Bobillot Damien
Dimitri Druelle
X 2002

Rapport de projet d'EA FPGA

Processeur Thumb avec contrôleur VGA

Introduction

Parmi la multitude de composants d'électronique numérique – contrôleurs divers, cartes de codage/décodage, calculs… – les processeurs restent les composants les plus fascinants, en particulier par leur aspect omnipotent. En effet, un processeur peut-être vu comme une machine universelle pouvant effectuer n'importe quelle tâche ne dépendant que du programme qu'on lui demande d'exécuter : c'est l'interface entre le monde virtuel – les programmes et les actions qu'on leur demande d'effectuer – et le monde réel – la création et l'envoi du signal électrique vers un écran par exemple. C'est un peu la raison pour laquelle nous nous sommes tourné vers le choix d'un tel composant.

Un processeur, vu par le principal client, un programmeur, c'est avant tout un jeu d'instructions. Nous aurions pu réinventer un jeu d'instruction, mais nous pensons qu'il est préférable d'en utiliser un déjà existant. Cela permet entre autres de découvrir quels sont les jeux d'instructions couramment utilisés ainsi que leurs spécificités lors des recherches préliminaires. Notre choix c'est tourné vers le jeu d'instructions Thumb (une variante de l'ARM, un acronyme pour *Advanced RISC Machines*.). C'est un jeu d'instruction assez simple et codé sur 16 bits – c'est important car cela permet de lire une instruction en RAM en un cycle.

Structure du processeur

Pour réaliser notre microprocesseur nous sommes parti d'une architecture de Von Neumann standard et nous l'avons implémenté à notre façon.

Dans notre projet presque tous les composants sont réalisés en VHDL. Ce fut un choix de notre part, afin d'un, d'utiliser la puissance de ce langage et deuxièmement afin de faciliter la création du contrôleur qui est le cœur du microprocesseur.

Le jeu d'instruction que nous avons utilisé est le Thumb. Comme tout jeu d'instructions, le Thumb à des avantages – être sur 16 bits donc pratique à mettre en œuvre lors de l'utilisation de mémoire accessibles par mots de 16 bits, être un jeu RISC donc avec un nombre limité d'instructions – et des inconvénients – un format d'instruction assez complexe vu qu'il s'agit d'une compression d'un jeu d'instruction sur 32 bits, l'ARM.

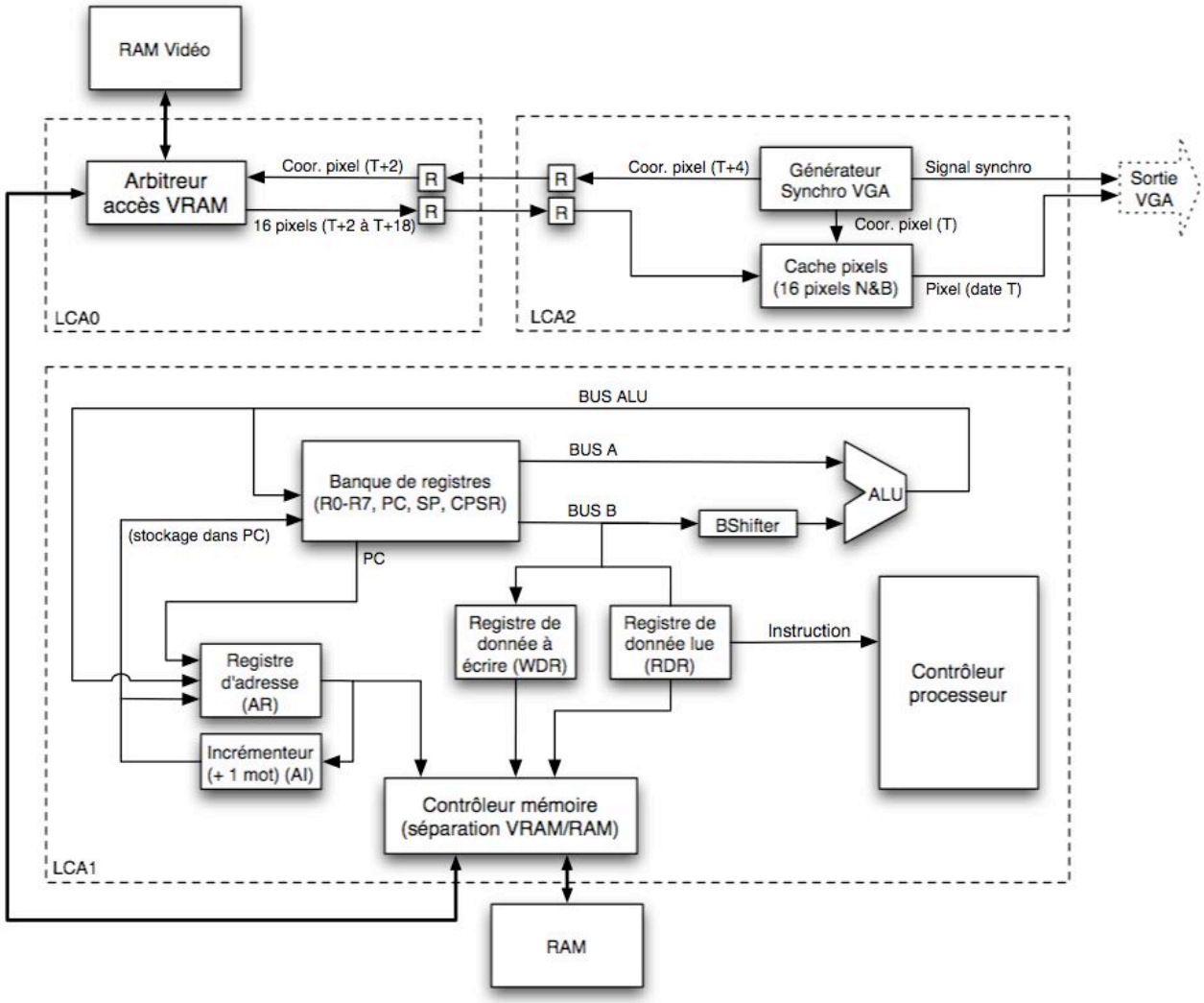
Notre processeur est construit autour d'une architecture Von Neumann : une banque de registres, une ALU, et pour les accès mémoires des registres d'adresse (AR), de données lues (RDR) et de données à écrire (WDR). À cela s'ajoute trois éléments :

- L'incrémenteur d'adresse (AI). Ce composant permet d'incrémenter l'adresse contenu dans l'AR à chaque cycles. Cela permet par exemple de lire une instruction par cycle dans la mémoire, ou de recharger des registres sauvegarder de manière consécutive sur la pile lors des appels de fonctions (instruction Load Multiple Registers).

- Le *Barrel Shifter* (BS) effectue une rotation des bits suivit d'un masque. Ce composant à pour but initial de récupérer les données immédiates situées dans l'instruction : une instruction lue est située au départ dans le RDR, de là elle est envoyée vers le contrôleur principal du processeur (chef d'orchestre dirigeant les communications entre les divers composants) et aussi sur le bus B. Le *Barrel Shifter* récupère alors la valeur immédiate.

- Le contrôleur mémoire. Il redirige les accès mémoire vers la RAM ou la VRAM selon l'adresse. La RAM correspond aux adresses dont le bit de poids fort est 0. La VRAM correspond aux autres adresses.

Schéma général du système :

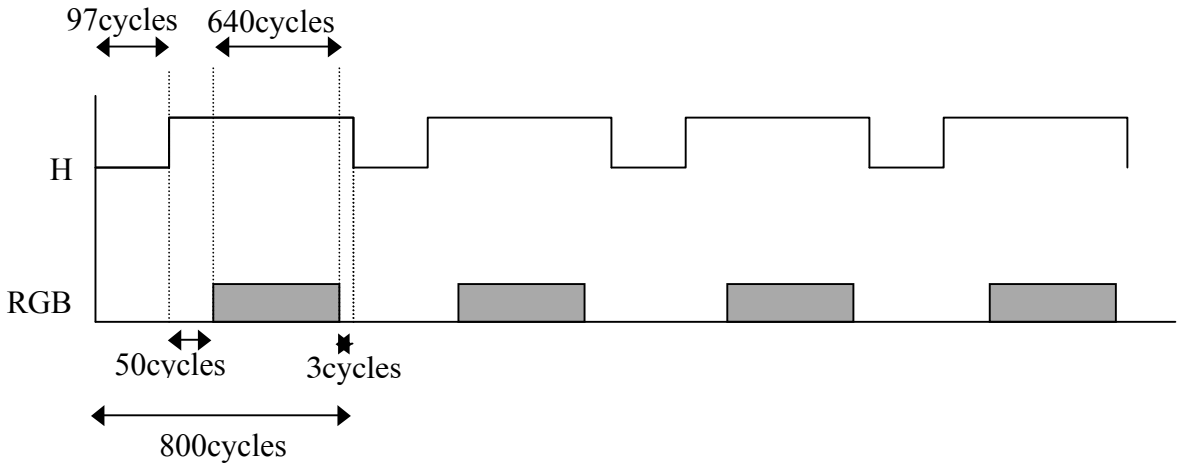


LCA0: ce composant contient le système de gestion de la ram vidéo.
LCA1: ce composant contient le microprocesseur.
LCA2: ce composant contient le système de gestion de l'affichage.

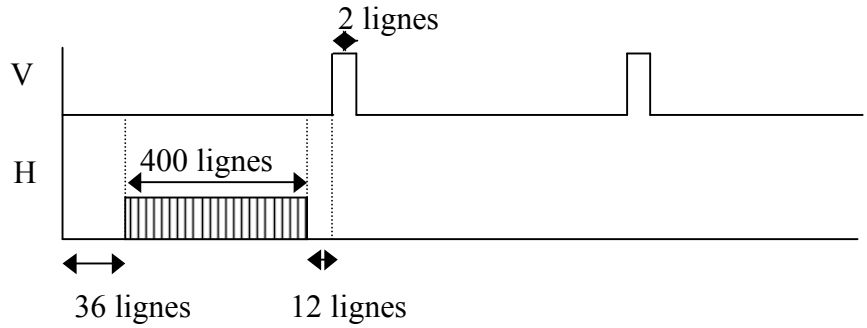
Contrôleur VGA

Une image est constituée de 640 pixels par ligne et de 400 lignes. Afin que l'image soit bien synchronisée, la norme VGA impose qu'il y ait plusieurs cycles pendant lesquels on envoie un signal noir à l'écran. Ainsi, au début de chaque image il faut qu'il y ait 1600 cycles de noir et au début de chaque ligne 160 cycles. Ce qui fait que pour afficher une image complètement il faut que nous attendions 360000 cycles. La durée d'affichage d'un pixel doit être de 40ns. Donc pour afficher une image cela prend 14,4ms.

Synchronisation horizontale (unité horizontale en nombre de cycles, 1cycle=40ns)



Synchronisation verticale (unité horizontale 1ligne=800cycles)



L'image affichée est stockée en Ram (celle que nous appelons VRam). Le pixel de coordonnée (x,y) est stocké dans le bit x&15 à l'adresse y<<6+x>>4.

Cette technique n'utilise pas toute la mémoire, mais permet de calculer l'adresse facilement à partir des coordonnées.

Problèmes rencontrés

Limitation de la mémoire

L'un des problèmes que nous avons vite rencontré fut celui de la taille de la mémoire vidéo. En effet si nous voulons coder tous nos pixels avec les 3 couleurs principales (Rouge Vert Bleu) la taille de la mémoire est insuffisante. Notre objectif n'étant pas de faire un affichage parfait mais juste un jeu type casse-briques, une image en noir et blanc nous suffisait largement. Par ailleurs, du fait que l'on adresse la mémoire par octet et que la taille des registres est limité à 16 bits, nous ne pouvons adresser que 64Ko de mémoire RAM et vidéo, bien que l'on dispose de 512Ko (deux fois 128K de mots de deux octets). Cela limite d'autant plus la taille de mémoire graphique utilisable.

Une solution, autre que le passage en 32 bits, aurait été de mapper juste une partie de la mémoire vidéo dans l'espace d'adressage disponible, et de mettre en place un mécanisme permettant de sélectionner la partie de mémoire vidéo mappée dans l'espace d'adressage (par exemple avec une valeur stockée à une position prédéfinie dans l'espace adressable). Par exemple, si on écrit la valeur 0 à l'adresse 0x7FFF, tout se passera comme si la plage d'adresse 0x8000-0xFFFF visible par le processeur était la plage 0x00000-0x07FFF de la VRAM. Si on écrit la valeur 1 à l'adresse 0x7FFF, tout ce passera maintenant comme si la plage d'adresse 0x8000-0xFFFF était la plage 0x08000-0x0FFFF de la VRAM et ainsi de suite.

Accès mémoire concurrentiels

Le but de la mise en place des systèmes RISC est de permettre de pipeliner les instructions, et ainsi de donner l'impression d'exécuter une instruction par cycle. Un des problèmes principaux que l'on a rencontré, et qu'un tel système doit pouvoir lire à chaque cycle une instruction dans la mémoire : il n'est donc jamais possible d'effectuer d'autre opération mémoire (chargement ou écriture d'un registre en RAM) sans casser ce rythme d'une instruction par cycle.

La solution utiliser pour résoudre ce problème est l'utilisation d'une mémoire permettant plusieurs accès simultanés. Une telle mémoire coûte chère et ne peut donc pas remplacer les mémoires RAM classiques. Cependant cela peut être utiliser sous forme de mémoires cache directement codées dans le processeur. Nous n'avons malheureusement pas eu le temps de l'inplémenter.

Vitesses de propagation des signaux

Nous avons un impératif sur la période d'horloge maximum sous laquelle peut tourner notre circuit. En effet, le contrôleur VGA a été conçu pour fonctionner à une période de 40ns, résolution des signaux VGA.

Cependant, la complexité du circuit définissant le processeur impose des périodes minimales supérieures à 40ns.

Deux solutions sont envisageables :

- Soit on fait tourner le contrôleur VGA à une fréquence plus faible : cela impose de réduire la résolution de l'affichage, par exemple en affichant des pixels deux fois plus gros.

- Soit on réduit la distance maximale entre deux registres dans le processeur (et ainsi, réduire le temps de propagation des signaux). Cela peut se faire en découpant l'exécution des instructions un nombre supérieur d'étapes, qui seront ainsi plus rapides à exécuter. Il faudrait alors allonger le pipeline ce qui peut rajouter des problèmes de dépendance entre des instructions en fin d'exécution dont le résultat sert à une instruction en début d'exécution.

Complexité du contrôleur processeur

Bien qu'ayant choisit un jeu d'instruction RISC, il reste tout de même 19 formats d'instructions, et surtout des instructions de format différent effectuant des opérations similaires (il y a par exemple 7 formats pour les instructions de type *load*). Ceci est dû au fait qu'il est difficile de créer un jeu d'instruction aussi complet sur uniquement 16 bits : tout les recoins possibles et imaginables sont utilisés.

Dans les processeurs ARM commercialisés, le jeu d'instruction natif est l'ARM : les instructions Thumb sont en fait d'abord converties en ARM avant d'entré dans le contrôleur du processeur. Cela extrait la complexité hors du contrôleur, ce qui plus facile à implémenter.

Conclusion

Notre objectif était de réaliser un microprocesseur sur lequel nous pourrions faire tourner une ou plusieurs applications. À l'heure où nous terminons ce rapport nous ne n'avons pas pu atteindre cet objectif. Il y a plusieurs raisons à ceci. La première est que réaliser un microprocesseur est un projet de grande envergure, demandant une réflexion avant codage importante, afin de déterminer quel structure il va falloir utiliser en fonction des caractéristiques propres des FPGA. Mais comme nous ne connaissons pas la carte et les composants XillinX, il y a beaucoup de problèmes matériels auxquels nous n'avions pas penser au début du projet et qui impose des restructurations complètes du projet, ce qui prends beaucoup de temps.

Néanmoins nous arrivons à un résultat satisfaisant mais non spectaculaire. En effet même si toutes les instructions ne sont pas codées, l'architecture est entièrement présente. Il suffit d'implémenter chaque instruction manquante dans le contrôleur.

Avec plus de temps nous aurions aimer mettre dans notre processeur de la cache sur la carte LCA3 qui est libre, finir toutes les instruction et surtout voir tourner notre casse-briques.

Formats d'instruction du Thumb :

	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Move Shift Register	0	0	0	OP		OFFSET					RS			RD		
Add/sub	0	0	0	1	1	L	OP		RN/OFFSET		RS			RD		
Move/comp./add/sub immediate	0	0	1	OP		RD			OFFSET							
ALU Operation	0	1	0	0	0	0	OP				RS			RD		
HI Register operation /branch exchange	0	1	0	0	0	1	OP		H1	H2	RS/HS			RD/HD		
PC-relative load	0	1	0	0	1	RD			WORD							
Load/store with register offset	0	1	0	1	L	B	0		RO		RB			RD		
Load/store sign-extended byte/halfword	0	1	0	1	H	S	1		RO		RB			RD		
Load/store with immediat offset	0	1	1	B	L	OFFSET					RB			RD		
Load/store halfword	1	0	0	0	L	OFFSET					RB			RD		
SP-relative load/store	1	0	0	1	L	RD			WORD							
Load Adress	1	0	1	0	SP	RD			WORD							
Add offset to stack pointer	1	0	1	1	0	0	0	0	S	WORD						
Push/Pop register	1	0	1	1	L	1	0	R	RLIST							
Multiple Load/store	1	1	0	0	L	RB			RLIST							
Conditional branch	1	1	0	1	COND				SOFFSET							
Software interrupt	1	1	0	1	1	1	1	1	VALUE							
Unconditionnal branch	1	1	1	0	0	OFFSET										
Long Branch with link	1	1	1	1	H	OFFSET										

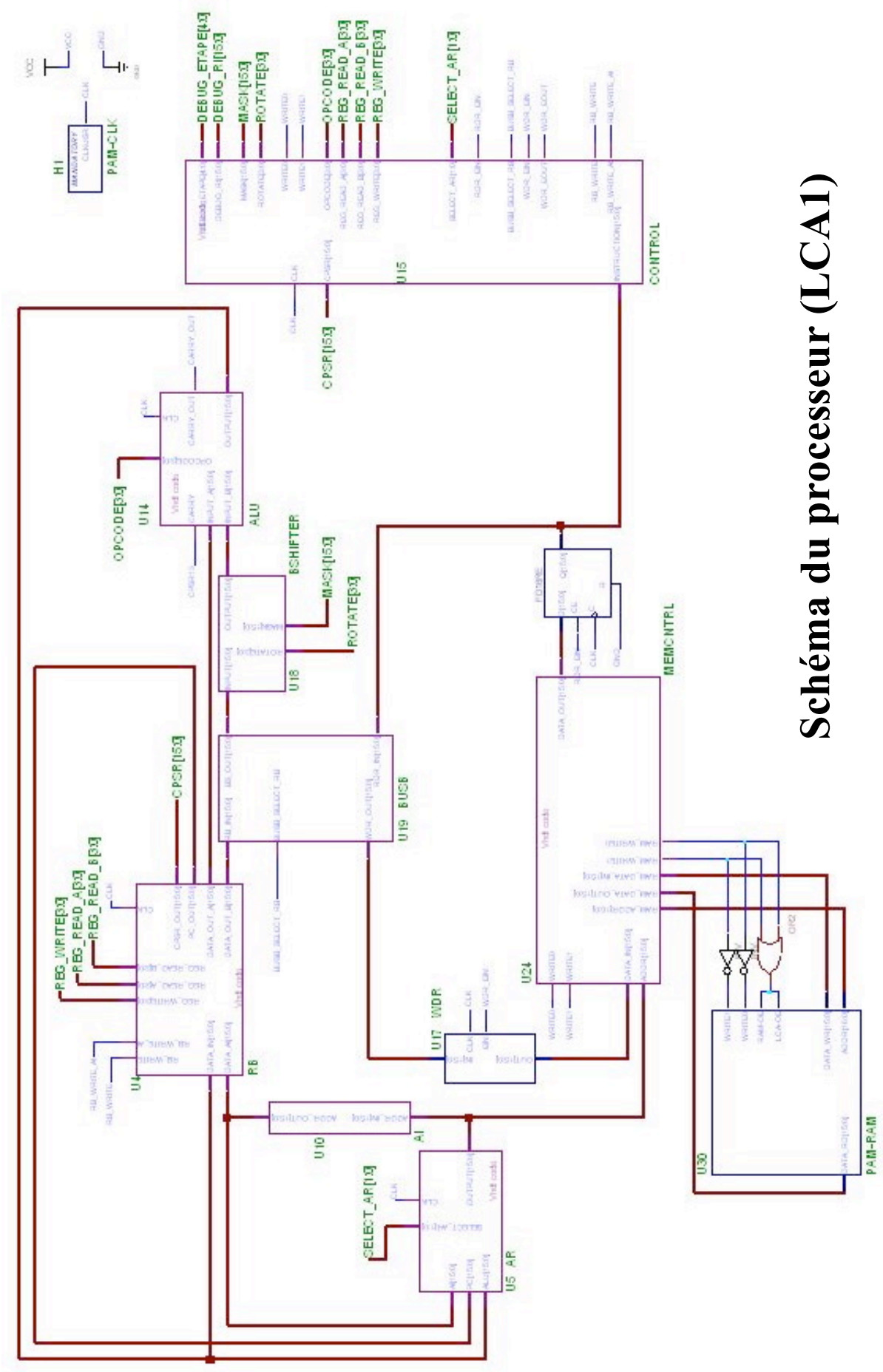
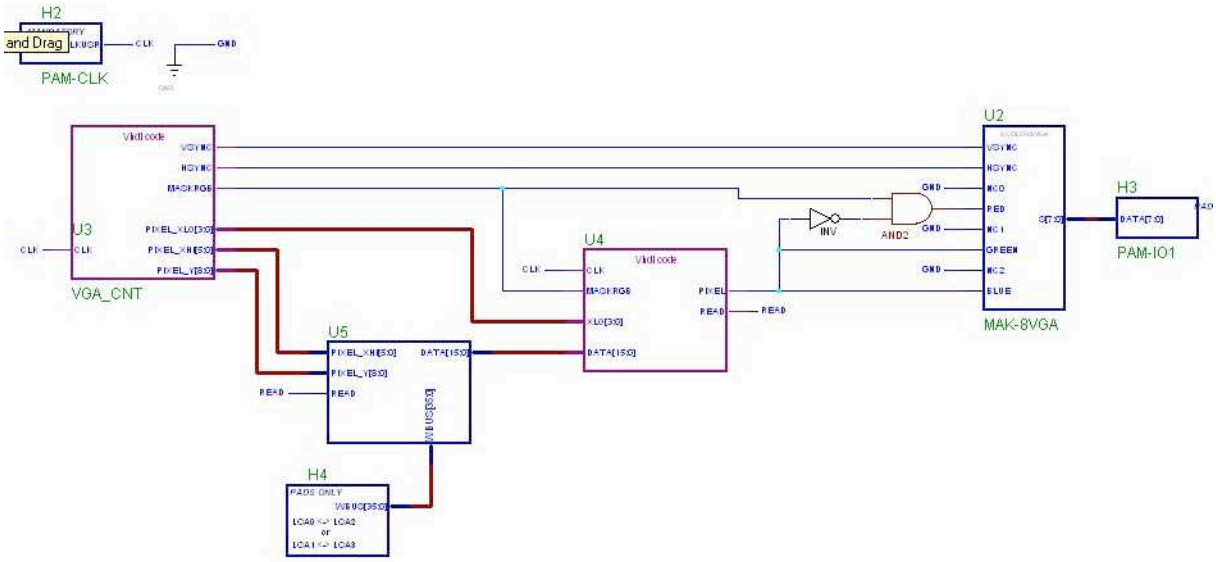
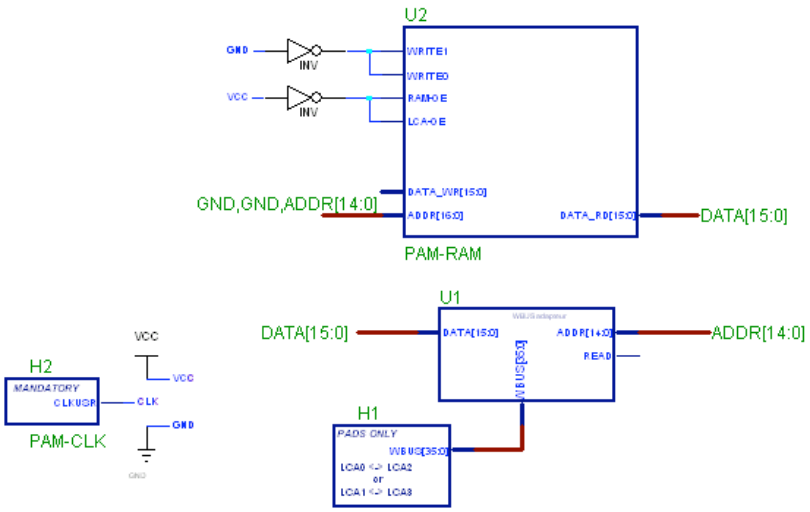


Schéma du processeur (LCA1)
CONTROL
U15
ALU
U14
BSHIFTER
U18
BUS6
U19
MEMCNTRL
U24
PAM-RAM
U30
WDR
U17
RB
U4
AR
U5
U10
H1
PAM-CLK
DEBUG_ETAPE[4:3]
DEBUG_PI[15:3]
MASK[15:3]
ROTATE[3:3]
OPCODE[3:3]
REG_READ_A[3:3]
REG_READ_B[3:3]
REG_WRITE[3:3]
SELECT_AR[1:3]
CPSR[15:3]

Contrôleur VGA (LCA2) :



Gestion de l'accès à la VRAM (LCA0) :



VGA_CNT : génération des signaux de synchronisation VGA

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity VGA_CNT is
    port (
        CLK: in STD_LOGIC;
        HSYNC: out STD_LOGIC;
        VSYNC: out STD_LOGIC;
        MASKRGB: out STD_LOGIC;
        PIXEL_XLO: out INTEGER range 0 to 15;
        PIXEL_XHI: out INTEGER range 0 to 63;
        PIXEL_Y: out INTEGER range 0 to 450
    );
end VGA_CNT;

architecture VGA_CNT_arch of VGA_CNT is
         signal H,V,RGB,Vmask : STD_LOGIC;
         signal hc : INTEGER range 0 to 800;
         signal vc : INTEGER range 0 to 450;
         signal xlo : INTEGER range 0 to 15;
         signal xhi : INTEGER range 0 to 63;
         signal y : INTEGER range 0 to 450;
begi

process (clk,H,V,RGB,hc,vc,xhi,xlo,y,Vmask)
begin
         if clk'event and clk='1' then
                  if hc=97 then
                           H<='1';
                  end if;
                  if hc=143 then
                           xhi<=0;
                  end if;
                  if hc=147 then
                           xlo<=0;
                           RGB<='1';
                  end if;
                  if hc=787 then
                           RGB <='0';
                           if Vmask='1' then
                                    y<=y+1;
                           end if;
                  end if;
                  if hc=800 then
                           hc<=1;
                           H<='0';
                           vc<=vc+1;
                           if vc=36 then
                                    Vmask<='1';
                           end if;
                           if vc=436 then
                                    Vmask<='0';
                           end if;
                           if vc=448 then
                                    V<='1';
                           end if;
                           if vc=450 then
                                    y<=0;
                                    vc<=0;
                                    V<='0';
                           end if;
                  else
                           hc<=hc+1;
                  end if;

                  if xlo=12 then
                           xhi<=xhi+1;
                  end if;
                  xlo<=xlo+1;
         end if;

         HSYNC    <= H;
         VSYNC    <= V;
         MASKRGB <= RGB and Vmask;
         PIXEL_Y <= y;
         PIXEL_XHI<= xhi;
         PIXEL_XLO <= xlo;
end process;
end VGA_CNT_arch;
```

DataReg : cache de pixels

```
library IEEE;
use IEEE.std_logic_1164.all;

entity DataReg is
    port (
        DATA:    in STD_LOGIC_VECTOR (0 to 15);
        XLO:     in INTEGER range 0 to 15;
        MASKRGB:in STD_LOGIC;
        CLK:     in STD_LOGIC;
        READ:    out STD_LOGIC;
        PIXEL:   out STD_LOGIC
    );
end DataReg;

architecture DataReg_arch of DataReg is
        signal Cache:STD_LOGIC_VECTOR (0 to 15);
begin
process (clk)
begin
        if clk'event and clk='1' then
                if XLO=0 then
                        Cache<=DATA;
                        PIXEL<=DATA(0) and MASKRGB;
                else
                        PIXEL<=Cache(XLO) and MASKRGB;
                end if;
                if xlo=12 then
                        READ<=MASKRGB;
                else
                        READ<='0';
                end if;
        end if;
end process;
end DataReg_arch;
```

RB : banc de registres

```
library IEEE;
use IEEE.std_logic_1164.all;

entity RB is
    port (
        DATA_IN: in STD_LOGIC_VECTOR (15 downto 0);
        DATA_AI: in STD_LOGIC_VECTOR (15 downto 0);
        RB_WRITE_AI: in STD_LOGIC;
        RB_WRITE: in STD_LOGIC;
        CLK: in STD_LOGIC;
        REG_WRITE: in STD_LOGIC_VECTOR (3 downto 0);
        REG_READ_A: in STD_LOGIC_VECTOR (3 downto 0);
        REG_READ_B: in STD_LOGIC_VECTOR (3 downto 0);
        PC_OUT: out STD_LOGIC_VECTOR (15 downto 0);
        DATA_OUT_A: out STD_LOGIC_VECTOR (15 downto 0);
        DATA_OUT_B: out STD_LOGIC_VECTOR (15 downto 0);
        CPSR_OUT: out STD_LOGIC_VECTOR (15 downto 0)
    );
end RB;

architecture RB_arch of RB is
        signal r0: STD_LOGIC_VECTOR (15 downto 0);
        signal r1: STD_LOGIC_VECTOR (15 downto 0);
        signal r2: STD_LOGIC_VECTOR (15 downto 0);
        signal r3: STD_LOGIC_VECTOR (15 downto 0);
        signal r4: STD_LOGIC_VECTOR (15 downto 0);
        signal r5: STD_LOGIC_VECTOR (15 downto 0);
        signal r6: STD_LOGIC_VECTOR (15 downto 0);
        signal r7: STD_LOGIC_VECTOR (15 downto 0);
        signal SP: STD_LOGIC_VECTOR (15 downto 0);
        signal LR: STD_LOGIC_VECTOR (15 downto 0);
        signal PC: STD_LOGIC_VECTOR (15 downto 0);
        signal CPSR: STD_LOGIC_VECTOR (15 downto 0);
        signal SPSR: STD_LOGIC_VECTOR (15 downto 0);
begin

process(r0,r1,r2,r3,r4,r5,r6,r7,SP,LR,PC,CPSR,SPSR,clk,REG_READ_A,REG_READ_B,RB_WRITE_AI,DATA_AI)
begin
        case REG_READ_A is
                when "0000" => DATA_OUT_A <= r0;
                when "0001" => DATA_OUT_A <= r1;
                when "0010" => DATA_OUT_A <= r2;
                when "0011" => DATA_OUT_A <= r3;
                when "0100" => DATA_OUT_A <= r4;
                when "0101" => DATA_OUT_A <= r5;
```

```
                when "0110" => DATA_OUT_A <= r6;
                when "0111" => DATA_OUT_A <= r7;
                when "1011" => DATA_OUT_A <= SP;
                when "1100" => DATA_OUT_A <= LR;
                when "1101" => DATA_OUT_A <= PC;
                when "1110" => DATA_OUT_A <= CPSR;
                when "1111" => DATA_OUT_A <= SPSR;
                when others => DATA_OUT_A <= "0000000000000000";
        end case;

        case REG_READ_B is
                when "0000" => DATA_OUT_B <= r0;
                when "0001" => DATA_OUT_B <= r1;
                when "0010" => DATA_OUT_B <= r2;
                when "0011" => DATA_OUT_B <= r3;
                when "0100" => DATA_OUT_B <= r4;
                when "0101" => DATA_OUT_B <= r5;
                when "0110" => DATA_OUT_B <= r6;
                when "0111" => DATA_OUT_B <= r7;
                when "1011" => DATA_OUT_B <= SP;
                when "1100" => DATA_OUT_B <= LR;
                when "1101" => DATA_OUT_B <= PC;
                when "1110" => DATA_OUT_B <= CPSR;
                when "1111" => DATA_OUT_B <= SPSR;
                when others => DATA_OUT_B <= "0000000000000000";
        end case;

        if clk'event and clk='1' then
                if RB_write='1' then
                        case REG_WRITE is
                                when "0000" => r0 <= DATA_IN;
                                when "0001" => r1 <= DATA_IN;
                                when "0010" => r2 <= DATA_IN;
                                when "0011" => r3 <= DATA_IN;
                                when "0100" => r4 <= DATA_IN;
                                when "0101" => r5 <= DATA_IN;
                                when "0110" => r6 <= DATA_IN;
                                when "0111" => r7 <= DATA_IN;
                                when "1011" => SP <= DATA_IN;
                                when "1100" => LR <= DATA_IN;
                                when "1101" => PC <= DATA_IN;
                                when "1110" => CPSR <= DATA_IN;
                                when others => SPSR <= DATA_IN;
                        end case;
                end if;
                if RB_WRITE_AI='1' then
                        PC <= DATA_AI;
                end if;
        end if;

        PC_OUT <= PC;
        CPSR_OUT <= CPSR;
end process;
end RB_arch;
```

BShifter : rotation et masque

```
library IEEE;
use IEEE.std_logic_1164.all;

entity BShifter is
    port (
         ROTATE :in INTEGER range 0 to 15;
         MASK :in STD_LOGIC_VECTOR (15 downto 0);
        INPUT: in STD_LOGIC_VECTOR (15 downto 0);
        OUTPUT: out STD_LOGIC_VECTOR (15 downto 0)
    );
end BShifter;

architecture BShifter_arch of BShifter is
begin
        rotate_loop:
        for i in 0 to 15 generate
                OUTPUT(i)<=INPUT((i+ROTATE)) and MASK(i);
        end generate;
end BShifter_arch;
```

ALU

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.std_logic_arith.all;
use IEEE.std_logic_unsigned.all;

entity ALU is
    port (
         clk: in STD_LOGIC;
        carry: in STD_LOGIC;
        carry_out: out STD_LOGIC;
        input_a: in STD_LOGIC_VECTOR (15 downto 0);
        input_b: in STD_LOGIC_VECTOR (15 downto 0);
        output: out STD_LOGIC_VECTOR (15 downto 0);
        opcode: in STD_LOGIC_VECTOR (3 downto 0)
    );
end ALU;

architecture ALU_arch of ALU is
         signal REGA: STD_LOGIC_VECTOR (15 downto 0);
         signal REGB: STD_LOGIC_VECTOR (15 downto 0);
         signal REGC: STD_LOGIC;

begin
process(REGA,REGB,REGC,clk, carry, input_a, input_b, opcode)
begin
if clk'event and clk='1' then
         REGA <= input_a;
         REGB <= input_b;
         REGC <= carry;
end if;
case opcode is
         when "0000" | "1000"      => output <= REGA and REGB;
         when "0001" | "1001"      => output <= REGA xor REGB;
         when "0010" | "1010"      => output <= REGA - REGB;
         when "0011"               => output <= REGB - REGA;
         when "0100" | "1011"      => output <= REGA + REGB;
         when "0101"               => output <= REGA + REGB;-- + REGC;
         when "0110"               => output <= REGA - REGB;-- + REGC - 1;
         when "0111"               => output <= REGB - REGA;-- + REGC - 1;
         when "1100"               => output <= REGA or REGB;
         when "1101"               => output <= REGB;
         when "1110"               => output <= REGA and not REGB;
         when "1111"               => output <= not REGB;
         when others               => output <= "0000000000000000";
end case;
carry_out <= '0';
end process;
end ALU_arch;
```

Control : Contrôleur processeur

```
library IEEE;
use IEEE.std_logic_1164.all;

entity control is
    port (
         CLK:in STD_LOGIC;
         INSTRUCTION: in STD_LOGIC_VECTOR (15 downto 0);
         CPSR: in STD_LOGIC_VECTOR (15 downto 0);
--Register Bank
         RB_WRITE: out STD_LOGIC;
         RB_WRITE_AI: out STD_LOGIC;
         REG_WRITE: out STD_LOGIC_VECTOR (3 downto 0);       -- imp
         REG_READ_A: out STD_LOGIC_VECTOR (3 downto 0);
         REG_READ_B: out STD_LOGIC_VECTOR (3 downto 0);
--Bus B
         BUSB_SELECT_RB:out STD_LOGIC;
--Adress Register
         SELECT_AR: out STD_LOGIC_vector (1 downto 0);
--ALU
         OPCODE: out STD_LOGIC_VECTOR (3 downto 0);
--WDR
         WDR_EIN: out STD_LOGIC;
         WDR_EOUT: out STD_LOGIC;
--RDR
         RDR_EIN:out STD_LOGIC;
--RAM
         WRITE1: out STD_LOGIC;
         WRITE0: out STD_LOGIC;

--BShifter
         MASK:out STD_LOGIC_VECTOR (15 downto 0);
```

```
        ROTATE: out INTEGER range 0 to 15;

--debug
        DEBUG_ETAPE:out INTEGER range 0 to 31;
        DEBUG_RI: out STD_LOGIC_VECTOR (15 downto 0)
    );

end control;

architecture control_arch of control is
        signal RI: STD_LOGIC_VECTOR (15 downto 0);
        signal ETAPE: INTEGER range 0 to 31;
begin
process(clk,RI,ETAPE)
begin
        if clk'event and clk='1' then
                case ETAPE i
                        when 0 =>       -- envoi de l'adresse de l'instruction
                                RB_WRITE <= '0';
                                WRITE0 <= '0';
                                WRITE1 <= '0';
                                RB_WRITE_AI <= '1';
                                ETAPE<=ETAPE+1;
                                SELECT_AR<="11";
                                RDR_EIN<= '1';

                        when 1 => -- recuperation de l'instruction dans RDR
                                ETAPE<=ETAPE+1;

                        when 2 => -- lecture/execution de l'instruction
                                RI <= INSTRUCTION;
                                RDR_EIN<= '0';

                                -- execution, etape 1
                                if INSTRUCTION(15 downto 11)="00011" then--ADD/SUB
                                        if INSTRUCTION(10)='0' then --REGISTER
                                                REG_READ_B <= '0'&INSTRUCTION(8 downto 6);
                                                BUSB_SELECT_RB <= '1';
                                                MASK <= "1111111111111111";
                                                ROTATE <= 0;
                                        else --immediat
                                                BUSB_SELECT_RB <= '0';
                                                MASK <= "0000000000000111";
                                        ROTATE <= 6;

                                        end if;
                                        REG_READ_A <= '0'&INSTRUCTION(5 downto 3);
                                        REG_WRITE  <= '0'&INSTRUCTION(2 downto 0);

                                        if INSTRUCTION(9)='0' then
                                                OPCODE <= "0100";
                                        else
                                                OPCODE <= "0010";
                                        end if;

                                       RB_WRITE_AI <= '0';
                                       WDR_EIN <= '0';
                                        WDR_EOUT <= '0';

                                elsif INSTRUCTION(15 downto 13)="011" then--Load/store Register
                                        BUSB_SELECT_RB <='0';
                                        RB_WRITE <= '0';
                                        RB_WRITE_AI <= '0';
                                        REG_READ_A <= '0'&INSTRUCTION(5 downto 3);
                                        MASK <= "0000000000011111";
                                        ROTATE <= 6;
                                        OPCODE <= "0100";
                                end if;
                                ETAPE<=ETAPE+1;

                        when 3 =>
                                if RI(15 downto 11)="00011" then --add/sub fin
                                        ETAPE<=0;
                                        RB_WRITE <= '1';


                                elsif RI(15 downto 13)="011" then--Load/store Register
                                        if RI(11)='0' then --Store
                                                BUSB_SELECT_RB <='1';
                                                REG_READ_B <= '0'&RI(2 downto 0);
                                                SELECT_AR<="10";
                                                WDR_EIN <= '1';
                                        end if;
                                        ETAPE<=ETAPE+1;
                                end if;

                        when 4 =>
```

```
                                if RI(15 downto 13)="011" then--Load/store Register
                                        if RI(11)='0' then --Store
                                                SELECT_AR<="00";
                                                WRITE0 <= '1';
                                                WRITE1 <= '1';
                                                WDR_EIN <= '0';
                                        end if;
                                        ETAPE<=ETAPE+1;
                                end if;

                        when 5 =>
                                if RI(15 downto 13)="011" then--Load/store Register
                                        if RI(11)='0' then --Store
                                                WRITE0 <= '0';
                                                WRITE1 <= '0';
                                        end if;
                                        ETAPE<=0;
                                end if;

                        when others =>
                                ETAPE<=0;
                end case;
        end if;
        DEBUG_ETAPE<=ETAPE;
        DEBUG_RI<=RI;
end process;
end control_arch;
```