System on Chip FPGA

Compteur de ’1’s

Yann Thoma

Reconfigurable and Embedded Digital Systems Institute
Haute Ecole d’Ingénierie et de Gestion du Canton de Vaud

Février 2024

Etude de cas

- Enoncé du problème:
  - Réaliser un composant capable de compter le nombre de 1 présentés à son entrée
  - Taille de l'entrée: 64 bits
  - Taille de la sortie: 7 bits
  - Pas de contrainte de temps

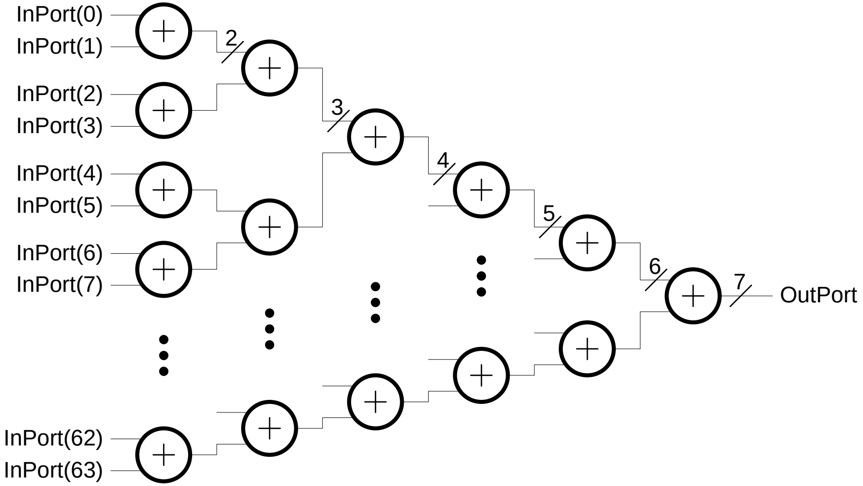
Déclaration d'entité

```vhdl
library ieee;
use ieee.std_logic_1164.all;

entity OneCtr is
port(
        clk      : in  std_logic;
        rst      : in  std_logic;
        start    : in  std_logic;
        InPort   : in  std_logic_vector(63 downto 0);
        OutPort  : out std_logic_vector(6 downto 0)
);
end OneCtr;
```

Architectures possibles

- Arbre d’additionneurs
- Arbre d’additionneurs en pipeline
- Un seul additionneur
- Convertisseur parallèle-série
- Multiplexeur d’entrée
- Processeur spécialisé

Arbre


InPort(0)
InPort(1)
InPort(2)
InPort(3)
InPort(4)
InPort(5)
InPort(6)
InPort(7)
InPort(62)
InPort(63)
2
3
4
5
6
7
OutPort

```vhdl
architecture combinatorial of OneCtr is

type inter1_type is array(0 to 63) of std_logic_vector(0 downto 0);
signal inter1:  inter1_type;
type inter2_type is array(0 to 31) of std_logic_vector(1 downto 0);
signal inter2:  inter2_type;
type inter3_type is array(0 to 15) of std_logic_vector(2 downto 0);
signal inter3:  inter3_type;
type inter4_type is array(0 to 7) of std_logic_vector(3 downto 0);
signal inter4:  inter4_type;
type inter5_type is array(0 to 3) of std_logic_vector(4 downto 0);
signal inter5:  inter5_type;
type inter6_type is array(0 to 1) of std_logic_vector(5 downto 0);
signal inter6:  inter6_type;

component adder
generic ( SIZE: integer:=2);
port (input1,input2: in std_logic_vector(SIZE-1 downto 0);
        output: out std_logic_vector(SIZE downto 0));
end component;
```

```vhdl
begin
gen1: for i in 0 to 31 generate
    inter1(2*i)(0)   <= InPort(2*i);
    inter1(2*i+1)(0) <= InPort(2*i+1);
    add: adder
    generic map(SIZE=>1)
    port map(
        input1=>inter1(2*i),input2=>inter1(2*i+1),output=>inter2(i));
end generate;

gen2: for i in 0 to 15 generate
    add: adder
    generic map(SIZE=>2)
    port map(
        input1=>inter2(2*i),input2=>inter2(2*i+1),output=>inter3(i));
end generate;

gen3: for i in 0 to 7 generate
    add: adder
    generic map(SIZE=>3)
    port map(
        input1=>inter3(2*i),input2=>inter3(2*i+1),output=>inter4(i));
end generate;
```

```vhdl
gen4: for i in 0 to 3 generate
    add: adder
    generic map(SIZE=>4)
    port map(
        input1=>inter4(2*i),input2=>inter4(2*i+1),output=>inter5(i));
end generate;
gen5: for i in 0 to 1 generate
    add: adder
    generic map(SIZE=>5)
    port map(
        input1=>inter5(2*i),input2=>inter5(2*i+1),output=>inter6(i));
end generate;
gen6: for i in 0 to 0 generate
    add: adder
    generic map(SIZE=>6)
    port map(
        input1=>inter6(2*i),input2=>inter6(2*i+1),output=>OutPort);
end generate;

end combinatorial;
```

Pipeline

```
architecture pipeline of OneCtr is

type inter1_type is array(0 to 63) of std_logic_vector(0 downto 0);
signal inter1:  inter1_type;
type inter2_type is array(0 to 31) of std_logic_vector(1 downto 0);
signal inter2:  inter2_type;
type inter3_type is array(0 to 15) of std_logic_vector(2 downto 0);
signal inter3:  inter3_type;
type inter4_type is array(0 to 7) of std_logic_vector(3 downto 0);
signal inter4:  inter4_type;
type inter5_type is array(0 to 3) of std_logic_vector(4 downto 0);
signal inter5:  inter5_type;
type inter6_type is array(0 to 1) of std_logic_vector(5 downto 0);
signal inter6:  inter6_type;

component adderreg
generic ( SIZE: integer:=2);
port (clk, rst: in std_logic;
      input1,input2: in std_logic_vector(SIZE-1 downto 0);
      output: out std_logic_vector(SIZE downto 0));
end component;
```

```
begin
gen1: for i in 0 to 31 generate
    inter1(2*i)(0)<=InPort(2*i);
    inter1(2*i+1)(0)<=InPort(2*i+1);
    add: adderreg
    generic map(SIZE=>1)
    port map(clk=>clk,rst=>rst,input1=>inter1(2*i),
             input2=>inter1(2*i+1),output=>inter2(i));
end generate;

gen2: for i in 0 to 15 generate
    add: adderreg
    generic map(SIZE=>2)
    port map(clk=>clk,rst=>rst,input1=>inter2(2*i),
             input2=>inter2(2*i+1),output=>inter3(i));
end generate;

gen3: for i in 0 to 7 generate
     add: adderreg
     generic map(SIZE=>3)
     port map(clk=>clk,rst=>rst,input1=>inter3(2*i),
              input2=>inter3(2*i+1),output=>inter4(i));
end generate;
```

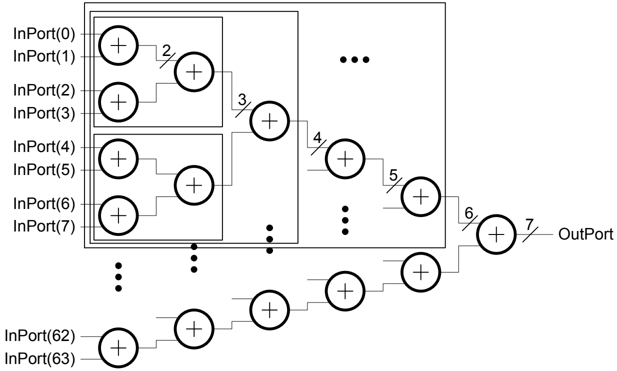
```
gen4: for i in 0 to 3 generate
     add: adderreg
     generic map(SIZE=>4)
     port map(clk=>clk,rst=>rst,
        input1=>inter4(2*i),input2=>inter4(2*i+1),output=>inter5(i));
end generate;
gen5: for i in 0 to 1 generate
    add: adderreg
    generic map(SIZE=>5)
    port map(clk=>clk,rst=>rst,
        input1=>inter5(2*i),input2=>inter5(2*i+1),output=>inter6(i));
end generate;
gen6: for i in 0 to 0 generate
    add: adderreg
    generic map(SIZE=>6)
    port map(clk=>clk,rst=>rst,
        input1=>inter6(2*i),input2=>inter6(2*i+1),output=>OutPort);
end generate;

end pipeline;
```

Version récursive

- Une structure telle que celle-ci est récursive!

⇒ Implémentation récursive...

- A chaque niveau:
  - 2 sous-blocs
  - 1 additionneur



Version récursive

Partie déclarative

```vhdl
entity OneCtr is
    generic(SIZE: integer:=8);
    port(
        InPort  : in  std_logic_vector(SIZE-1 downto 0);
        OutPort : out std_logic_vector(ilog(SIZE+1) downto 0)
   );
end OneCtr;

architecture struct of OneCtr is

    signal op1         : std_logic_vector(ilog(SIZE+1) downto 0);
    signal op2         : std_logic_vector(ilog(SIZE+1) downto 0);
    signal OutPortInt0 : std_logic_vector(ilog(SIZE/2+1) downto 0);
    signal OutPortInt1 : std_logic_vector(ilog(SIZE-SIZE/2+1) downto 0);

begin

    ....
```

Version récursive

Si la taille des entrées est 1

```
gen1: if SIZE=1 generate
    OutPort(0) <= InPort(0);
    OutPort(1) <= '0';
end generate;
```

Version récursive

Instanciation des sous-blocs si nécessaire

```
genmix: if (SIZE>1) generate
    c0: OneCtr
    generic map(SIZE => SIZE/2)
    port map(
        InPort  => InPort(SIZE/2-1 downto 0),
        OutPort => OutPortInt0
    );

    c1: OneCtr
    generic map(SIZE => SIZE-SIZE/2)
    port map(
        InPort  => InPort(SIZE-1 downto SIZE/2),
        OutPort => OutPortInt1
    );

    ...
```

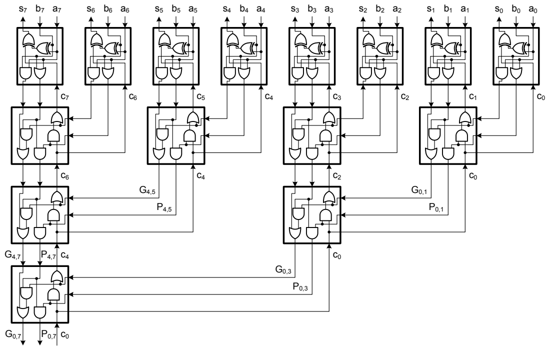
Version récursive

Préparation des opérandes et addition

```
    process(all)
    begin
        op1 <= (others => '0');
        op1(ilog(SIZE/2+1) downto 0) <= OutPortInt0;
    end process;

    process(all)
    begin
        op2 <= (others => '0');
        op2(ilog(SIZE-SIZE/2+1) downto 0) <= OutPortInt1;
    end process;

    OutPort<=std_logic_vector(unsigned(op1) + unsigned(op2));

end generate;
```

Récursion: potentiels

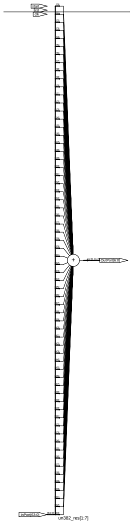
- Les utilisations peuvent être variées
- Exemples:
  - Arbre d'additionneurs
  - Arbre de multiplexeurs
  - Réseau de neurones en couche
  - Additionneur à anticipation de retenue

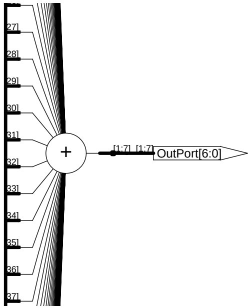


Un seul additionneur

```vhdl
architecture oneadder of OneCtr is
begin

    process(all)
        variable res : std_logic_vector(6 downto 0);
    begin
        res := (others => '0');
        for i in 0 to 63 loop
            res := res + InPort(i);
        end loop;
        OutPort <= res;
    end process;
end oneadder;
```

Un seul additionneur

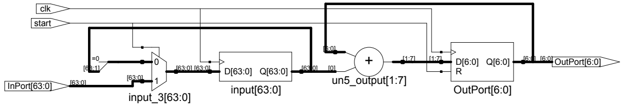




Convertisseur parallèle-série

```vhdl
architecture PISO of OneCtr is
    signal input  : std_logic_vector(63 downto 0);
    signal output : std_logic_vector(6 downto 0);
begin

    process(all)
    begin
        if rising_edge(clk) then
            if start = '1' then
                input <= InPort;
                output <= (others => '0');
            else
                input <= '0' & input(63 downto 1);
                output <= output + input(0);
            end if;
        end if;
    end process;

    OutPort<=output;

end PISO;
```

Convertisseur parallèle-série



Multiplexeur d’entrée

```vhdl
architecture Mux of OneCtr is
    signal counter : std_logic_vector(5 downto 0);
    signal output  : std_logic_vector(6 downto 0);
begin

    process(all)
    begin
        if rising_edge(clk) then
            if start = '1' then
                counter <= (others => '0');
                output <= (others => '0');
            else
                counter <= counter + 1;
                output <= output + InPort(to_integer(unsigned(counter)));
            end if;
        end if;
    end process;

    OutPort<=output;

end Mux;
```

Comparaison

- Synthèse pour un XCV100-4 TQ144, avec Synplify

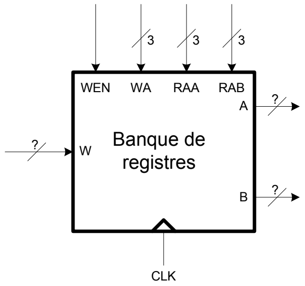
Architecture	LUTs	DFF	Fréquence (MHz)	Pas
Arbre	140	0	53.1	1
Direct	140	0	53.1	1
Pipeline	183	183	177.0	6
PISO	70	71	177.0	64
Mux	49	21	98.1	64
Processeur	135	9	123.3	472

Algorithme possible

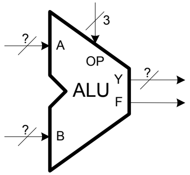
```
repeat forever{
    wait for Start
    Result ← 0
    Mask   ← 1
    Data   ← InPort
    Count  ← 0
    while (Count<>64) {
        Tmp    ← Data and Mask
        Result ← Result + Tmp
        Data   ← Data >> 1
        Count  ← Count + 1
    }
    OutPort ← Result
}
```

Blocs de base nécessaires

- Stockage de cinq variables
  - Result, Mask, Data, Count, Tmp



- $Y = A + B$
- $Y = A \gg 1$
- $Y = A\ and\ B$
- $F = (A == B)$



Standardisation de la précision

Count $\leftarrow$ Count + 1	7 bits
Data $\leftarrow$ Data » 1	64 bits
Tmp $\leftarrow$ Data and Mask	64 bits
Result $\leftarrow$ Result + Tmp	7 bits

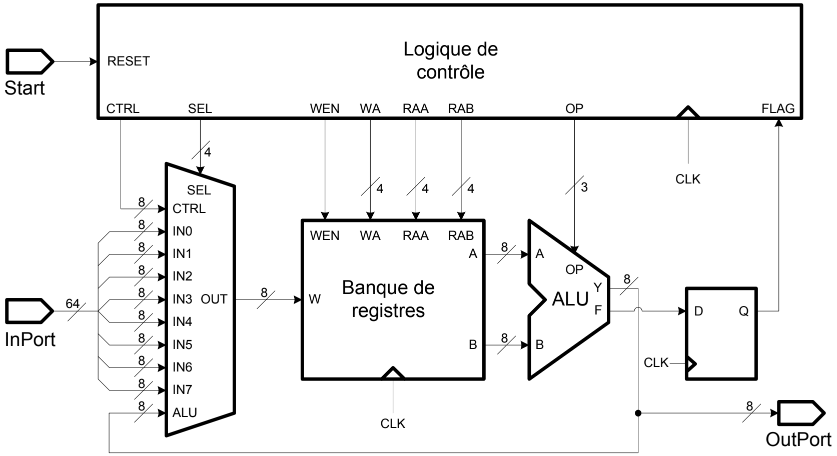
- Dans l'optique de réaliser ce traitement avec un processeur, il faut standardiser les données.

$\Longrightarrow$ On place tout sur 8 bits

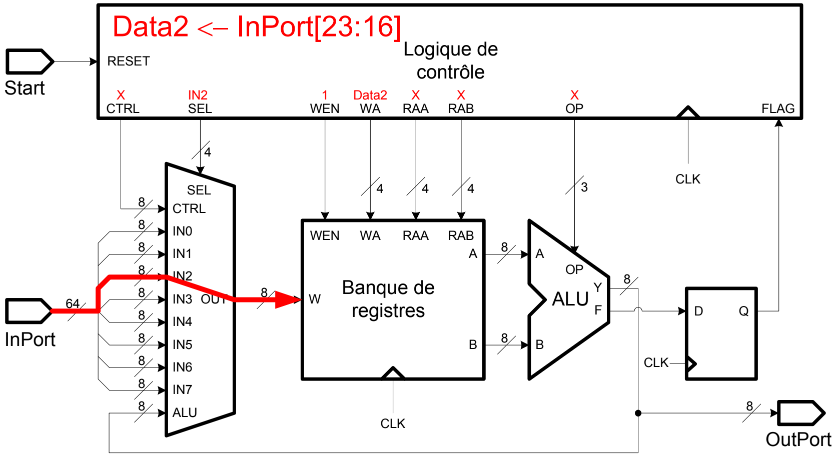
Adaptation de l’algorithme à la précision

```
repeat forever{
    wait for Start
    Result ← 0
    Mask   ← 1
    Data0  ← InPort[7:0]
    Data1  ← InPort[15:8]
    ...
    Data7  ← InPort[63:56]
    Count  ← 0
    while (Count<>8) {
        Tmp    ← Data0 and Mask
        Result ← Result + Tmp
        Data0  ← Data0 >> 1
        ...
        Tmp    ← Data7 and Mask
        Result ← Result + Tmp
        Data7  ← Data7 >> 1
        Count  ← Count + 1
    }
    OutPort ← Result
}
```

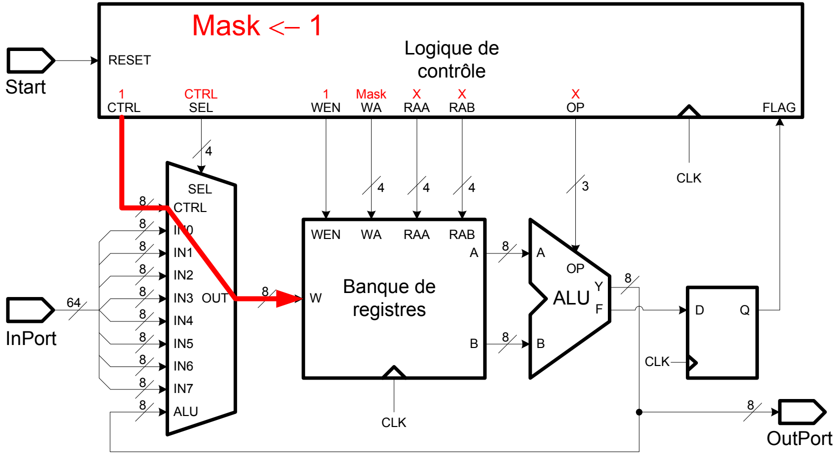
Une structure possible pour le traitement des données



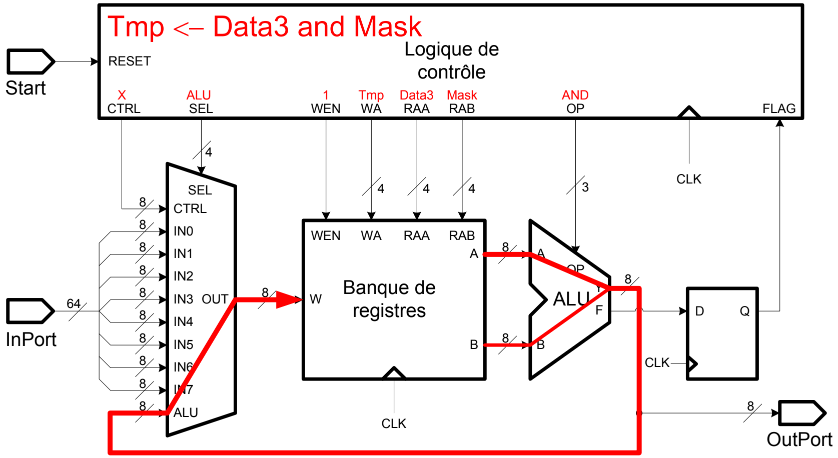
Exemple de chargement



Exemple de chargement



Exemple de calcul



Signaux de contrôle

SEL

CTRL	0000
IN0	0001
IN1	0010
IN2	0011
IN3	0100
IN4	0101
IN5	0110
IN6	0111
IN7	1000
ALU	1001

WA
RAA
RAB

Result	0000
Mask	0001
Data0	0010
Data1	0011
Data2	0100
Data3	0101
Data4	0110
Data5	0111
Data6	1000
Data7	1001
Count	1010
Tmp	1011
Zero	1100
One	1101
Eight	1110

OP

ADD	000	$Y = A + B$
SHR	001	$Y = A \gg 1$
EQ	010	$F = (A == B)$
AND	011	$Y = A\ and\ B$
MOV	100	$Y = A$

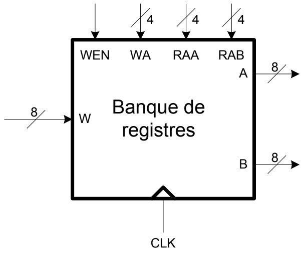
Caractéristiques de la banque de registres

- Les sorties sont disponibles après un délai combinatoire

```
A <= registers(to_integer(unsigned(RAA)));
B <= registers(to_integer(unsigned(RAB)));
```

- La mise à jour se fait sur le flanc montant de l'horloge

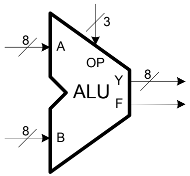
```
if rising_edge(clk) then
    if WE = '1' then
        registers(to_integer(unsigned(WA))) <= W;
    end if;
end if;
```



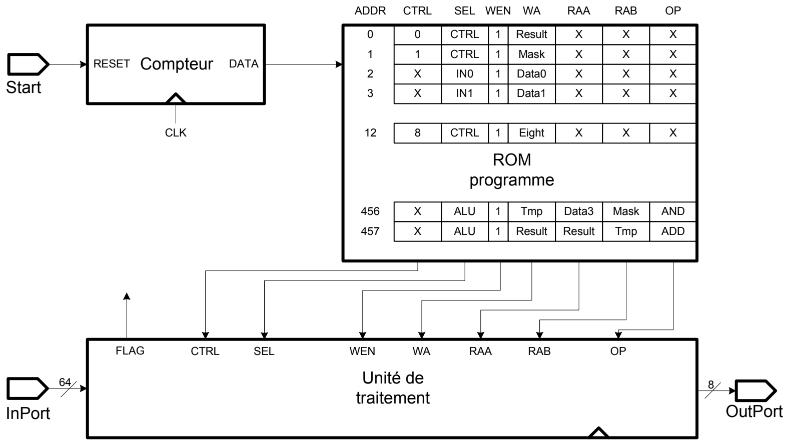
Unité arithmétique et logique

- Purement combinatoire
- Plusieurs opérations à disposition

```
case OP is
    when ADD =>
        Y <= A+B;
    when SHR =>
        Y <= '0' & A(7 downto 1);
    ...
end case;
```



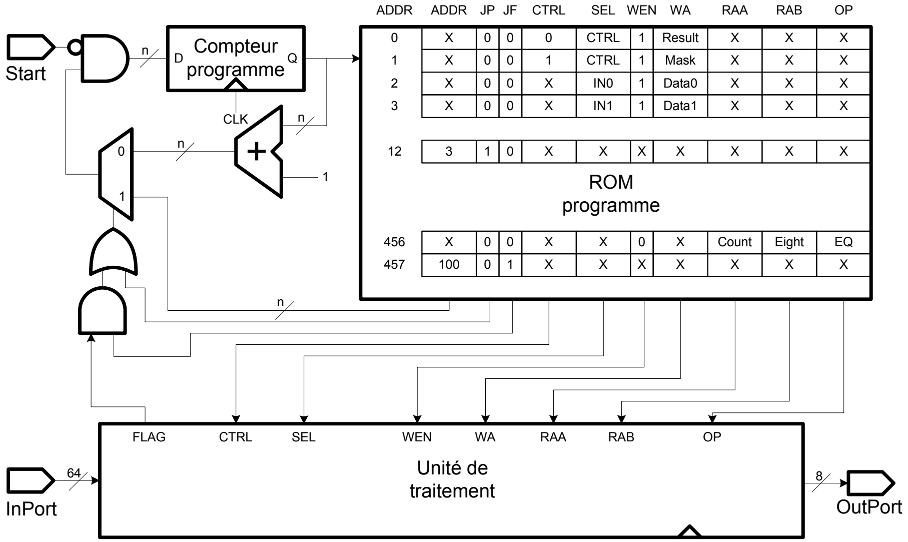
Comment générer les signaux de contrôle?



ADDR	CTRL	SEL	WEN	WA	RAA	RAB	OP
0	0	CTRL	1	Result	X	X	X
1	1	CTRL	1	Mask	X	X	X
2	X	IN0	1	Data0	X	X	X
3	X	IN1	1	Data1	X	X	X
12	8	CTRL	1	Eight	X	X	X
456	X	ALU	1	Tmp	Data3	Mask	AND
457	X	ALU	1	Result	Result	Tmp	ADD

Utilisation d'une ROM et lecture séquentielle des instructions. Mais, comment gérer le "while count/=8"?

Autorisation des sauts conditionnels



ADDR	ADDR	JP	JF	CTRL	SEL	WEN	WA	RAA	RAB	OP
0	X	0	0	0	CTRL	1	Result	X	X	X
1	X	0	0	1	CTRL	1	Mask	X	X	X
2	X	0	0	X	IN0	1	Data0	X	X	X
3	X	0	0	X	IN1	1	Data1	X	X	X
12	3	1	0	X	X	X	X	X	X	X
456	X	0	0	X	X	0	X	Count	Eight	EQ
457	100	0	1	X	X	X	X	X	X	X

Exemple de l'implémentation de la boucle while

- Calcul du fanion, puis branchement conditionnel

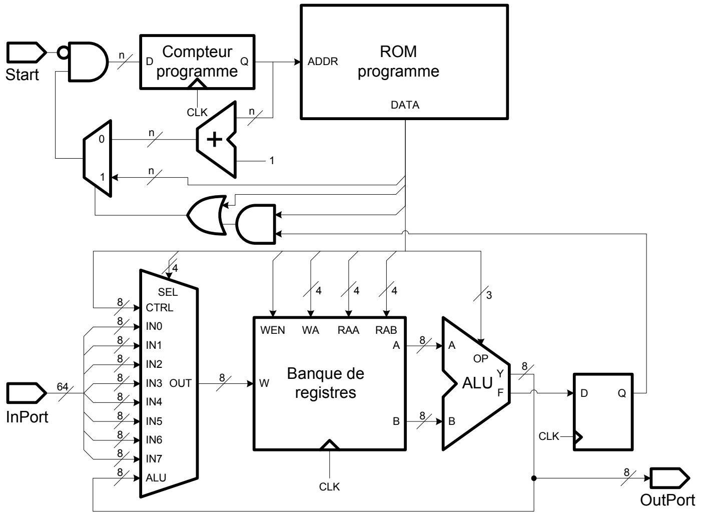
```
            ...
LoopBegin:  Tmp ← Data0 and Mask                 (début du while)
            ...
            Count ← Count + 1
            Flag ← Count EQ 8
            if Flag set, jump to LoopOut
            Jump to LoopBegin
LoopOut:    ...                                  (fin du while)
```

Besoin d'un registre supplémentaire

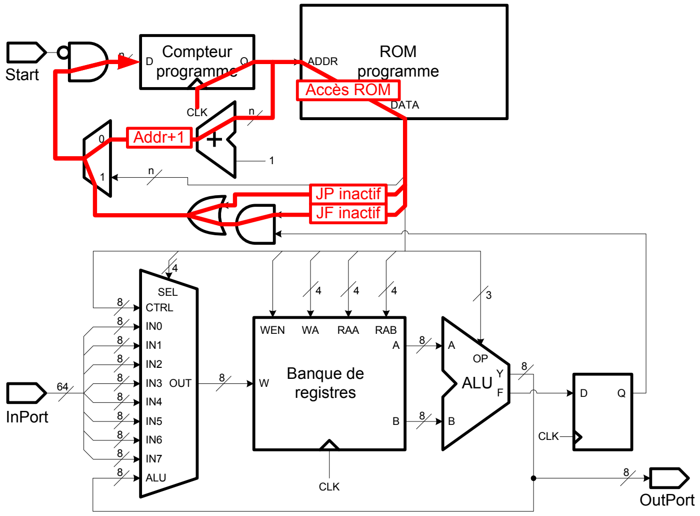
- Il n'existe pas de chemin de CTRL à ALU, il faut un registre intermédiaire

```
           Eight ← 8
           ...
LoopBegin: Tmp ← Data0 and Mask
           ...
           Count ← Count + 1
           Flag ← Count EQ Eight
           if Flag set, jump to LoopOut
           Jump to LoopBegin
LoopOut:   ...
```

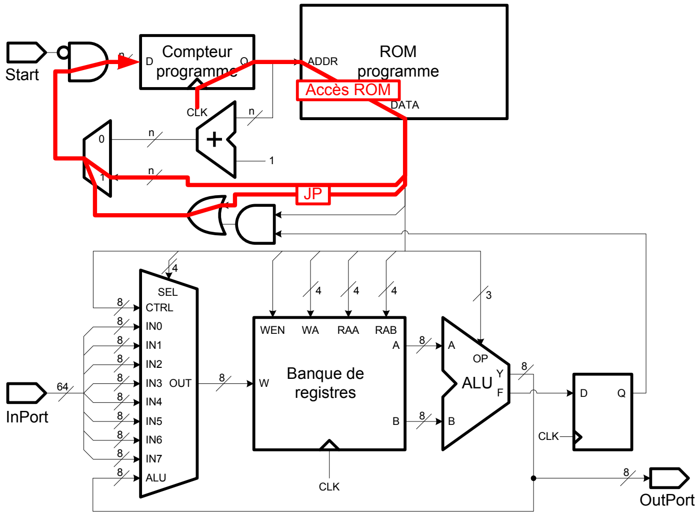
Compteur de ’1’ complet



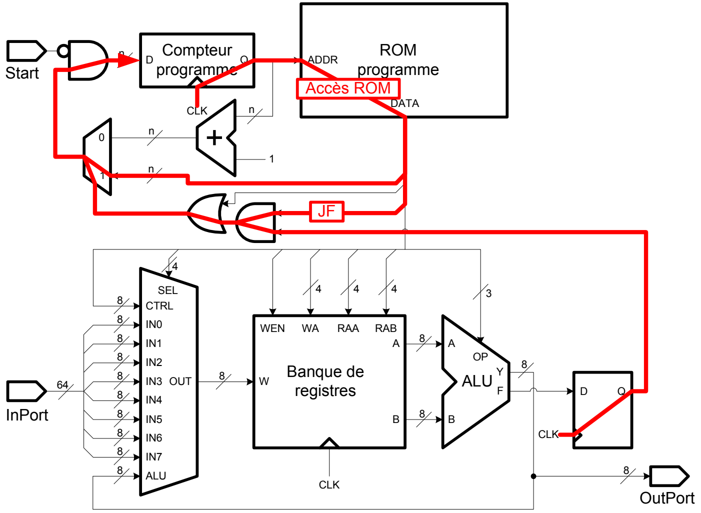
Propagation des signaux



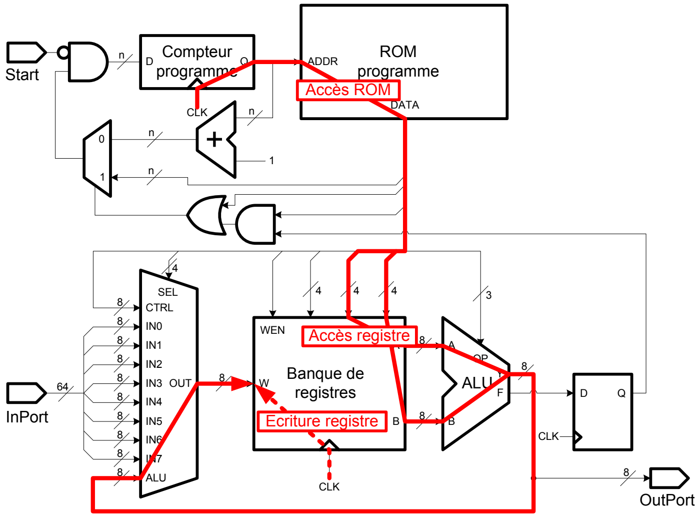
Propagation des signaux



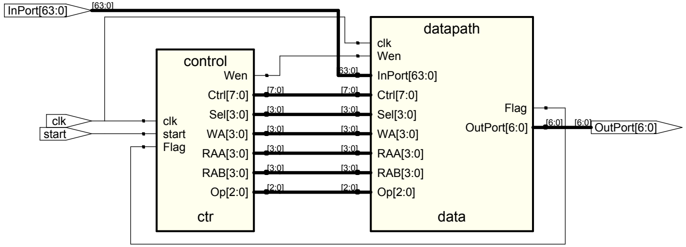
Propagation des signaux



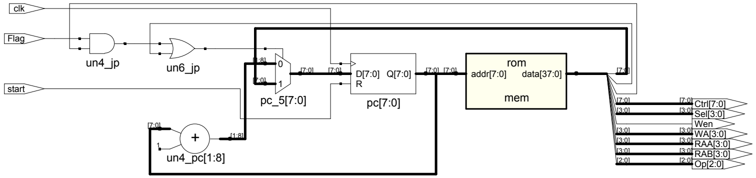
Propagation des signaux



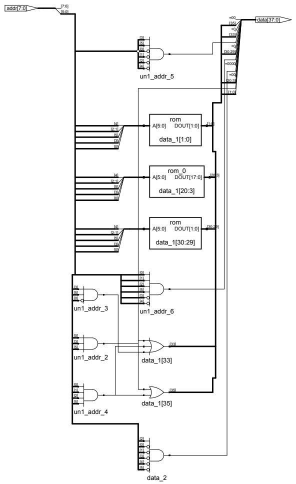
Synthèse du processeur



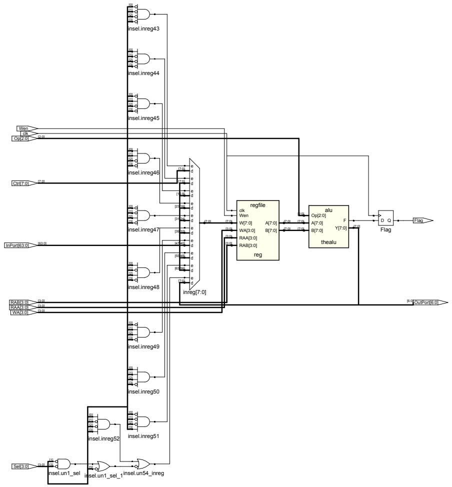
Synthèse de l’unité de contrôle



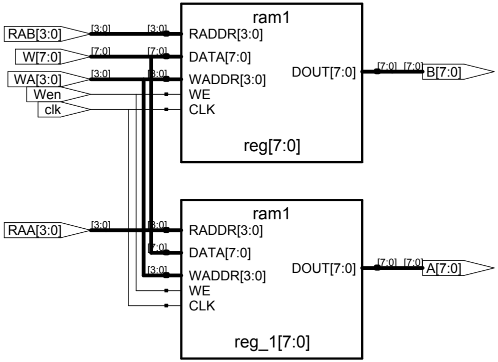
Synthèse de la ROM



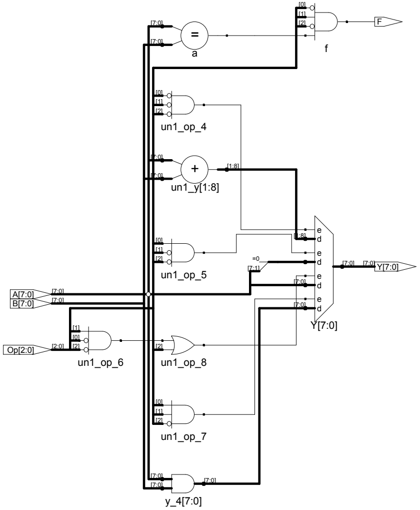
Synthèse de l'unité de traitement



Synthèse du banc de registres



Synthèse de l'ALU



A[7:0]
B[7:0]
Op[2:0]
a
f
F
un1_op_4
un1_y[1:8]
un1_op_5
un1_op_6
un1_op_8
un1_op_7
y_4[7:0]
Y[7:0]
=0