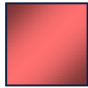
XC886/888CLM Cordic & MDU Module

May 2006

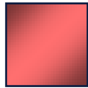
Infineon

Never stop thinking

Agenda

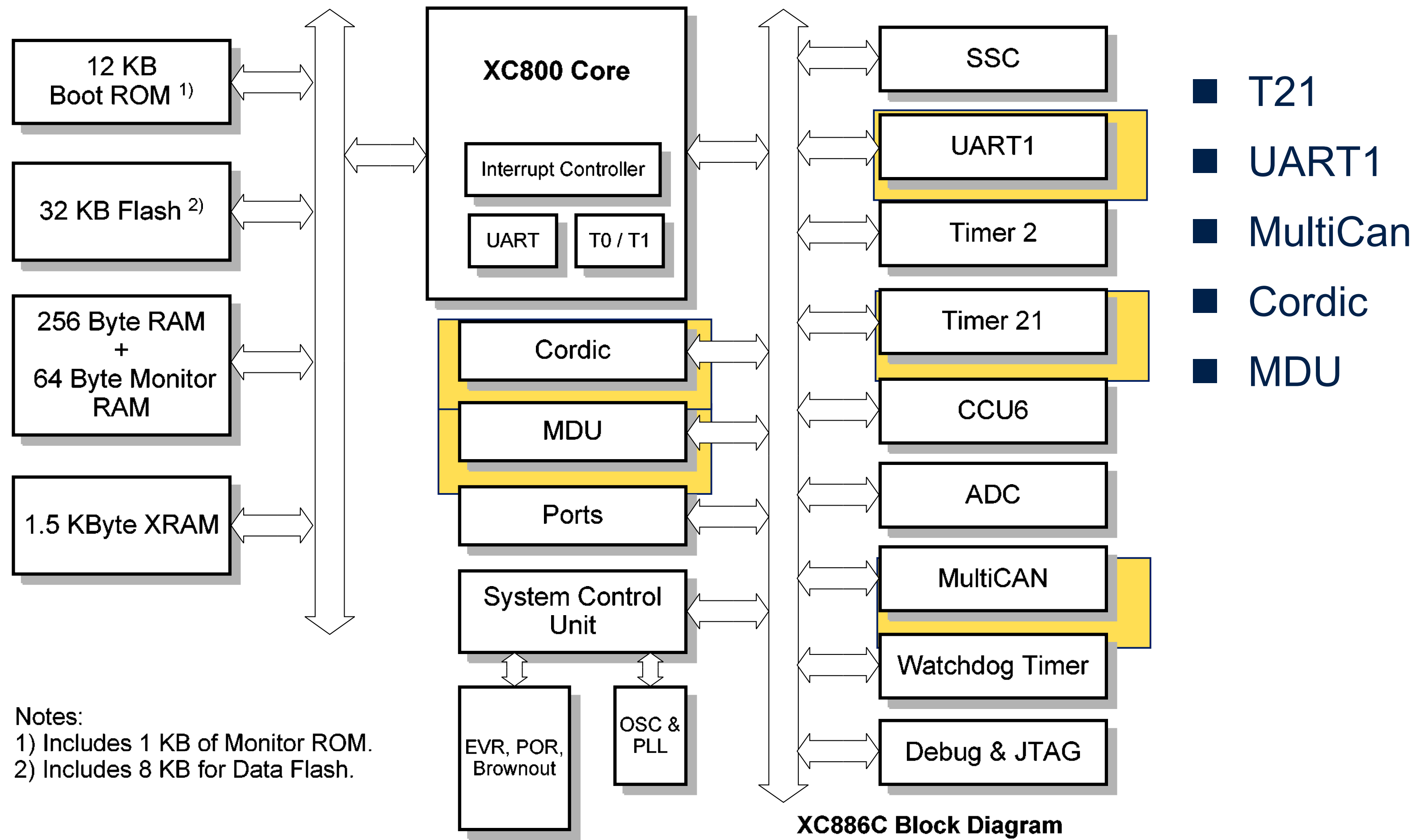


XC888/886CLM – Cordic Coprocessor



XC886/888CLM – MDU Coprocessor

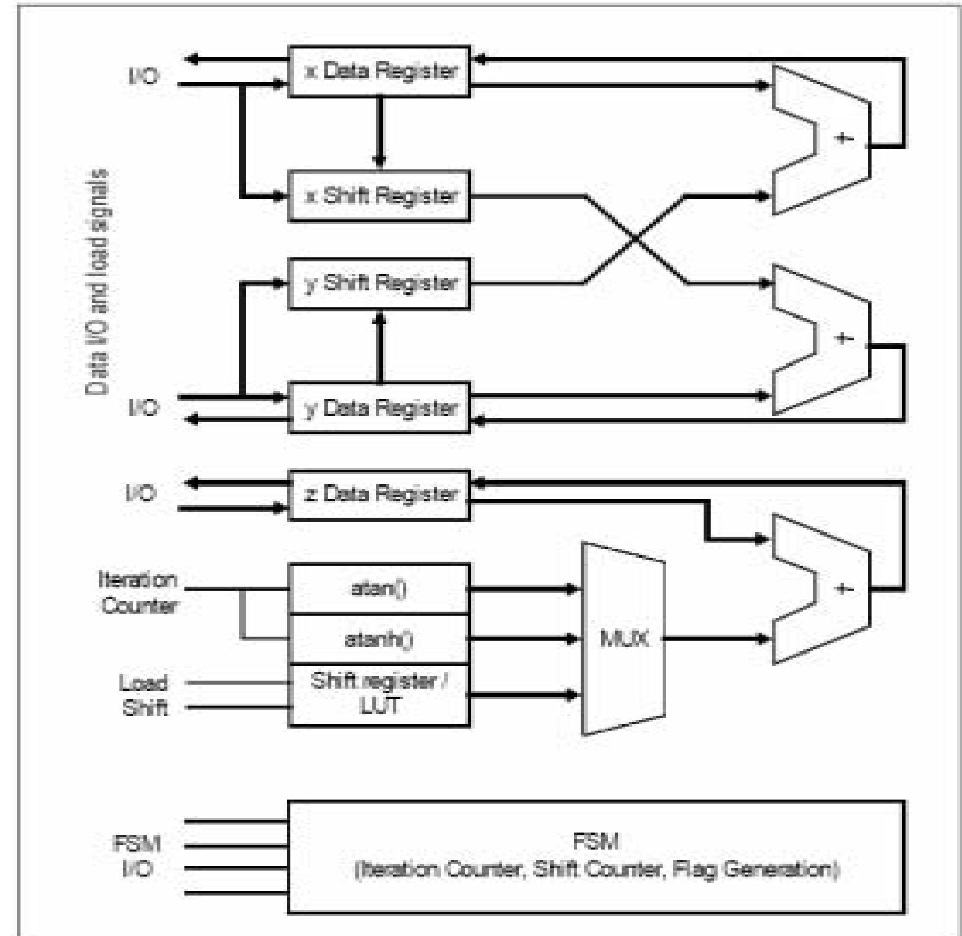
XC886/888CLM – New Peripherals



- T21
- UART1
- MultiCan
- Cordic
- MDU

XC886/888CLM – Cordic Basics

- Cordic (**CO**ordinate **R**otation **DI**gital **C**omputer)
  - Mathematical co-processor for 16bit trigonometric, hyperbolic and linear functions (e.g. to solve SIN, COS, LOG, EXP, SQRT...)
  - Hardcoded - Look Up Table based on iterative approximation algorithm (16 iterations, max 41 cycles)
- 8 SFRs in the mapped area
- one interrupt vector on a shared node @0x0043



Infineon

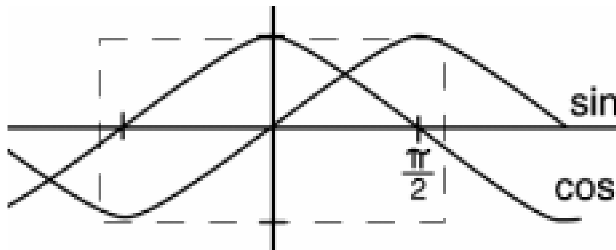
XC886/888CLM – Cordic History

- CORDIC algorithms belong to the class of shift-add algorithms. (introduced by Jack E. Volder in 1951 for trigonometric functions, generalized and extended with linear and hyperbolic functions by J.S. Walther 1971)
- The idea behind the CORDIC is, to have all mathematic functions in a microcontroller available with most less costs, since shift and add hardware can be easy implemented.
- Mainly the CORDIC is used to perform a rotation in a plane using a series of specific incremental rotation angles selected so that each is performed by a shift and add operation. There is also a mechanism for computing the magnitude and phase angle of an input vector.
- One CORDIC iteration involves 2 shifts, 1 table lookup, 3 additions

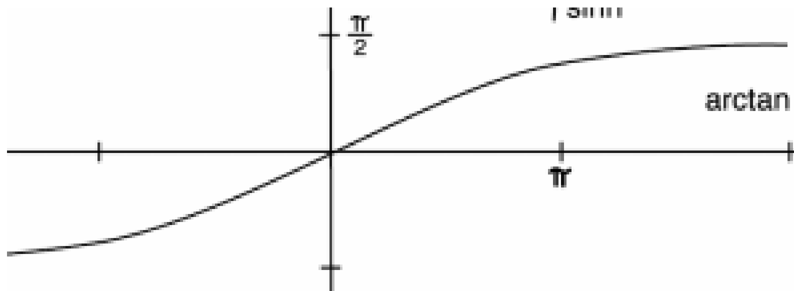
Never stop thinking

XC886/888CLM – Some Math Basics

- Some basic functions
  - sine (x), cosine (x)



  - arctan (x)



- Taylor series expansion

$$\sin x = x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} \pm \ldots;$$

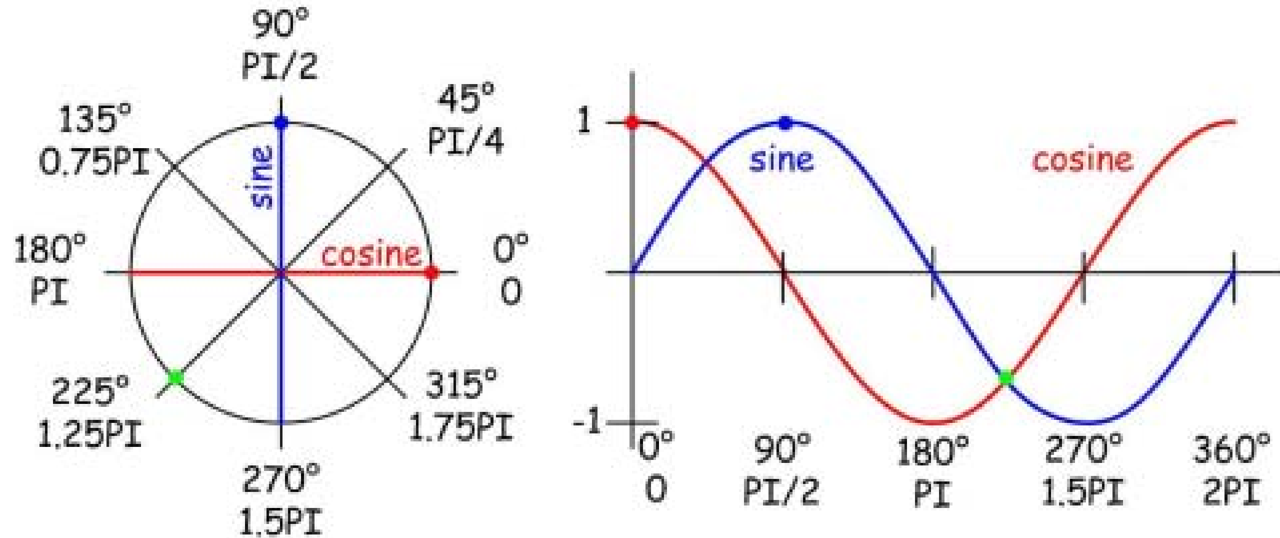
$$\arctan x = x - \frac{x^3}{3} + \frac{x^5}{5} - \frac{x^7}{7} \pm \ldots$$

- Factorial
  - e.g. 5! = 5*4*3*2*1 = 120

Infineon

XC886/888CLM – Some Math Basics cont.

- The Unite Circle



- Radian
  - deg to rad $$\text{angle}\,[\text{rad}] = \text{angle}\,[\text{deg}] \bullet \frac{\pi}{180}$$
    - e.g. 45 (deg) → 45/180*PI = 0.7854 (rad)
    - e.g. 1 (rad) = PI → 180 (deg)

Never stop thinking

XC886/888CLM – Cordic Coordinate Systems

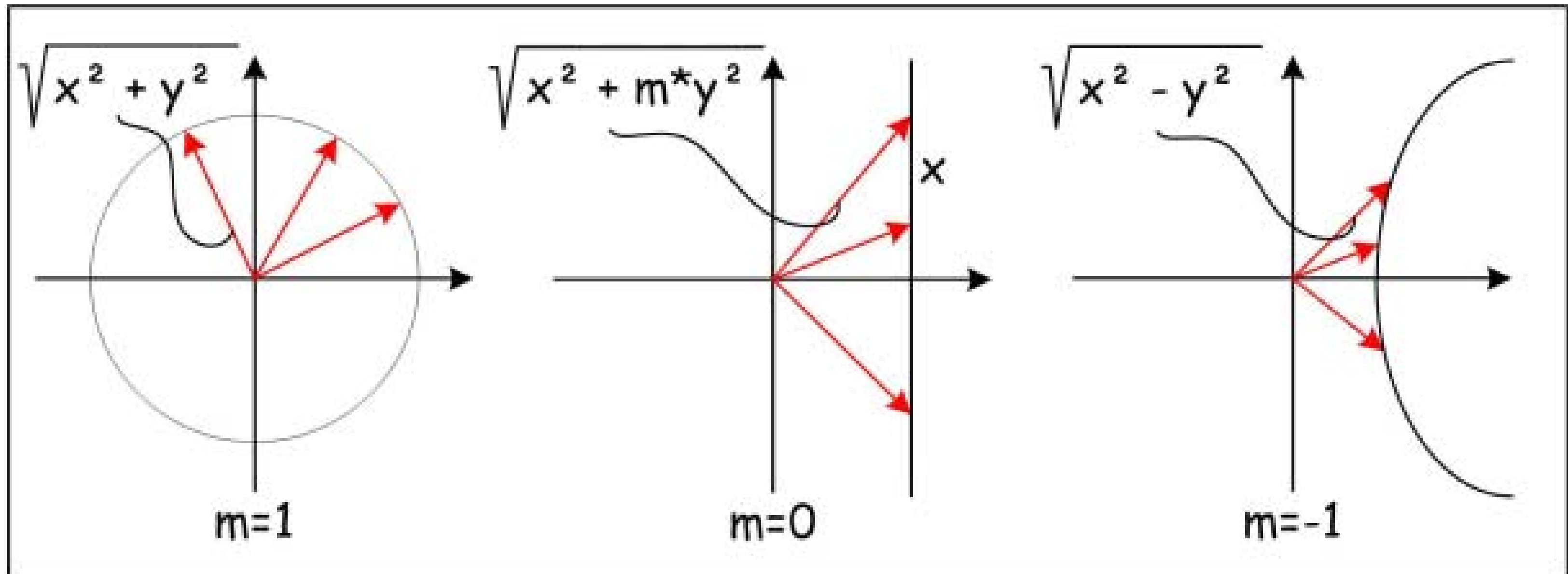
- CORDIC equations

$$x^{(i+1)} = x^{(i)} - m\, d_i\, y^{(i)}\, 2^{-i}$$

$$y^{(i+1)} = y^{(i)} + d_i\, x^{(i)}\, 2^{-i}$$

$$z^{(i+1)} = z^{(i)} - d_i\, e^{(i)}$$

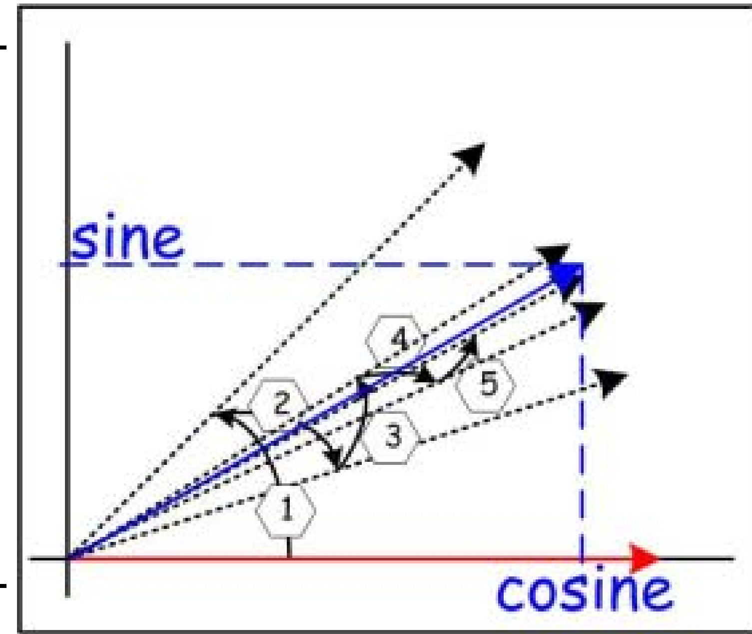
m	Coordinate System	Domain of Convergence
1	circular	1.74 radian
0	linear	2
-1	hyperbolic	1.11 radian



XC886/888CLM – Cordic Iterations

- Starting with a phase of 45 degrees, the phase of each successive R multiplier is a little over half of the phase of the previous
- “Binary search" on phase by adding or subtracting successively smaller phases to reach the "target" phase

i	*e*(*i*) in degrees	*e*(*i*) in radians
0	45.0	0.785 398 163
1	26.6	0.463 647 609
2	14.0	0.244 978 663
3	7.1	0.124 354 994
4	3.6	0.062 418 810
5	1.8	0.031 239 833
6	0.9	0.015 623 728
7	0.4	0.007 812 341
8	0.2	0.003 906 230
9	0.1	0.001 953 123



- **Example: 30° angle**

30.0 ≅ 45.0 – 26.6 + 14.0 – 7.1 + 3.6 + 1.8 – 0.9 + 0.4 – 0.2 + 0.1

= 30.1

XC886/888CLM – Cordic Math Functions

CORDIC		Rotation	Vectoring
Linear MUL/DIV/MAC K = 1	IN	x_n, z_n, y_n	x_n, y_n, z_n
	OUT	$y_{n+1} = x_n * z_n + y_n$	$z_{n+1} = y_n / x_n + z_n$
Circular SINE/COSINE angle/ magnitude K = 1.64676	IN	z_n=angle, $x_n=K, y_n=0$	$x_n, y_n, z_n=0$
	OUT	x_{n+1} = cos(angle) y_{n+1} = sin(angle)	$z_{n+1} = \arctan(y_n/x_n)$ $x_{n+1} = 1/K*(x_n^2 + y_n^2)^{1/2}$
Hyperbolic K = 0.828	IN	z_1=angle, $x_1=K', y_1=0$	$y_1 < x_1, z_1=0$
	OUT	x_{n+1} = cosh(angle) y_{n+1} = sinh(angle)	$z_{n+1} = \text{arctanh}(y_1/x_1)$ $x_{n+1} = 1/K'*(x_1^2 - y_1^2)^{1/2}$

XC886/888CLM – Cordic Data Format

- S15
  - S15 or e.g. S5.10 or S10.5.
  - signed integer
    - one sign bit | 15 number bits
  - rational number (fractional)
    - one sign | n . (15-n) bit fractional
    - decimal point can be freely chosen
    - fraction means S | $2^{1} + 2^{0} . 2^{-1} + 2^{-2} + 2^{-3} + \ldots$
  - negative numbers in Twos Complement
    - e.g. -25dec = 1|111 1111 1110 0110 +1
      = 1|111 1111 1110 0111

```
Example S15
25       x      -25 =    -625
0x0019 x 0xFFE7 = 0xFD8F

2.5      x     -2.5 =   -6.25
0x0028 x 0xFFD8 = 0xF9C0

0.25     x    -0.25= 0.0625
```

- 4Q16 (S4.11)
  - decimal point is fixed

```
Example S4.11
                  S|     4.11
 2.5          = 0|001'0.100'0000'0000       = 0x1400
(-5.625       = 0|010'1.101'0000'0000       = 0x2D00
twos compl)= 1|101'0.010'1111'1111 + 1 = 0xD300
```

XC886/888CLM – Cordic Data Format cont.

- Input:
  - X, Y can be integer (S15) *or* rational number (fraction) in all operating modes. They *must have* the *same form* and in case of fraction the *same number of bits for the decimal place.*
  - Z for linear functions it is always in signed 4Q16 (S4.11)
  - Z for circular and hyperbolic functions it is always handled as scaled normalized integer.

```
input Z = real Z (in rad) * 32768/PI
```

- Results:
  - X, Y can be integer (S15) *or* rational number (fraction) in all operating modes. Must be same format as input format.
  - X in circular vectoring mode can be interpreted
    - as twos complement *or*
    - as 16-bit unsigned depending on bit X_USIGN

```
X, Y Result Data = CD_Result / MPS
MPS = magnitude prescaler (1, 2, 4)
```

  - Z for linear functions is signed 4Q16 format (S4.11)
  - Z for circular and hyperbolic functions is a scaled normalized integer

```
real result Z (in rad) = Z* PI/32768
```

XC886/888CLM – Cordic Accuracy & Deviation

- Input inaccuracy: representation of a value is limited by the number of bits
- The CORDICs accuracy is also limited by the number of iterations. Therefore an absolute deviation can be calculated. This is the difference between the expected and the real value.
- The normalized deviation refers to the magnitude of the result, i.e. the absolute error for small numbers is big, but if this is put into relationship to the magnitude it is still small.
- Properly scaled values will produce small errors. The max. error can be calculated and is depending on the chosen mode.
- Some examples for Linear and Circular mode

Infineon

XC886/888CLM – Cordic Accuracy & Deviation cont

- Some example calculation in LINEAR mode
  - y = z * x
    - |z| ≤ 2 → z in S1.11
    - with x = 3000, 6000, ...30000 in S15
    - and z = 1, 0.1, 0.01, 0.001 in S4.11

```
// result magnitude in S15 = 32768
//  calculate 0*0.01, 3000*0.01, 6000*0.01 ... 30000*0.01

for (x=0;x<32000;x+=3000){ //(x, z=0.01 S4Q16)
  product =     CD_multiply  (x, 0x0014); }
```

  - here: the deviation depends on how accurate the value z can be coded in S4.11 (Input inaccuracy)
    - → max. error: $\mathbf{2^{-11}}$ = 0,000488281

Z		1	0.1	0.01	0.001
Z_{coded}	hex	0x0800	0x00CD	0x0014	0x0002
	dec	1.000000	0.100098	0.009766	0.000977

Never stop thinking

XC886/888CLM – Cordic Accuracy & Deviation cont

X / Z	3000	6000	9000	12000	15000	18000	21000	24000	27000	30000
1 0x0800	3000	6000	9000	12000	15000	18000	21000	24000	27000	30000
0.1 0x00CD	300	600	901	1201	1501	1802	2102	2403	2701	3003
0.01 0x0014	29	58	88	117	146	176	205	235	264	293
0.001 0x0002	3	6	9	12	15	18	21	24	27	30

– Input format of X: integer (S15) or fraction

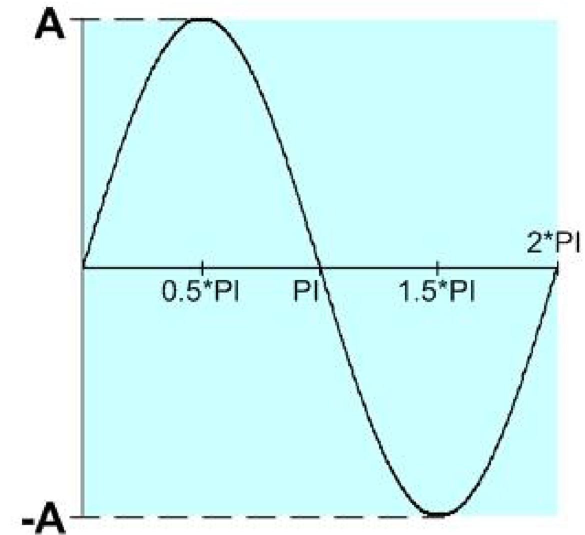
– It is necessary to reserve as many bits as possible after the decimal point to minimize the output truncation error of the result.

- i.e. 18.5 → possible: 0|000 0001 0010.1000 = 0x0128
  better: 0|001 0010.1000 0000 = 0x1280

XC886/888CLM – Cordic Scaling the Amplitude

- Example for circular rotation mode – sine calculation
  - Y_{FINAL} = K*X*sin(Z)   K = 1,64676
  - For scaling the amplitude A = K*X, set X = A/K
    note: choose the amplitude as high as possible to increase the resolution of the range of sin(Z) and to decrease the result error
    note: A_{max} = 32768

X	A	Y_{FINAL}
1	1.64676	{-1.64676…+1.64676}
100	164.676	{-164.676…+164.676}
12145	19999.9	{-19999.9…+19999.9}
19898	32767.3	{-32767.3…+32676.3}



  - Angle Z in normalized scaled integer
    - Min. adjusting angle: $\frac{180°}{32768} = \mathbf{0.00549°}$

XC886/888CLM – Cordic Example *SINE()*

- How to calculate ***sine(z)***

z0=angle, x0=K,y0=0
yn+1 = sine(angle)

~1µs

```
SYSCON0 |= 0x01;              // switch into mapped area
CD_STATC = 0x00;              // clear keep bits - disable CORDIC IRQ
CD_CON   = 0x0A;              // circular rotation mode - auto start
CD_CORDYL= 0x00;
CD_CORDYH= 0x00;              // y0=0
CD_CORDXH= 0x17;              // x0hi=K
CD_CORDZL= z;
CD_CORDZH= (z>>8);            // input angle z in integer
CD_CORDXL= 0xB9;              // x0lo=K and start Cordic
while(!(CD_STATC & 0x04)); // wait on CORDIC ready
printf ("SIN CORDIC = %d%d\n",CD_CORDYH,CD_CORDYL ); // print result
SYSCON0 &= 0xFE;              // switch to  non-mapped area
```

XC886/888CLM – Cordic Example *ARCTAN()*

- How to calculate an ***angle***

xn, yn, zn=0

zn+1 = arctan(yn/xn)

```
CD_S15    CORDIC_CalculateAtan (CD_S15 Xval ,CD_S15 Yval)
{
 if(Xval == 0x0000) {
   g_CORDIC_atan.WORD = 0;          // Avoid error at 'divide by 0'
 }
 else    {
   SYSCON0    |= 1;
   CD_STATC    = 0;
   CD_CON     =(MPS_1+TWOS_COMPLEMENT+START_AUTO+MODE_VECTORING+FUNC_CIRCULAR);
   CD_CORDZH  =   CD_CORDZL = 0;
   CD_CORDYH  =   (unsigned char)  Yval;
   CD_CORDYL  =   (unsigned char) (Yval>>8);
   CD_CORDXH  =   (unsigned char)  Xval;
   CD_CORDXL  =   (unsigned char) (Xval>>8);

   while(!(CD_STATC & 0x04)); // wait on CORDIC ready

   g_CORDIC_atan.BYTE.H = CD_CORDZH;
   g_CORDIC_atan.BYTE.L = CD_CORDZL;
   SYSCON0   &=   0xFE;
   }
}
```

~1µs

XC886/888CLM – Cordic Example Multiply & Accumulate

- How to do a ***MAC*** in S4Q16 (S4.11)

xn, zn, yn

*yn+1 = xn * zn + yn*

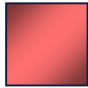
~1µs

~1µs

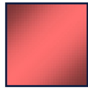
```
// mac (multiply & accumulate) -- input format x, y in S15, z in S4.11
// x=4096 = 0x1000, y=4096 =0x1000, z=2 =0x1000  >> yn+1 = 12288 = 0x3000
SYSCON0 |= 0x01; // set RMAP
CD_CON   =(MPS_1+TWOS_COMPLEMENT+START_WITH_ST_SET+MODE_ROTATION+FUNC_LINEAR);
CORDIC_LOAD_XYZ_REGISTERS(0x1000,0x1000,0x1000); // (x, y, z)
CORDIC_START();   // set start bit
while(!(CD_STATC & 0x04));          // Polling until CORDIC is ready - EOCflag
g_CORDIC_mac.BYTE.H = CD_CORDYH;
g_CORDIC_mac.BYTE.L = CD_CORDYL;    // result yn+1 = 12288 = 0x3000

// additional accumulate
CD_CON   &= 0xEF;                   // Set AutoStart
CD_STATC |= 0x40;                   // KEEP Y
CD_CORDZH = 0x08;                   // z=1 = 0x0800
CD_CORDZL = 0x00;
CD_CORDXH = 0x10;                   // x=4096 = 0x1000
CD_CORDXL = 0x00;                   //CORDIC starts here automatically
while(!(CD_STATC & 0x04));          // Polling until CORDIC is ready - EOC
g_CORDIC_mac.BYTE.H = CD_CORDYH;
g_CORDIC_mac.BYTE.L = CD_CORDYL;    // result yn+1 = 16384 = 0x4000
SYSCON0 &= 0xFE;                    // clr RMAP
```

Infineon

Agenda



XC888/886CLM – Cordic Coprocessor



XC886/888CLM – MDU Coprocessor

Never stop thinking

XC886/888CLM – MDU

- Mathematical co-processor for Multiply and Divide
  - Fast signed/unsigned 16x16bit multiplication
  - Fast signed/unsigned 32/16bit or 16/16bit division
  - 32bit unsigned normalization
  - 32bit arithmetic/logical shift operations
- CPU has to MOVE the operands and result (including waitstate for code fetching from Flash) = 8clk / 8bit
  - e.g. signed multiplication 16bit x 16bit:
    32 (load operands) + 16 (multiplication) + 32 (fetch result) = 80clk = 3.3µs @ 24MHz
- For continues MDU operation, next operands can be loaded in parallel to current calculation
  - e.g. continues (interleaved) signed multiplication 16bit x 16bit:
    16 (multiplication) || 32 (load operands for next multiplication) + 32 (fetch result) = 64clk = 2.7µs @ 24MHz

Infineon

XC886/888CLM – MDU Performance

Table 6-1 MDU Operation Characteristics

Operation	Result	Remainder	No. of Clock Cycles used for calculation
Signed 32-bit/16-bit	32-bit	16-bit	33
Signed 16-bit/16bit	16-bit	16-bit	17
Signed 16-bit x 16-bit	32-bit	–	16
Unsigned 32-bit/16-bit	32-bit	16-bit	32
Unsigned 16-bit/16-bit	16-bit	16-bit	16
Unsigned 16-bit x 16-bit	32-bit	–	16
32-bit normalize	–	–	No. of shifts + 1 (Max. 32)
32-bit shift L/R	–	–	No. of shifts + 1 (Max. 32)

Never stop thinking

XC886/888CLM – MDU ROM-Library

- MDU is very useful for some mathmetical functions such as
  - UIDIV, LMUL, ULDIV, LShift
- ROM contains most valuable MDU-routines which can be called by user
  - this saves codespace and runtime (no waitstate for ROM access)
- easy integration with KEIL compiler
- MDU Library support in ROM only available for devices marked with "M"
- ROM Library support from ***AB-step*** onwards

Routine Name	Num of Bytes		Num of Clock Cycle		ROM
	Software	MDU	Software	MDU	
?C?IMUL	18	20	57	53	MDU Library
?C?UIDIV	85	24	267	61	MDU Library
?C?LMUL	139	61	213	123 / 105	MDU Library
?C?ULDIV	146	197	1148 / 886	101 / 249	MDU Library
?C?LNEG	14	X	31	X	--
?C?ULSHR	19	28	10 + (30 * n)	50 + n	MDU Library
?C?SLSHR	20	28	10 + (30 * n)	50 + n	MDU Library
?C?LSHL	20	28	10 + (30 * n)	50 + n	MDU Library

XC886/888CLM – Floating Point ROM-Library

- ROM contains most valuable floating point routines which can be called by user
  - this saves codespace and runtime (no waitstate for ROM access)
- easy integration with KEIL compiler
- Float Library support in ROM available for all devices
- Fload Library support from ***AB-step*** onwards

Routine Name	Num of Bytes	Num of Clock Cycle	ROM
	Software	Software	
?C?FPADD	~1kbyte	from 309 to 488	Float Library
?C?FSUB		~408	Float Library
?C?FPMUL		~385	Float Library
?C?FPDIV		~1757	Float Library
?C?FPCMP3		~87	Float Library
?C?FPCASTC		from 197 to 231	Float Library
?C?FPCASTI		from 271 to 305	Float Library
?C?FPCASTL		from 699 to 733	Float Library
?C?FPCASTF		from 630 to 980	Float Library

Never stop thinking

Any Questions?

Example PI-Control without Cordic in ASM

```
;******* PI-Regler 16*********
_pi_regler16_B:   ; Eingang_istwert  (16 Bits) = Drehzahl
                  ; Eingang_sollwert (16 Bits) = sollwert16
                  ; Ausgang          ( 8 Bits)  = sollwert
        clr   c
        mov   a,sollwert16+1  ; e = Regelabweichung = Sollwert - Iswert
        subb  a,Drehzahl+1
        mov   r2,a            ; R2 = e_L
        mov   a,sollwert16
        subb  a,Drehzahl      ; A = e_H
        jc  pi16_neg_B        ; Überlauftest
        cjne  a,#0,pi16_1_B   ; High-teil von Regelabweichung e_H test
        jmp  pi16_2_B
pi16_1_B:
        mov  a,#0FFh     ; positive Regelabweichung e_L Begrenzung
        mov  r2,a        ; positive Regelabweichung e_L
pi16_2_B:
        mov  a,r2        ; positive Regelabweichung e_L
        mov  b,ki16      ; Integrierer  yint16 = yint16 + ki*e_L
        mul  ab          ;  ki * e_L
        add a,yint16+2   ; A = yint16_L + L
        mov yint16+2,a   ; yint16_L = yint16_L + L
        mov a,b          ; A = H
        addc a,yint16+1  ; A = yint16_M + H + C
        mov yint16+1,a   ; yint16_M = yint16_M + H + C
        clr  a           ; A = 0
        addc a,yint16    ; A = yint16_H + C
        jnc   pi16_3_B   ; Überlauftest
        mov   a,ymax16   ; Ausgang und Integrierer Begrenzung
        mov   yint16,a
        mov   sollwert,a
        ret
```

```
pi16_3_B:
        cjne  a,ymax16,pi16_4_B  ; Ausgang und Integrierer
        Begrenzung
pi16_4_B:
        jc  pi16_5_B
        mov  a,ymax16
        mov  yint16,a
        mov  sollwert,a
         ret
pi16_5_B:
        mov  yint16,a
pi16_51_B:                ; positive Regelabweichung e_L
        mov  a,r2         ; Proportional Anteil
        mov  b,kp16
        mul  ab           ; kp * e_L
        jnb   b.7,pi16_52_B ; Überlauftest vor der
        Multiplication
        mov   b,#07FH
pi16_52_B:                ; Multiplication mal 2
   add  a,ACC             ; A = L * 2
   xch  a,b               ; A = H  / B = L * 2
   addc a,ACC             ; A = H * 2
   xch  a,b               ; A = L * 2  / B = H * 2
   add  a,yint16+1
   mov  a,b
   addc a,yint16
   jc     pi16_7_B        ; Überlauftest
   cjne a,ymax16,pi16_6_B     ; Ausgang Begrenzung
pi16_6_B:
        jc            pi16_8_B
pi16_7_B:
        mov           a,ymax16
```

```
pi16_8_B:
        mov           ; sollwert,a
        ret
pi16_neg_B:           ; negative Regelabw e_L Begrenzung
   cjne a,#0FFh,pi16_9_B
   mov  a,r2
   jz   pi16_9_B      ; negative Regelabweichung e_L
   cpl   a
   inc   a
   sjmp pi16_10_B
pi16_9_B:
   mov   a,#0FFh      ; negative Regelabweichung e_L
pi16_10_B:
   mov  r2,a          ; R2 = maximale Regelabweichung e_L
   mov  b,ki16        ; Integrierer , yint = yint - ki*e
   mul  ab            ; ki * e_L
   mov  r3,a
   mov  a,yint16+2    ; yint16_L = yint16_L - L
   clr   c            ; C = 0
   subb a,r3          ; A = yint16_L - L
   mov  yint16+2,a    ; yint16_L = yint16_L - L
   mov  a,yint16+1    ; A = yint16_M
   subb a,b           ; A = yint16_M - H - C
   mov  yint16+1,a    ; yint16_M = yint16_M - H - C
   mov  a,yint16      ; A = yint16_H
   subb a,#0          ; A = yint16_H - C
   jnc   pi16_11_B    ; Überlauftest
   clr   a
pi16_11_B:
   mov  yint16,a
pi16_101_B:           ; negative Regelabweichung e_L
   mov   a,r2         ; Proportional Anteil
```

Example PI-Control with Cordic

```
int pi_control(int IST, int SOLL) {
 // y = Kp * e + Ki * integral(e*dt)
 error = SOLL-IST; // calculation of the error
 SYSCON0 |= 0x01;  // switch into mapped area
 CD_STATC = 0x00;  // clr keep bit - int dis
 CD_CON   = 0x08;  // LinRot - auto start MPS=1
 CORDY = 0x0000;   // Y = 0
 CORDZ = 0x08CD;   // Kp = 1.1
 CORDX = error;    // CORDIC start auto
 NOP               // wait on ready - 25nops
 if(ERROR) result = result_max;
 CD_STATC = 0x40;  //set keep bit for CORDY
 CORDZ = 0x0733;   // Ki = 0.9
 CORDX = error_int;//CORDIC start auto
 NOP               // wait on ready - 25nops
 error_int += error;// calc error integral
 if(ERROR) result = result_max;
 printf ("PI = %04d dec \n",(int)result);
 SYSCON0 &= 0xFE;
return (int)result;
}
```