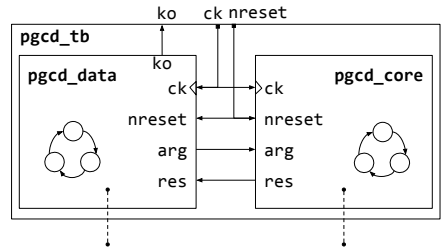
MOCCA

Conception d'un circuit
sous alliance

PGCD

un exemple de ce qu'on peut faire
pour valider un circuit simple

Premier exemple : un PGCD



envoie des nombres en argument et attend le résultat qu'il vérifie et signale les erreurs

reçoit les nombres en argument, calcule le résultat et l'envoie

Le PGCD est un modèle plus simple que CORDIC, mais la méthode de validation est semblable.

Objectif

- Modéliser en vhd les modèles tb, data et core
- Valider les modèles

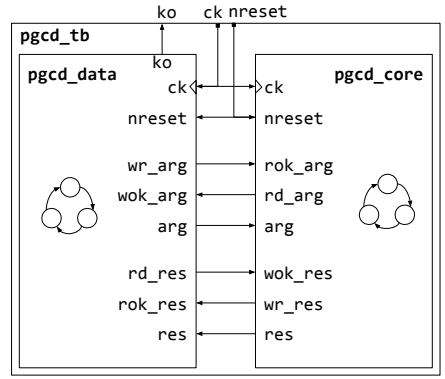
Usage d'Alliance

- `vasy` vhd → vst / vbe
- `genpat` générateur de patterns
- `asimut` simulateur

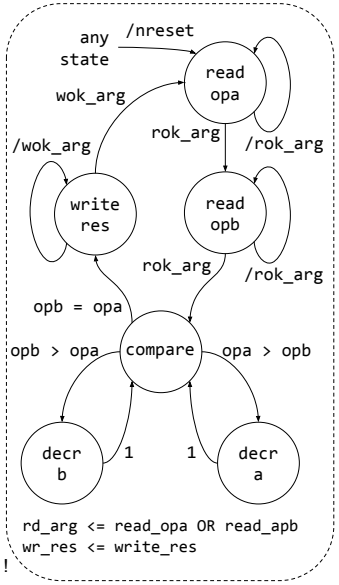
Mais aussi de

- `Makefile`
- `gcc`
- `awk (pour le fun)`

PGCD

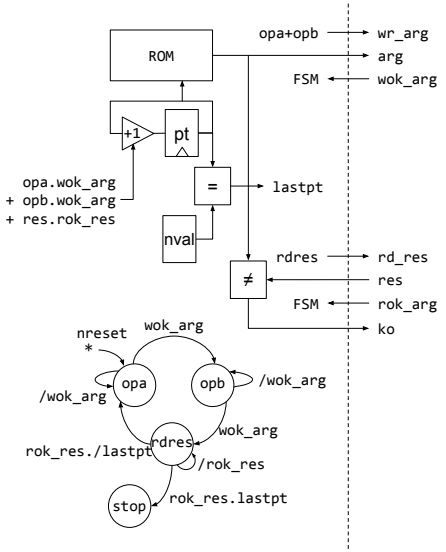



```
unsigned pgcd( unsigned opa,  unsigned opb) {
    while (opa != opb) {
            if (opa > opb) opa -= opb;
        else if (opa < opb) opb -= opa;
    }
    return opa;
}
```



Remarque: write et read sont des ordres, donc toujours des sorties !

pgcd_data en vhdl



- Ce modèle envoie des nombres pris dans une ROM en utilisant le protocole FIFO, puis attend le résultat qu'il compare à ce qu'il a dans sa ROM.
- L'intérêt de cette technique, c'est qu'il n'est pas nécessaire de connaître la durée de calcul, il faut juste connaître le résultat attendu.
- Ici la rom va être produite par un programme en C, puis insérée dans le modèle vhdl de pgcd_data

pgcd_data en vhdl ... *avec du C pour la rom*

interface et signaux internes

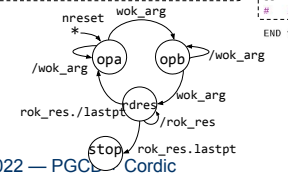
```
ENTITY pgcd_data IS
PORT(
    ck          : IN  std_logic;
    nreset      : IN  std_logic;

    wr_arg_p    : OUT std_logic;
    arg_p       : OUT std_logic_vector(VALWD-1 DOWNTO 0);
    wok_arg_p   : IN  std_logic;

    rd_res_p    : OUT std_logic;
    res_p       : IN  std_logic_vector(VALWD-1 DOWNTO 0);
    rok_res_p   : IN  std_logic;

    ko_p        : OUT std_logic
);
END pgcd_data;

ARCHITECTURE vhd OF pgcd_data IS

    SIGNAL          -- FSM states
        opa,        -- set first operande
        opb,        -- set second operande
        res,        -- get result
        stop,       -- it's over
        lastpt      -- 1 when pt = address of the last filled
    : std_logic;

    SIGNAL
        pt          -- rom_pointer
    : std_logic_vector(ADDRWD-1 downto 0);

    SIGNAL
        value       -- rom_value
    : std_logic_vector(VALWD-1 downto 0);
```

transition one-hot et pointeur de rom

```
BEGIN

    REG : PROCESS (ck) begin
    if ((ck = '1') AND NOT(ck'STABLE)) then
        if (nreset = '0') then
            opa  <= '1';
            opb  <= '0';
            res  <= '0';
            stop <= '0';
            pt   <= (others=>'0');
        else
            opa  <= (res AND rok_res_p AND not lastpt)
                 OR (opa AND not wok_arg_p);
            opb  <= (opa AND wok_arg_p)
                 OR (opb AND not wok_arg_p);
            res  <= (opb AND wok_arg_p)
                 OR (res AND not rok_res_p);
            stop <= (res AND rok_res_p AND lastpt)
                 OR stop;
            if ((opa AND wok_arg_p) OR (opb AND wok_arg_p) OR (res AND rok_res_p)) then
                pt   <= pt + 1;
            end if;
        end if;
    end if;
    end process REG;

    lastpt   <= (pt = LASTPT);
    wr_arg_p <= opa OR opb;
    rd_res_p <= res;
    arg_p    <= value;
    ko_p     <= res AND rok_res_p AND (value /= res_p);

--  #include <rom.txt> incudes a file with a generated ROM, defined as below
--  value       <= x"12"    when pt = 0
--             else x"60"    when pt = 1
--             else x"06"    when pt = 2
--             else x"00";
#   include "rom.txt"

END vhd;
```

ROM

génération Moore & Mealy



il faut utiliser le préprocesseur du C pour insérer le contenu de la ROM

rom.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#ifndef ADDRWDMAX   /* number of bits of rom address*/
#define ADDRWDMAX   16
#endif

void usage(char *message) {
    fprintf( stderr, "\nERROR : %s\n\n", message);
    fprintf( stderr, "USAGE rom <addrwd> <valwd>\n");
    fprintf( stderr, "   <addrwd>: number of bits of the rom address (max is %d)\n", ADDRWDMAX);
    fprintf( stderr, "             the number of triplets [opa,opb,pgcd(opa,opb)]\n");
    fprintf( stderr, "             is then the maximum number possible in 2**addrwd\n");
    fprintf( stderr, "  <valwd> : range of aperandes are between 1 to 2**valwd\n\n");
    fprintf( stderr, "Ex: rom 5 8 -> gives 10 triplets with operands from 1 to 2**8-1 (255)\n\n");
    exit (1);
}

// value returns a number from 1 to range
unsigned value( unsigned valrange) {
    return 1 + (rand()%(valrange-1));
}

// return the pgcd of opa and opb
unsigned pgcd( unsigned opa,  unsigned opb) {
    while (opa != opb) {
            if (opa > opb) opa -= opb;
        else if (opa < opb) opb -= opa;
    }
    return opa;
}

unsigned twopow(unsigned n) {
    unsigned res = 1;
    while (n--) res *= 2;
    return res;
}
```

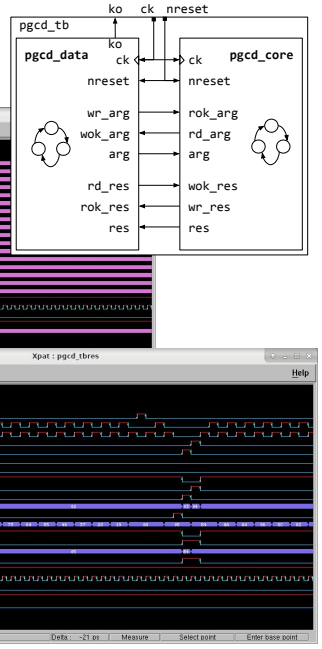
On veut produire ça

```
value <= x"82" when pt = 0
    else x"91" when pt = 1
    else x"05" when pt = 2
    else x"b4" when pt = 3
    else x"0a" when pt = 4
    else x"0a" when pt = 5
    else x"7a" when pt = 6
    else x"aa" when pt = 7
    else x"02" when pt = 8
    else x"6f" when pt = 9
    else x"0d" when pt = 10
    else x"01" when pt = 11
    else x"be" when pt = 12
    else x"a4" when pt = 13
    else x"02" when pt = 14
    else x"c5" when pt = 15
    else x"de" when pt = 16
    else x"01" when pt = 17
    else x"00";
```

```
int main( int argc, char * argv[]) {

    if (argc < 3) usage("Too few arguments");
    if (argc > 3) usage("Too much arguments");
    unsigned addrwd = atoi(argv[1]);
    unsigned valwd  = atoi(argv[2]);
    if (addrwd > ADDRWDMAX) usage("<addrwd> too big (change ADDRWDMAX in source code)");

    unsigned valrange = twopow(valwd)-1;
    unsigned valuenb = twopow(addrwd)/3;

    char *name = "value";
    unsigned namelen = strlen(name);
    unsigned rangelen = 1+(valwd-1)/4;

    for(int i=0; i < valuenb; i++) {
        unsigned opa = value(valrange);
        unsigned opb = value(valrange);
        unsigned res = pgcd(opa, opb);
        if (i==0)
            printf ("%*s <= x\"%0*x\" when pt = %d\n", namelen, name, rangelen, opa, i);
        else
            printf ("%*s x\"%0*x\" when pt = %d\n", namelen+3, "else", rangelen, opa, 3*i);
        printf ("%*s x\"%0*x\" when pt = %d\n", namelen+3, "else", rangelen, opb, 3*i+1);
        printf ("%*s x\"%0*x\" when pt = %d\n", namelen+3, "else", rangelen, res, 3*i+2);
    }
    printf ("%*s x\"%0*x\";\n", namelen+3, "else", rangelen, 0);

    fprintf (stderr, "rom generated with %d triplets (opa, opb, res)\n", valuenb);
    return 0;
}
```


pgcd_pat.c

```
int main ()
{

// since it is not possible to get arguments with genpat,
// we use environment variables. The GETENV() macro try to get
// the variable env value and we can choose a default value for each
#   define GETENV(var,def) getenv(var)?getenv(var):def

    char *    PATNAME= GETENV("PATNAME","default");
    unsigned VALWD  = atoi(GETENV("VALWD","1"));
    unsigned ADDRWD = atoi(GETENV("ADDRWD","1"));
    unsigned CYCLES = atoi(GETENV("CYCLES","100"));

    DEF_GENPAT (PATNAME);

    // interface
    // the 1st port must be "ko" port which is true in case of error
    DECLAR ("ko_p",      ":2", "B", OUT, "", "");

    DECLAR ("vdd",       ":2", "B", IN,  "", "");
    DECLAR ("vss",       ":2", "B", IN,  "", "");
    DECLAR ("ck",        ":2", "B", IN,  "", "");
    DECLAR ("nreset",    ":2", "B", IN,  "", "");

    // It is possible to see internal signals
    DECLAR ("wr_arg",    ":2", "B", SIGNAL, "", "");
    DECLAR ("arg",       ":2", "X", SIGNAL, vector(VALWD-1,0), "");
    DECLAR ("rd_arg",    ":2", "B", SIGNAL, "", "");
    DECLAR ("rd_res",    ":2", "B", SIGNAL, "", "");
    DECLAR ("res",       ":2", "X", SIGNAL, vector(VALWD-1,0), "");
    DECLAR ("wr_res",    ":2", "B", SIGNAL, "", "");

    DECLAR ("data.pt",   ":1", "x", REGISTER, vector(ADDRWD-1,0), "");
    DECLAR ("data.opa",  ":1", "B", REGISTER, "", "");
    DECLAR ("data.opb",  ":1", "B", REGISTER, "", "");
    DECLAR ("data.res",  ":1", "B", REGISTER, "", "");
    DECLAR ("data.stop", ":1", "B", REGISTER, "", "");
    DECLAR ("data.lastpt", ":1", "B", SIGNAL, "", "");

    DECLAR ("core.read_opa", ":1", "B", REGISTER, "", "");
    DECLAR ("core.read_opb", ":1", "B", REGISTER, "", "");
    DECLAR ("core.compare",  ":1", "B", REGISTER, "", "");
    DECLAR ("core.decr_a",   ":1", "B", REGISTER, "", "");
    DECLAR ("core.decr_b",   ":1", "B", REGISTER, "", "");

    AFFECT (cycle (0), "vdd", "1");
    AFFECT (cycle (0), "vss", "0");
    AFFECT (cycle (0), "nreset", "0");
    AFFECT (cycle (1), "nreset", "1");

    // la génération du signal d'horloge
    int c;
    for (c = 0; c <= CYCLES; c++) {
        AFFECT (cycle (c), "ck", inttostr (0));
        AFFECT (next_cycle (c), "ck", inttostr (1));
    }
    AFFECT (cycle (0), "nreset", "0");

    SAV_GENPAT ();
    return 0;
}
```



Makefile

Makefile contenant le processus de validation du PGCD (ici ensemble de "script shell")

```
CFLAGS  = -Wall -O3 -std=c99 #-DDEBUG
LDFLAGS = -lm

MODEL   = pgcd#     model name to validate
ADDRWD  = 8#        bit width of addresses for the rom
VALWD   = 8#        bit width of operands for the rom
CYCLES  = 4000#     number of cycles to simulate (should be suffisant for all triplets)

# VALNB must contains the number of triplets [opa,opb,res]
# VALNB depends on the address width ADDRWD, since VALNB = (2**ADDRWD)/3
# LASTPT is the pointer (pgcd_data.pt) for the last res in rom (pgcd_data.pt is from 0 to 3*VALNB-1)
# the lines below show how to compute an expresssion in the makefile with awk command
VALNB   = $(shell awk -v ADDRWD=$(ADDRWD) 'BEGIN{print int((2**ADDRWD)/3) }')
LASTPT  = $(shell awk -v VALNB=$(VALNB) 'BEGIN{print 3*VALNB-1}')

valid_pgcd:
    $(CC) $(CFLAGS) rom.c -o rom
    ./rom   $(ADDRWD) $(VALWD) > rom.txt
    export PATNAME=$(MODEL)_tb ADDRWD=$(ADDRWD) VALWD=$(VALWD) CYCLES=$(CYCLES);\
    genpat $(MODEL)_pat
    gcc -w -E -DADDRWD=$(ADDRWD) -DVALWD=$(VALWD) -DLASTPT=$(LASTPT) $(MODEL)_data.vhd.c\
    | grep -v "^#" > $(MODEL)_data.vhd
    vasy -a -I vhd -p -o $(MODEL)_data $(MODEL)_data
    vasy -a -I vhd -p -o $(MODEL)_core $(MODEL)_core
    vasy -a -I vhd -p -o $(MODEL)_tb   $(MODEL)_tb
    asimut  $(MODEL)_tb $(MODEL)_tb $(MODEL)_tbres |\
    awk '/pattern/{printf("->"$$3" "$$4"\r")}END{print}'
    @grep ": ?1" $(MODEL)_tbres.pat || echo "Lucky no error"

clean:
    rm  Makefile.*\
        $(MODEL)_core.vbe\
        $(MODEL)_data.vbe\
        $(MODEL)_data.vhd\
        $(MODEL)_tb.vst\
        $(MODEL)_tb.pat\
        $(MODEL)_tbres.pat\
        default.pat\
        rom rom.txt\
        2> /dev/null || true
```

awk, n'est pas indispensable….

Si la séquence des opérations à réaliser est courte alors la description détaillée des dépendances de fichiers est inutile, cela alourdi la description du processus de construction et c'est une source d'erreurs

Cordic

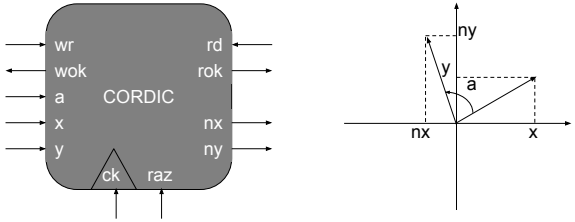
Objectif

Conception d'un circuit ASIC* avec Alliance

- Passer d'un algorithme à un modèle RTL (modélisation et synthèse logique)
- Passer d'un modèle RTL au dessin des masques (synthèse physique)

ASIC utilisant l'algorithme CORDIC

- Calcul de la rotation d'un vecteur (x,y,a) → (nx,ny)
- L'algorithme est « simple », c'est une boucle for en C
- Il permet plusieurs modèles RTL en fonction des contraintes de réalisation

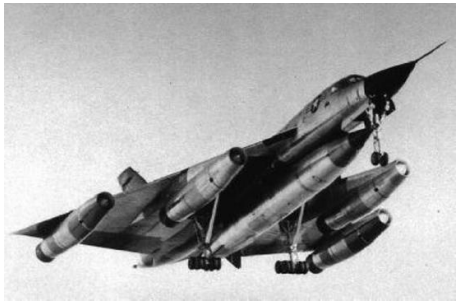


 * ASIC (Application Specific Integrated Circuit) ≠ processeur

Algorithme CORDIC

CORDIC signifie COordinate Rotation DIgital Computer

- Algorithme conçu en 1956 par Jack Volder
- Remplacement du calculateur analogique (composant électromécanique) de navigation des B-58.



source : http://aviation-militaire.kazeo.com/convair-b-58-hustler-a121956030

Algorithme CORDIC

source Wikipédia

Usages de CORDIC

- CORDIC permet au départ de calculer les fonctions trigonométriques en utilisant seulement les opérateurs d'addition, de soustraction et de décalage.
- J. Walther a généralisé l'algorithme (Unified CORDIC) pour calculer aussi les fonctions hyperboliques, exponentielles, logarithmiques, de multiplication, de division et de racine carré.
- CORDIC a permis la conception de la 1ère calculatrice scientifique HP-35 en 1972 et il était présent dans les 1ers coprocesseurs de calcul jusqu'au 80487.

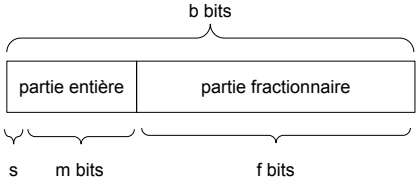
https://www.wikiwand.com/fr/HP-35

- CORDIC demande peu de matériel, peu d'énergie et il est très rapide.

Nombre à virgule fixe

Les nombres sont codés en virgule fixe

- Une partie entière et une partie fractionnaire de taille fixe
- Les nombres sont signés en complément à 2^n ⇒ -a = (2^n - a) ou -a = /a + 1

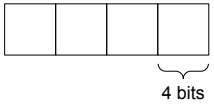


Exemple de codage sur 8 bits : 1-3-4
1 bit de signe
3 bits de partie entière
4 bits de partie fractionnaire

Quelques nombres :

```
1.0  ⇒ 0-001-0000
1.5  ⇒ 0-001-1000
-1   ⇒ 1-111-0000
```

- Dans les calculatrices, les nombres étaient codés en BCD (Décimal Codé Binaire) : chaque chiffre de 0 à 9 est codé sur 4 bits parce que ça simplifie l'affichage



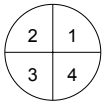
Quelques nombres (entiers naturels)

```
10  ⇒ 0000-0000-0001-0000
 8  ⇒ 0000-0000-0000-1000
```

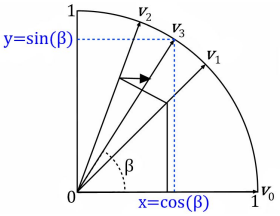

Principes de recherche par dichotomie

Objectif : Calculer le sin(ß) et cos(ß) du vecteur d'angle ß quelconque

Principe



- Supposons que l'angle ß soit entre -90° et +90° (1e et 4e quadrant)
- On utilise une approche "*dichotomique*" (division par 2 de l'angle): Rotations successives du vecteur V_0 d'angle 0° par des angles $\pm ß_i$ de plus en plus petits tels que $-45° <= ß_i <= 45°$ pour approcher le vecteur d'angle ß



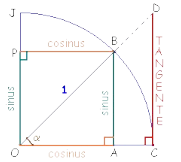
- Chaque rotation élémentaire approche le bon résultat $ß = 0° + \sum \pm ß_i$ avec i compris entre 0 et n
- On connait le cos(0°) et on suppose être capable de calculer $cos(ß) \approx cos(0° + \sum \pm ß_i)$ (resp. pour sin(ß))
- L'astuce c'est de bien choisir les $ß_i$

Rotation élémentaire : choix des $ß_i$

Rotation élémentaire

- La rotation positive élémentaire d'un vecteur v_i par un angle $ß_i$ rend v_{i+1}

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = \begin{pmatrix} cos(ß_i) & -sin(ß_i) \\ sin(ß_i) & cos(ß_i) \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$



http://www.trigofacile.com/maths/trigo/notions/fonctions/tangente.htm

- En mettant $cos(ß_i)$ en facteur

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = cos(ß_i) \begin{pmatrix} 1 & -tan(ß_i) \\ tan(ß_i) & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

- Puisque $-45° < ß_i < 45°$ alors $-1 <= tan(ß_i) <= 1$
- On choisi $ß_i = \sigma_i arctan(2^{-i})$ ⇒ $\sigma_i tan(ß_i) = \sigma_i tan(arctan(2^{-i})) = \sigma_i 2^{-i}$ avec $\sigma_i = \pm 1$ en fonction du sens de la rotation

$$\begin{pmatrix} x_{i+1} \\ y_{i+1} \end{pmatrix} = cos(arctan(2^{-i})) \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} x_i \\ y_i \end{pmatrix}$$

⇒ Soit $K_i = cos(arctan(2^{-i}))$ et sachant que $a * 2^{-i} = a \gg i$ (≫ décalage droite)
alors on obtient : $x_{i+1} = K_i*(x_i - \sigma_i*y_i*2^{-i})$ $\quad x_{i+1} = K_i*(x_i - \sigma_i*y_i \gg i)$
$y_{i+1} = K_i*(y_i + \sigma_i*x_i*2^{-i})$ $\quad y_{i+1} = K_i*(y_i + \sigma_i*x_i \gg i)$

Algorithme complet

- On effectue **n+1** rotations du vecteur (1,0) *(**n** est le nombre de bits après la virgule)*
  soit $K = \prod_{i=0}^{n} K_i = \prod_{i=0}^{n} cos(arctan(2^{-i}))$ *(si **n** est connu alors K est une constante)*
  $$\begin{pmatrix} x_n \\ y_n \end{pmatrix} = K \prod_{i=0}^{n} \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$
- On connaît les angles ß$_i$ que l'on peut mettre dans une table
  - *ß$_0$ = arctan (2^0) = 0,7853 rad = 45°*
  - *ß$_1$ = arctan (2^{-1}) = 0,4636 rad = 26,57°*
  - *ß$_2$ = arctan (2^{-2}) = 0,2449 rad = 14,04°*
  - …
  - *ß$_7$ = arctan (2^{-7}) = 0,0078 rad = 0,45°*

  on voit que les angles décroissent, c'est presque dichotomique (div 2))
- En résumé, l'algorithme consiste à : *(si l'angle de rotation ß est entre -90° et 90°)*
  - Faire **n** rotations de σ_iß$_i$ avec $\sigma_i = \pm 1$ (sans multiplier par K$_i$)
    une rotation est une boucle d'itération : $x_{i+1} = x_i - \sigma_i * y_i \gg i$; $y_{i+1} = y_i + \sigma_i * x_i \gg i$
  - Puis à la fin, à multiplier par la constante K (le produit des K$_i$)
- La précision du résultat dépend du nombre d'itérations (au plus **n+1**)
  ⇒ à chaque itération, on ajoute un nouveau chiffre après la virgule !

Principes (rotation)

Prologue et épilogue de l'algorithme

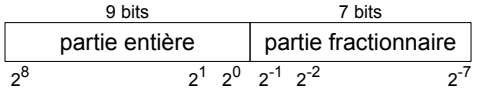
- Si l'angle de départ est dans les 2^e, 3^e ou 4^e quadrants (> 90°)
  alors il faut faire des rotations de ±90° pour amener l'angle dans le 1^e quadrants,
  il faut se souvenir de la rotation pour retrouver le signe du résultat

  En effet, on sait : *cos(a+b) = cos(a)cos(b) - sin(a)sin(b)*
  donc on a : *cos(ß+90°) = -sin(ß)*
  on montre aussi que : *sin(ß+90°) = cos(ß)*
- On part donc du vecteur (1,0) et on fait à la fin des itérations, on obtient un vecteur (x, y) qu'il faut multiplier par la constante K : (cos(x), sin(y)) = K ∗ (x, y)
- **On ne veut pas utiliser un multiplieur, on utilise des additions et des décalages**
  soit : $K = a_n 2^n + a_{n-1} 2^{n-1} + \ldots + a_0 2^0$ où $a_i \in \{0, 1\}$
  $$xr = cos(x) = xK = xa_n 2^n + xa_{n-1} 2^{n-1} + \ldots + xa_0 2^0$$
  $$xr = a_n(x \ll n) + a_{n-1}(x \ll (n-1)) + \ldots + a_0 x$$
  ⇒ Si K a peu de bits à 1 (c.-à-d. peu de a_n à 1), il y a peu de termes à sommer

Choix d'un codage en virgule fixe

Codage des nombres pour le circuit réalisé

- Les nombres sont en complément à 2^n
- Nous allons utiliser un codage sur 16 bits : 1-8-7



Opérations

- La conversion est réalisée par de simples décalages
  - Soit E un nombre entier sur 8 bits en complément à 2: E ∈ [-128, 127]
  - Soit F un nombre avec 7 bits après la virgule sur 16 bits :
    On a 9 bits pour la partie entière mais on choisit de limiter F ∈ [-128, 128[
    - -128 = `0b1.1000.0000.0000.000`
    - +128 ≈ `0b0.0111.1111.1111.111` = 127,99609375
  - en 1-8-7 l'intervalle théorique est plus grand mais c'est pour éviter les dépassements de capacité lors des calculs…
  - Les conversions E ←→ F se font par des décalages
    F = E << 7 et E = F >> 7 (dans ce dernier cas, on perd la partie fractionnaire)
- Les opérations arithmétiques +/- restent inchangées

Multiplication par K avec le codage 1-8-7

Rappel de l'algo : pour faire une rotation ß d'un vecteur (a,b)
- on fait les 8 rotations élémentaires ß$_i$ pour obtenir le vecteur intermédiaire (x,y)
- puis, on multiplie par K pour obtenir, le vecteur correct (rx, ry)

$$\begin{pmatrix} x \\ y \end{pmatrix} = \prod_{i=0}^{7} \begin{pmatrix} 1 & -\sigma_i 2^{-i} \\ \sigma_i 2^{-i} & 1 \end{pmatrix} \begin{pmatrix} a \\ b \end{pmatrix} \qquad \begin{pmatrix} rx \\ ry \end{pmatrix} = K \begin{pmatrix} x \\ y \end{pmatrix}$$

On sait que :
si on multiplie des nombres à virgule, on fait comme s'il n'y avait pas de virgule, puis on place la virgule sur le résultat en sautant autant de chiffres qu'il y en avait dans les opérandes : aa,aa * bb,bb = cccc,cccc

Ici, les nombres sont codés en 1-8-7
⇒ on va multiplier des nombres de 16 bits et faire un décalage à droite de 14 bits

$$K_{real} = \prod_{i=0}^{7} K_i = \prod_{i=0}^{7} cos(arctan(2^{-i})) = 0.607259$$
$$K_{fixed} = 0.607259 \ll 7 = 78 = 0x4E = 0000000001001110$$
$$rx = ((x \ll 6) + (x \ll 3) + (x \ll 2) + (x \ll 1)) \gg 14$$

Pour ne pas avoir à coder inutilement ***rx*** sur 32 bits, on peut anticiper les décalages

$$rx = (((x \gg 7) \ll 6) + ((x \gg 7) \ll 3) + ((x \gg 7) \ll 2) + ((x \gg 7) \ll 1)) \gg 7$$
$$rx = ((x \gg 1) + (x \gg 4) + (x \gg 5) + (x \gg 6)) \gg 7$$

Exemple du calcul de cosinus avec CORDIC

Calcul de *cos(78°) = cos(1,3613 rad) = 0.2079*

- Ici, on va faire seulement 4 itérations et utiliser la base 10.
- On va donc faire 4 rotations d'angle ß$_i$ (i de 0 à 3) du vecteur (1,0)
  - *ß$_0$ = arctan (2^0) = 0,7853 rad = 45°*
  - *ß$_1$ = arctan (2^{-1}) = 0,4636 rad = 26,57°*
  - *ß$_2$ = arctan (2^{-2}) = 0,2449 rad = 14,04°*
  - *ß$_3$ = arctan (2^{-3}) = 0,1243 rad = 7.125°*
- *78 → 45 + 26,57 + 14,04 - 7,125 = 78.48°*
- *K_0= cos(arctan(2^0)) = 0,7071 ; K_1= 0,8944 ; K_2= 0,9701 ; K_3= 0,9922 ;*
- Etapes *(ici on multiplie par K_i à chaque étape pour plus de clarté)*

$$\begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = K_0 \begin{pmatrix} 1 & -2^0 \\ 2^0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 0 \end{pmatrix} = 0.7071 \begin{pmatrix} 1 \\ 1 \end{pmatrix} = \begin{pmatrix} 0.7071 \\ 0.7071 \end{pmatrix}$$

$$\begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = K_1 \begin{pmatrix} 1 & -2^{-1} \\ 2^{-1} & 1 \end{pmatrix} \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} = 0.8944 \begin{pmatrix} 0,7071 - 0,7071/2 \\ 0.7071/2 + 0,7071 \end{pmatrix} = 0,8944 \begin{pmatrix} 0,3535 \\ 1.060 \end{pmatrix} = \begin{pmatrix} 0.3162 \\ 0,9486 \end{pmatrix}$$

$$\begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = K_2 \begin{pmatrix} 1 & -2^{-2} \\ 2^{-2} & 1 \end{pmatrix} \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = 0.9701 \begin{pmatrix} 0.3162 - 0.9486/4 \\ 0.3162/4 + 0.9486 \end{pmatrix} = 0.9701 \begin{pmatrix} 0.079 \\ 1.028 \end{pmatrix} = \begin{pmatrix} 0.077 \\ 0,9974 \end{pmatrix}$$

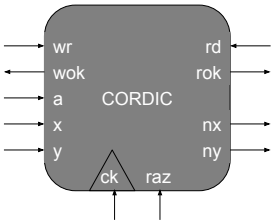
$$\begin{pmatrix} x_4 \\ y_4 \end{pmatrix} = K_3 \begin{pmatrix} 1 & 2^{-3} \\ -2^{-3} & 1 \end{pmatrix} \begin{pmatrix} x_3 \\ y_3 \end{pmatrix} = 0.9922 \begin{pmatrix} 0,077 + 0.9974/8 \\ -0,077/8 + 0.9974 \end{pmatrix} = 0.9922 \begin{pmatrix} 0.2016 \\ 0,9877 \end{pmatrix} = \begin{pmatrix} 0.2000 \\ 0,9800 \end{pmatrix}$$

- cos(78°) = 0,2079 ≈ 0,200 (3,8% d'erreur)
  notez que l'on calcule aussi sin(78°) avec moins de 1% d'erreur (sin(78°)=0.9781)

Circuit CORDIC

Objectif

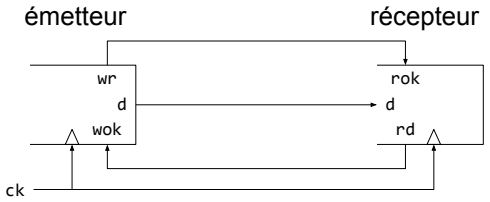
modéliser et valider un circuit CORDIC calculant les coordonnées **(nx,ny)** d'un vecteur **(x,y)** après la rotation d'un angle a.



Caractéristiques

- CORDIC présente une interface FIFO en entrée et en sortie.
- Les nombres à l'interface sont des entiers sur 8 bits (pas de virgule)
- La latence et le débit des calculs dépendent de l'architecture interne

Interface de communication FIFO



- La communication est synchronisée sur une horloge ck
- L'émetteur et le récepteur sont indépendants
  chacun dit s'il est prêt à envoyer ou à recevoir une donnée
  - si **wr** est actif l'émetteur a une donnée **d** et informe le récepteur sur **rok**
  - si **rd** est actif le récepteur a une place pour **d** et informe l'émetteur sur **wok**
  - si wr et rd sont actifs au même cycle alors une donnée **d** est transmise
- Notez une chose importante concernant le choix du nom des signaux de contrôle (j'ai mis du temps à comprendre ça…) :
  - **wr** et **rd** sont des **ordres** donc des **sorties**
  - **wok** et **rok** sont des **informations** d'état donc des **entrées**

Modèle comportemental en C

Le comportement du circuit sans horloge

```
void cossin(double a_p, char x_p, char y_p, char *nx_p,char *ny_p)
{
    *nx_p = (char) (x_p * cos(a_p) - y_p * sin(a_p));
    *ny_p = (char) (x_p * sin(a_p) + y_p * cos(a_p));
}
```

Modèle comportemental en C

```
void cordic(short a_p, char x_p, char y_p, char *nx_p, char *ny_p)
{
    unsigned char i, q;
    short a, x, y, dx, dy;

    // conversion en virgule fixe :
    // 7 chiffres après la virgule
    a = a_p;          // angle de départ
    x = x_p << 7;     // coordonnée x initiale
    y = y_p << 7;     // coordonnée y initiale

    // normalisation de l'angle pour être
    // dans le 1° quadrant (q = n° quadrant)
    q = 0;
    while (a >= F_PI/2) {
        a = a - F_PI/2;
        q = (q + 1) & 3;
    }

    // 8 rotations successives
    for (i = 0; i <= 7; i++) {
        dx = x >> i;
        dy = y >> i;
        if (a >= 0) {     // rotation positive
            x -= dy;      // calcul des coordonnées
            y += dx;      // après rot de +ATAN[i]
            a -= ATAN[i]; // nouvel angle de rot.
        } else {          // rotation négative
            x += dy;      // calcul des coordonnées
            y -= dx;      // après rot de -ATAN[i]
            a += ATAN[i]; // nouvel angle de rot.
        }
    }
    // produit du résultat par les cosinus
    // des angles : K=0x4E=01001110
    x = ((x>>6) + (x>>5) + (x>>4) + (x>>1))>>7;
    y = ((y>>6) + (y>>5) + (y>>4) + (y>>1))>>7;

    // placement du point dans le quadrant initial
    switch (q) {
    case 0:
        dx = x;
        dy = y;
        break;
    case 1:
        dx = -y;
        dy = x;
        break;
    case 2:
        dx = -x;
        dy = -y;
        break;
    case 3:
        dx = y;
        dy = -x;
        break;
    }
    *nx_p = dx;
    *ny_p = dy;
}
```

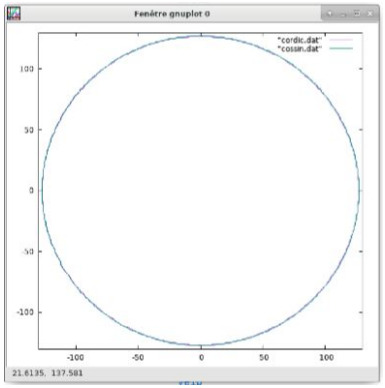
```
// variable globale
short ATAN[8] = {
    0x65, // ATAN(2^-0)
    0x3B, // ATAN(2^-1)
    0x1F, // ATAN(2^-2)
    0x10, // ATAN(2^-3)
    0x08, // ATAN(2^-4)
    0x04, // ATAN(2^-5)
    0x02, // ATAN(2^-6)
    0x01, // ATAN(2^-7)
};
```

cossin() = cordic()

```
void cossin(double a_p, char x_p, char y_p, char *nx_p,char *ny_p)
{
    *nx_p = (char) (x_p * cos(a_p) - y_p * sin(a_p));
    *ny_p = (char) (x_p * sin(a_p) + y_p * cos(a_p));
}
```

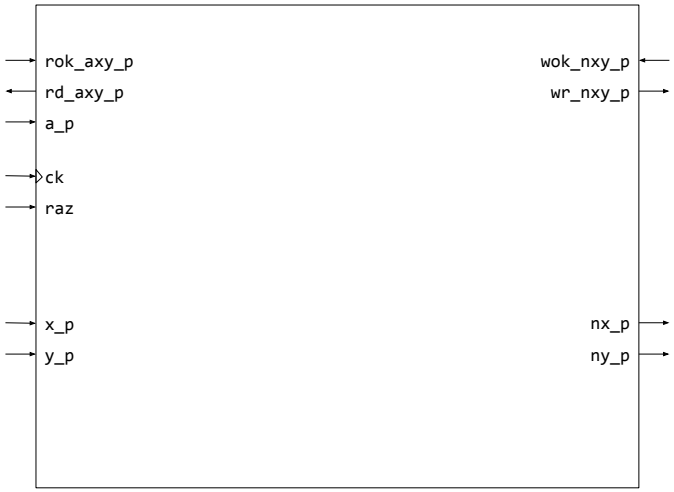

différences entre cossin() et cordic()

```
int main()
{
    FILE *f;
    double t,K;
    int i;

    f = fopen("cossin.dat", "w");
    for (double a = 0; a <= M_PI * 2; a += 1. / 64) {
        char nx_p;
        char ny_p;
        cossin(a, 127, 0, &nx_p, &ny_p);
        fprintf(f, "%4d %4d\n", nx_p, ny_p);
    }
    fclose(f);

    f = fopen("cordic.dat", "w");
    for (short a = 0; a <= 2 * F_PI ; a += 1) {
        char nx_p;
        char ny_p;
        cordic(a, 127, 0, &nx_p, &ny_p);
        fprintf(f, "%4d %4d\n", nx_p, ny_p);
    }
    fclose(f);

    return 0;
}
```

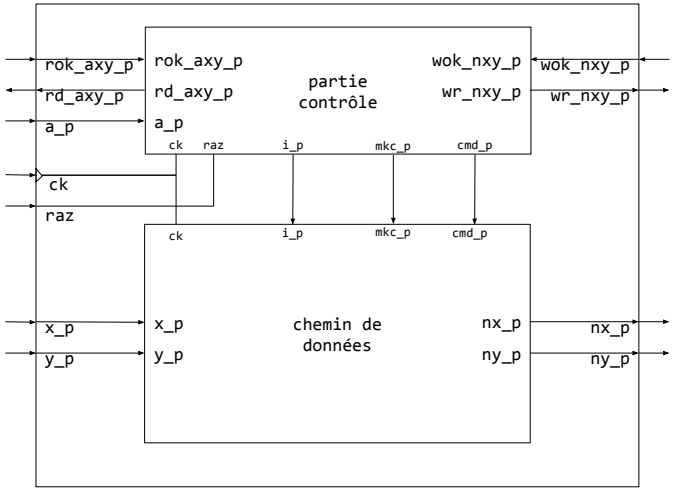



```
plot: cercle
    ./cercle
    gnuplot \
            -e 'plot [-130:130] [-130:130] "$(MODEL).dat" with lines;'\
            -e 'replot "cossin.dat" with lines;' \
            -e 'pause -1'
```

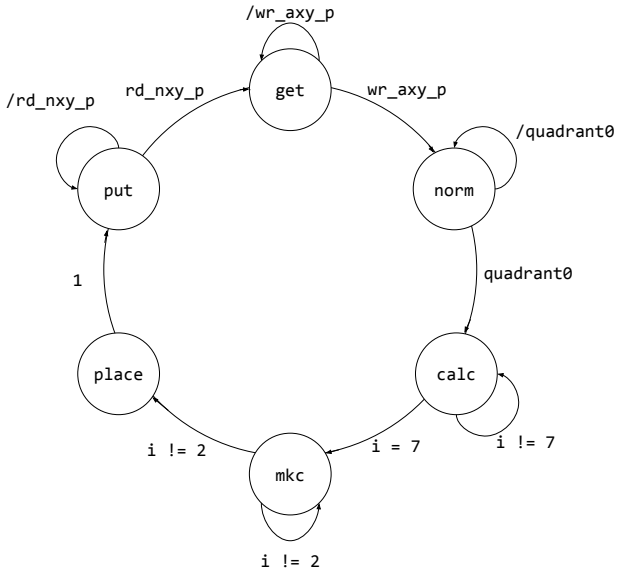

Interface de cordic



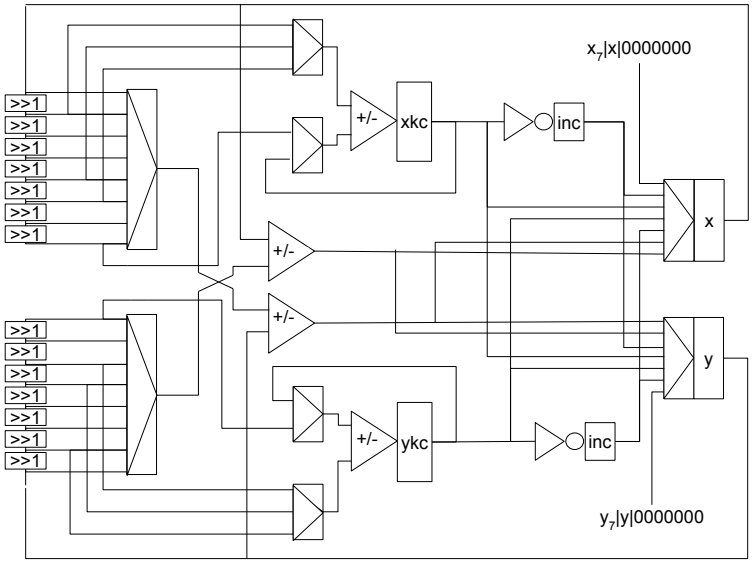
Décomposition en partie contrôle et données



automate



Chemin de données



Travaux Pratiques - PGCD

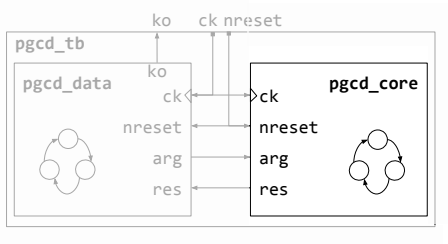
Le but est de réaliser un ASIC complet (peut-être avec les plots)

Vous partez :

- d'un fichier VHDL incomplet

Vous devez pour le moment :

- compléter et valider le modèle VHDL du PGCD
- synthétiser pour obtenir une netlist sur SXLIB



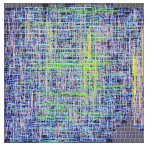
Travaux Pratiques - Cordic

Vous allez travailler sur Cordic, le code qui vous est donné fonctionne, mais vous devez le faire évoluer, Il y a plusieurs possibilités de difficultés croissantes :

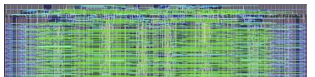
- Créer un test bench comme pour PGCD (vous n'êtes pas obligé de faire beaucoup de tests, car on ne va pas fabriquer le circuit…)
- Réduire le nombre d'entrées-sorties
  - entrer x, y et a séquentiellement
  - sortir nx et ny séquentiellement
- Augmenter le débit en créant un pipeline à deux ou trois étages, p. ex.
  - lecture et normalisation
  - calcul
  - placement + multiplication et écriture

Vous devez synthétiser Cordic sur SXLIB.

Pour le placement-routage, vous pourrez faire évoluer l'architecture et éventuellement ajouter des plots



version sxlib



version avec un chemin de données