LiteX : une approche haut niveau pour simplifier l'intégration d'algorithmes sur FPGA

G. Goavec-Merou - F. Kermarrec - J.-M. Friedt
Email: gwenhael.goavec-merou@trabucayre.com - florent@enjoy-digital.fr - jmfriedt@femto-st.fr

Indépendant, Le Conquet, France
Enjoy-Digital, Landivisiau, France
FEMTO-ST Time & Frequency department, Besançon, France

2 octobre 2026

Contexte

Nous allons parler exclusivement de logiciel et de description.

- le matériel évolue en permanence
  - nouvelles générations de FPGA
  - nouveaux ADC/DAC, plus rapides et à meilleure résolution
  - nouvelles cartes d'extension
  - révision d'une carte existante
- il change également selon les besoins :
  - vitesse ou résolution
  - taille de la matrice
  - interfaces et communication

→ Le matériel est adapté au besoin.

- le logiciel doit rester stable, viable et (idéalement) à l'état de l'art
- les algorithmes évoluent seulement si nécessaire (FIR / FFT / DDC / DUC)

→ Le logiciel doit rester réutilisable.

Dans la durée :

- un nouveau matériel peut imposer un nouvel environnement (Spartan-6 → Zynq)
- les langages lourds compliquent la maintenance, l'évolution et la migration
- la dépendance à un logiciel propriétaire expose à l'abandon de l'outil et aux ruptures de compatibilité (abandon de `PetaLinux` par AMD/Xilinx à partir de 2026.02)

Pourquoi encore changer ? Cela fonctionne depuis des années

Les HDL historiques restent efficaces, mais ils sont verbeux, complexes et manquent d'expressivité.

VHDL :

```
signal count : std_logic_vector(7 downto 0);

count <= std_logic_vector(unsigned(count) + 1);
```

Verilog :

```
reg [7:0] count;

count <= count + 1'b1;
```

- le Block Design facilite l'assemblage, mais le format graphique reste difficile à relire, à comparer et à fusionner dans un VCS (petite modification → beaucoup de lignes changées)
- l'approche TCL :
  - rend la construction textuelle et reproductible,
  - sert à instancier et à câbler les modules
  - implique l'utilisation d'un second langage dans le flux de développement
  - est **totalement liée** à la plate-forme, aux procédures et au logiciel du fournisseur

→ Réutiliser ou transposer le projet vers un autre outil reste complexe/difficile/impossible.

- changement d'outil
- changement de FPGA/CPU
- changement de procédures TCL

Deux approches alternatives :

- nouveaux langages : Silice, SpinalHDL, Chisel
- surcouches : Migen, Amaranth

⇒ **Pourquoi ne pas passer à un environnement simple, cohérent et indépendant du matériel ?**

Migen / LiteX - Python pour générer le matériel

```
import numpy as np
from scipy.signal import firls, get_window
from migen import *
from litex.gen import *

class FIR(LiteXModule):
    def __init__(self, n, width, fs,
                 f_pass, f_stop, window="hann"):
        taps = firls(n, [0, f_pass, f_stop, fs/2],
                        [1, 1, 0, 0], fs=fs)
        taps *= get_window(window, n, fftbins=False)

        peak   = np.max(np.abs(taps))
        gain   = (2**(width-1)-1) / peak
        coeffs = np.round(taps * gain).astype(int)

        self.coeff_addr = Signal(max=n)
        self.coeff_dat  = Signal((width, True))

        rom = Memory(width, n, init=coeffs.tolist())
        rd  = rom.get_port(async_read=True)
        self.specials += rom, rd
        self.comb += [
            rd.adr.eq(self.coeff_addr),
            self.coeff_dat.eq(rd.dat_r)
        ]
        [...]
```

Principes

- comme sous GNU Octave : `firls`, window, norm et arrondi
- `peak = max(abs(taps))`
- normalisation des coefficients par le gain (FSR) `width`
- ROM chargée avec `coeff` (en entier)
- `coeff_addr` adresse la ROM/LUT

Exemple

`FIR(63, 16, 125e6, 10e6, 12e6, "hann")`

→ Le même environnement pour la préparation et la description du core

LiteX - Écosystème Python pour FPGA

- écosystème Python reposant sur Migen
- description, intégration et interconnexion de cores
- gestion complète de la chaîne d'outils :
  - génération du HDL et des contraintes
  - création du projet et appel des outils
  - build et chargement du bitstream
- intégration de processeurs :
  - softcores (VexRiscV, Serv, femtoRV, ...)
  - cœurs ARM intégrés aux Zynq-7000, ZynqMP, SoCFPGA, ...



Source : Enjoy-Digital/LiteX

Prend en charge la majorité des outils :

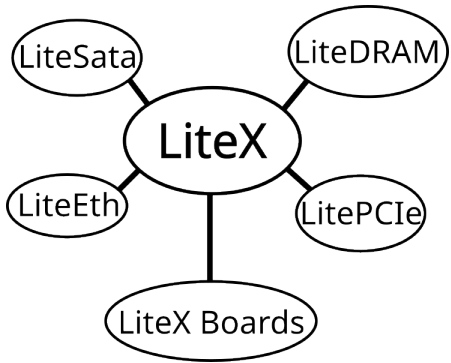
- Xilinx : Vivado, ISE et Open-Source (prjxray)
- Altera : Quartus
- Lattice : Diamond, Radiant et Open-Source (icestorm, prjtrellis, prjoxyde)
- CologneChip : prjpeppercorn (Open-Source) et p_r (obsolète)
- Efinix : Efinity
- Microchip : Libero
- Gowin : EDA et Open-Source (Apicula)

→ Des dépôts satellites ajoutent des fonctions selon les besoins.

Dépôts satellites - Liste non exhaustive

LiteX ou une target peut utiliser ces dépôts pour ajouter des capacités additionnelles en fonction des besoins et des caractéristiques de la carte.

- LiteSata : contrôleur SATA (protocole et bas niveau)
- LiteDRAM : contrôleur DRAM/SDRAM/DDRx (spécification et bas niveau)
- LiteEth : contrôleur Ethernet (protocoles IPv4/UDP/ARP/TFTP et bas niveau)
- LitePCIe : contrôleur PCIe (Gen1/2/3) et bas niveau
- **LiteX Boards** : définition des cartes supportées :
  - `platforms` : description de la plate-forme (brochage/modèle de FPGA, ...)
  - `targets` : assemblage/configuration des horloges. Script exécutable pour produire le bitstream

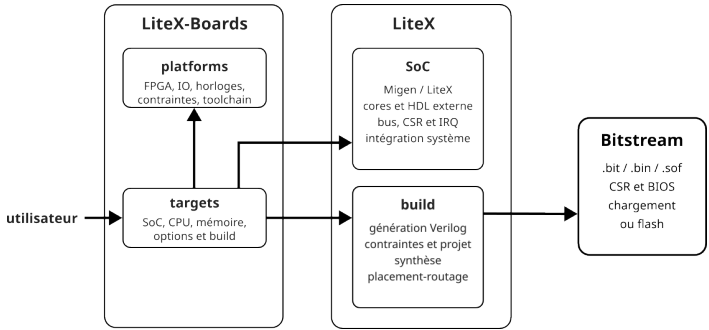


Pour l'ensemble des dépôts, le code est découpé entre :

- l'implémentation du protocole/de la spécification
- la partie `phy` : code spécifique à chaque type/famille de FPGA

Autres exemples : LiteSPI (flash), LiteSDCard (carte SD), LiteI2C (i2c), LiteJESD204B/C.

Du target au bitstream



- les plates-formes décrivent la carte (IO/périphériques, modèle de FPGA, ...)
- les targets
  - définissent le SoC (périphériques, PLL, ...)
  - utilisent xSoC comme classe de base
  - les cores pour PLL/CPU/contrôleurs, ...

- build est défini par architecture/outils pour :
  - créer le fichier Verilog
  - créer le fichier de contraintes
  - créer le projet
  - piloter l'outil en vue de la génération du bitstream
  - générer des fichiers .json/.csv décrivant memory-map et constantes

Langage - Bases : Migen

Module, signaux et logique

```
class MyCore(LiteXModule):
    def __init__(self, nbits):
        self.enable = Signal()
        self.count  = Signal(nbits)
        self.active = Signal()
        self.comb += self.active.eq(self.→
            ↪count != 0)
        self.sync += If(self.enable,
            self.count.eq(self.count + 1))

self.myCoreInst = MyCore(nbits=32)
```

- `Signal`, `comb` et `sync` décrivent fils, logique combinatoire et registres
- `LiteXModule` enregistre les sous-modules affectés à `self.*`

Bas niveau : `specials`

```
self.specials += Instance("BUFG",
    i_I = clk_in,
    o_O = clk_out)
```

Primitive, mémoire ou module HDL hors de `comb`/`sync` ; `p_`, `i_`, `o_`, `io_` décrivent paramètres et ports.

Machine à états

```
self.busy   = Signal()
self.result = Signal(16)
self.fsm = fsm = FSM(reset_state="IDLE")

fsm.act("IDLE",
    self.busy.eq(0),
    If(start, NextState("RUN")))
fsm.act("RUN",
    self.busy.eq(1),
    If(done,
        NextValue(self.result, data),
        NextState("IDLE")))
```

- `eq()` fixe une valeur combinatoire tant que l'état est actif
- `NextValue()` met à jour un registre au prochain front ; sa valeur reste mémorisée jusqu'à la prochaine modification

Langage - Bases : interfaces LiteX

Flux matériels : `stream.Endpoint`

```
from litex.soc.interconnect import stream

layout = [
    ("data", 32), ("channel", 2),
]
self.sink   = stream.Endpoint(layout)
self.source = stream.Endpoint(layout)
self.comb  += self.sink.connect(self.source)
```

- le layout décrit le payload transporté
- l'Endpoint ajoute `valid`, `ready`, `first` et `last`
- transfert lorsque `valid & ready` vaut 1
- `connect()` câble payload et handshake

Registres logiciel/matériel : CSR

```
from litex.soc.interconnect.csr import *

self.control = CSRStorage(fields=[
    CSRField("enable", reset=1),
    CSRField("start", pulse=True),
])
self.status = CSRStatus(fields=[
    CSRField("busy"),
    CSRField("errors", size=8),
])
self.comb += [
    self.status.fields.busy.eq(busy),
    self.status.fields.errors.eq(errors),
]
```

- `CSRStorage` : CPU → logique
- `CSRStatus` : logique → CPU
- `CSRField` décrit nom, taille, offset, reset et impulsion

Intégration de codes HDL existants

Adopter LiteX/Migen n'impose pas de réécrire les blocs existants.

Mécanisme

- Verilog/SystemVerilog : ajout direct des sources au projet
- VHDL : utilisation directe si l'outil accepte les projets mixtes
- sinon, `VHD2VConverter` adapte le VHDL en Verilog pour la toolchain
- un wrapper Python décrit :
  - paramètres, ports, horloges et resets
  - fichiers sources et dépendances
  - connexion au reste du SoC avec `Instance`
- la plate-forme sélectionne les sources et le flow propre à chaque outil

Exemples

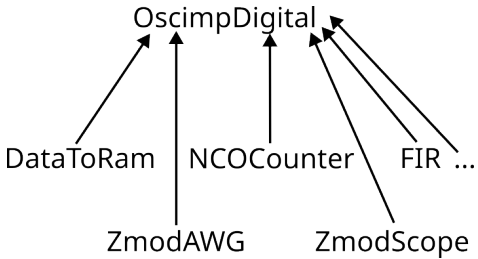
- `oscimpDigital_litex` : intégration de blocs HDL existants dans un projet LiteX
- `LimeSDR_GW` : refonte du gateware LimeSDR autour d'une architecture commune
  - LiteX pour l'intégration et le build
  - blocs RF réutilisables en VHDL avec LimeDFB
  - mêmes blocs sur plusieurs cartes et toolchains : XTRX, Mini V1/V2, LimeSDR USB, etc.

`github.com/trabucayre/oscimpDigital_litex`

`github.com/myriadrf/LimeSDR_GW`

oscimpDigital_litex

Un exemple d'intégration de codes VHDL



- `OscimpDigital` : classe de base
  - prise en charge de l'ajout des sources
  - instanciation du module `VHD2VConverter`
- `DataToRam`, `ZmodAWG`, ... : Wrappers autour de modules VHDL
  - ajout des `stream` source/sink
  - définition des `CSR` pour la configuration/pilotage des modules
  - ajout/définition des sources VHDL à intégrer.

- installation : `pip3 install --user -e .`
- utilisation par `from oscimpDigital_litex import xxx`
- les extensions `Zmod AWG` et `Zmod Scope` sont intégrées dans ce dépôt. Des tests plus poussés restent nécessaires
- la carte `Digilent Eclypse Z7` est disponible dans `litex-boards`
- en cours : adaptation des exemples de la `Red Pitaya`
- en cours : finalisation d'un script JSON → dtso (Device Tree overlay)

`https://github.com/trabucayre/oscimpDigital_litex`

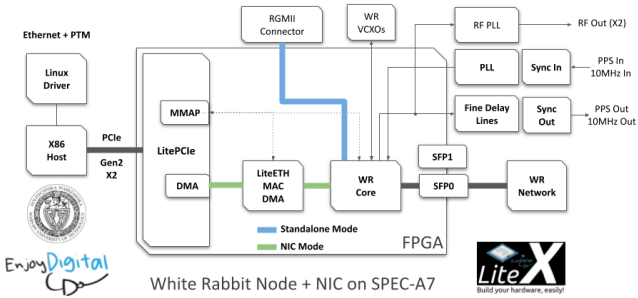
White Rabbit

White Rabbit

- distribution du temps et de la fréquence sur un réseau Ethernet fibré
- synchronisation à l'échelle de la subnanoseconde entre plusieurs équipements
- transport des données et distribution du PPS et du 10 MHz

`Litex_wr_nic`

- cœur White Rabbit intégré à l'écosystème LiteX
- modes Slave, Master ou GrandMaster
- NIC Ethernet 1 Gb/s et support PCIe PTM



White Rabbit Node + NIC on SPEC-A7

Source : `https://github.com/enjoy-digital/litex_wr_nic`

- intégration comme un core, au même titre que LiteEth ou LiteDRAM
- cibles Xilinx 7-Series et Gowin GW5A, avec une calibration métrologique à finaliser pour Gowin

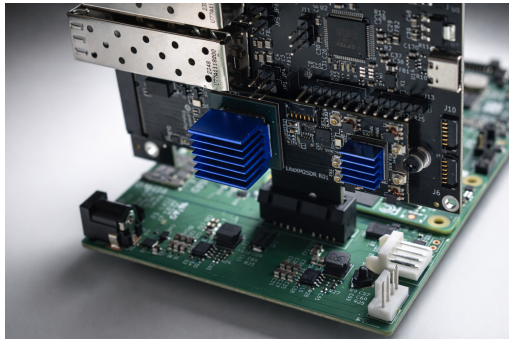
LiteX-M2SDR + baseboard

Une plate-forme SDR entièrement LiteX

- un même design pour le traitement, les transports et le contrôle
- PCIe : réception (RX) et émission (TX) à haut débit
- SATA : acquisition, stockage et rejeu des données
- SFP 0 : flux RX/TX et contrôle Ethernet ; PoE possible selon l'équipement réseau
- SFP 1 : White Rabbit pour asservir l'oscillateur du front-end RF
- PPS distribué pour synchroniser les acquisitions

Les interfaces de communication deviennent des variantes de build : les mêmes cores restent réutilisables d'une plate-forme à l'autre.

- RF Frontend ADi AD9361
- FPGA : Xilinx Artix-7 XC7A200T
- Format : M.2 2280 Key M



Source : Enjoy-Digital, LiteX-M2SDR / Baseboard

`https://github.com/enjoy-digital/litex_m2sdr`

Conclusion

- **Développement simplifié**
  - Python pour le prétraitement et la description du core
  - calcul, configuration et génération réunis dans un même langage
- **Cohérence et réutilisation**
  - algorithmes indépendants du matériel
  - mêmes cores réutilisables sur plusieurs plateformes
- **Outils transparents pour l'utilisateur**
  - génération du projet, contraintes et build pris en charge par LiteX
  - changement de FPGA, de plateforme ou de toolchain avec un impact limité
- **Communications adaptables**
  - accès depuis un processeur ARM, PCIe ou Ethernet
- **Synchronisation avec White Rabbit**
  - référence commune de temps et de fréquence
  - cohérence temporelle des conversions et asservissement des horloges

→ Le matériel et les interfaces évoluent ; les algorithmes restent.

- `https://github.com/enjoy-digital/litex`
- `https://github.com/trabucayre/oscimpDigital_litex`
- `https://github.com/enjoy-digital/litex_wr_nic`

Disponible pour développements sur FPGA / systèmes embarqués

gwenhael.goavec-merou@trabucayre.com / 06.13.06.70.19