Conception et exploitation des processeurs

Frédéric Pétrot

Équipe pédagogique :
Julie Dumas, Claire Maiza, Olivier Muller, Frédéric Pétrot, Lionel Rieg (resp.),
Manu Selva et Sebastien Viardot

Année universitaire 2021-2022

C1	Présentation du projet CEP et rappels de VHDL
C2	Chaîne de compilation et assembleur RISC-V
C3	Conventions pour les appels de fonctions
C4	Gestion des interruptions par le logiciel

Sommaire

Introduction

Objectif :

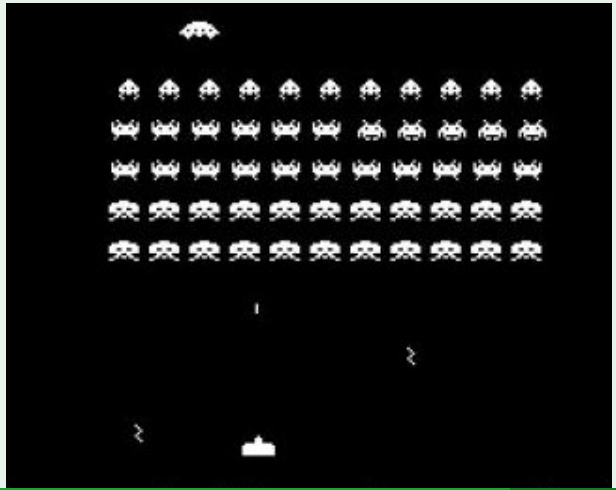
Conception d'un ordinateur minimaliste

Comprenant :

- un processeur
- de la mémoire
- des périphériques simples : LEDs, boutons poussoir, interrupteurs
- des périphériques moins simples : horloge, ...
- des périphériques complexes : contrôleur HDMI, ...

Introduction

Challenge, ...



Introduction

Comment ?

Conception du processeur

- ISA : RISC-V `rv32i`
- stratégie de conception : PC/PO, cf. cours Circuits numériques et éléments ...
- langage de conception : VHDL
- sur les cartes Zybo de digilent (autour d'un FPGA Xilinx Zynq-7000)

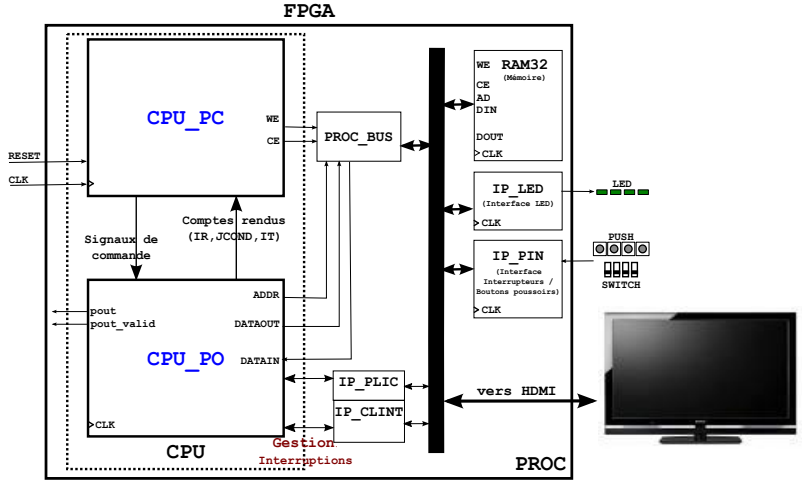
Notez bien : on ne part pas d'une feuille blanche, ...

Conception de l'environnement

- utilisation de blocs existants : mémoire par ex.
- conception de périphériques : VHDL
- utilisation d'un « bus » permettant la connexion des composants

Sommaire

Structure processeur + périphériques



Mise en œuvre de la mémoire

Mémoire RAM synchrone : ce n'est pas un bloc combinatoire !

Positionne `addr` => `datain` disponible *au cycle suivant*
Positionne `addr/dataout/we=1` => `dataout` échantillonné `clk` ↑
Comportement *registre*, toujours lisible lorsque `we=0`

Impact vital !

- sur la lecture de l'instruction à exécuter (*ifetch*)
  positionner `pc` dans un état, mettre à jour `ir` dans le suivant
- sur la lecture de la donnée lors d'un `lw`
  positionner `ad` dans un état, mettre à jour `rd` dans le suivant

Mise en œuvre du bus de communication

Bus pour accès différents composants : principes

Écriture combinatoire :

- processeur positionne `mem_addr/mem_dataout/mem_we=1/mem_ce=1`
- bus décode adresses pour calcul `bus_ce(i)` et `bus_we(i)` (mutuellement exclusifs)
- `mem_dataout` échantillonné par composant avec `bus_ce(i) = 1` et `bus_we(i) = 1` sur `clk` ↑

Lecture séquentielle :

- processeur positionne `mem_addr/mem_we=0/mem_ce=1`
- sur `clk` ↑, composants produisent `bus_datai(i)`
- bus utilise `ce(i)` du cycle *précédent* pour choisir quel signal produire sur `mem_datain`

Sommaire

Processeur

Rappel des caractéristiques globales de l'architecture `rv32i` :

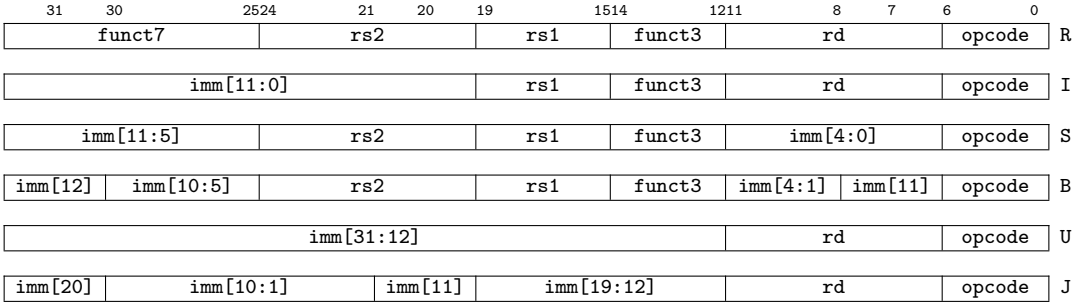
- bus de données et d'adresse sur 32 bits
- taille de l'instruction : 32 bits
- ISA 3-adresses
- 32 registres opérandes des instructions, notés `x0` à `x31`
  - registres `x1` à `x31` : usage général
  - registre `x0` : vaut toujours 0
- registre `pc` : adresse de l'instruction suivante
- registre `ir` : instruction en cours d'exécution
- registre `ad` : adresse effective lors d'un accès mémoire

Rappel des formats

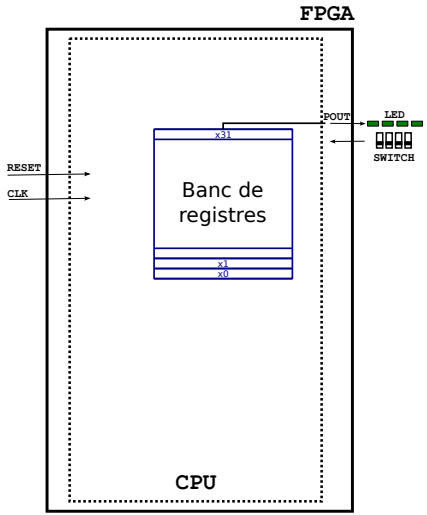
Où récupérer l'information dans l'instruction

- *registre*
  - 5 bits dans l'instruction, 1, 2 ou 3 fois, pour désigner un registre
- *immédiat*
  - 5 bits dans l'instruction, pour les décalages
  - 12 bits dans l'instruction, différentes interprétations possibles
  - 20 bits dans l'instruction, pour les sauts inconditionnels
- *indirect registre + déplacement*
  - 5 bits pour identifier le registre x_i plus 12 bits pour la constante
  - accès mémoire et sauts indirects
- *relatif à pc*
  - 12 bits pour les branchements conditionnels
  - 20 bits pour les sauts absolus

Détail de l'encodage des instructions



Processeur

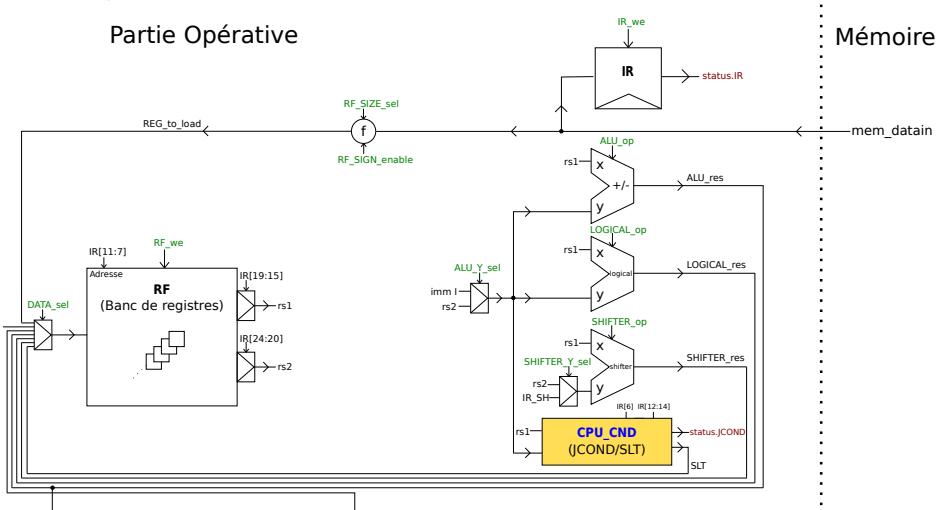


Debug minimal :
`x31` toujours visible sur les LEDs
Sélection d'un quartet parmi 8 grâce aux interrupteurs

int_2	int_1	int_0	$LED_{3...0}$
0	0	0	$pout_{3...0}$
0	0	1	$pout_{7...4}$
0	1	0	$pout_{11...8}$
0	1	1	$pout_{15...12}$
1	0	0	$pout_{19...16}$
1	0	1	$pout_{23...20}$
1	1	0	$pout_{27...24}$
1	1	1	$pout_{31...28}$

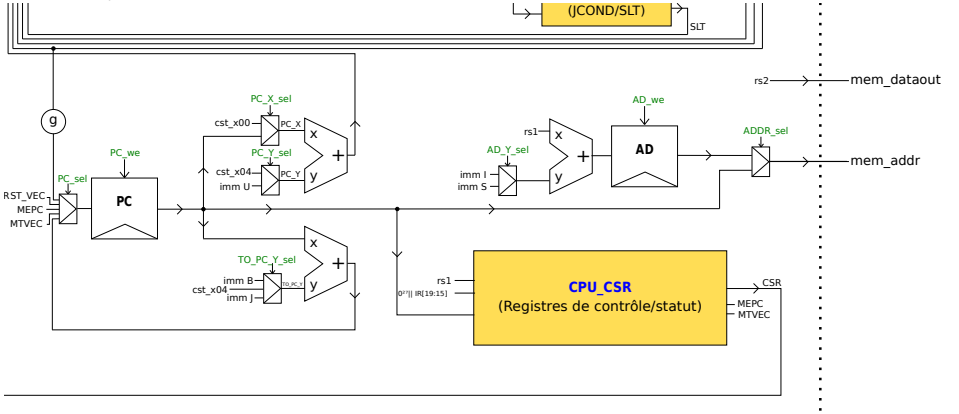
Processeur

Partie opérative



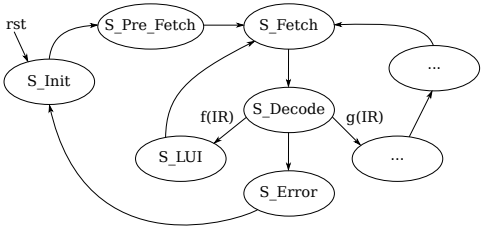
Processeur

Partie opérative



Processeur

Partie contrôle



États	Opérations
INIT	`pc` ← 0x1000
PREFECTCH	`mem_addr` ← `pc` émission de `pc` sur le bus d'adresse
FETCH	`ir` ← `mem_datain` réception `ir` sur bus données
DECODE	`pc` ← `pc` + 4 [1] décodage de `ir`
LUI	`rd` ← $\texttt{ir}_{31\cdots12}\|\|0^{12}$, `mem_addr` ← `pc` émission de `pc` sur le bus d'adresse

1. `pc` *n'est pas toujours incrémenté* dans l'état **DECODE**

Note

1 état entre positionnement `pc` et reception `ir`
exemple états **PREFECTCH** et **LUI**

Sommaire

Fournitures initiales

Partie contrôle : automate d'états

- interface complètement spécifiée
- registre d'état
- squelette fonction transition et fonction génération

Partie opérative : interconnexion d'unités fonctionnelles

- interface complètement spécifiée
- comportement complètement spécifié, et majoritairement fourni
- sauf : calcul des conditions de saut et interruptions

Outils disponibles

- outils de développement croisés pour RISC-V : `asm, cc, ld, objdump, ...`
- base de tests profs (activé lors des `git push`) :
  validation du comportement des instructions
  évaluation de la qualité de vos propres tests

Objectifs en terme d'instructions

Séance	Instruction typique	Famille
1	États : **FETCH, DECODE,...** Inst. : `lui`, `addi` Programme : Afficher valeur sur LEDs	

2	`add`, `sll`
	`addi`, `slli`
	`auipc`
	Programme : Compteur sur LEDs
	Chenillard minimaliste sur LEDs
		`sub`, `or`, `and`, `xor`, `srl`, `sra`
		`andi`, `ori`, `xori`, `srli`, `srai`

3	`jal`
	`beq`
	Programme : Chenillard à motif (rotation)
	Multiplication Egyptienne
		`jalr`
		`bne`, `blt`, `bltu`, `ble`, `bleu`

4	`lw`, `sw`
	`slt`
	Système : HDMI
	Programme : tracé de droite "Bresenham"
		`sltu`, `slti`, `sltiu`

5	PO : interruption 1 source `mret` Programme : démonstration d'interruption Jackpot : jouer à *space invaders*	

Partie contrôle :
« Factorisez » si possible les états de l'automate

- `lw` et `sw` doivent toutes deux calculer une adresse effective
- de nombreuses instructions utilisant l'ALU peuvent être factorisées

Partie opérative :

- ajout du calcul des conditions
- ajout du support des interruptions

Objectifs

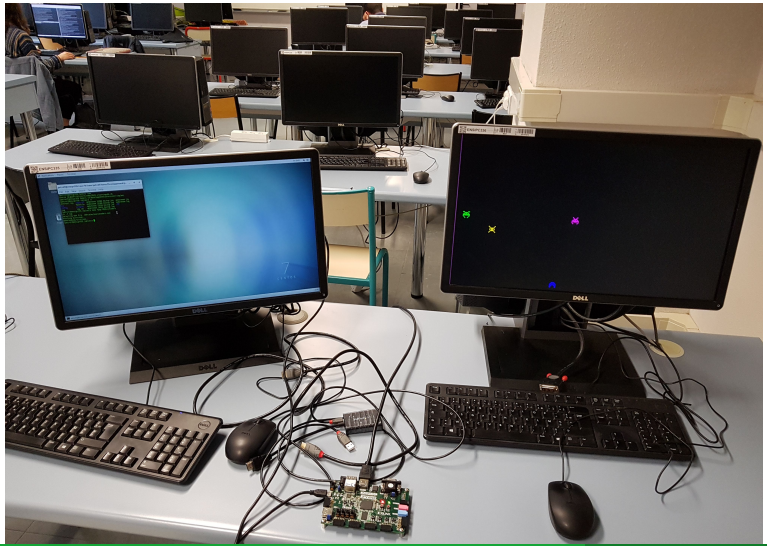
Méthode

1. En premier lieu
   - lire le cahier des charges complètement une première fois
2. Ensuite
   - faire les étapes dans l'ordre imposé
   - choisir quelques extensions en fonction de vos objectifs et vitesse de progression

Vérification et suivi

1. développer des tests unitaires :
   vérifier implantation de chaque instruction vs la spécification
2. outil de validation web fourni permet de voir votre progression
   ce n'est pas un outil de test!

Et voilà !



Sommaire

VHDL

VHDL : VHDL is a Hardware Description Language

Descriptions synthétisables

- structurelles
- comportementales
  - flot de données
  - séquentielles

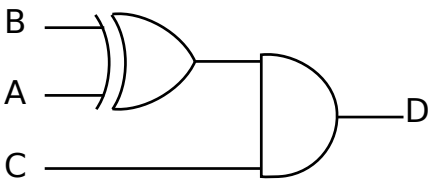
Certaines « tournures » peuvent ne pas être réalisables en matériel !

Sommaire

Description structurelle

De base : Déjà vue moult fois

```
entity BOX is
   port(A, B, C : in    std_logic;
        D       : out   std_logic);
end BOX;
architecture STRUCTURAL  of BOX is
   signal S1 : std_logic;
   component XOR2
      port(I0, I1 : in    std_logic;
           O      : out   std_logic);
   end component;
   component AND2
      port(I0, I1 : in    std_logic;
           O      : out   std_logic);
   end component;
begin
   BLOC1 : XOR2
      port map(B, A, S1);
   BLOC2 : AND2
      port map(I0=>C, I1=>S1, O=>D);
end STRUCTURAL;
```



Ou plus aisément :

```
D <= C and (A xor B);
```

Sommaire

Description comportementale

Exprime un comportement séquentiel ou concurrent

Séquentiel

- processus, avec liste de sensibilité
  `process(i0, i1, sel) ...`
- instructions similaires à celles d'Ada :
  - instructions conditionnelles : `if`, `case`
  - boucles : `loop`, `for`, `while`
- variables internes au processus
  `x := y;` affecte *immédiatement* `y` dans `x`

Concurrent

- affectations hors de tout processus :
  `a <= b xor c;`
- instructions spécifiques :
  - sélection combinatoire : `with ... select ...`, `when ... else ...`

Description comportementale

Attention !

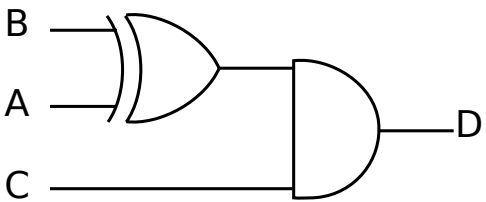
Production de matériel lors de la synthèse

- `if`s produisent des multiplexeurs, `case`s produisent des décodeurs, ...
- `loop`, `for`, `while` produisent autant d'éléments que d'itérations de boucles, ...
  Si cœur de boucles complexes, alors beaucoup de matériel

Description comportementale

Notion fondamentale : l'assignation notée <=

Assignation dite *postée*
<= prend effet lorsque tous les membres de droite ont été évalués
`a <= b; b <= a` échangent les contenus de `a` et `b`
Temps de l'évaluation = Δ-cycle




```
D <= C and (A xor B);
```
peut s'écrire
```
X <= A xor B;
D <= C and X;
```
ou encore
```
D <= C and X;
X <= A xor B;
```

Assignation ré-évaluée si événement sur membre de droite
=> Comportement idem portes logiques

Description comportementale

Notion fondamentale : liste de sensibilité

`process(a, b, c, d, ...):`

process ré-évalué si événement sur signal de la liste de sensibilité

Tous signaux lus dans le process
⇒ processus combinatoire, typique pour fonction de transition/génération de FSM

```
process(CUR_STATE, A, B, C, D)
begin
  case CUR_STATE is
    when S0 => if A ... then
                  NXT_STATE <= S1
               else
                  NXT_STATE <= S2
               end if;
    when S1 => ...
  end case;
end process;
```

Quelques signaux lus dans le process
⇒ processus mémorisant

```
entity DFF is ...  end DFF;
architecture SEQ of dff is
begin
    process(ck)
    begin
        if rising_edge(ck) then
            q <= d
        end if;
    end process;
end SEQ;
```

Exemple : MUX 1 parmi 4

Séquentiel

```
architecture SEQ of MUX is
begin
process(I0, I1, I2, I3, SEL)
begin
  case SEL is
    when 0 => O <= I0;
    when 1 => O <= I1;
    when 2 => O <= I2;
    when others =>
                O <= I3;
  end case;
end process;
end SEQ;
```

Concurrent

```
architecture CUR of MUX is
begin
with SEL select
  O <= I0 when 0,
       I1 when 1,
       I2 when 2,
       I3 when others;
end CUR;
```

Note : affectation concurrente ≡ `process` avec membres de droite dans liste de sensibilité et affectant membre de gauche

Autres éléments mémorisants

Reset synchrone

```
entity DFF is ...  end DFF;
architecture SEQ of dff is
begin
    process(ck)
    begin
        if rising_edge(ck) then
           if reset = '1' then
             q <= '0';
           else
             q <= d
        end if;
    end process;
end SEQ;
```

Reset asynchrone, à éviter

```
entity DFF is ...  end DFF;
architecture SEQ of dff is
begin
    process(ck, reset)
    begin
        if reset = '1' then
           q <= '0';
        elsif rising_edge(ck) then
           q <= d
        end if;
    end process;
end SEQ;
```

Attention !

Vérifiez que les affections de `q` sont *toutes* conditionnées par le même front d'horloge, sinon *latches* => pas bon !

Types et attributs

Définition de types dérivés

```
subtype w32 is std_logic_vector(31 downto 0);
type w32_vec is array (natural range <>) of w32;
```

Définition de types énumérés

```
type STATE_TYPE is ( INIT,   -- 00
                     FETCH,  -- 01
                     DEDCODE -- 10
);
```

Types et attributs

Définition d'agrégats

```
type PO_status is record
    IR    : w32;
    JCOND : boolean;
    -- Compléter pour les interruptions :
    IT    : boolean;
end record;
```

Typiquement utilisé pour simplifier l'écriture des connexions

```
entity CPU_PO is
  port (
    ....
    status : out PO_status;
    ....
entity CPU_PC is
  port (
    ....
    status : in  PO_status;
    ....
```

Types et attributs

Attributs : chaque élémént connaît des choses sur lui-même (introspection)

```
subtype TRIBYTE is std_logic_vector(23 downto 0);
type VIDEO_MEM is array(positive range <>, positive range <>) of TRIBYTE;
variable SCREEN : VIDEO_MEM(1 to 640, 1 to 480);
variable PIXEL  : TRIBYTE;
...
   PIXEL <= VIDEO_MEM(I, J);
```

```
PIXEL'length  : 24                    SCREEN'range(1) : 1 to 640
PIXEL'left    : 23                    SCREEN'range(2) : 1 to 480
PIXEL'right   : 0                     SCREEN'right(1) : 640
PIXEL'range   : 23 downto 0           SCREEN'left(2)  : 1
```

Fonctions

Fonctions : intégrées *lors de la synthèse*

Définition

```
function is_br_or_auipc(ir : w32)
    return boolean is
begin
    if ir(6 downto 0) = "...." or ir(6 downto 0) = "...." then
        return true;
    else
        return false;
    end if;
end function is_br_or_auipc;
```

Utilisation

```
case S_Decode :
    if is_br_or_auipc(ir) = false then
        -- pc <- pc+4
    end if;
    ...
```

Sommaire

Résumé

Langage spécifique

- à la production de matériel
- avec une sémantique du temps
- simulable
- synthétisable (sous ensemble que nous utiliserons)