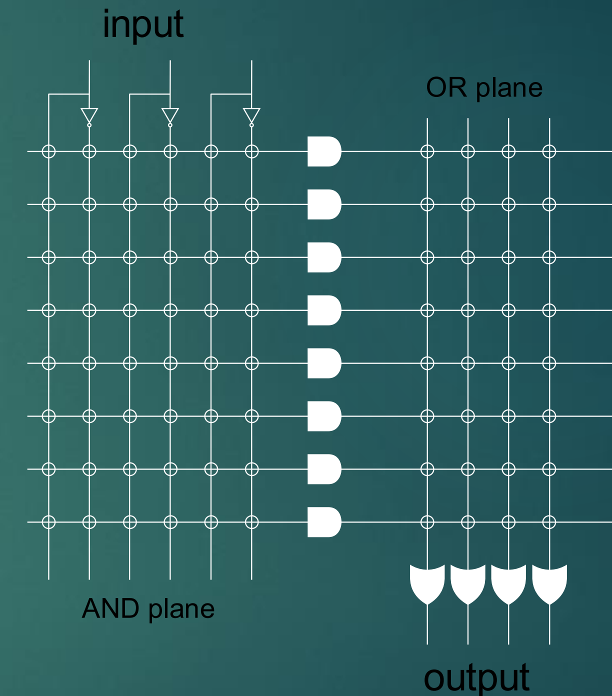
COMMENT CONCEVOIR LE CIRCUIT LOGIQUE ?

BUS D'INTERCONNEXION

POINT DE SYNTAXE LANGAGE C

Années '70
Programmable Logic Array (PLA)

- Matrice de fonction logiques
- On choisit les connexions à réaliser pour une fonction logique spécifique
- On « programme » le composant à la fabrication en concevant le masque utilisé pour la fabrication du circuit intégré

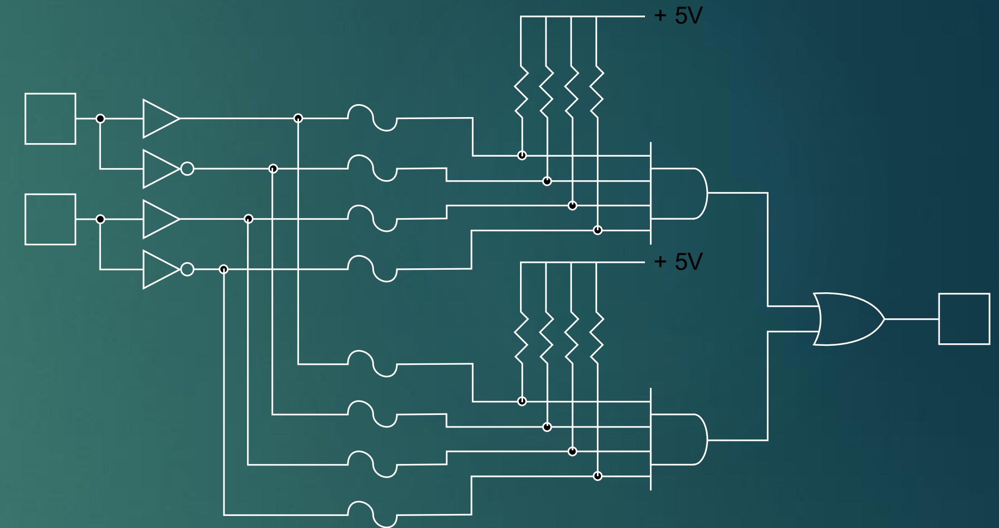


Un circuit PLA.
Les cercles représentent des connexions possibles
À la fabrication, on choisit quelles connexions sont réalisées

Programmable logic array (schematic drawing example).svg © Ilia Kr. – CC BY-SA 3.0

Années ’70
Programmable Arrays Logic (PAL)

- Matrice de fonction logiques
- On choisit les connexions à réaliser pour une fonction logique spécifique
- On programme le composant en flashant le contenu d’une mémoire qui pilote les connexions programmables
- Selon la génération, la mémoire peut être :
  - PROM (programmable une seule fois, antifusibles)
  - EPROM (la mémoire peut être effacée par UV pour être programmée à nouveau)
  - EEPROM (la mémoire est entièrement reprogrammable) – on parle alors aussi de Generic Array Logic (GAL)



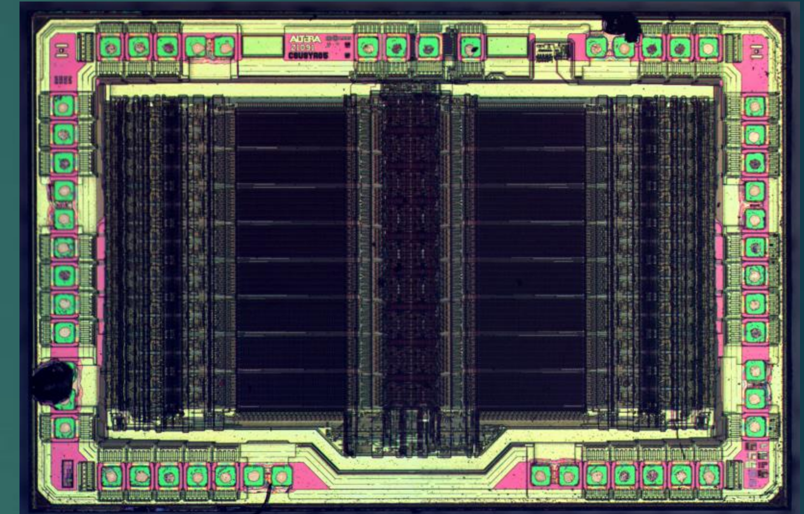
Simplified programmable logic device

Un circuit PAL. Les vagues représentent des connexions programmables

Programmable Logic Device.svg © DnetSvg – Public domain

Années ’80
Complex Programmable Logic Device (CPLD)

- Équivalent de nombreux GAL sur un seul circuit
- La programmation se fait toujours par choix du routage des bits individuels



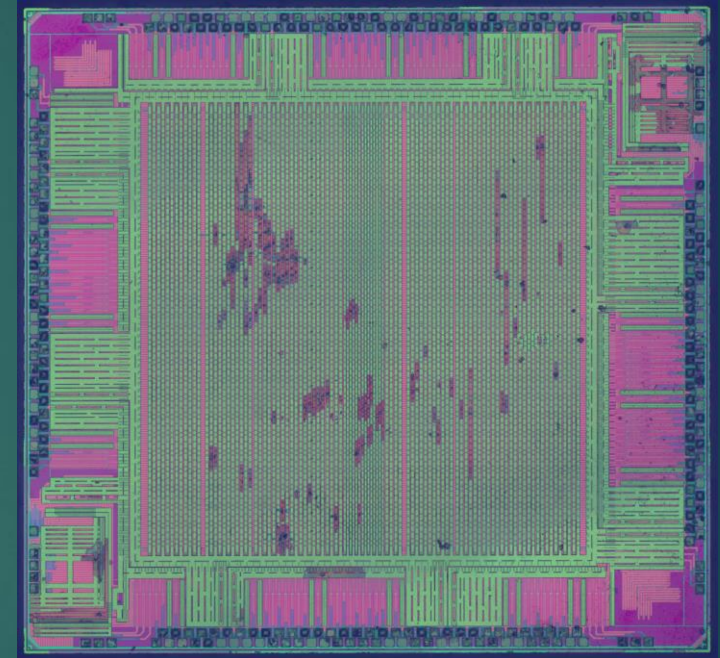
Altera EPM7032

Altera-epm7032-HD.jpg © ZeptoBars – CC BY 3.0

Années ’80

Field-Programmable Gate Array (FPGA)

- Programmation sur la base de Configurable Logic Blocks (CLB), qui rassemblent au minimum une ou plusieurs LUT et des bascules D
  - Les FPGA peuvent également intégrer des blocs spécialisés (additionneurs, multiplicateurs, etc.)
- La mémoire est en général volatile (ne survit pas à une perte d’alimentation)
  - Nécessite l’adjonction d’une mémoire externe pour être automatiquement reprogrammée au boot



Altera Cyclone 1

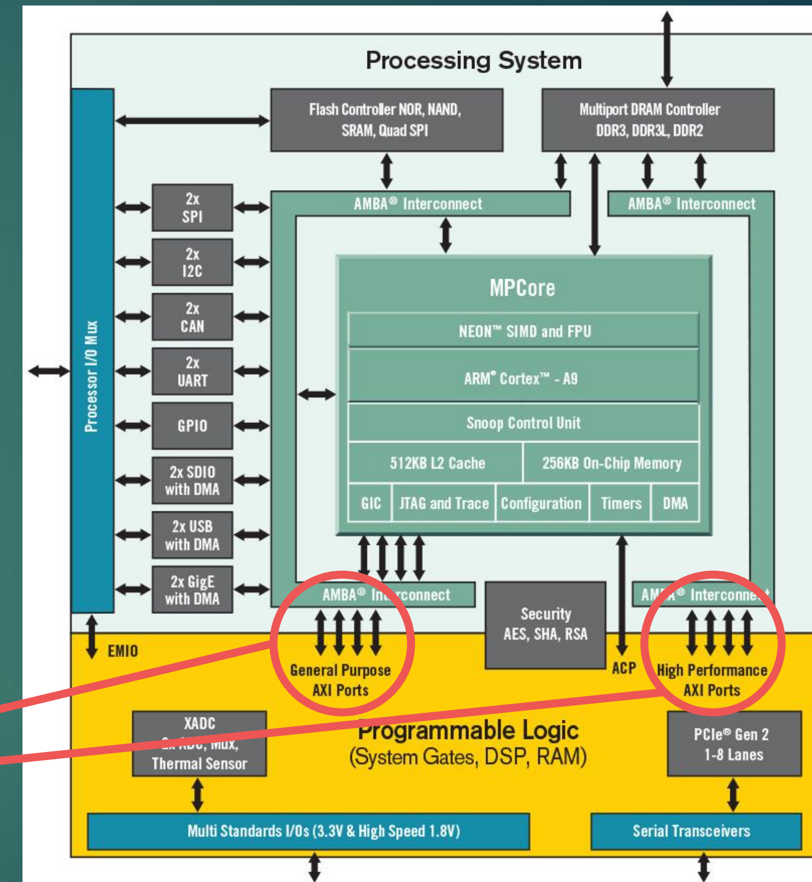
Altera-cyclone-1-fpga-HD.jpg © ZeptoBars – CC BY 3.0

Évolution des FPGA

- Le développement des FPGA a suivi une progression rapide
  - On a maintenant des centaines de millions de portes logiques, des blocs spécialisés et de la mémoire dans un FPGA
- Il est rapidement devenu possible de déployer des cœurs de processeurs sur un FPGA (on parle alors de processeur « softcore »)
  - Wait! Pourquoi faire ? On combine pas les désavantages du matériel avec les désavantages du logiciel, du coup ?
- L'idée est de créer des designs sur mesure, sur une seule puce : un ou plusieurs processeur(s), accompagné(s) d'un environnement spécialisé en déployant des périphériques spécialisés dédiés à une tâche
  - On parle alors de System On Programmable Chip (SOPC)
- La communication entre le processeur et les autres composants à l'intérieur du SOPC se fait généralement par un bus de communication

Du FPGA au SOPC

- L'utilisation de processeurs softcore se généralise
  - C'est quand même bien pratique, parfois, le software !
- Pourquoi ne pas intégrer directement un microcontrôleur dans le même chip que le FPGA ?
  - On parle alors de « hardwired core »
- Le bus du µP se prolonge dans la zone programmable, pour pouvoir ajouter des périphériques personnalisés



Xilinx Zynq 7000 : un microcontrôleur ARM couplé à de la logique programmable

Image © Xilinx

NOTIONS DE BASE : LOGICIEL ET MATÉRIEL

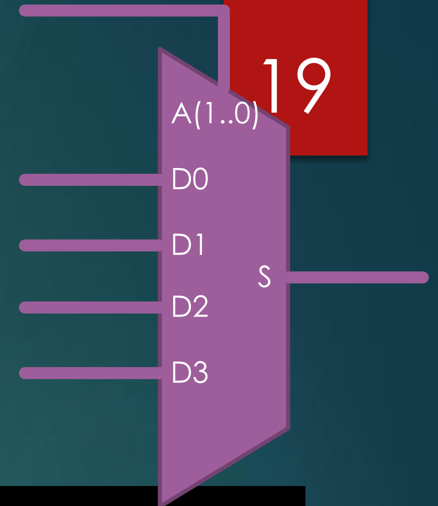
BREF HISTORIQUE DU MATÉRIEL RECONFIGURABLE

Comment concevoir le circuit logique ?

BUS D'INTERCONNEXION

POINT DE SYNTAXE LANGAGE C

Mais comment coder tout ça ?



- Dans les premières générations de composants logiques programmable, on déterminait bit à bit la valeur de chaque interconnexion via des équations logiques
- Dans les composants plus récents, c'est tout simplement impossible en raison de la complexité
- Des langages de description matériels (Hardware Description Language – HDL) ont donc été développés
- Le VHDL et le Verilog sont les deux langages de description les plus répandus

- Exemple de code en VHDL : multiplexeur 4 vers 1

```
entity Mux_4v1 is
port(D : in  std_logic_vector(3 downto 0);
     A : in  std_logic_vector(1 downto 0);
     S : out std_logic
    );
end entity;

architecture ar of Mux_4v1 is
begin
  with A select S <=
    D(0) when "00",
    D(1) when "01",
    D(2) when "10",
    D(3) when others;
end architecture;
```

A(1..0)	S
00	D(0)
01	D(1)
10	D(2)
11	D(3)

Vous étudierez le VHDL en 3ème année

Les IP

- Le développement des FPGA a permis de mettre à disposition des composants de la même manière que des bibliothèques logicielles
- On parle d'« Intellectual Property Blocks » (Blocs de propriété intellectuelle), ou IP Blocks, ou tout simplement IP
- Les IP peuvent être distribuées :
  - Directement sous forme de code HDL (open source hardware)
  - Sous forme illisible par un humain, le code pouvant alors être sécurisé pour n'être utilisable qu'avec une licence

Exemples de sites proposant du code matériel open source :
https://opencores.org
https://github.com/search?q=language:vhdl

NOTIONS DE BASE : LOGICIEL ET MATÉRIEL

BREF HISTORIQUE DU MATÉRIEL RECONFIGURABLE

COMMENT CONCEVOIR LE CIRCUIT LOGIQUE ?

Bus d'interconnexion

POINT DE SYNTAXE LANGAGE C

Bus sur puce (On-chip buses)

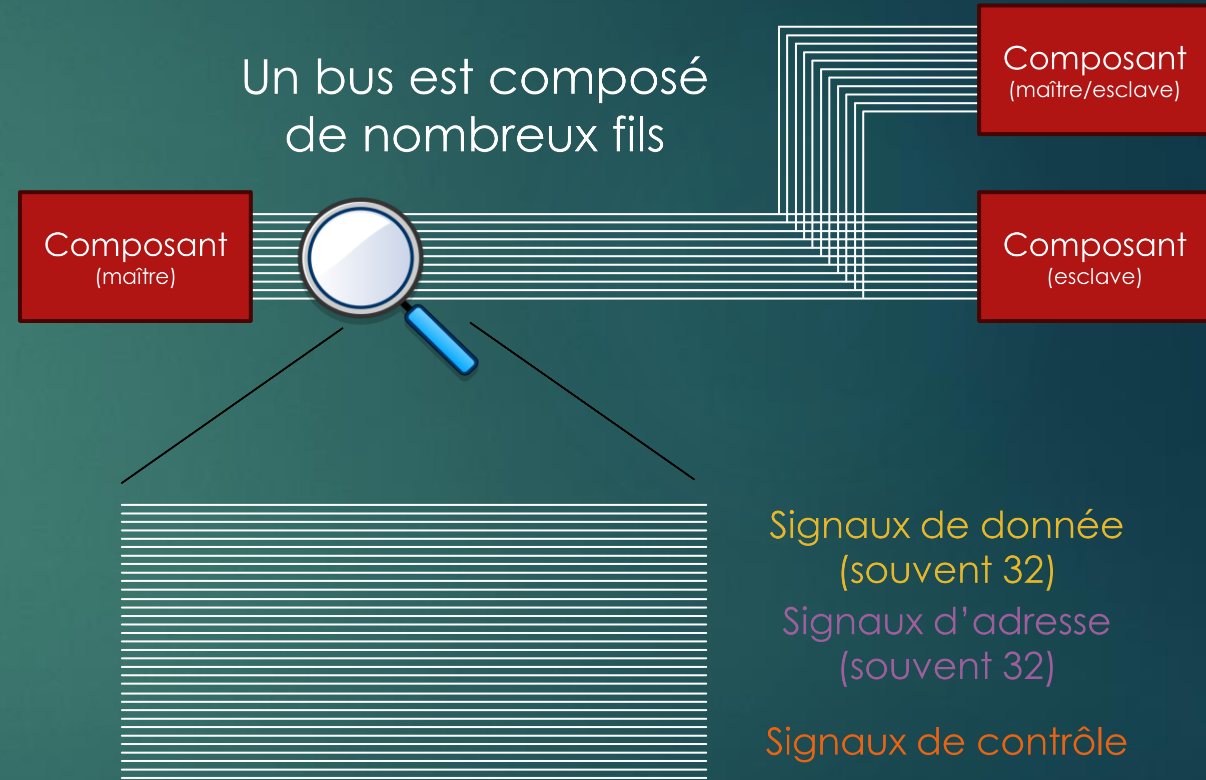
- Le processeur est toujours maître
- Un périphérique peut être maître et/ou esclave
- Une mémoire est toujours esclave

- Lorsque l'on travaille sur une puce, on utilise en général des bus parallèles plutôt que série, car le nombre de fils n'est pas un problème
  - Par exemple, un bus de 32 bits aura 32 fils de données
  - Les adresses sont aussi envoyées en parallèle (souvent sur 32 bits aussi)
- Quelques standards de bus
  - Avalon (Altera/Intel)
  - ARM AMBA (Xilinx/AMD)

- Les bus servent à faire transiter :
  - Des données
    - Entre le processeur et un périphérique
    - Entre le processeur et une mémoire
    - Entre périphériques
    - Entre un périphérique et une mémoire
  - Des instructions
    - entre le processeur et une mémoire
- Généralement, on utilise deux bus distincts : un pour les données et un pour les instructions
  - on parle alors d'architecture *Harvard*, par opposition à l'architecture *Von Neumann* qui n'utilise qu'un seul bus

Structure d’un bus parallèle

- Un bus relie 2 composants ou plus
- Les signaux (ou fils) composant le bus sont séparés en trois « catégories », ou « canaux » :
  - Signaux de donnée
    - Pour faire transiter les données entre les périphériques
  - Signaux d’adresse
    - Pour indiquer à quel périphérique on s’adresse
  - Signaux de contrôle
    - Pour paramétrer la transaction (lecture/écriture, nombre de mots envoyés, taille des données envoyées, etc.)



Gnome-searchtool blue.svg © David Vignoni – LGPL 2.1

Horloge et reset

- En plus d'être reliés au(x) bus, tous les composants doivent être reliés à deux signaux supplémentaires : l'horloge et le signal de reset
- Les composants (ou tout au moins leur interface de bus) doivent être synchronisés pour que les signaux sur le bus soient compris de tous : ils partagent donc une horloge commune
  - Les composants peuvent par ailleurs disposer d'autres horloges pour leur comportement interne
- Chaque composant doit être initialisé au démarrage, et pouvoir être réinitialisé au besoin : un signal de reset est donc également fourni à tous les composants
  - On parle de reset « asynchrone », car il peut intervenir à n'importe quel moment
  - Les composants peuvent par ailleurs disposer d'autres manières de se réinitialiser, par exemple par écriture dans un registre : on parle alors de reset « synchrone », car l'information de reset sera prise en compte, comme n'importe quelle autre information, sur un front d'horloge

Topologies de bus sur puce

- Médium partagé (Shared-medium)
  - Un seul canal d'adresse (*n* bits) et un seul canal de données (*m* bits), partagés dans le temps entre les périphériques
- Crossbar
  - Plusieurs canaux directs entre périphériques, utilisables en parallèle (pour *c* canaux, $c \times (n+m)$ bits)
- Réseau sur puce (Network-on-chip – NOC)
  - Réseau commuté avec des routeurs faisant transiter les données entre les IP
  - Type de réseau lui-même configurable, voir par exemple Wormhole switching pour votre culture générale

Notion de registre et d’adresse

- Chaque composant a une adresse de base et une taille mémoire
  - Par exemple, un composant placé à l’adresse 0x2000 avec une taille de 0x10 occupera toutes les adresses de 0x2000 à 0x200F
  - 0x2000 – 0x200F est ce que l’on appelle la *plage mémoire* du composant
- Un *registre* est une case mémoire disponible dans un composant à une *adresse* spécifique par rapport à son adresse de base
  - Toutes les adresses dans la plage d’un composant ne sont pas forcément des registres
- Un registre peut être en lecture seule (R), en écriture seule (W) ou en lecture/écriture (R/W)
  - Attention ! La lecture d’un registre R/W ne retourne pas forcément ce qui y a été écrit !

Attention au calcul ! Adresse de fin de la plage mémoire = début + taille - 1

Exemple de registres : Xilinx AXI IIC Bus interface

- Périphérique I2C fourni par Xilinx
  - Voir la doc complète
- Il s'agit des adresses au sein du périphérique : il faut ajouter l'adresse de base du périphérique pour obtenir l'adresse d'un registre sur le bus
  - Si on place le périphérique à l'adresse 0x1000 sur le bus, quelle sera l'adresse de son registre CR ?
  - CR sera alors à l'adresse 0x1100
- En général, les registres sont alignés sur la largeur du bus
  - Par exemple, sur un bus 32 bits (= 4 octets), les adresses sont alignés sur des multiples de 4
- Notez que certaines adresses n'existent pas dans ce composant ! (elles sont inutilisées)

Par exemple, les adresses 0x044 à 0x0FF sont inutilisées

Le bus étant sur 32 bits, le registre CR utilise les adresses 0x100, 0x101, 0x102 et 0x103 pour stocker ses 4 octets

Le registre suivant est donc à l'adresse 0x104

Pas de « -1 » ici : il peut y avoir un registre à l'adresse 0 de la plage mémoire du composant !

Address (hex)	Register Name	Access Type	Default Value (hex)
	Interrupt Registers		
0x01C	nterrupt Enable (GIE)[1]	Read/Write	0x0
0x020	nterrupt Status Register (ISR)[1]	Read/Toggle[3] on Write	0xD0
0x028	Interrupt Enable Register (IER)[1]	Read/Write	0x0
	Soft Reset		
0x040	Soft Reset Register (SOFTR)[2]	Write Only	N/A
	IIC Configuration, Control, Data		
0x100	Control Register (CR)	Read/Write	0x0
0x104	Status Register (SR)		
0x108	Transmit		
	O (RX_FIFO)		
	ss Register (ADR)		
	O Occupancy Register (TX_FIFO_OCY)	Read	0x0
	O Occupancy Register (RX_FIFO_OCY)	Read	0x0
	t Address Register (TEN_ADR)	Read/Write	0x0
	O Programmable Depth Interrupt Register PIRQ)	Read/Write	0x0
0x124	General Purpose Output Register (GPO)	Read/Write	0x0
0x128	Timing Parameter TSUSTA Register	Read/Write	See Note[4]
0x12C	Timing Parameter TSUSTO Register	Read/Write	See Note[4]
0x130	Timing Parameter THDSTA Register	Read/Write	See Note[4]
0x134	Timing Parameter TSUDAT Register	Read/Write	See Note[4]
0x138	Timing Parameter TBUF Register	Read/Write	See Note[4]
0x13C	Timing Parameter THIGH Register	Read/Write	See Note[4]
0x140	Timing Parameter TLOW Register	Read/Write	See Note[4]

Image © Xilinx

Exemple – Bus AXI

- Le bus AXI (Advanced eXtensible Interface) fait partie de la famille de bus AMBA (Advanced Microcontroller Bus Architecture), développée par ARM
- Utilisé par la famille de FPGA Zynq de Xilinx
- Composé de 5 canaux (chacun ayant ses propres signaux)
  - Read Address channel (AR)
  - Read Data channel (R)
  - Write Address channel (AW)
  - Write Data channel (W)
  - Write Response channel (B)

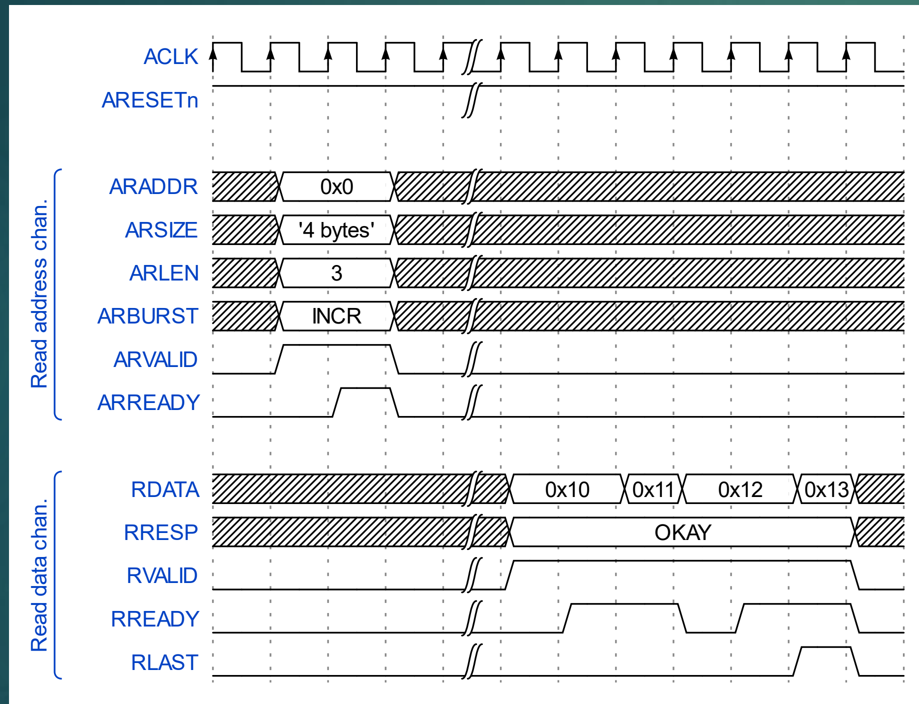
Deux canaux d'adresse, deux canaux de données : le bus est full duplex

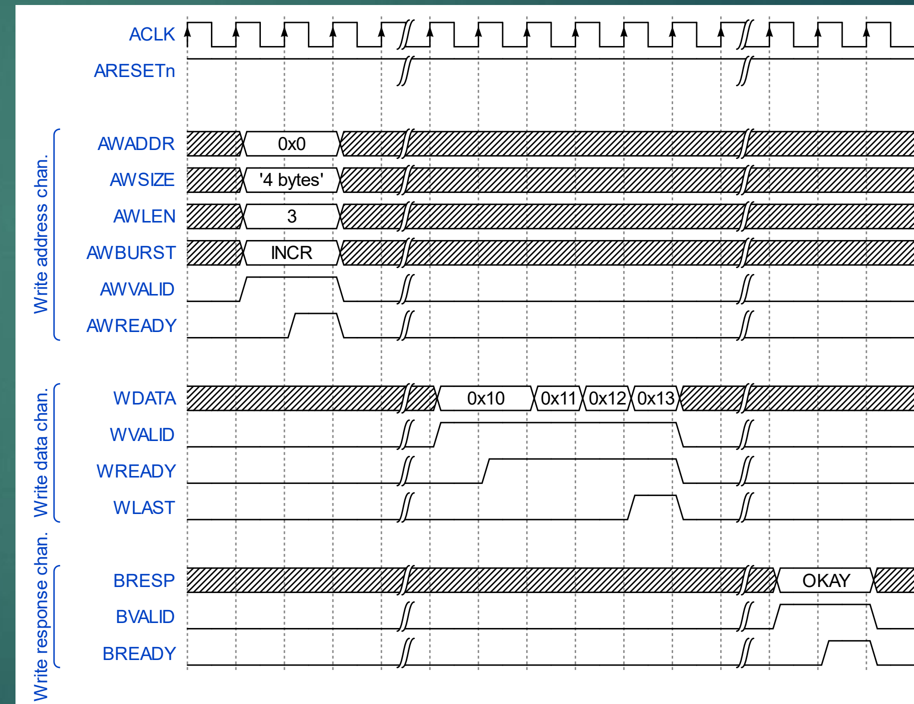
- Les bus de données peuvent être configurés de 8 bits à 1024 bits
- Topologies supportées :
  - Bus d'adresse et de données partagés (shared medium)
  - Bus d'adresse partagé, multiples bus de données
  - Bus d'adresse et de données multiples (crossbar)
- Voir la spécification complète ici

Exemple – Bus AXI
Quelques exemples de signaux

- Signaux globaux
  - ACLK : horloge
  - ARESETn : reset
- Canal « Write address »
  - AWADDR : adresse d'écriture
  - AWLEN : nombre de mots à transférer
  - AWSIZE : taille des mots à transférer
  - AWVALID : validation de l'adresse (= enable)
  - … et 8 autres signaux
- Canal « Write data »
  - WDATA : donnée
  - WVALID : validation de la donnée (= enable)
  - … et 5 autre signaux
- Canal « Write response »
  - BVALID : indique que l'adresse est valide
  - BREADY : indique que le maître accepte une réponse au write
  - … et 3 autres signaux

Exemple – Bus AXI Chonogrammes

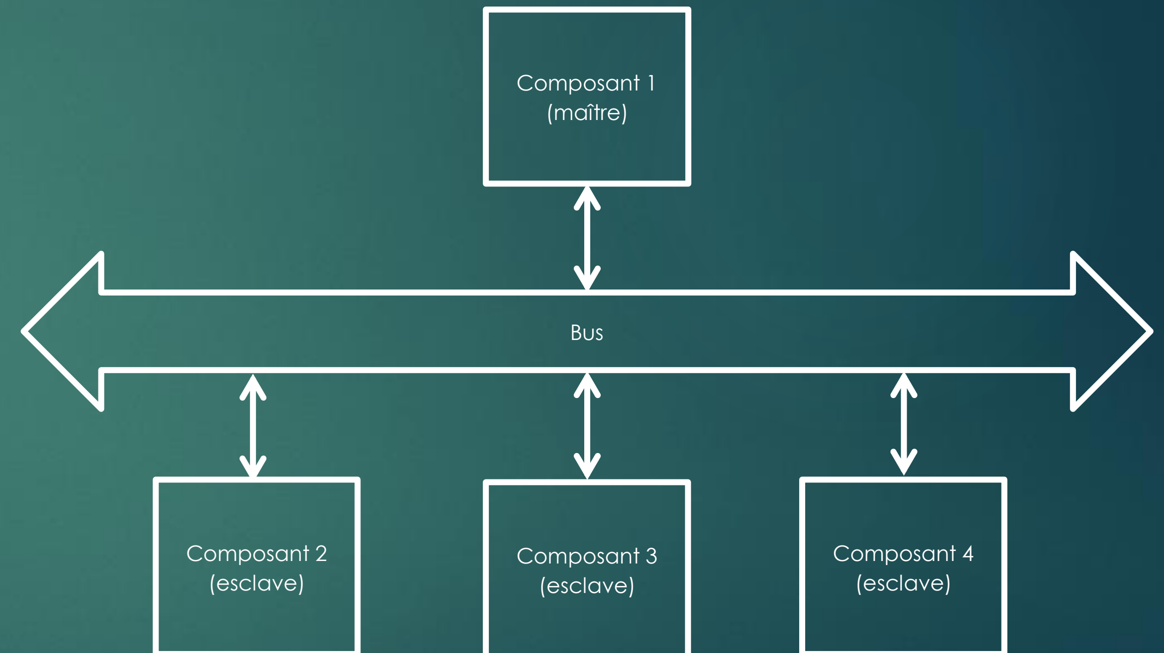




AXI read transaction.svg © Nicola Corna – CC BY-SA 4.0
AXI write transaction.svg © Nicola Corna – CC BY-SA 4.0

Représentation simplifiée d'un système basé sur un bus

- Une représentation simplifiée d'un système architecturé autour d'un bus peut être faite de la manière ci-contre
- Par convention, on place les composants maître au-dessus du bus, et les composants esclaves en dessous
  - Et si un composant est maître-esclave ?
  - On le met là où il y a le plus de place…



NOTIONS DE BASE : LOGICIEL ET MATÉRIEL
BREF HISTORIQUE DU MATÉRIEL RECONFIGURABLE
COMMENT CONCEVOIR LE CIRCUIT LOGIQUE ?
BUS D’INTERCONNEXION

Point de syntaxe langage C

Point de syntaxe langage C : Adresses et pointeurs

- Le langage C (également en C++) définit un type de variable particulier : les pointeurs
- Un pointeur est une adresse : adresse d'un registre sur le bus, ou adresse d'une variable dans la mémoire
  - Dans ce cours, nous utiliserons les pointeurs uniquement pour accéder à des registres sur le bus
  - Des informations complémentaires pour l'utilisation sur des variables sont disponibles dans les annexes en fin de cours
- Un pointeur permet donc d'accéder au contenu d'une adresse (ici, au contenu d'un registre), en lecture et/ou en écriture
- La syntaxe des pointeurs peut être un peu déroutante au début, soyez très attentifs à la suite !

Point de syntaxe langage C : Adresses et pointeurs

- L'opérateur indiquant un pointeur est « * »
- Lors de la définition d'une variable de type pointeur, il faut ajouter cet opérateur après le type de la variable
- Par exemple, pour un bus d'adresse de 32 bits, on déclarera une adresse vers un registre de 8 bits comme ceci :

```
uint8_t* monAdresse;
monAdresse = 0x00001100;
```

- Pour manipuler la donnée située dans le registre (et non le pointeur) on ajoute l'opérateur « * » avant le nom de la variable :

```
*monAdresse = 0x01;
uint8_t valeurRegistre = *monAdresse;
```

- Ainsi, on ne modifie pas *le pointeur* mais *le registre situé à l'adresse stockée dans le pointeur*
- Mnémotechnique : regardez la déclaration, et ajoutez mentalement des parenthèses autour de ce qui vous intéresse :

```
(uint8_t*) (monAdresse)
(uint8_t) (*monAdresse)
```

*monAdresse est de type uint8_t : c'est donc une valeur

monAdresse est de type uint8_t* : c'est donc un pointeur

Par exemple

- Sur l'exemple de tout à l'heure, on peut donc manipuler les registres CR et SR de la manière suivante
- En supposant que le périphérique est à l'adresse 0x0001000 sur le bus :

```c
// Adresse de base du périphérique et offset :
#define I2C_BASE_ADDR 0x00001000
#define I2C_CR_OFFSET 0x100
#define I2C_SR_OFFSET 0x104

// Définitions de pointeurs sur les registres :
volatile uint8_t*  const I2C_CONTROL_REG = I2C_BASE_ADDR + I2C_CR_OFFSET;
volatile uint16_t* const I2C_STATUS_REG  = I2C_BASE_ADDR + I2C_SR_OFFSET;

// Manipulation des pointeurs :
*I2C_CONTROL_REG = 0x01; // Activer le périphérique I2C
uint16_t statut = *I2C_STATUS_REG; // Lire le statut
```

Les #define se placent en haut du fichier

Ces variables peuvent être également placées en haut du fichier (variables globales)

Ce code va dans une fonction !

Attention : certain registres ne sont accessibles qu'en lecture ou qu'en écriture

Address (hex)	Register Name	Access Type	Default Value (hex)
	Interrupt Registers		
0x01C	Global Interrupt Enable (GIE)[1]	Read/Write	0x0
0x020	Interrupt Status Register (ISR)[1]	Read/Toggle[3] on Write	0xD0
0x028	Interrupt Enable Register (IER)[1]	Read/Write	0x0
	Soft Reset		
0x040	Soft Reset Register (SOFTR)[2]	Write Only	N/A
	IIC Configuration, Control, Data		
0x100	Control Register (CR)	Read/Write	0x0
0x104	Status Register (SR)	Read	0xC0
0x108	Transmit FIFO (TX_FIFO)	[illegible]	0x0
0x10C	Receive FIFO (RX_FIFO)	Read	N/A
0x110	Slave Address Register (ADR)	Read/Write	0x0
0x114	Transmit FIFO Occupancy Register (TX_FIFO_OCY)	Read	0x0
0x118	Receive FIFO Occupancy Register (RX_FIFO_OCY)	Read	0x0
0x11C	Slave Ten Bit Address Register (TEN_ADR)	Read/Write	0x0
0x120	Receive FIFO Programmable Depth Interrupt Register (RX_FIFO_PIRQ)	Read/Write	0x0
0x124	General Purpose Output Register (GPO)	Read/Write	0x0
0x128	Timing Parameter TSUSTA Register	Read/Write	See Note[4]
0x12C	Timing Parameter TSUSTO Register	Read/Write	See Note[4]
0x130	Timing Parameter THDSTA Register	Read/Write	See Note[4]
0x134	Timing Parameter TSUDAT Register	Read/Write	See Note[4]
0x138	Timing Parameter TBUF Register	Read/Write	See Note[4]
0x13C	Timing Parameter THIGH Register	Read/Write	See Note[4]
0x140	Timing Parameter TLOW Register	Read/Write	See Note[4]

#define est une syntaxe particulière permettant remplacer dans votre code une valeur (ici I2C_BASE_ADDR) par une autre (ici 0x00001000). Ne pas utiliser de constante dans le calcul des pointeurs !

Image © Xilinx

Et les « volatile » et les « const » ?


```
volatile uint8_t*  const I2C_CONTROL_REG = I2C_BASE_ADDR + 0x100;
volatile uint16_t* const I2C_STATUS_REG  = I2C_BASE_ADDR + 0x104;
```

- Dans la déclaration ci-dessus, deux mots-clés supplémentaires :
  - volatile
    - Indique que cette variable ne doit pas être optimisée par le compilateur
  - const
    - Vous le connaissez, mais pas placé à cet endroit
    - Placé ici (après l'*), il signifie que le pointeur est constant (l'adresse du périphérique ne change effectivement pas), mais pas le contenu du registre
- Pour en savoir plus sur ces qualificateurs, voir les annexes en fin de cours
- La syntaxe à retenir est la suivante :

```
volatile <taille>* const <nom du registre> = <adresse du registre>;
```

Point d'entrée du programme

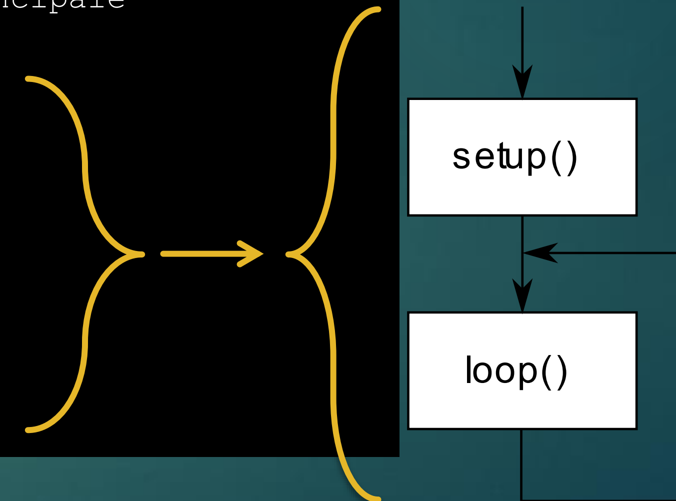
- Vous avez l'habitude des fonctions `setup()` et `loop()`
- En réalité, le point d'entrée d'un programme C ou C++ est le `main()`
- Les programmes que vous avez écrit jusqu'à présent ont déjà un `main()` (caché) qui a la forme ci-contre
- Dans ce module, vous devrez travailler avec le `main()`
- On conservera la même structure :
  - Initialisation une seule fois
  - Boucle principale à l'infini

```
void setup()
{
	// Initialisation
}

void loop()
{
	// Boucle principale
}

void main()
{
	setup();

	while(1)
	{
		loop();
	}
}
```



Annexes

Annexes

- Les slides présentés dans cette section ne font pas partie du cours
- Mais si vous avez envie de mieux comprendre certaines notions de langage C, vous pouvez vous y référer
- Ces notions seront d'ailleurs très probablement utiles dans des cours ultérieurs, donc pourquoi ne pas prendre un peu d'avance 😉 ?

Annexe : Position des qualificateurs

- Un qualificateur de type (par exemple « `const` » ou « `volatile` ») se place toujours *après* ce qu'il modifie

```
uint32_t volatile * const I2C_CONTROL_REG;
```

  - Ici, `volatile` s'applique donc au `uint32_t` (la valeur pointée), et `const` s'applique au `*` (le pointeur)
  - On a donc un pointeur dont l'adresse ne peut pas changer vers une variable qui ne doit pas être optimisée
- Oui, mais c'est pas ce qui était écrit sur les slides précédents !

```
volatile uint32_t* const I2C_CONTROL_REG;
```

  - C'est une exception à la règle : un qualificateur placé en début de ligne s'applique à ce qui la suit
  - C'est pourquoi vous avez toujours écrit :

```
const uint32_t MA_CONSTANTE = 10;
```

  - Mais maintenant, vous savez que c'est la même chose que :

```
uint32_t const MA_CONSTANTE = 10;
```

Annexe : Le qualificateur `volatile`

- Le mot-clé `volatile` « indique que cette variable ne doit pas être optimisée par le compilateur », mais ça veut dire quoi ?
- Lorsque votre code est compilé, le compilateur l'optimise
- Par exemple, si vous écrivez

```
const int x = 10;
const int y = 2;
int z = x + y;
```

- Il est tout à fait possible que le compilateur supprime les variables x et y, et compile en fait :

```
int z = 12;
```

- De manière générale, si le contenu d'une case mémoire n'a pas été modifié entre deux utilisations, le code généré par le compilateur ne va pas lire à nouveau sa valeur mais réutiliser la valeur précédente sans ré-accéder à la mémoire
- Ça épargne des accès mémoire, c'est bien !
- MAIS si c'est un registre, dont la valeur peut changer à tout instant indépendamment du code, alors il y a problème ! Si on fait plusieurs lectures successives (par exemple dans une boucle pour attendre une valeur dans un registre), il n'aura pas d'accès réel au registre ➔ boucle infinie !

Annexe :
Pointeur vers une variable

- Pour pointer vers une variable, le pointeur doit indiquer le type de la variable pointée
- Par exemple, un pointeur vers une variable de type int se note :

```
int* monPointeur;
```

- Contrairement à un registre, dont on connait précisément l'adresse sur le bus, on ne fixe jamais l'adresse d'une variable dans la mémoire !

```
int* monPointeur = 0x00000010;
```

Interdit !
Il y a peut-être déjà une variable à cette adresse, ou cette adresse n'est peut-être même pas dans la mémoire !

- À la place, on utilise l'opérateur « & » pour obtenir l'adresse d'une autre variable :

```
int maVariable = 42;
int* monPointeur = &maVariable;
*monPointeur = 21;
```

Ici, on manipule `maVariable` via son adresse

Annexe : Arithmétique des pointeurs

- Pourquoi faut-il utiliser des `#define` pour déclarer les adresses de base et les offset ? Une constante ferait l'affaire, non ?
- Voyons ce qu'il se passe dans les cas suivants :

```
volatile uint16_t* const PERIPH_1_BASE_ADDR = 0x1000;
volatile uint8_t*  const PERIPH_2_BASE_ADDR = 0x2000;

const int REG_1_OFFSET = 1;

volatile uint16_t* const PERIPH_1_REG_1 = PERIPH_1_BASE_ADDR + REG_1_OFFSET;
volatile uint8_t*  const PERIPH_2_REG_1 = PERIPH_2_BASE_ADDR + REG_1_OFFSET;
```

- À votre avis, combien valent `PERIPH_1_REG_1` et `PERIPH_2_REG_1` ?
- L'arithmétique des pointeurs prend en compte la taille des variables pointées : par exemple une variable de 16 bits « utilise » 2 adresses sur le bus, un pointeur vers une telle variable sera donc incrémenté par pas de 2 (1 pour 8 bits ; 4 pour 32 bits ; 16 pour 64 bits, etc.)

Vaut 0x1002

Vaut 0x2001