System on Chip (SoC)

ELT5

2021-2022

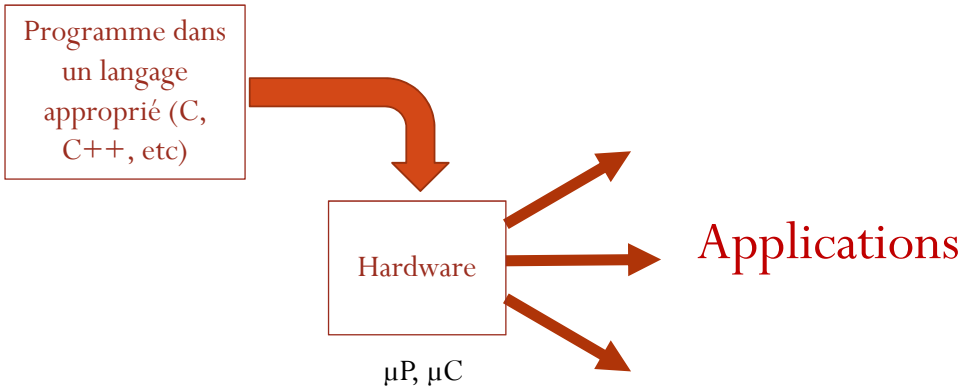
Vahid MEGHDADI

Introduction

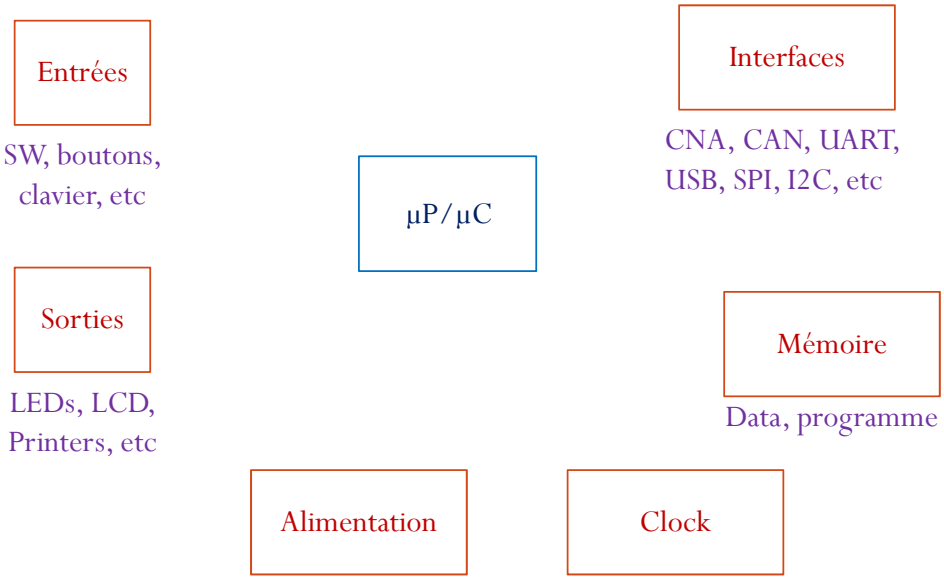
- Qu'est ce qu'un système embarqué
  - Quelle est son architecture typique
  - Comment peut-on le programmer
- Qu'est ce qu'un System-on-Chip (SoC) et quels sont ses avantages
- Les APIs (Application Programming Interface)
- Nous travaillerons en VHDL pour construire un SoC simple pour illustrer les principes

Système embarqué

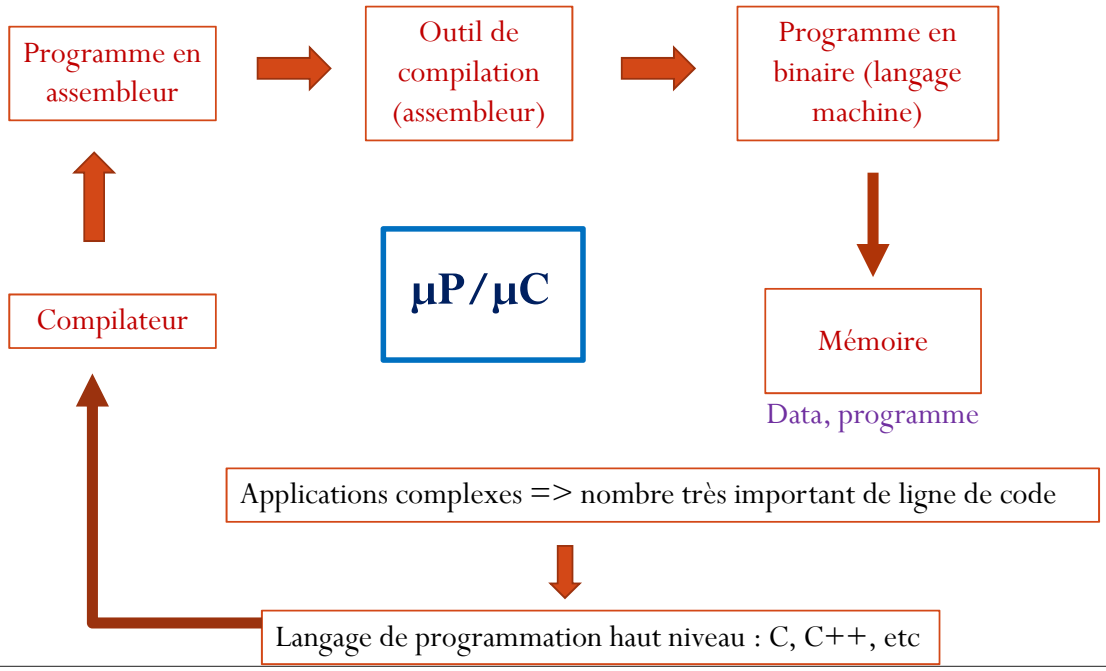
- Quand vous avez un programme informatique (SW) et que vous le mettez dans un circuit matériel compatible (HW) pour effectuer une application particulière, vous avez créé un système embarqué.



Un système embarqué typique



Programmer un système embarqué

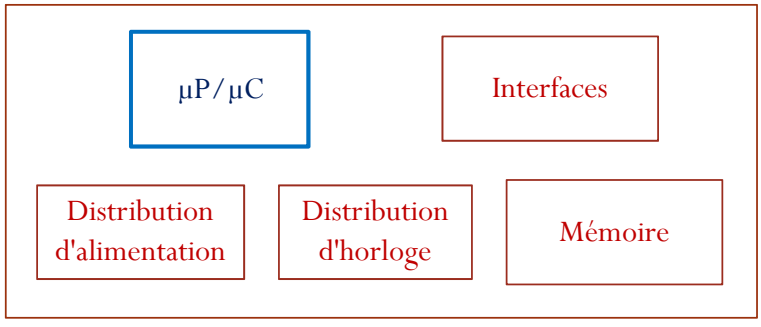


Système on chip (SoC)

La loi de Moore: tous les 2 ans, le nombre de transistor sur une puce double (Cela va peut être fini en 2022 !),

⇒ Plus de choses peuvent être intégrées dans la même puce.

⇒ Peut-on intégrer un système embarqué sur une seule puce ?



Avantages de SoC

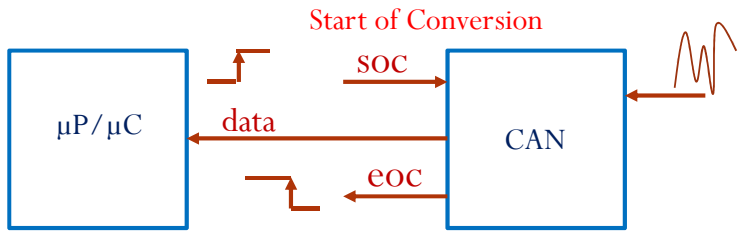
- Taille
  - Tous les composants sont maintenant sur la même puce, donc un gain de place important
- Vitesse
  - Les distances entre les composants sont réduites, donc la fréquence d'horloge peut augmenter considérablement
  - Les composants sur puce et leur connexion sont plus facilement prévisibles, moins de marge
- Coût
  - Un circuit imprimé moins complexe
  - Moins de soudage
  - Fréquence d'horloge faible sur la carte mais importante sur puce, cela réduit les frais liés à PCB

Programmation SoC

- Le SoC est un système embarqué
- Programmation se fait par
  - C, C++
  - API (Application Programming Interface)

Il est important de comprendre les principes de API. Son rôle est de faciliter la programmation liée aux interfaces.

APIs



Etape 1 : Activation soc
Etape 2 : Attente de eoc
Etape 3 : Lecture de donnée

→ Programme en C/C++

Le programmeur va juste appeler l'API correspondant : x = read_from_ADC();

CAN : Convertisseur Analogique/Numérique ; soc : start of conversion; eoc : end of conversion

APIs

- Quand vous programmer un SoC, vous devez chercher les APIs développées pour la plateforme
- Quand vous créez vous-même un SoC, vous devrez vous-même développer des APIs avec une bonne documentation pour fournir aux utilisateurs de votre SoC.
- Les APIs sont basées sur les détails du HW utilisé et sont compilées pour le microprocesseur de votre SoC.
- Exemples: printf(…), input(…), read_ADC(), write_DAC(…), write_LCD(…), initialize_board(), etc.
- Le programme, ainsi écrit, est concis et plus facilement compréhensible

PSoC, programmable SoC

- Deux types
  - Le processeur lui-même est implanté dans le FPGA par un programme (VHDL, Verilog, etc.)
  - Le processeur est gravé physiquement sur le FPGA, on programme des périphérique et autres logiques sur la partie programmable du FPGA
- Exemples chez xilinx
  - Sur le FPGA Artix-7, on peut implanter le microprocesseur 32 bits MicroBlaze (soft core) avec une vitesse maximale de 300 MIP
  - Le FPGA Zynq contient en dur le microprocesseur ARM cortex-A9 double cœur

Objectifs de cours

- La création d'un PSoC (programmable SoC) simple
- Le microprocesseur possède une architecture très simple, écrit en VHDL (PicoBlaze)
- Puisqu'on n'a pas de compilateur pour ce petit processeur, on programme en assembleur
- Nous y ajoutons différents périphériques en VHDL et nous créons des API pour les utiliser
- Implanter le tout dans le FPGA de la carte BASYS 3

Déroulement de cours

- La prise en main du processeur
  - Architecture
    - Registres internes, les bus, entrée/sortie, interruption
  - Implantation en VHDL
  - Jeu d'instructions
  - Chronogrammes des signaux relatifs aux instructions
- Création des périphériques et leur connexion au bus périphérique (hardware)
- Création des APIs (software)
- Création d'une application démonstrative

Evaluation

- 3H cours (coeff 10), 9H de TP (coeff 15)
- Evaluation cours
  - Un exam écrit (20 points)
- Evaluation TP (notation individuelle)
  - Séance 1 et 2 notées (sur votre engagement et non pas forcément sur le résultat) (4 points)
    - Une absence non justifiée, séance non rattrapée : 2 points perdus
    - Une absence non justifiée, séance rattrapée : 1 point perdu
    - Absence justifiée, séance non rattrapée : 1 point perdu
    - Absence justifiée, séance rattrapée : 0 point perdu
  - Séance 3 notée sur le résultat et la compétence individuelle (8 points)
  - Evaluation de compétence par rapport aux TP effectués par équipe, **une personne tirée au sort sera oralement questionnée, la note obtenue sera mise pour tous les membres de l'équipe**. Avant le début des TP, vous me remettez la liste des participants de votre équipe.
- Définition : Une équipe est constituée de deux binômes/trinômes de TP (4 à 6 personnes).
- Attention : Bien s'assurer que vos membres d'équipe suivent et comprennent les TP.

Microprocesseure PicoBlaze (HW)

Architecture

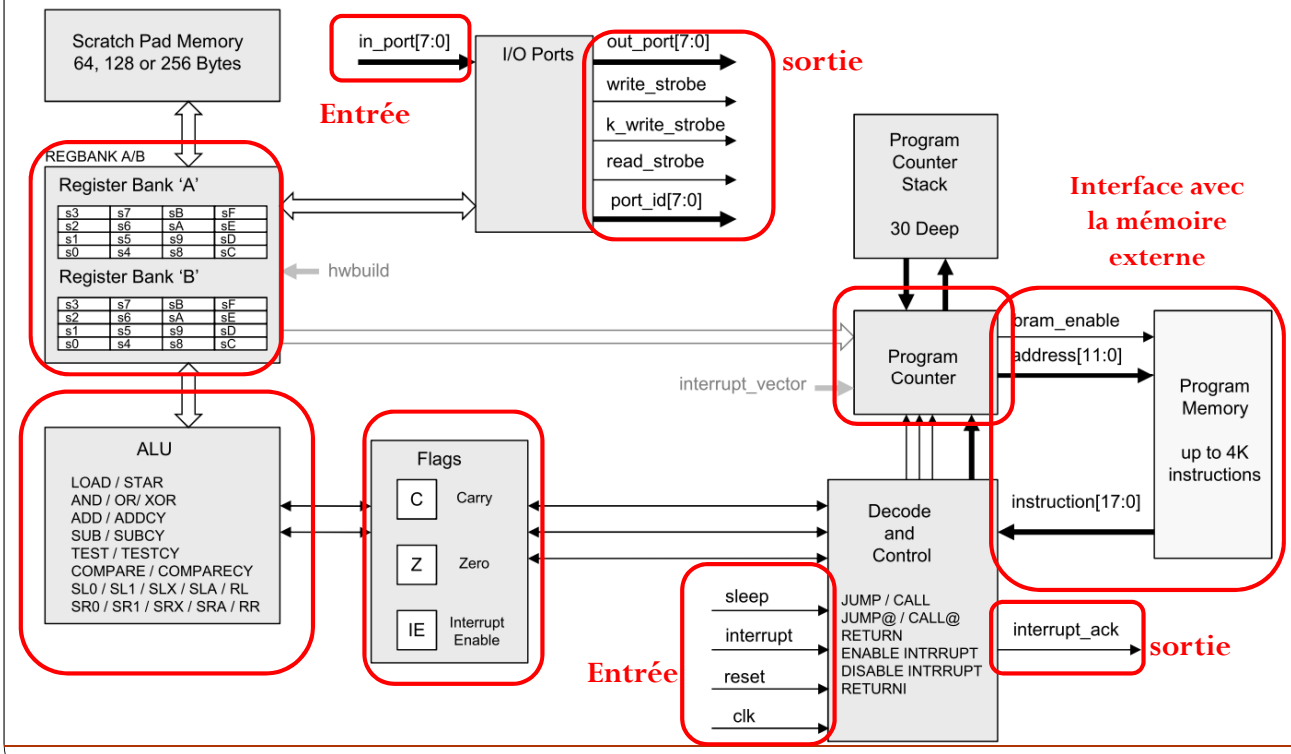
Déclaration et instanciation

Connexion d'une mémoire contenant le programme du µP

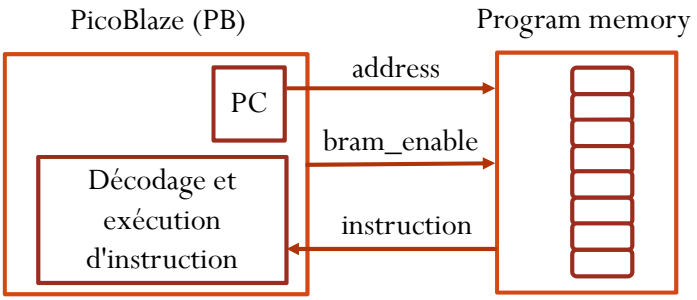
PicoBlaze, un petit µP « soft-core »

- Le programme VHDL de ce processeur est disponible
- Outils software existant autour de PicoBlaze
  - Assembleur pour générer le code opératoire (exécutable ou encore les codes objets) à partir des instructions écrites en assembleur
  - Des utilités pour créer une mémoire contenant ces codes opératoires

Architecture de PicoBlaze



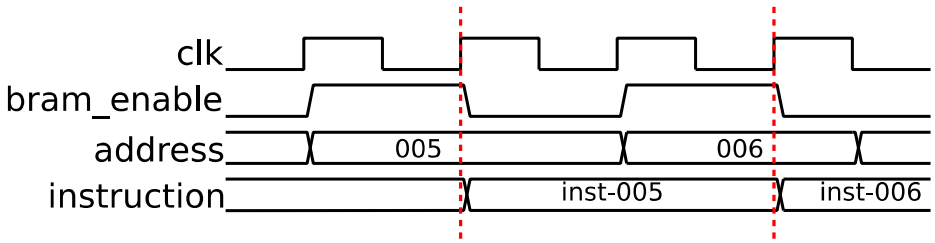
Exécution d'une instruction



Etape 1: PB met l'adresse de l'instruction sur le bus "address" et active le signal bram_enable

Etape 2: La mémoire met le code objet déjà stocké à l'adresse pointée, sur le bus "instruction"

Etape 3: le PB décode et exécute l'instruction.



Exemple

- L'instruction "LOAD s5,42" signifie : charger x"42" dans le registre s5.
- Le code opératoire de l'instruction générale de "LOAD sX,kk" est

  instruction(17 downto 0) = x"01Xkk" : 00 0001 XXXX kkkk kkkk
- Donc

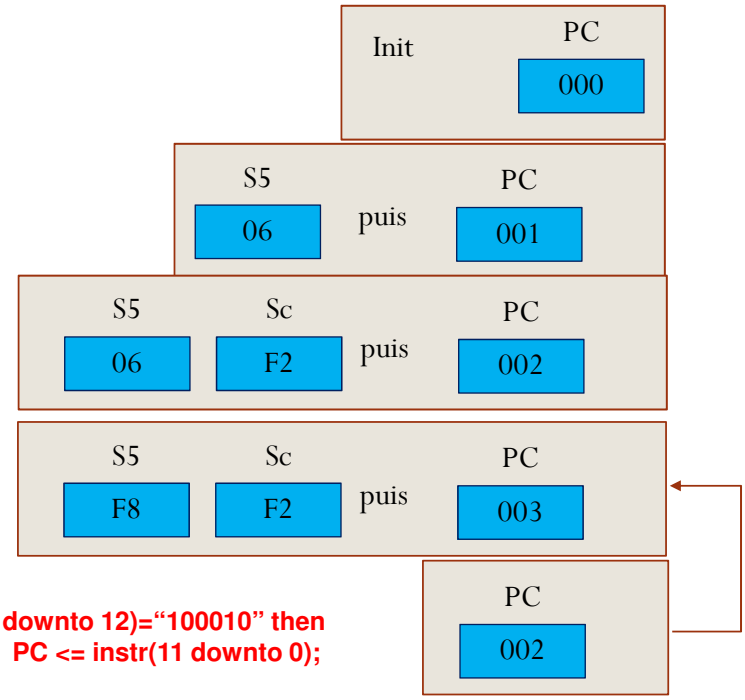
  LOAD s5,42 = 00 0001 0101 0100 0010

- Le bloc décodeur examine d'abord les 6 MSB et ensuite exécute

  S(instruction(11 downto 8) <= instruction(7 downto 0);

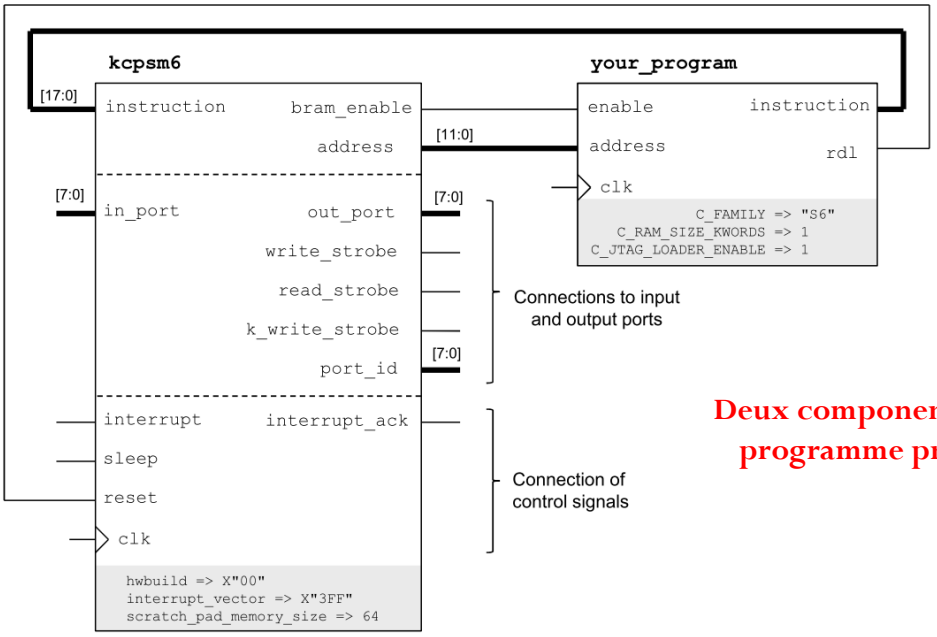
Exemple d'un programme

Case mémoire	Code opératoire	Mnémonique
000	00506	LOAD s5,06
001	00CF2	LOAD sC,F2
002	105C0	ADD s5,sC
003	22002	JUMP 002



if instr(17 downto 12)="100010" then
PC <= instr(11 downto 0);

Instanciation en VHDL



Deux components dans le programme principal.

Déclaration de PicoBlaze dans le programme principal

Deux étapes pour réaliser le µP: **1) déclaration du component** 2) instanciation

Déclaration dans la zone declarative (avant begin)

```
component kcpsm6
  generic(                 hwbuild : std_logic_vector(7 downto 0) := X"00";
                  interrupt_vector : std_logic_vector(11 downto 0) := X"3FF";
           scratch_pad_memory_size : integer := 64);
  port (                    address : out std_logic_vector(11 downto 0);
                        instruction : in std_logic_vector(17 downto 0);
                        bram_enable : out std_logic;
                            in_port : in std_logic_vector(7 downto 0);
                           out_port : out std_logic_vector(7 downto 0);
                            port_id : out std_logic_vector(7 downto 0);
                       write_strobe : out std_logic;
                     k_write_strobe : out std_logic;
                        read_strobe : out std_logic;
                          interrupt : in std_logic;
                      interrupt_ack : out std_logic;
                              sleep : in std_logic;
                              reset : in std_logic;
                                clk : in std_logic);
end component;
```

Instanciation

Deux étapes pour réaliser le µP: 1) déclaration du component **2) instanciation**

```
processor: kcpsm6
    generic map (                hwbuild => X"00",
                        interrupt_vector => X"3FF",
                scratch_pad_memory_size => 64)
    port map(                   address => address,
                            instruction => instruction,
                            bram_enable => bram_enable,
                                port_id => port_id,
                           write_strobe => write_strobe,
                         k_write_strobe => k_write_strobe,
                               out_port => out_port,
                           read_strobe => read_strobe,
                                in_port => in_port,
                              interrupt => interrupt,
                          interrupt_ack => interrupt_ack,
                                  sleep => kcpsm6_sleep,
                                  reset => kcpsm6_reset,
                                    clk => clk );
```

Des paramètres de configuration

Max 105 MHz pour spartan 6

Les broches du PicoBlaze

Les signaux à déclarer dans le P.P.

Déclaration et instanciation de la mémoire

```
component your_program
  generic(             C_FAMILY : string := "S6";
           C_RAM_SIZE_KWORDS : integer := 1;
        C_JTAG_LOADER_ENABLE : integer := 0);
  Port (      address : in std_logic_vector(11 downto 0);
         instruction : out std_logic_vector(17 downto 0);
              enable : in std_logic;
                 rdl : out std_logic;
                 clk : in std_logic);
end component;
```

```
program_rom: your_program
  generic map(                C_FAMILY => "S6",
                       C_RAM_SIZE_KWORDS => 1,
                     C_JTAG_LOADER_ENABLE => 1)
  port map(                  address => address,
                         instruction => instruction,
                              enable => bram_enable,
                                 rdl => kcpsm6_reset,
                                 clk => clk);
```

Des paramètres de configuration

Les ports de la mémoire

Les signaux à déclarer dans le P.P.

Création de la mémoire de programme

Il faudra créer la mémoire remplie avec les codes opératoires. On suit les étapes suivantes:

- Ecrire le programme en assembleur dans un fichier texte
- Lancer une utilité fournie qui assemble votre programme et génère un fichier contenant le code opératoire. L'outil crée ensuite un programme VHDL qui réalise la mémoire avec son contenu.
- On n'a plus qu'intégrer ce programme VHDL en tant qu'un component à notre programme main.

Microprocesseur PicoBlaze (SW)

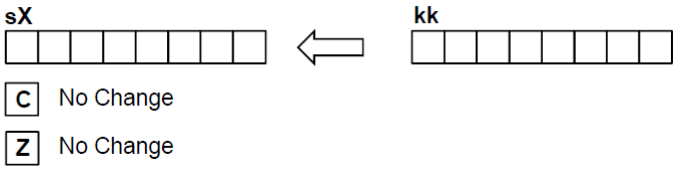
- Jeu d'instructions

Jeu d'instruction: LOAD

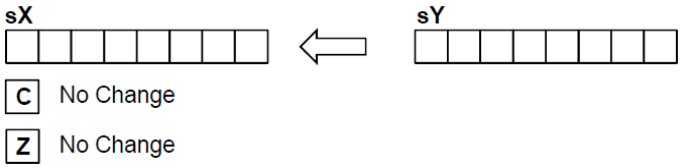
Register loading

```
00xy0   LOAD sX, sY
01xkk   LOAD sX, kk
16xy0   STAR sX, sY
```

LOAD sX, kk sX = kk



LOAD sX, sY sX = sY



Exemples: LOAD s5,sF ; LOAD sA,25 ;

Jeu d'instruction: opérations logiques

Logical

```
02xy0   AND sX, sY
03xkk   AND sX, kk
04xy0   OR sX, sY
05xkk   OR sX, kk
06xy0   XOR sX, sY
07xkk   XOR sX, kk
```

sX est le premier opérande mais aussi la destination
sY est le deuxième opérande
kk est la valeur immédiate du deuxième opérande

Exemples:

AND s2,s4 signifie [(s2) "et-logique" bit à bit (s4)] est chargé dans s2
XOR s2,s2 efface le contenu de s2
AND s2,0F force les 4 MSB de s2 à '0' et garde les 4 LSB
OR s2,0F force les 4 LSB de s2 à '1' et garde les 4 MSB

Jeu d'instruction: opérations arithmétiques

Arithmetic

```
10xy0   ADD sX, sY
11xkk   ADD sX, kk
18xy0   SUB sX, sY
19xkk   SUB sX, kk
```

sX est le premier opérande mais aussi la destination
sY est le deuxième opérande
kk est la valeur immédiate du deuxième opérande

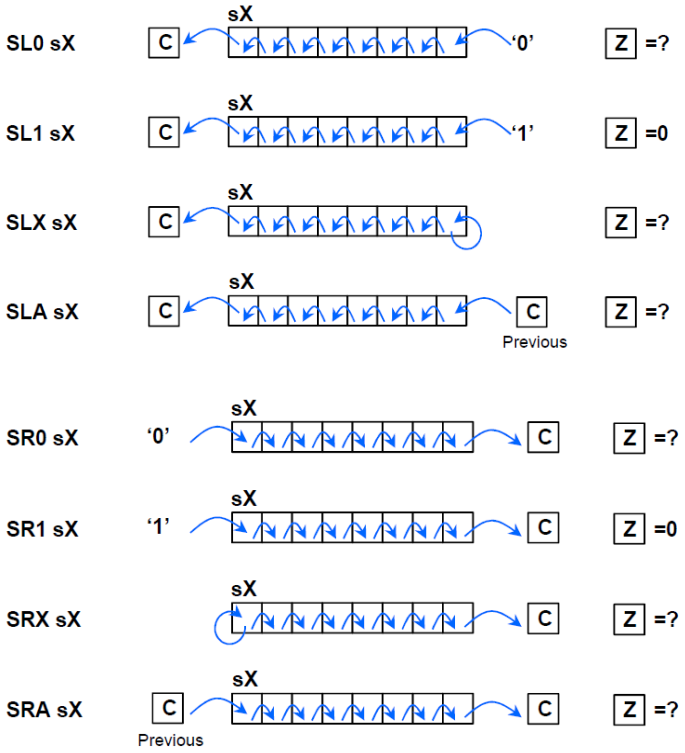
Exemples:

- **ADD s2,sB** signifie [(s2) + (sB)] est chargé dans s2
- **SUB s2,s2** efface le contenu de s2
- **ADD s2,01** incrément le contenu de s2
- **SUB s2,01** décrément le contenu de s2

Jeu d'instruction : décalage et rotation

Shift and Rotate

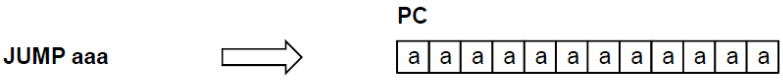
```
14x06   SL0 sX
14x07   SL1 sX
14x04   SLX sX
14x00   SLA sX
14x02   RL sX
14x0E   SR0 sX
14x0F   SR1 sX
14x0A   SRX sX
14x08   SRA sX
14x0C   RR sX
```

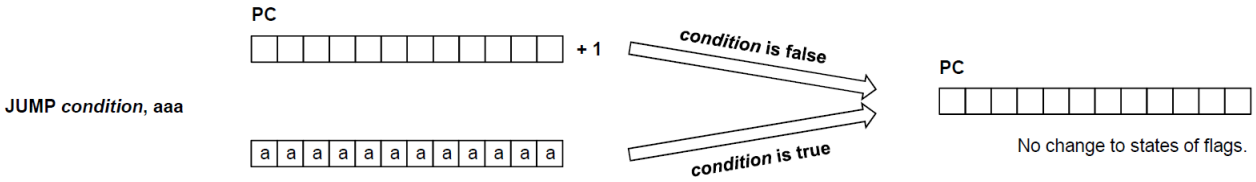


Jeu d'instruction : Branchement

Jump

```
22aaa   JUMP aaa
32aaa   JUMP Z, aaa
36aaa   JUMP NZ, aaa
3Aaaa   JUMP C, aaa
3Eaaa   JUMP NC, aaa
```



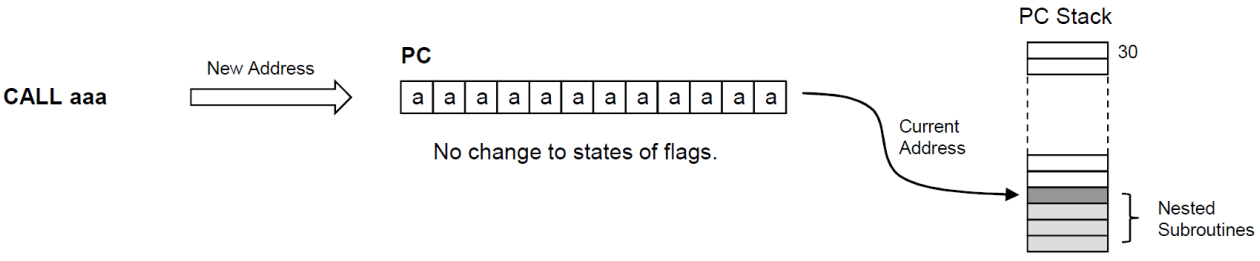



```
 Z : si le drapeau Z est actif   (Z='1')
NZ : si le drapeau Z est inactif (Z='0')
 C : si le drapeau C est actif   (C='1')
NC : si le drapeau C est inactif (C='0')
```

Jeu d'instruction: sous-programme

Subroutines

```
20aaa   CALL aaa
30aaa   CALL Z, aaa
34aaa   CALL NZ, aaa
38aaa   CALL C, aaa
3Caaa   CALL NC, aaa
```

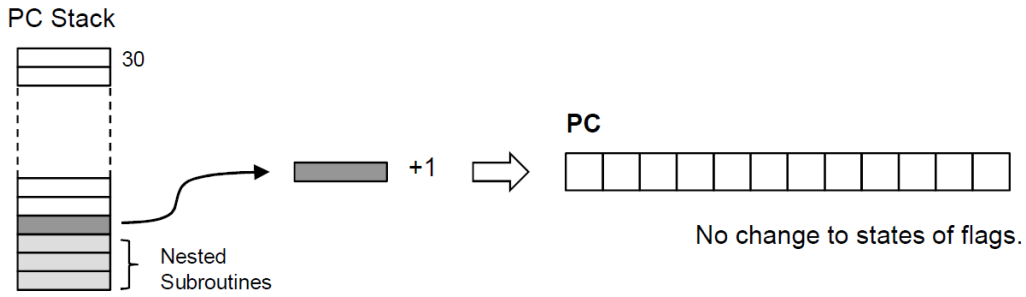


- "CALL" est comme un "JUMP" sauf qu'il garde l'adresse de retour sur la pile
- Si la condition n'est pas vérifiée, le µP passe, tout simplement, à l'instruction suivante :

(PC <= PC + 1)

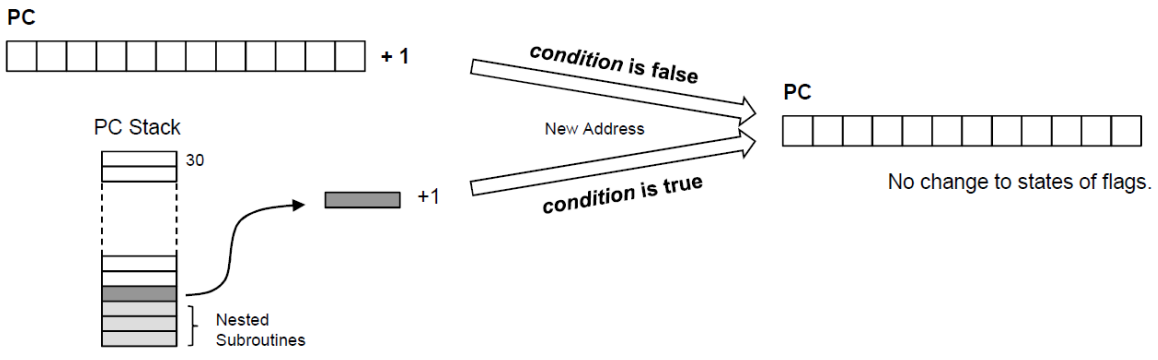
Jeu d'instruction: sous-programme

```
25000    RETURN
31000    RETURN Z
35000    RETURN NZ
39000    RETURN C
3D000    RETURN NC
```



Jeu d'instruction: sous-programme

```
25000    RETURN
31000    RETURN Z
35000    RETURN NZ
39000    RETURN C
3D000    RETURN NC
```



Jeu d'instruction (suite)

Input and Output

```
08xy0    INPUT  sX, (sY)
09xpp    INPUT sX, pp
2Cxy0    OUTPUT sX,(sY)
2Dxpp    OUTPUT sX, pp
2Bkkp    OUTPUTK kk, p
```

Test and Compare

```
0Cxy0    TEST sX, sY
0Dxkk    TEST sX, kk
0Exy0    TESTCY sX, sY
0Fxkk    TESTCY sX, kk
1Cxy0    COMPARE sX, sY
1Dxkk    COMPARE sX, kk
1Exy0    COMPARECY sX, sY
1Fxkk    COMPARECY sX, kk
```

Register Bank Selection

```
37000    REGBANK A
37001    REGBANK B
```

Interrupt Handling

```
28000    DISABLE INTERRUPT
28001    ENABLE INTERRUPT
29000    RETURNI DISABLE
29001    RETURNI ENABLE
```

Scratch Pad Memory

(64, 128 or 256 bytes)

```
2Exy0    STORE sX,(sY)
2Fxss    STORE sX, ss
0Axy0    FETCH sX, (sY)
0Bxss    FETCH sX, ss
```

Exemple: sous-programme

Créer un sous programme qui calcule la fonction $f(x) = 4x + 5$
x est une donnée signée sur 8 bits. La sortie est aussi signée sur 8 bits. On suppose qu'il n'y a pas de dépassement. La donnée x est passée au sous-programme dans le registre sF. Le sous programme retourne la réponse dans sF.

```
    LOAD     SF,07
    CALL     CALCUL
FIN:
    JUMP     FIN

CALCUL:
    SL0      SF ; fois 2
    SL0      SF ; encore fois 2
    ADD      SF,05
    RETURN
; on pouvait aussi utiliser ADD
; ADD sF,sF  ; fois 2
; Add sF,sF  ; encore fois 2
```

```
000  01F07          LOAD sF, 07
001  20003          CALL 003[CALCUL]
002             FIN:
002  22002          JUMP 002[FIN]
003          CALCUL:
003  14F06          SL0 sF    ; fois 2
004  14F06          SL0 sF    ; encore fois 2
005  11F05          ADD sF, 05
006  25000          RETURN
```

Sous-programme délai

```
        instruction 1
        CALL    DELAI
        instruction 2
FIN:    JUMP    FIN

DELAI:  LOAD    s0,10'd  ; ou LOAD s0,0A
BCL:    SUB     s0,01
        JUMP    NZ,BCL
        RETURN
```

Calcul du montant de délai.
Horloge = 100 MHz, Période = 10 ns; chaque instruction est exécutée en 2 cycles d'horloge: 20 ns

Les instructions en rouge sont exécutées une fois
Les instructions en vert sont exécutées 10 fois
Le délai total : $(1 + 1 + 10(1 + 1) + 1) \times 20ns = 460\ ns$

Le délai maximale est obtenu si on avait mis `LOAD s0,00` :

$$1 + 1 + 256(1 + 1) + 1) \times 20ns = 10{,}3\ \mu sec$$

Exercice

Ecrire un sous programme pour obtenir un délai de 10 µsec.

```
        CALL    DELAI
FIN:    JUMP    FIN

DELAI:  LOAD    s0,??
BCL:    SUB     s0,01
        JUMP    NZ,BCL
        RETURN
```

Il faudra exécuter $\frac{10\mu sec}{20ns} = 500$ instructions

Nombre d'instructions exécutées=
$(1 + 1 + (1 + 1).x + 1) = 500$
$\Rightarrow 2x + 3 = 500$
$x = 248{,}5$

Si on prend $x = 248$, le nombre d'instruction exécutées sera $2x + 3 = 499$.
Il faudra donc exécuter une instruction de plus : une instruction "nulle".

$$(248)_{10} = (F8)_{16}$$

```
        CALL    DELAI
FIN:    JUMP    FIN

DELAI:  LOAD    s0,F8
BCL:    SUB     s0,01
        JUMP    NZ,BCL
        ADD     s0,00 ; instr nulle
        RETURN
```

Remarque : sous programme

Dans le programme précédent, le sous-programme délai écrase le contenu de s0.
Il est possible que s0 contenait une valeur importante dans le programme principale.

Deux alternatives

On peut utiliser le banc B pour ne pas écraser les registres du banc actif.

```
         CALL    DELAI
FIN:     JUMP    FIN

DELAI:   REGBANK B
         LOAD    s0,10'd
BCL:     SUB     s0,01
         JUMP    NZ,BCL
         REGBANK A
         RETURN
```

On peut utiliser le "Scratch Pad" pour sauvegarder l'ancien valeur de s0.

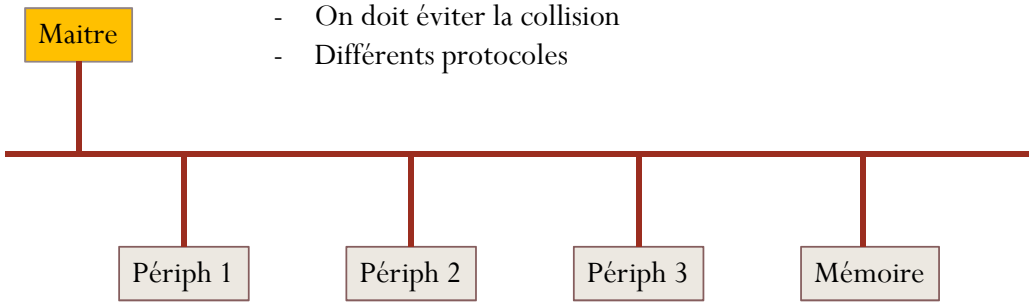
```
         CALL    DELAI
FIN:     JUMP    FIN

DELAI:   STORE   s0,00
         LOAD    s0,10'd
BCL:     SUB     s0,01
         JUMP    NZ,BCL
         FETCH   s0,00
         RETURN
```

Bus périphérique

Utilité d'un Bus

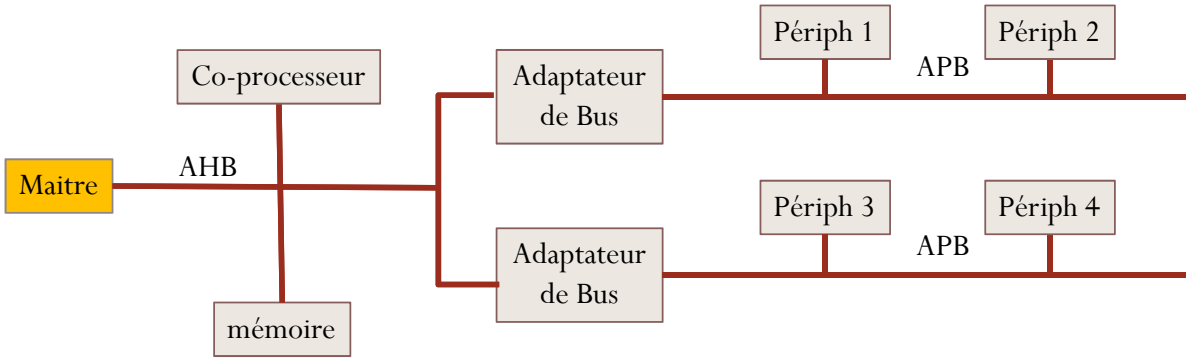
- Permet de connecté un "maitre" aux plusieurs esclaves.
- Economiser le nombre de fil à tirer
- Cependant, le débit sera partagé entre les "périphériques"

- Les périphériques partagent les mêmes fils.
- On doit éviter la collision
- Différents protocoles



Bus pour des "system on chip"

- OCB: On Chip Bus
- Le plus utilisé : AMBA (Advanced Microcontroller Bus Architecture)



AHB: Advanced High-performance Bus, haut débit, latence courte
APB: Advanced Peripheral Bus, moins de débit

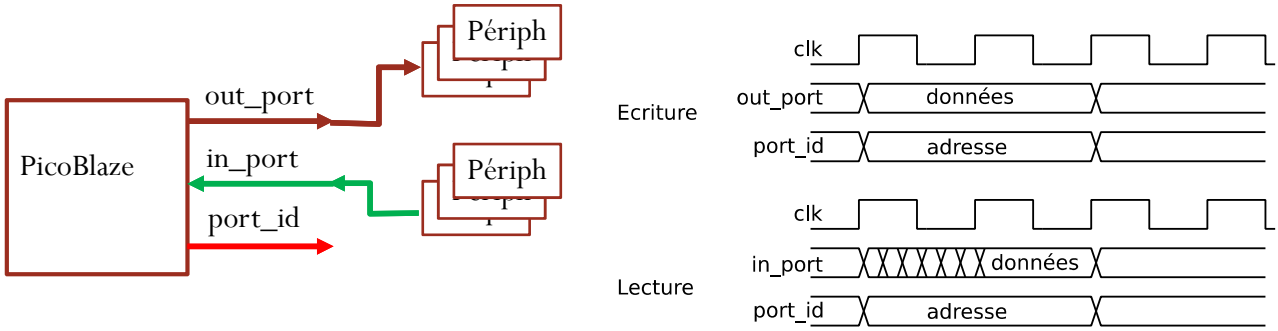
Bus ARM AMBA / AXI

- La normalisation facilite le design des interfaces et des périphériques.
- Les périphériques peuvent être remplacés sans souci de compatibilité
- Les coûts baissent
- L'évolution de bus AMBA continue:
  - version 1 en 1996 (ASB et APB)
  - Version 2 en 1999 (introduction de AHB)
  - Version 3 en 2003(AXI)
  - Version 4 en 2010 (AXI4)
- On parle maintenant de "network on chip" qui remplacera les bus. Les données sont transmises par paquet comme sur un réseau.

ASB: Advanced System Bus ; AXI : Advanced eXtensible Interface (for FPGA)

PicoBlaze et ses connexions

- Extrêmement simple, très bon pour des petits circuits
- PicoBlaze supporte deux bus
  - Entre le cœur et la mémoire
  - Entre le cœur et les périphériques.
    - Deux bus: **lecture**/**écriture** séparés



Ecrire sur un port de sortie

Instruction entrée/sortie

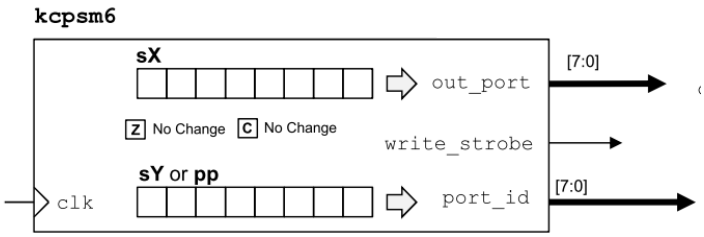
Input and Output

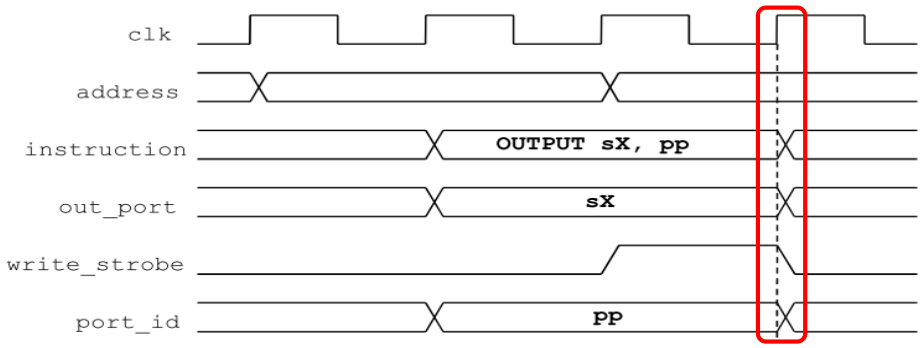
```
08xy0    INPUT  sX, (sY)
09xpp    INPUT sX, pp
2Cxy0    OUTPUT sX,(sY)
2Dxpp    OUTPUT sX, pp
```

Port de sortie

L'instruction à utiliser est:

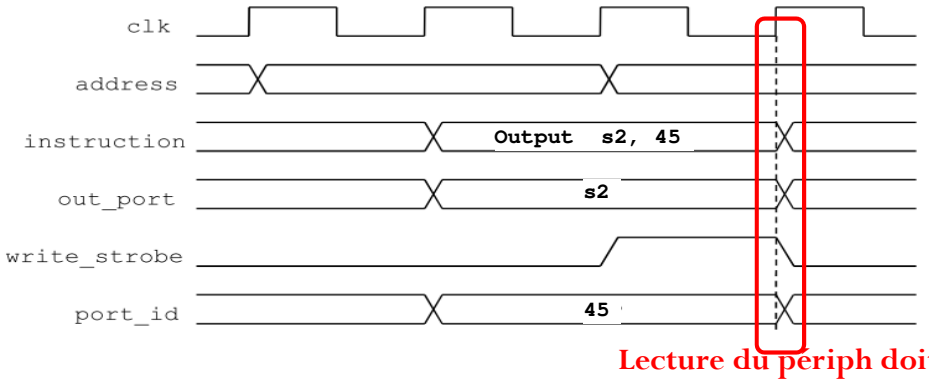
```
output  Sx,pp
output  Sx,(Sy)
```





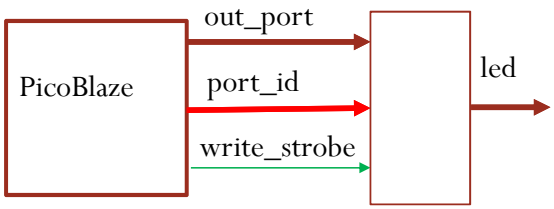
Lecture du périph doit se faire ici.

Exemple



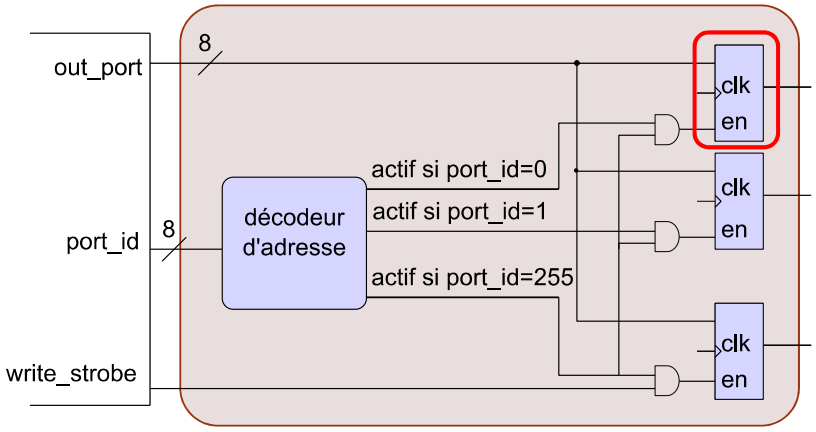
Lecture du périph doit se faire ici.

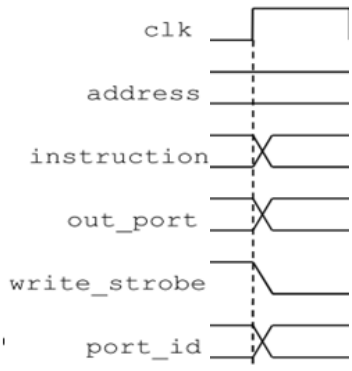
Output s2,45 est supposé écrire sur 8 leds le contenu du s2.




```
process(clk)
begin
    if clk'event and clk='1' then
        if write_strobe='1' then
            if port_id = x"45" then
                led <= out_port;
            end if;
        end if;
    end if;
end process;
```


Port de sortie, périphérique multiple





Décodage complet de l'adresse, possibilité d'avoir 256 ports de sortie

Par exemple, pour écrire dans le registre 0, en assembleur on écrit: OUTPUT S5,00
Grâce au timing du µP, le contenu du S5 sera figé dans le registre du haut mais pas dans les autres.

Lire d'un port d'entrée

La commande "INPUT"

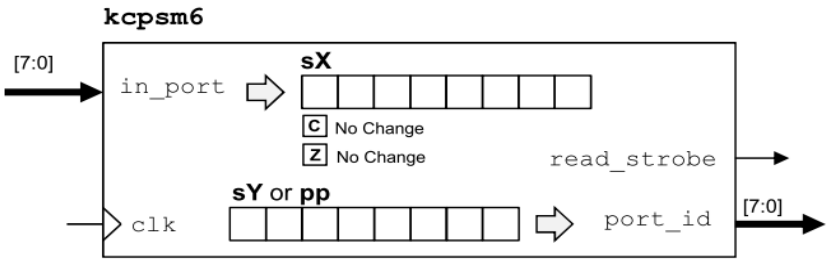
Input and Output

```
08xy0    INPUT  sX, (sY)
09xpp    INPUT sX, pp
2Cxy0    OUTPUT sX,(sY)
2Dxpp    OUTPUT sX, pp
```

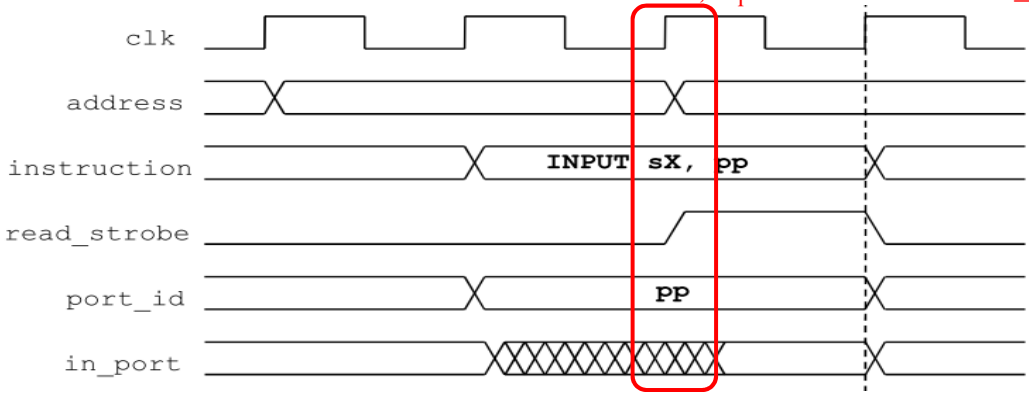
Port d'entrée

L'instruction à utiliser :

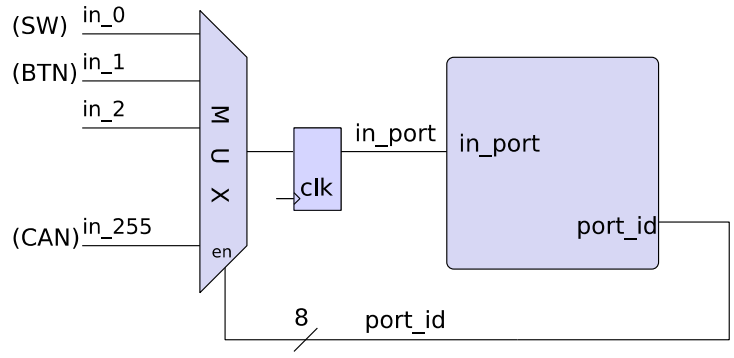
```
input Sx,pp
input Sx,(Sy)
```

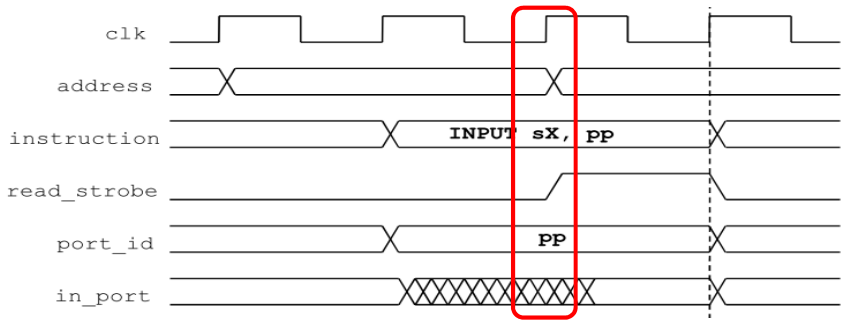


Ici, le périf met sa donnée sur in_port



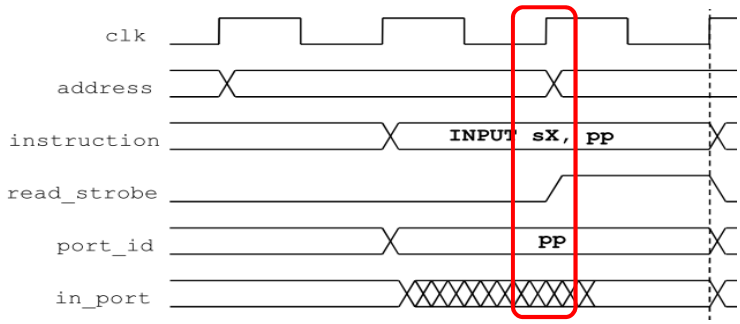
Port d'entrée





Process:
if CLK'event and CLK='1'
if port_id est le bon
mettre la donnée sur
in_port, le processeur
la lira sur le front
montant suivant

Interface port d'entrée

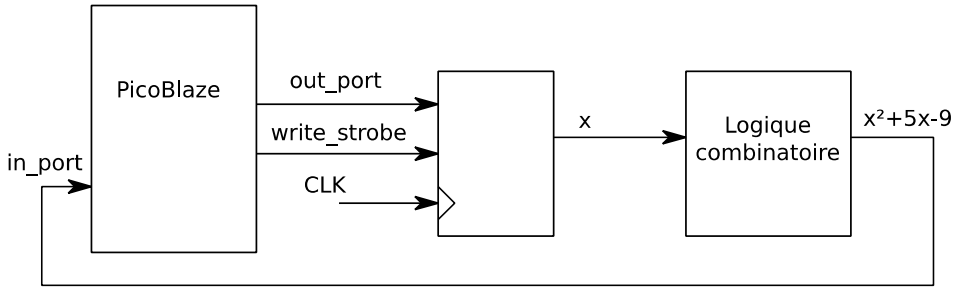



```
process(clk)
begin
   if clk'event and clk='1'
then
      if port_id = x"00" then
         in_port <= sw;
      end if;
      if port_id = x"01" then
         in_port <= btn;
      end if;
...
      if port_id = x"FF" then
         in_port <= CAN;
      end if;
   end if;
end process;
```

Exercice

- Nous souhaitons réaliser le calcul « $x^2 + 5x - 9$ » par la partie logique du FPGA pour accélérer le calcul. Ainsi, si on souhaite effectuer ce calcul en assembleur on écrit les lignes suivantes (pour $x = 11'd = x"0B"$):

```
LOAD    S0,0B
OUTPUT  S0,00   écrire sur le port de sortie 0
INPUT   S1,00   lire le résultat sur le port d'entrée 0
```



Exercice, suite ...

La partie VHDL:

```
Process(CLK)
Bgin
    if CLK'event and CLK='1' then
        if write_strobe = '1' then
            x <= out_port;
        end if;
    end if;
End process;

in_port <= x*x-5*x+9;
```

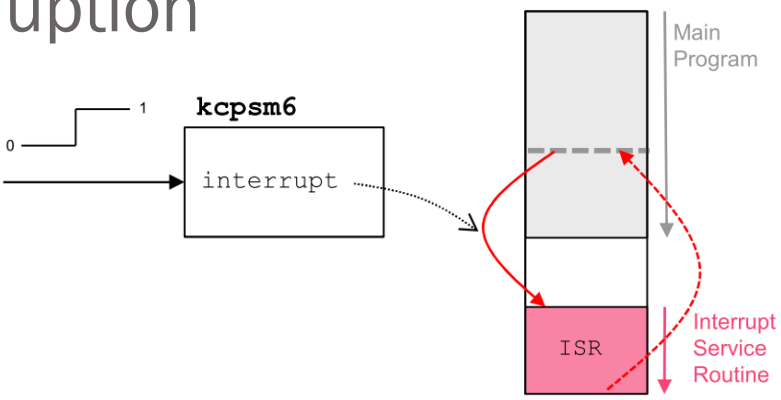
Ici, nous n'avons pas géré le dépassement. On aurait pu calculer la sortie sur 16 bits. A ce moment-là il aurait fallu considérer deux ports d'entrée, un pour le MSD et l'autre pour le LSD.

Sous programmes et interruption

Sous-programme

- Pour appeler un sous-programme déjà placé à l'adresse « aaa », on fait exécuter l'instruction « CALL aaa ».
- Avec cette commande
  - PC courant est sauvegardé sur la pile
  - PC est chargé par "aaa".
- Pour un retour d'un sous-programme, on fait exécuter la commande "RETURN".
  - Le contenu de pile + 1 est chargé dans PC.

Interruption



To state the obvious, an interrupt is used to interrupt the normal program execution sequence of KCPSM6. This means that when the 'interrupt' input is driven High ('1'), it will force KCPSM6 to abandon the code that it is executing, save its current operational state and divert its attention to executing a special section of program code known as an Interrupt Service Routine (ISR). Once the interrupt has been serviced, KCPSM6 returns to the program at the point from which it was interrupted and restores the operational states so that it can resume execution of the program as if nothing had happened.

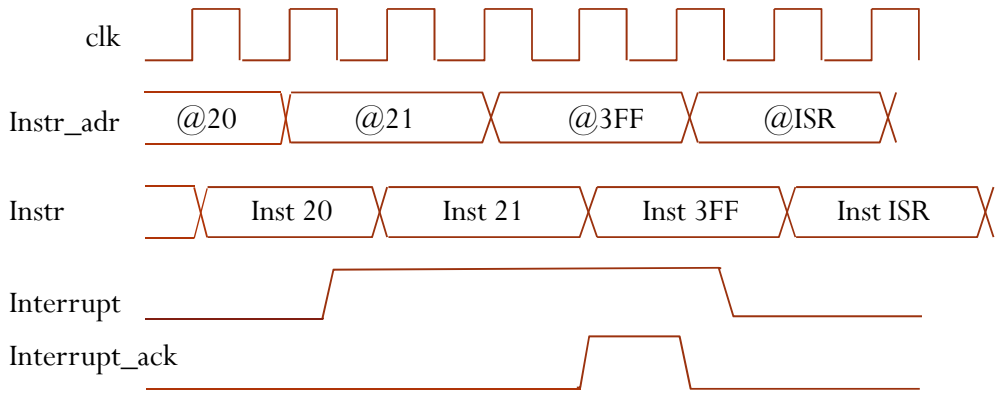
Routine d'interruption

- Quand l'interruption arrive (un niveau 1 sur l'entrée « interrupt »)
  - PicoBlaze termine l'instruction en cours, puis saute à une adresse prédéfinie (celle que nous avions indiquée au moment de l'instanciation du KCPSM6).

```
processor: kcpsm6
   generic map (      hwbuild => X"00",
                interrupt_vector => X"3FF",
          scratch_pad_memory_size => 64)
   port map(           address => address,
```

  - C'est à la charge du programmeur de mettre sa routine à cette adresse.

Exécution d'une interruption



Instruction Inst 21 est abandonnée et sera exécutée au retour de l'interruption.

interrupt_vector => X"3FF"
Inst_3FF = JUMP ISR

Exercice

- Comment imaginez-vous les lignes VHDL qui réalisent la prise en charge de l'interruption à l'intérieur du PicoBlaze ?

```
process(CLK)
begin
    if CLK'event and CLK='1' then
        if IE='1' and interrupt='1' then
            pile <= pc; -- simplifié
            pile <= 'Z', 'C' 'REGBANK'; --simplifié
            pc <= interrupt_vector;
            IE <= '0'
        else
            ….
```

Exercice

- Comment imaginez-vous les lignes VHDL qui réalisent la prise en charge de l'instruction RETURNI à l'intérieur du PicoBlaze ? (code opératoire 29000 pour « RETURNI DISABLE » et 29001 pour « RETURNI ENABLE »)

```
process(CLK)
begin
    if CLK'event and CLK='1' then
        …
        case code(17 downto 12) is
        when "101001" => -- 29
            'Z', 'C' 'REGBANK'<= pile; --simplifié
            pc <= pile; -- simplifié
            IE <= code(0);
        when …
```


Exemple d'interruption

Enoncé: A chaque interruption, incrémenter le registre SF.

Instanciation

```
processor: kcpsm6
    generic map (      hwbuild => X"00",
                  interrupt_vector => X"3FE",
           scratch_pad_memory_size => 64)
    port map(           address => address,…
```

Option 1

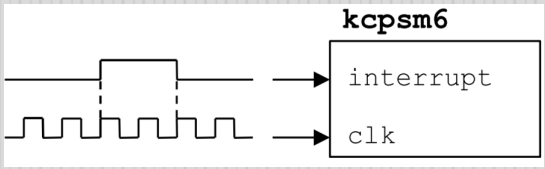
```
          enable   interrupt
          ...

          address  3FE (directive d'assembleur)
ISR :     add      SF,01
          returni  enable
```

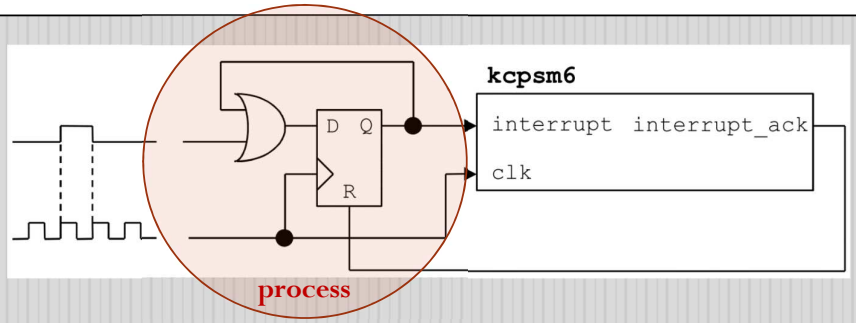
Option 2

```
          enable   interrupt
          ...
ISR :     add      SF,01
          returni  enable

          address  3FE ; on ne connaît pas la
          JUMP     ISR ; taille de la routine,
```

Mise en place du circuit

« Open loop », il est nécessaire de faire durer la demande de l'interruption au moins pour deux cycles d'horloge => pas de circuit supplémentaire.



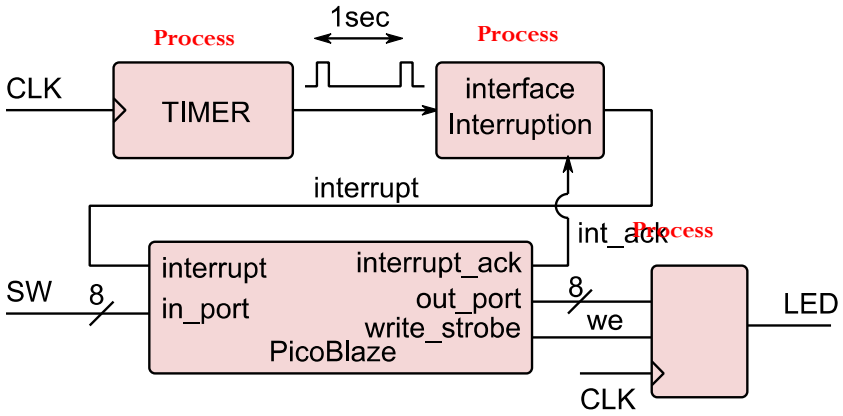


« Closed-loop », l'interruption est maintenue à '1' jusqu'à ce que le processeur la serve.

Exemples

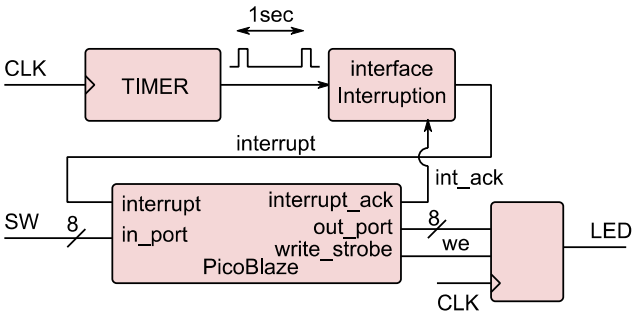
Exemple 1 : Timer

- Créer un Timer qui déclanche une interruption toutes les secondes. La routine d'interruption se charge de changer l'état des 8 LED de la carte: soit l'état des switches soit son inverse.



Instanciation KCPSM6

```
processor: kcpsm6
   generic map ( hwbuild => X"00",
        interrupt_vector => X"3FF",
 scratch_pad_memory_size => 64)
  port map(      address => address,
             instruction => instruction,
             bram_enable => bram_enable,
                 port_id => open,
            write_strobe => we,
          k_write_strobe => open,
                out_port => out_port,
             read_strobe => open,
                 in_port => SW,
               interrupt => interrupt,
           interrupt_ack => int_ack,
                   sleep => '0',
                   reset => kcpsm6_reset,
                     clk => clk);
```

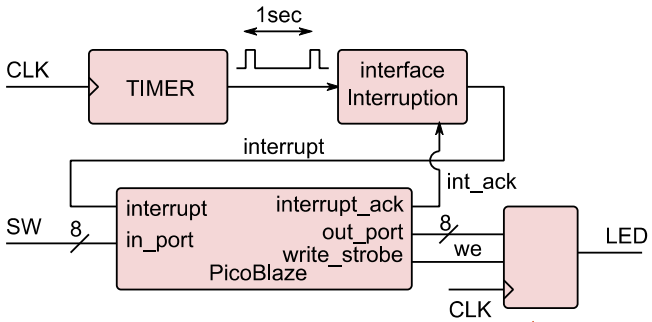


Connecter directement aux switches car il n'y a qu'un seul port d'entrée.

Instanciation Timer et outport

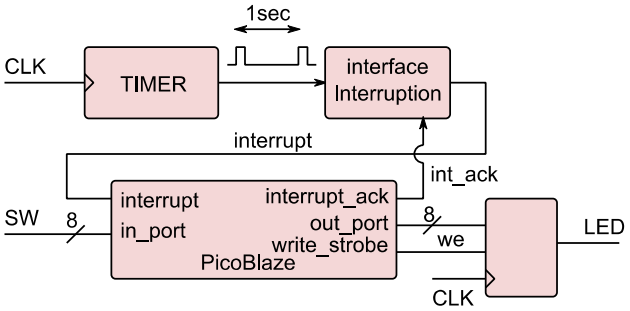
```
TIMER: process(clk)
begin
  if clk'event and clk = '1' then
     cmp <= cmp + 1;
     timer_trig <= '0';
     if cmp = 99999999 then
        cmp <= 0;
        timer_trig <= '1';
     end if;
  end if;
end process;
```

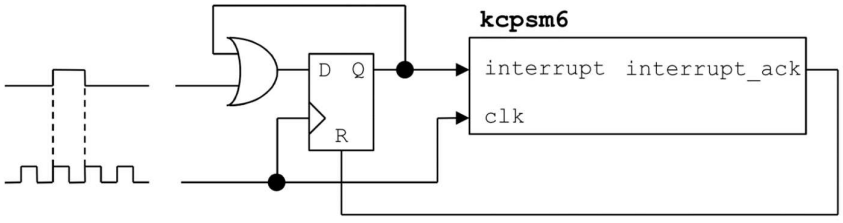
```
output_ports: process(clk)
  begin
    if clk'event and clk = '1' then
      if we = '1' then
         LED <= out_port;
      end if;
    end if;
  end process output_ports;
```



Interruption software

```
interrupt_control: process(clk)
begin
   if clk'event and clk='1' then
      if int_ack = '1' then
         interrupt <= '0';
      else
         if timer_trig = '1' then
            interrupt <= '1';
         end if;
      end if;
   end if;
end process interrupt_control;
```





Software

```
            ENABLE      INTERRUPT
            LOAD        S1,00
LOOP:       JUMP        LOOP

ISR:        INPUT       S0,00
            COMPARE     S1,00
            JUMP        Z,INVERS
            JUMP        AFFICH
INVERS:
            XOR         S0,FF
AFFICH:
            OUTPUT      S0,00
            XOR         S1,FF
            RETURNI     ENABLE

            ADDRESS     3FF
            JUMP        ISR
```

S0 contient l'état des switches
S1 est un drapeau qui s'inverse à chaque interruption.

On exécute le programme kcpsm6.exe pour générer la mémoire ROM qui contiendra les codes opératoires du programme en assembleur.

On n'a qu'à ajouter cette mémoire au projet.

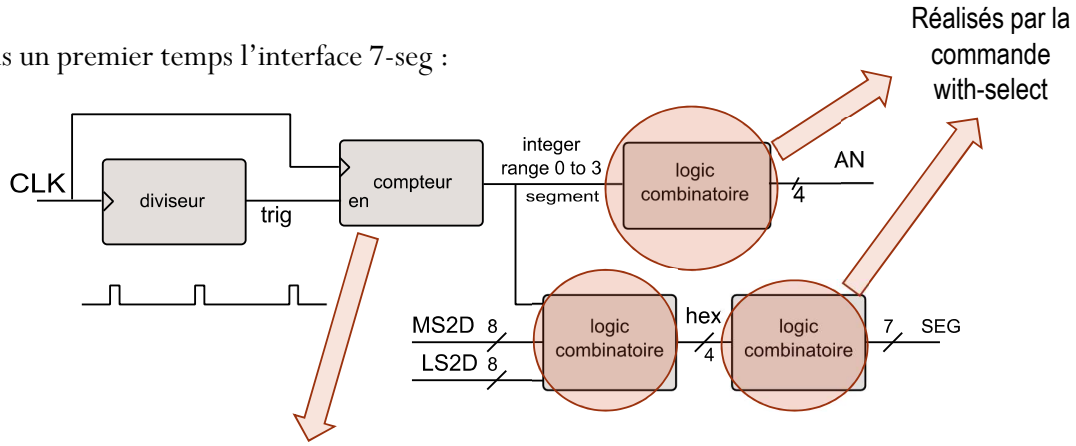
Exercice : Affichage 7-Seg

- On va créer le circuit nécessaire pour simplifier l'affichage d'une valeur hexadécimale 4 digits sur les sept-segments de la carte Nexy3. Ainsi le programme (API) pour afficher la valeur « 1234 » sur les 7-segment serait :

```
            LOAD    s0,12
            LOAD    s1,34
            CALL    AFF_7SEG
            ….
            ….
AFF_7SEG:
            OUTPUT  s0,01       ; l'adresse du port pour les 2 MSD
            OUTPUT  s1,02       ; l'adresse du port pour les 2 LSD
            RETURN
```

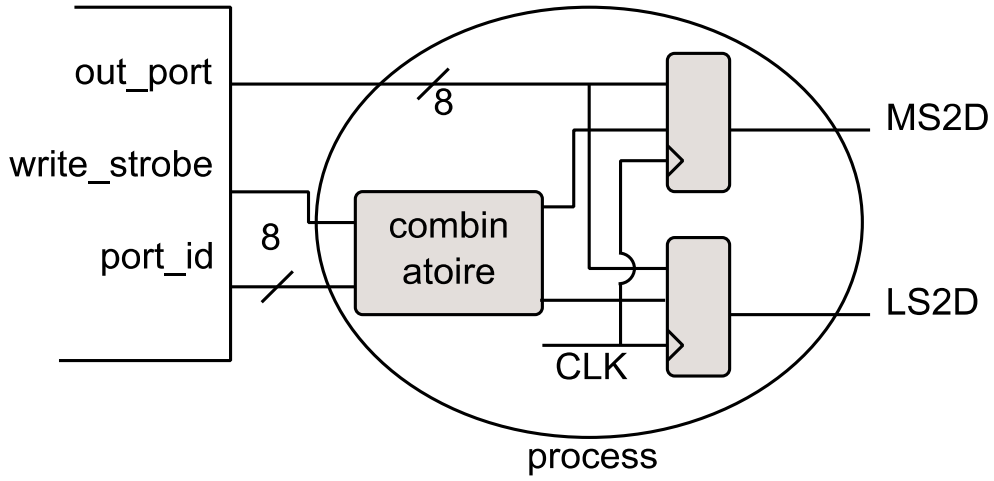
Suite : Conception circuit

Dans un premier temps l'interface 7-seg :




```
process(CLK)
begin
   if CLK'event and CLK='1' then
      if trig='1' then segment <= segment + 1; end if;
   end if;
end process;
```


Suite : Interface avec PicoBlaze



Exercice : Ajout d'un UART

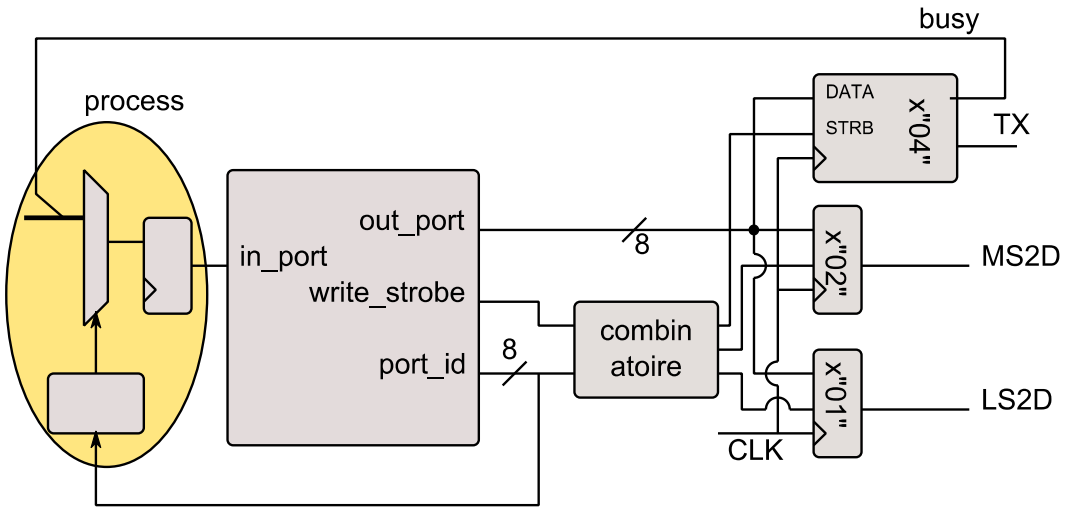
- En programmation, on vérifie si UART n'est pas "busy" on lui envoie un octet:

```
; API UART
UART:   INPUT   SF,01     ; Busy est à lire sur le port 1
        AND     SF,01     ; Busy est le LSB
        JUMP    NZ,UART   ; attend tant que Busy = '1'
        OUTPUT  S0,04     ; envoyer à l'interface le carac
        RETURN            ; ASCII
```

Pour appeler :

```
LOAD    S0,"A"
CALL    UART
```


Suite : Ajout de l'UART



Ici, puisqu'il n'y a qu'une seule entrée sur le port « in_port »:

```
in_port <= "0000000" & busy;
```

Exercice : Ajout d'un timer

- On va maintenant mettre en place un timer qui donne une interruption toutes les secondes. La routine d'interruption va incrémenter un registre 16-bits puis envoie le contenu à l'interface qui l'affiche sur les 7-segments.
- Le hardware a été expliqué plus haut. Le programme sera:

```
ISR:    ADD      SA,01
        ADDC     SB,00
        OUTPUT   SA,01
        OUTPUT   SB,02
        RETURNI  ENABLE
```


Affichage du timer sur PC

```
ISR: ADD     SA,01
     ADDCY   SB,00
     OUTPUT  SA,01
     OUTPUT  SB,02
     LOAD    S0,SB
     ACALL   HEX2ASCII
     LOAD    S0,SE
     CALL    UART
     LOAD    S0,SF
     CALL    UART
     LOAD    S0,SA
     CALL    HEX2ASCII
     LOAD    S0,SE
     CALL    UART
     LOAD    S0,SF
     CALL    UART
```

```
; sous-programme qui transforme une
; donnée numérique 8 bits en deux codes
; ascii pour les deux digits
; "4F"   →  52 et 65
; on suppose la donnée 8 bits dans S0
; la routine retourne deux codes ascii
; dans SE et SF
HEX2ASCII:
          LOAD      SF,S0
          LOAD      SE,S0
          AND       SF,0F
          ADD       SF,30
          COMPARE   SF,3A
          JUMP      C,DONE1 (si SF ≥ 3A)
          ADD       SF,07
DONE1:    SR0       SE
          SR0       SE
          SR0       SE
          SR0       SE
          ADD       SE,30
          COMPARE   SE,3A
          JUMP      C,DONE2
          ADD       SE,07
DONE2:    RETURN
```

Carac	ASCII Hexa
0	30
1	31
…	…
9	39
A	41
B	42
…	…
F	46