The GH VHDL Library

An
www.OpenCores.org
Project

ghuber@opencores.org

Revision History

Revision	Date	Author	Description
1.0	3 Sept 2005	G Huber	Initial Revision
1.1	10 Sept 2005	G Huber	Add parts, fix some typo's
2.0	17 Sept 2005	H LeFevre	1. Add LFSR's 2. Add gh_ prefix to the name of some parts. (See chapter 4 for explanation on this change) 3 Mod parts to use gh_ parts (where required)
2.1	18 Sept 2005	G Huber	Add decoder/mux, clock divider, and NCO
2.2	24 Sept 2005	S A Dodd	Add pulse generator
2.3	1 Oct 2005	G Huber	Add sweep generator
2.4	4 Oct 2005	H LeFevre	Add Random Number Generator/CASR
3.0	8 Oct 2005	G Huber	Reorganize library, add a couple of shift registers
3.1	15 Oct 2005	S A Dodd	Add parity generator, FIFO's, integer counters
3.2	23 Oct 2005	G Huber	Add programmable LFSR's
3.3	29 Oct 2005	G Huber	Add Configuration Registers
3.4	13 Nov 2005	S A Dodd, G Huber	Add some memory parts
3.5	14 Jan 2006	G Huber	Add delay lines
3.6	21 Jan 2006	S A Dodd G Huber	Add Control Registers Add a fixed delay line for a bus
3.7	28 Jan 2006	H LeFevre	Add a baud rate generator
3.8	4 Feb 2006	S A Dodd H LeFevre	Add FIFO with sync clear Add an In Place Multiplier
3.9	11 Feb 2006	H LeFevre G Huber	Add two more In Place Multipliers (one has both inputs unsigned and the other both inputs are signed) Add another shift register (shifts left)
3.10	18 Feb 2006	G Huber	Add another shift register (also shifts left) Finished adding the gh_ prefix to all parts
3.11	25 Mar 2006	H LeFevre S A Dodd	Add one more In Place Multiplier Mod gh_sincos to use Cordic $\pm$ 45
3.12	13 May 2006	S A Dodd	Add FIR Filter
3.13	26 May 1006	G Huber	Add debounce, stretch low
3.14	16 Sept 2006	G Huber	Add a counter, 18 bit multipliers
3.15	23 Sept 2006	SA Dodd	Add complex math parts
3.16	23 Dec 2006	H LeFevre	Replace async FIFO (to use gray code)
3.17	27 Dec 2006	G Huber HL/SD	Add Gray code converters Update FIFO's
3.18	13 Jan 2007	H LeFevre	add async FIFO's with ¼. ½, and ¾ full flags
3.19	27 Jan 2007	SA Dodd	add digital attenuator
3.20	3 Feb 2007	H LeFevre	add parallel FIR Filter
3.21	10 Feb 2007	H LeFevre	add FIR Filters of odd order, negative symmetry

3.22	9 June 2007	G Huber SA Dodd	add programmable delay bus, FASM dual port Ram with reset, 3 multipliers with generics
3.23	30 June 2007	H LeFevre G Huber	add 2 in-place multipliers, with all data bits out add MAC with full generics and an unsigned array divider
3.24	15 July 2007	S A Dodd	add a FIR filter and Pulse time/width module
3.25	12 Aug 2007	S A Dodd	mod/add filter
3.26	16 Aug 2007	S A Dodd	add (two clock multiply) complex multipliers
3.27	14 Oct 2007	H LeFevre	add some filters w/o multipliers
3.28	21 Oct 2007	S A Dodd G Huber	Add rev A of rectangular to polar (CORDIC application) – increases pipelining add 4 byte memory
3.29	22 Nov 2007	H LeFevre	add VMEbus Slave Interface Module parts
3.30	25 Nov 2007	H LeFevre	add FIR filter, rev A for NCO
3.31	8 Dec 2007	G Huber	add VME read Modules
3.32	30 Dec 2007	H LeFevre	add 3 multiplier complex multipliers
3.33 3.33a	3 May 2008 4 May 2008	H LeFevre	Add random number scalar (serial multiplier) Add random number scalar (parallel multiplier)
3.34	24 May 2008	H LeFevre	Add two asynchronous fifo's (with UART style flags)
3.35	27 May 2008	G Huber	Add programmable delay line using generics
3.36	1 June 2008	S A Dodd	3 complex multipliers, with an extra register delay for higher operating frequency Data Mux(2:1) /DeMux (1:2) set
3.37	4 July 2008	G Huber	Add some NCO type accumulators
3.38	1 Sept 2008	H LeFevre	Add versions of a couple frequency syntheses parts
3.39	20 Sept 2008	H LeFevre	Add programmable Stretch parts, add init to some of the memory parts
3.40	27 Sept 2008	H LeFevre	Add 4 byte GPIO
3.41	04 Oct 2008	H LeFevre	Add Burst Generator
3.42	11 Oct 2008	H LeFevre	Add CORDIC's with 28 bit atan functions
3.43	26 Oct 2008	H LeFevre	Add Sin Cos ROM's
3.44	1 Nov 2008	H LeFevre	Add Sin Cos ROM's with quarter size tables
3.45	8 Nov 2008	H LeFevre	Add config registers (3072, 4096 bits), fix notes
3.46	25 Jan 2009	H LeFevre	Add watch dog timers
3.47	28 Feb 2009	H LeFevre	Add Pulse Width Modulator

Table of Contents

1 Introduction

The GH VHDL Standard Parts Library is a collection of basic VHDL parts that may be included in larger designs. There is nothing wrong with modifying library parts so that they will meet the system requirements.

1.1 Purpose

- Educational – this is a set of design examples that demonstrate some of the more important language constructs.
- To have a set of building blocks to aid in the building of a VHDL design – Large designs can be broken up into smaller blocks. When there are common functions in these blocks, time can be saved when these common functions can be designed once and reused many times.

Note: The library is setup as a collection of design files – this makes it easy to examine the design of each part. Some may want to put them together as a "proper" VHDL library.

1.2 What the Library is Not

- A VHDL language reference.
- Complete – Contributions are encouraged, which may be added the library (or ignored) at our discretion.
- Perfect. Look for ways to improve it – even if we do not like your "improvements," if they make your life easier, use them anyway.

1.3 GH VHDL License

distribution of a fuse map or other hardware implementation in CPLD, FPGA, ASIC or other form of custom IC).

4. THE LESSER IP IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED.
5. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY ARISING FROM, OR IN CONNECTION WITH THE USE OF THE LESSER IP.

2 Basic Registers and Gates

Here are the basic parts that make up many larger systems. For some of these, it may be argued that it is more work to instantiate them than it is to rewrite the function. However, a number of design entry tools allow the use of Block diagrams. When using block diagrams, it is useful to have these parts available.

2.1 D Flip Flop

The D Flip Flop is almost too simple to be in the library, but it is here anyway.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
D	I	Input Data
Q	O	Output Data

File name: gh_DFF.vhd

2.2 JK Flip Flop

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
J	I	J Input
K	I	K Input
Q	O	Output Data

File name: gh_JKFF.vhd

Truth Table for the JKFF

CLK	rst	J	K	Q
X	1	X	X	0
↑	0	1	0	1
↑	0	0	1	0
↑	0	1	1	toggle
↑	0	0	0	no change

2.3 Basic Register and Latch

These parts have the generic “size” which sets the data width.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
LE	I	Latch enable (1 = transparent D to Q 0 = hold Q)
CE	I	Clock enable
D(size-1 downto 0)	I	Input Data
Q(size-1 downto 0)	O	Output Data

Parts	CLK	rst	LE	CE	D	Q	comments
gh_latch.vhd			x		x	x	
gh_register.vhd	x	x			x	x	
gh_register_ce.vhd	x	x		x	x	x	

2.4 XOR Bus

This is just a XOR gate with a programmable length (using the generic “size”). Its purpose is to make it easier to combine two LFSR’s (of different length), or a LFSR with a CASR (Cellular Automata Shift Register), to improve the characteristics of the generated random numbers.

I/O		Function
A(size downto 1)	I	Size number of bits from LFSR A
B(size downto 1)	I	Size number of bits from LFSR B
Q(size downto 1)	O	output

File name: gh_xor_bus.vhd

2.5 Comparators

While Comparators are not strictly gates, they are included here because they are simple enough that some people will find it easier to rewrite the code, than it is to instantiate a component.

I/O		Function
A (size-1 downto 0)	I	A input vector
B (size-1 downto 0)	I	B input vector
min (size-1 downto 0)	I	
max (size-1 downto 0)	I	
D (size-1 downto 0)	I	
AGB	O	ABS of A is greater than the ABS of B when high
AEB	O	ABS of A is equal to the ABS of B when high
ALB	O	ABS of A is less than the ABS of B when high
AS	O	A sign bit
BS	O	B sign bit
ABS_A(size-1 downto 0)	O	ABS of A
ABS_B(size-1 downto 0)	O	ABS of B
Y	O	Y = '1' when D is between min and max

Parts	A	B	min	max	D	AGB	AEB	ALB	AS	BS	ABS_A	ABS_B	Y	Comments
gh_compare.vhd	x	x				x	x	x						Unsigned data
gh_compare_ABS.vhd	x	x				x	x	x	x	x	x	x		Signed data
gh_compare_BMM.vhd			x	x	x								x	Unsigned data
gh_compare_BMM_s.vhd			x	x	x								x	signed data
gh_compare_ABS_reg.vhd (clk and rst inputs added)	x	x				x	x	x	x	x	x	x		Signed data- adds pipeline registers

2.6 Decoders

The design of the decoders is based on the 75LS138, except that the outputs, when active, are high.

I/O		Function
A	I	Address/select input
G1	I	Output enable, active high
G2n	I	Output enable, active low
G2n	I	Output enable, active low
Y(8 or 16 downto 0)	O	Output bus, only 1 output is active (high) at a time- when all enables are active

Parts	A	G1	G2n	G3n	Y	Comments
gh_decoder_2to4.vhd	x	x	x	x	4 bits	Output bit, which corresponds with value of A, is high
gh_decoder_3to8.vhd	x	x	x	x	8 bits	Output bit, which corresponds with value of A, is high
gh_decoder_4to16.vhd	x	x	x	x	16 bits	Output bit, which corresponds with value of A, is high

2.7 Multiplexers

I/O		Function
sel	I	Selects which input becomes the output
A - P	I	Data inputs A input sel = 0, B input sel =1, C input sel = 2, D input sel =3 etc
Y	O	Output

Parts	sel	Data	Y	Comments
gh_mux_2to1.vhd	1 bit	A, B	x	
gh_mux_2to1_bus.vhd	1 bit	A, B	x	Uses generic size to set width of data bus
gh_mux_4to1.vhd	2 bits	A – D	x	
gh_mux_4to1_bus.vhd	2 bits	A – D	x	Uses generic size to set width of data bus
gh_mux_8to1_bus.vhd	3 bits	A – H	x	
gh_mux_16to1_bus.vhd	4 bits	A - P	x	

2.8 Shift Registers

These are just a simple shift registers – the input D is loaded into Q(0) [when shifting right] with each clock edge. The data Q(n) is shifted to Q(n+1) at the same time [or Q(n+1) is shifted to Q(n) when shifting left]. The Shift Register's have the generic "size" which sets the number of bits to be shifted.

It should be noted the "shift left" and "shift right" refers to shifting the data as if it is lined up: q0 q1 q2 q3 q4…qn. Default for this library is shift right (_sl in the name means it shifts left, _slr means it can shift either left or righ).

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
srst	I	Synchronous reset, active high
LOAD	I	Parallel Load command
SE	I	Shift enable
MODE	I	Mode Bits : 00 hold (do nothing) 01 shit right ($Q_i = Q_{i-1}$) 10 shift left ($Q_i = Q_{i+1}$) 11 Parallel Load
DSL	I	Serial data in for shift left
DSR	I	Serial data in for shift right
D or D(size-1 downto 0)	I	Data bit(s) to be shifted and/or loaded
Q(size-1 downto 0)	O	Shifted bits out

Parts	CLK	rst	srst	LOAD	SE	MODE	DSL	DSR	D	Q	Comments
gh_shift_reg.vhd	x	x							x	x	
gh_shift_reg_rs.vhd	x	x	x						x	x	Reset can be changed to Preset w/generic
gh_shift_reg_PL.vhd	x	x		x	x				x	x	Parallel Load, shift right
gh_shift_reg_PL_sl.vhd	x	x		x	x				x	x	Parallel Load, shift left
gh_shift_reg_PL_SLR.vhd	x	x				x	x	x	x	x	Parallel Load, shift left or right
gh_shift_reg_se_sl.vhd	x	x	x		x				x	x	

2.9 Four Byte Configuration Registers

Here is a collection of registers intended for use as configuration/control - set up so that they may be initialized by byte, word, or long word access on a 32 bit data buss.

With FPGA gate counts of over 3 million, how many configuration bits are required?

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
CSn	I	Chip Select, active low
WR	I	Write strobe, active high
BE(3 downto 0)	I	Byte enable bits
A	I	Address bits (Long Word addressing, BE is used to identify which byte)
D(31 downto 0)	I	Data buss in
RD(31 downto 0)	O	Read Configuration Data
Q	O	Configuration Bits

Parts	CLK	rst	CSn	WR	BE	A	D	RD	Q	Comments
gh_4byte_reg_32.vhd	x	x		x	x		x		x	Used on larger parts
gh_4byte_reg_64.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_128.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_256.vhd	x	x	x	x	x	x	x	x	x	Used on larger parts
gh_4byte_reg_512.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_768.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_1024.vhd	x	x	x	x	x	x	x	x	x	Used on larger parts (3072,4096)
gh_4byte_reg_1536.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_2048.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_3072.vhd	x	x	x	x	x	x	x	x	x	
gh_4byte_reg_4096.vhd	x	x	x	x	x	x	x	x	x	

3 Counters

3.1 Binary Counters

All of these counters use standard logic vectors and use the generic "size" to set the number of bits used in the counter.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
srst	I	Synchronous Reset, active high
CE	I	Count enable, active high
LOAD	I	Parallel load control
UP_D	I	Up/down control
D(size-1 downto 0)	I	Parallel load Data
TC	O	Terminal Count
one	O	Active when Count = 1
Q(size-1 downto 0)	O	Count value out

Parts	CLK	rst	srst	CE	LOAD	UP_D	D	TC	one	Q	comments
gh_counter.vhd	x	x		x	x	x	x	x		x	Universal Up/down counter
gh_counter_up_sr_ce.vhd	x	x	x	x						x	Up counter
gh_counter_up_ce.vhd	x	x		x						x	Up counter
gh_counter_up_ce_tc.vhd	x	x		x				x		x	Up counter
gh_counter_up_ce_ld.vhd	x	x		x	x		x			x	Up counter
gh_counter_up_ce_ld_tc.vhd	x	x		x	x		x	x		x	Up counter
gh_counter_down_ce_ld.vhd	x	x		x	x		x			x	Down counter
gh_counter_down_ce_ld_tc.vhd	x	x		x	x		x	x		x	Down counter
gh_counter_down_one.vhd	x	x		x	x		x	x	x	x	Useful as an event counter
gh_counter_fr.vhd	x	x									A free running binary counter

Why have so many counters in the library, when the first one is a super set of (most) the rest? After all, the synthesis tools will remove the excess logic. Logic verification is the answer. If a code coverage (and/or toggle coverage) tool is used to verify the design, some of the excess logic will show up as untested.

3.2 Modulo Counter

The Modulo counter is a specialized counter. It is incremented by the input N, and will roll over at the generic modulo. It will increment by the specified value even as it rolls over. The terminal count will go active the clock period before the roll over, for all values of N.

I/O		Function
CLK	I	Clock, rising edge is used
Rst	I	Asynchronous Reset, active high
CE	I	Count enable, active high
N(size-1 downto 0)	I	Increments by this value
TC	O	Terminal Count
Q(size-1 downto 0)	O	Count value out

Parts	CLK	rst	CE	N	TC	Q	comments
gh_counter_modulo.vhd	x	x	x	x	x	x	Note: size must be large enough to count up to modulo

3.3 Integer Counters

The Integer Counters us integers, rather than standard logic vectors for holding the count values. They have one generic, max_count. The chief advantage this counter have is that they can be set to count to any value, without having a vector size to set.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
LOAD	I	Parallel load control
CE	I	Count enable, active high
D	I	Parallel load Data
Q	O	Count value out

Parts	CLK	rst	CE	LOAD	D	Q	comments
gh_counter_integer_up.vhd	x	x	x	x	x	x	Counts up to max _count
gh_counter_integer_down.vhd	x	x	x	x	x	x	Counts down to zero, rolls over to max_count

4 Custom MSI Parts

This is a collection of parts that have functions that are not normally found in Standard MSI parts, but are not particularly complex in design.

4.1 Pulse Stretcher

The fixed Pulse Stretchers have the generic “stretch_count” which sets the number of clock periods that the pulse will be stretched. The programmable Pulse Stretches use the generic “size” the set the number of bits used to control the stretch count.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
D (Dn)	I	Input pulse to be stretched
stretch(size -1 downto 0)	I	Number of clocks to stretch pulse by
Q (Qn)	O	Stretched pulse out

For the fixed Pulse Stretchers, an integer Variable is used to control the pulse stretching. This means only one generic is needed to be control the stretch time. If a STD_LOGIC_VECTOR had been used, the number of bits in the vector would also need to be adjustable.

Parts	CLK	rst	D(n)	stretch	Q	Comments
gh_stretch.vhd	x	x	x		x	stretches a high pulse
gh_stretch_low.vhd	x	x	x		x	stretches a low pulse
gh_stretch_programmable.vhd	x	x	x	x	x	stretches a high pulse
gh_stretch_programmable_low.vhd	x	x	x	x	x	stretches a low pulse

4.2 Edge Detector

This part will detect edges on the data input. When the input is asynchronous, the "s" outputs should be used to avoid missing edges. With synchronous inputs, the "s" outputs will add a clock delay the non "s" outputs, without a gain in reliability.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
D	I	Input data bit
re	O	Rising edge detected (needs a synchronous input)
fe	O	Falling edge detected (needs a synchronous input)
sre	O	Rising edge detected (Data sampled before detection)
sfe	O	Falling edge detected (Data sampled before detection)

File name : gh_edge_det.vhd

4.3 Clock Divider

This uses a generic to set the dived ratio, the number of high speed clocks per low speed clock. The output is one clock period wide, designed to drive a clock enable pin on the parts running at the lower clock rate.

I/O		Function
CLK	I	Higher rate Clock
rst	I	Asynchronous Reset, active high
Q	O	Lower rate "clock enable" output

File name : gh_clk_ce_div.vhd

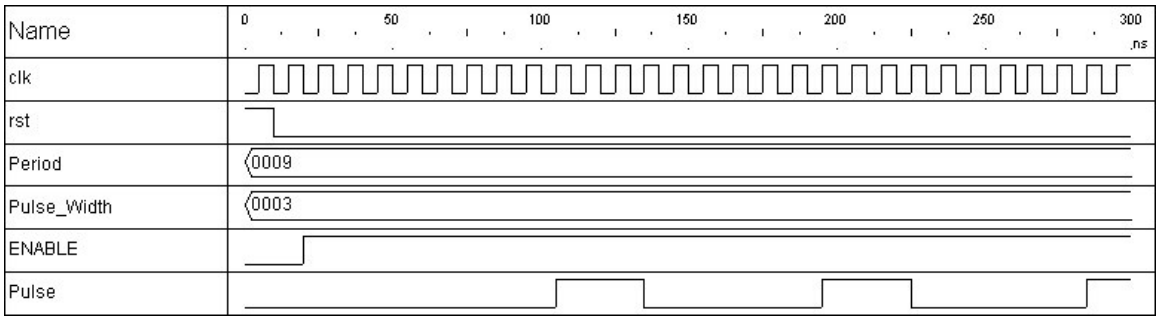
This part was designed specifically to be used by the TVFD_filter, the CIC_filter and any other part that requires two related clocks, where the lower rate "clock" is a clock enable pulse with the correct period.

For the TVFD_filter, the Q output drives the START input. For the CIC filtes, the Q output drives the ND input.

4.4 Pulse Generator

Does this belong here? Well, where else?? The Pulse Generator it is a simple application of two counters. If the Pulse Width is set to be equal to or larger than the Period, the output pulse will be a constant high.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
Period (size_Period-1 DOWNTO 0)	I	The number of clocks between pulses
Pulse_Width (size_Period-1 DOWNTO 0)	I	The Pulse width, in clock periods
ENABLE	I	Enable, active high
Pulse	O	The Output Pulse



Here is a simulation of the Pulse generator with the period set to 9 and the pulse width set to 3. File name : gh_pulse_generator.vhd

4.5 Parity Generator

This is a serial parity generator. It needs to be before the start of a data word. The SD (sample data command) is included so that it is easy to use a clock that is greater that the data rate.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
srst	I	Synchronous Reset, active high
SD	I	Sample Data control
D	I	Serial data in
Q	O	Parity Bit

File name : gh_parity_gen_Serial.vhd

4.6 Delay Lines

Here is a collection of registered delay lines. All of the delay lines use shift registers, so it is not just an edge that is delayed, it will delay the entire serial data string.

The fixed length delay line uses the generic "clock_delays" to set the number of register delays.

The programmable delay lines use a number of fixed delay lines, each with a multiplexer at the input to select the source of the input, also a multiplexer is used to select the source for the output. This avoids the need for a single large multiplexer to select the delay tap.

Note: For the programmable delay lines- when the delay changes, any data in the shift registers may be at the "wrong delay." If it is not cleared, it will take the DELAY (or ½ max delay, which ever is less) number of clocks to shift out the "bad" data.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
srst	I	Synchronous Reset, active high
D	I	Data input
DELAY (7, 6, 5, 4, or 3 downto 0)	I	Sets the programmable delay
Q	O	Output data

Parts	CLK	rst	Srst	D	DELAY	Q	Comments
gh_delay.vhd	x	x	x	x		x	Uses generic "clock_delays" to set number of clock delays
gh_delay_bus.vhd	x	x	x	x		x	Uses generic "clock_delays" to set number of clock delays and the generic "size" to set bus width
gh_delay_programmable_15.vhd	x	x	x	x	x	x	
gh_delay_programmable_31.vhd	x	x	x	x	x	x	
gh_delay_programmable_63.vhd	x	x	x	x	x	x	
gh_delay_programmable_127.vhd	x	x	x	x	x	x	
gh_delay_programmable_255.vhd	x	x	x	x	x	x	
gh_delay_programmable_255_bus.vhd	x	x	x	x	x	x	
gh_delay_programmable_bus.vhd	x	x		x	x	x	Uses generics for data width and size of possible delay (address size of internal RAM)

4.7 Baud Rate Generator

This 16 bit baud rate generator is designed to be a building block in UART's. It has separate clocks for loading the baud rate register and for the generating baud rate. Valid baud rate divide ratio's are from 2-65535. Divide values of 1 or 0 will disable the generator. The counter will be reloaded with a write to either byte.

I/O		Function
clk	I	Clock, rising edge is used
BR_clk	I	Baud rate counter clock
rst	I	Asynchronous Reset, active high
WR	I	Write, active high
BE(1 downto 0)	I	Byte enable, active high bit 1 for bits 15 downto 8 bit 0 for bits 7 downto 0
D(15 downto 0)	I	data in
RD(15 downto 0)	O	The baud rate register
rCE	O	Baud rate clock (Typically 16x of UART's baud rate- one BR_clk period wide)
rCLK	O	Baud rate clock (duty cycle about 50%)

File name : gh_baud_rate_gen.vhd

4.8 Control Registers

Embedded Systems often have Control Registers, where the software folks would like to be able to set or clear individual bits. If they can not do this, they may need to do a Read-Modify-Write, or use a shadow register so that only the desired bits are changed.

These Control Registers allow individual bits to be set, reset, or inverted. This is done by setting the MODE bits. The four operations are to write entire register, set any number of individual bits, clear any number of bits, or invert any number of bits.

One easy way of controlling the MODE bits is to tie them the to lower address bits.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
CE	I	Clock enable, active high Note: this signal must by synchronous with clk, and must go low between data writes
CSn	I	Chip Select, active low
WE	I	Write strobe
BE(3 downto 0)	I	Byte enable bits
MODE(1 downto 0)	I	Mode bits "00" writes D into Q "01" sets bits that are '1' in Q "10" clears bits that are '1' in Q "11" inverts bits that are '1' in Q
A	I	Address (Long Word addressing, BE is used to identify which byte)
D(size-1 downto 0) or (31 downto 0)	I	Data input
Q(size-1 downto 0) or (31 downto 0)	O	Output data

Parts	CLK	rst	CE	CSn	WE	BE	MODE	A	D	Q	Comments
gh_register_control_ce.vhd	x	x	x				x			x	Uses generic "size" to set bit width of the register
gh_4byte_control_reg_32.vhd	x	x		x	x	x	x			x	
gh_4byte_control_reg_64.vhd	x	x		x	x	x	x	x	x	x	
gh_4byte_control_reg_128.vhd	x	x		x	x	x	x	x	x	x	
gh_4byte_control_reg_256.vhd	x	x		x	x	x	x	x	x	x	

4.9 A Switch de-bouncer

Here is a logic module that will help in de-bouncing a switch. It has two generics that affect how it works:

min_pw (an integer) number of clocks wide a pulse needs to be to change states
hold (an integer) number of clocks to hold the output level

The check for minimum pulse width can help in filtering out noise that may be on the line, while the hold time should wide enough to allow any ringing (or switch bouncing) to settle out. It is setup to work the same way on signals that are active high as active low.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
D	I	data in
Q	O	De-bounced output

File name: gh_debounce.vhd

4.10 An Edge Detector for changing Clock Domains

This part is designed so that the edges of a pulse generated in one clock domain, can be sampled in a different clock domain. Although either clock may have a higher frequency than the other, if the output clock has the higher frequency, the gh_edge_detector.vhd will use fewer resources.

Note: the period of the input D (rising to rising edge, or falling edge to falling edge) should be at lest three times the slower clock frequency for proper operation. Also, a narrow input pulse may cause both outputs to be active at the same time (the greater the difference in clock frequencies, the higher probability this will happen).

I/O		Function
iclk	I	Input clock for sampling D
oclk	I	Clock which will be synchronous with outputs re and fe
rst	I	Asynchronous Reset, active high
D	I	Input data bit, if not synchronous with iclk, should be as lest as wide as an iclk period + (a register) setup time + hold time
re	O	Rising edge detected
fe	O	Falling edge detected

File name : gh_edge_det_XCD.vhd

4.11 Gray code converters

In a standard binary sequence, multiple bits may change at the same time (for example, in going from 2 to 3, two bits change at the same time). In contrast, Gray codes have only one bit change at a time.

Gray code counters offer a major advantage in Asynchronous FIFO design. For example, when the write count value is sampled by the read clock (for generating the EMPTY flag), with only one bit changing at a time, the worst that will happen is that the EMPTY flag will be high for one clock period longer than "ideal."

The two converters use combinational logic, and the generic "size" to set the data width.

File Names: gh_binary2gray.vhd
gh_gray2binary.vhd

Reference

1. Clive "Max" Maxfield, *Bebop to the Boolean Boogie, Second Edition*, Newnes 2003 – page 361

4.12 Pulse Width/Time Measurement

This module will measure the pulse width, and provide a relative time of arrival (TOA), for a series of pulses. The "current" time is also available. There are separate generics for the pulse width and time measurements.

For best operation, T_size >= pw_size

I/O		Function
clk	I	Input clock for sampling D
rst	I	Asynchronous Reset, active high
Pulse(pw_size-1 DOWNTO 0)	I	Pulse to measure
NEW_PULSE	O	New Pulse detected, PW & TOA valid
PW	O	Pulse Width measurement
TOA(T_size-1 DOWNTO 0)	O	Time of Arrival for pulse, relative to TTIME
TTIME(T_size-1 DOWNTO 0)	O	Free running counter to provide relative time
ACTIVE	O	goes high with a pulse input, goes low when there is no new pulse in the wrap around time of the free running counter.

File name : gh_pw_wTOA.vhd

4.13 Lower Rate Clock Mirror

This module, in systems that have a 1x clock and a 2x clock, will generate a logic mirror of the 1x clock. Works with the Data DeMux and Data Mux modules.

In systems that use multiple, related clocks (multi-rate systems), the higher rate clock may need to sample the lower rate clock so that their phase relationship is known. When using FPGA's, it is highly recommended to avoid using clocks as anything other than a clock input for a register.

I/O		Function
clk_2x	I	Clock, rising edge used - higher rate
clk_1x	I	Clock, rising edge used – lower rate
rst	I	Asynchronous Reset, active high
mirror	O	Logical mirror of 1x clock

File name: gh_clk_mirror.vhd

4.14 Data DeMux 1 to 2

This module will split a stream of data into two – at half the data rate of the input stream.

I/O		Function
clk_2x	I	Clock, rising edge used - higher rate
clk_1x	I	Clock, rising edge used – lower rate
rst	I	Asynchronous Reset, active high
mux_cnt	I	Controls demux timing (use mirror output of gh_clk_mirror.vhd module)
D(size-1 downto 0)	I	Input data, sync to 2x clock
Qa(size-1 downto 0)	O	Output data, sync to 1x clock (1^{st} sample)
Qb(size-1 downto 0)	O	Output data, sync to 1x clock (2^{nd} sample)

File name: gh_de_mux.vhd

4.15 Data Mux 2 to 1

This module will split a combine of two data streams into one – at twice the data rate of the input stream. It does not use the lower rate clock.

I/O		Function
clk_2x	I	Clock, rising edge used - higher rate
rst	I	Asynchronous Reset, active high
mux_cnt	I	Controls mux timing (use mirror output of gh_clk_mirror.vhd module)
Da(size-1 downto 0)	I	Input data, sync to 1x clock (1st sample)
Da(size-1 downto 0)	I	Output data, sync to 1x clock (2nd sample)
Q(size-1 downto 0)	O	Output data, sync to 2x clock

File name: gh_mux.vhd

4.16 Four Byte GPIO

The four byte GPIO module is a modification of the four byte control register (see paragraph 4.8) to include tri-state IO.

This Module allows flexible chip IO – the direction is controllable on a byte (8 bits) basis. Each bit (for the bytes that are driving) can be controlled on an individually.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
CSn	I	Chip Select, active low
WE	I	Write strobe
DRIVE(3 downto 0)	I	Per byte direction control (1 = drive, 0 = receive)
BE(3 downto 0)	I	Byte enable bits
MODE(1 downto 0)	I	Mode bits (see paragraph 4.8 for operation)
D (31 downto 0)	I	Data input
RD(31 downto 0)	O	Read back Data
Q(31 downto 0)	IO	Input/Output data

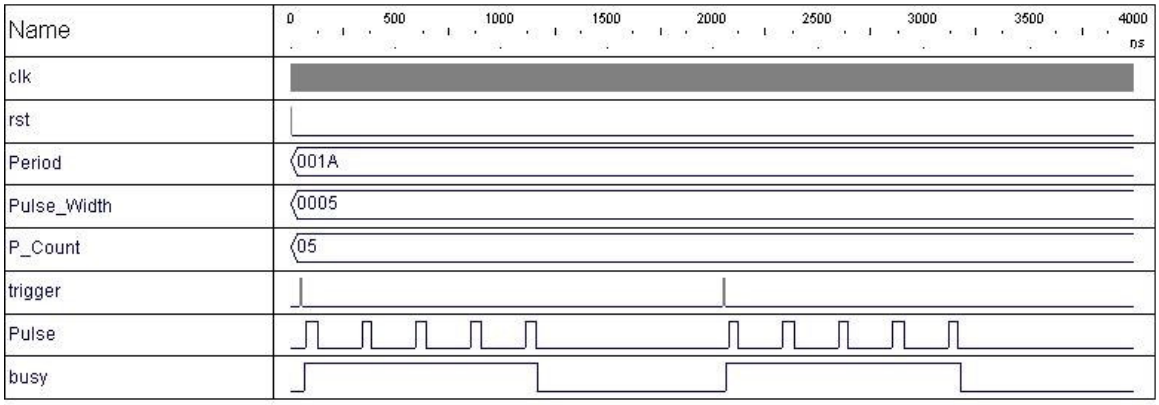
File name: gh_4byte_gpio_32.vhd

4.17 Burst Generator

The Burst Generator it is a simple application of three counters. A fourth counter can be used to create the trigger signal, making the bust periodic (an exercise left to the user).

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
Period (size_Period-1 DOWNTO 0)	I	The number of clocks between pulses
Pulse_Width (size_Period-1 DOWNTO 0)	I	The Pulse width, in clock periods
P_Count (size_pcount-1 downto 0)	I	The number of pulses in a burst
trigger	I	Starts burst, active high
Pulse	O	The Output Pulse
busy	O	Active high between 1st pulse through the end of the last pulse

Below is a simulation of the gh_Burst_Generator.vhd file:



4.18 Watch Dog Timers

A Watch Dog Timer is a free running counter- which, if the system fails to reset before it times out, will (typically) re-start a system (reset, re-boot, or interrupt). A generic, ticks, sets the number of clock ticks for the time out period in the fixed length version. The programmable version uses a generic size that sets the maximum time out.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
T_en	I	Timer enable
t	I	Toggle input to reset counter
t_time (size-1 downto 0)	I	Timer time out clocks
Q	O	High after time out period has elapsed

Parts	clk	rst	T_en	t	t_time	Q	Comments
gh_wdt.vhd	x	x		x	x	x	
gh_wdt_programmable.vhd	x	x	x		x	x	

4.19 Pulse Width Modulator

A pulse width modulator is one way of converting digital data to analog. The output is a series of pulses with the pulse widths (or duty cycle) that are proportional to the input digital value.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
d_format	I	Input Data format 0 = 2's complement, 1 = offset binary
DATA(size-1 downto 0)	I	Input data
PWMo	O	Pulse Width Modulator output
ND	O	New Data Sample strobe

The minimum recommended clock frequency for the pulse width modulator module is:

$$F_{PWM} = 2 \times F_{range} \times R$$

F_{PWM} = PWM module clock frequency
F_{range} = maximum frequency of input data stream
R = resolution (typically a multiple of 2N, where N is the number of bits)

Parts	clk	rst	d_format	DATA	PWMo	ND	Comments
gh_PWM.vhd	x	x	x	x	x	x	

Pulse width modulators can be used to control the intensity of LED's, audio play back, and motor control. The reference (listed below) is highly recommended reading.

Reference:

1. Rafael Camarota, *How to control analog output from a CPLD using a pulse width modulator,* Programmable Logic Design Line, February 24, 2009 http://www.industrialcontroldesignline.com/howto/motorcontrol/214502805

5 Math Functions

5.1 Accumulator

The Accumulator has the generic "size" which sets the number of bits to be accumulated. When CE is high, the value of D is added to the value of Q. No provision is made for overflow or underflow.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
srst	I	Synchronous Reset, active high
LOAD	I	Load data w/o accumulate
CE	I	Clock enable, active high
D (size-1 downto 0)	I	Input Data
Q(size-1 downto 0)	Q	Shifted bits out

Parts	CLK	rst	srst	LOAD	CE	D	Q	Comments
gh_acc.vhd	x	x	x		x	x	x	
gh_acc_ld.vhd	x	x		x	x	x	x	Loads Data with out accumulate (has priority over CE)

5.2 Multipliers

The multipliers will be recognized, at lest by the Xilinx ISE synthesis tool, and placed into one of the multiplier blocks. The two clock delay Multipliers are expected to operate at higher clock rates than the single clock delay multipliers.

I/O		Function
clk	I	Clocks, rising edge is used for A/B ports
DA(15 downto 0)	I	A input data port
DB(15 downto 0)	I	B input data port
Q(15 downto 0)	O	Output data

Parts	CLK	DA/DB	Q	Comments
gh_mult_g16.vhd	x	x	x	Has two clock delay, signed data
gh_mult_g18.vhd	x	x	x	Has two clock delay, signed data
gh_mult_g18_sc.vhd	x	x	x	Has a single clock delay, signed data

5.3 Multipliers using Generics

The multipliers will be recognized, at lest by the Xilinx ISE synthesis tool, and placed into one of the multiplier blocks. These use generics for the size of the input ports – the output port size is the sum of the size of the two input ports.

I/O		Function
clk	I	Clocks, rising edge is used for A/B ports
I(size-1 downto 0)	I	signed input data
Scale(ssize-1 downto 0)	I	unsigned input data
A(asize-1 downto 0), B(bsize-1 downto 0)	I	input data
Q	O	Output data

Parts	CLK	I	Scale	A/B	Q	Comments
gh_scaling_mult.vhd	x	x	x		x	Has one clock delay
gh_mult_gs.vhd	x			x	x	Has one clock delay
gh_mult_gus_sc.vhd	x			x	x	Has one clock delay

5.4 Multiplier Accumulator

The Multiplier Accumulator is a basic building block used in digital filters.

I/O		Function
clk	I	Clocks, rising edge is used for A/B ports
rst	I	Asynchronous Reset, active high
srst	I	Synchronous Reset, active high
LOAD	I	Load data w/o accumulate
ce	I	Clock enable
DA(15 downto 0)	I	A input data port
DB(15 downto 0)	I	B input data port
Q(15 downto 0)	O	Output data

Parts	CLK	rst	srst	LOAD	CE	DA/DB	Q	Comments
gh_MAC_16bit.vhd	x	x	x		x	x	x	
gh_MAC_16bit_ld.vhd	x	x		x	x	x	x	Loads Data with out accumulate (has priority over CE)
gh_MAC_ld.vhd	x	x		x	x	x	x	includes generics to set size (separate for DA/DB) and for bit expansion to avoid accumulator overflow

5.5 Random Number Generation

There are a number of was to generate pseudo random numbers in hardware. The LFSR may be the one most used, but a CASR is also included.

5.5.1 The Linear Feedback Shift Register (LFSR)

The Linear Feedback Shift Register (LFSR) is used for generating pseudo random numbers. These use the Fibonacci implementation, where the output from some of the registers are exclusive ORed together and feedback to the input of the beginning of the shift register.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
LOAD	I	Load the seed value
seed(size of LFSR downto 1)	I	LFSR seed (initial value)
Q(size of LFSR downto 1)	O	output

Parts	CLK	rst	LOAD	Seed	Q	Approximant time pattern runs before repeating (when the clock rate is 100MHz)
gh_lfsr_24.vhd	x	x			x	167.8ms
gh_lfsr_36.vhd	x	x			x	11.45 minutes
gh_lfsr_48.vhd	x	x			x	32.578 days
gh_lfsr_64.vhd	x	x			x	5,849 years
gh_lfsr_gfb4.vhd	x	x			x	Has generics for size (first feedback tap) and three more feedback taps (these taps may be set to zero to have tap ignored). If Taps are picked that give a maximum sequence length: $(2^{size} -1) * 10$ ns
gh_lfsr_gfb4_ld.vhd	x	x	x	x	x	Adds Load / Seed inputs

Bits	Taps	Bits	Taps	Bits	Taps	Bits	Taps
9	9,5	19	19,6,2,1	32	32,22,2,1	67	67,66,58,57
10	10,7	21	21,19	33	33,20	72	72,66,25,19
11	11,9	23	23,18	38	38,6,5,1	77	77,76,47,46
12	12,6,4,1	25	25,22	43	43,42,38,37	81	81,77
13	13,4,3,1	26	26,6,2,1	47	47,42	89	89,51
15	15,14	28	28,25	51	51,50,36,35	96	96,94,49,47
16	16,15,13,4	29	29,27	55	55,31	97	97,91
17	17,14	30	30,6,4,1	57	57,50	99	99,97,54,52
18	18,11	31	31,28	61	61,60,46,45	100	100,63

A selection of different length LFSR feedback taps. For those who are interested in LFSR's of different lengths should consult one of the references listed.

According to *The Art of Electronics*, for all of the maximum length LFSR's that have two feedback taps, the smaller feedback tap (n) can be replaced by the value of m – n (where m is the length of the LFSR).

References:

2. Paul Horowitz, Winfield Hill, *The Art of Electronics, Second Edition*, Cambridge Press, 1989 (lists taps for LFSR's from 3 to 39 that need only two taps for maximum length)
3. Synthaholic's Electronic Music Site*, LFSR Feedback Taps to 168 bits*, http://home1.gte.net/res0658s/electronics/LFSRtaps.html
4. Xilinx Inc, *Linear Feedback Shift Register v3.0*, LogiCore, March 28, 2003

5.5.2 CASR and Random Number Generator

The CASR (Cellular Automata Shift Register), and the Random Number Generator are added to the library with reservations. The referenced paper does not suggest a seed value for the CASR – when it was simulated, with most seed values, the pattern repeats every 1.74762 ms (based upon a 100MHz clock rate – in contrast a 36 bit LFSR will run over 11 minutes before it will repeat). A seed value of all ones will repeat faster.

The Random Number Generator XOR's 32 bits of the CASR output with 32 bits from a LFSR. The generic flavor of the LFSR is used. All of the generics of the LFSR and the seed value and load of the CASR are also on the Random Number generator to make experimenting with it easier. The generated number is 32 bits wide. Also, 32 bits from the two shift registers are also brought out to make it easier to play with.

They are included with the hope that it will inspire someone to send me some additional references on its proper usage.

CASR I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
load	I	Active high, will load the seed value
seed(37 downto 1)	I	start value for the shift register
Q(37 downto 1)	O	output

File names: gh_casr_37.vhd
gh_random_number.vhd

Reference

1. Thomas Tkacik, *A Hardware Random Number Generator*, a PDF file with Motorola's "intelligence everywhere" logo, August 14, 2002. a copy can be found at http://ece.gmu.edu/crypto/ches02/talks_files/Tkacik.pdf

5.5.3 Programmable LFSR's

For applications when the user may want dynamic control over the length, or start of the pseudo random number sequence, here are a couple of programmable LFSR's. One word of caution: area. This programmable LFSR's consume a lot more logic than the fixed length versions. For comparison (in a Xilinx Virtex2p FPGA):

LFSR	Slice Flip-Flops	# of Slices	# of LUT's
gh_lfsr_gfb4.vhd (with default, with a length of 43)	43	32	15
gh_lfsr_PROG_16.vhd	16	49	91
gh_lfsr_PROG_32.vhd	32	94	178

Programmable LFSR pin list

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
LOAD	I	Load value of D into LFSR
TAPS(1 down to 0)	I	Sets the number of feedback taps 00 = 1 feedback TAP used 01 = 2 feedback TAPs used 10 = 3 feedback TAPs used 11 = 4 feedback TAPs used
fb1,fb2, fb3, fb4 (4 or 3 downto 1)	I	Value for fbx = feedback TAP -1
D(32 or 16 downto 1)	I	Seed value in
Q(32 or 16 downto 1)	O	Shift register out

5.5.4 Random Number Scalars

There are cases where a random number within a specific range is needed. This module uses the maximum and minimum values to get a range, which is used to as a scale value that is multiplied with a pseudo random number. The product is then added with the minimum value for the result. The basic equation used is:

$$Scaled\_number = mim \oplus (random \otimes (\max - \min))$$

Note: the modules uses some rounding which is not shown.

If an LFSR is used to generate the pseudo random number input, it should run at full clock rate, with only one sample per Nsam period.

The nature of 2's complement binary math allows the input data to be either unsigned or signed, with the following limitations:

1. Data type may not be mixed: all must be signed, or all must be unsigned.
2. The maximum must be larger than the minimum value for the output to be in the proper range.

CASR I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
Nsam	I	New Sample command (minimum period is size+5 clock periods)
Max(size-1 downto 0)	I	Maximum number wanted
Min(size-1 downto 0)	I	Minimum number wanted
random(size-1 downto 0)	I	Random number (from LFSR, for example)
Sran(size-1 downto 1)	O	Output random number (Min ≤ Random number ≤ Max)

Parts	clk	rst	Nsam	Max	Min	random	Sran	
gh_ran_scale.vhd	x	x	x	x	x	x	x	minimum output sample period is size+5 clock periods)
gh_ran_scale_par.vhd	x			x	x	x	x	Uses parallel multiplier, output sample data rate is clock rate Suggestion: avoid using single LFSR for input – XOR two, or XOR with CASR (see par 5.5.2)

5.6 In Place Multipliers

These Multipliers are slow (size + 3 clock cycles), but will use relatively little logic. Some of them truncate the lower half of the output – both input data ports are the same size, which is set by the Generic "size."

These Multipliers use the shift-and-add technique known as Booth's Algorithm. Negative numbers have their 2's complement sent through Booth's Algorithm, if the output is negative (one negative input), a 2's complement is done on the output data as well. The shift register and the adder share the same set of registers, hence the name "In Place Multipliers."

Multiplier pin list

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
start	I	Start calculation, ignored while BUSYn is active
A(size-1 downto 0)	I	A data input
B(size-1 downto 0)	I	B data input
Q([2*]size-1 downto 0)	O	Output
BUSYn	O	Active low while calculating product

Note: Some synthesis tools do not handle these in place multipliers gracefully.

Parts	Comments
gh_mult_ip_usus.vhd	Both A and B inputs are unsigned
gh_mult_ip_usus_mg.vhd	Gain modified so that when A (or B) are all high, the output will follow the B (or A) inputs
gh_mult_ip_sus.vhd	A input is signed, B input is unsigned – when all B input bits are set high, output Q follows input A
gh_mult_ip_ss.vhd	Both A and B inputs are signed - gain is modified – 16 bit examples: - this part: x"8000" times x"8000" = x"7FFF" - "normal" multiplier output (upper 16 bits) = x"4000" - this part: x"7FFF" times x"7FFF" = x"7FFE" - "normal" multiplier output (upper 16 bits) = x"3FFF"
gh_mult_ip_usus_ab.vhd	Both A and B inputs are unsigned – all bits on output
gh_mult_ip_sus_ab.vhd	A input is signed, B input is unsigned – all bits on output

Reference

1. Clive "Max" Maxfield, *Bebop to the Boolean Boogie, Second Edition*, Newnes 2003 – page 78

5.7 Unsigned Array Divider

An unsigned divider is borrowed from Reto Zimmermann's public domain "VHDL Library of Arithmetic Units." Some minor edits were made it to so that it would fit better in this library. (rz_ was added to the names, the full adder was placed in the same file with the divider, etc.)

Multiplier pin list

I/O		Function
X(widthX-1 downto 0)	I	dividend
Y(widthY-1 downto 0)	I	divisor
Q(widthX-widthY downto 0)	O	quotient
R(widthY-1 downto 0)	O	remainder out

File names: rz_DivArrUns.vhd

The unsigned divider is implemented as a combination device. Also, do not forget that division is a much slower process than multiplication.

Those interested in VHDL Arithmetic, may be interested in Reto Zimmermann's complete public domain "VHDL Library of Arithmetic Units," which may be found at: www.iis.ee.ethz.ch/~zimmi/arith_lib.html

Note: the singed divider in "VHDL Library of Arithmetic Units" does not work (as admitted to in the comments in the file) which is why only the unsigned part is included.

(Note: this part is ***not*** covered by the GH VHDL License, no rights to it are claimed by anyone on the GH_VHDL_LIB team.)

5.8 Complex Math

Complex Math is used in processing quadrature signals, which are used in many digital signal processing applications. For example:

- Communications systems
- Single sideband modulators/demodulators
- Antenna beamforming applications
- Time difference of arrival processing in radio direction finding schemes
- Coherent pulse measurement systems
- Radar systems

Quadrature signals are two dimensional, whose value at an instant in time can be specified by a single complex number. Traditionally, the two parts have been called the *real part* and the *imaginary part*. Communications engineers prefer to call them the in-phase (I) and quadrature phase (Q).

Complex Math pin list

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
IA(size-1 downto 0)	I	I phase A data input
IB(size-1 downto 0)	I	I phase B data input
Scale(ssize-1 downto 0)	I	Scale Data input (unsigned)
iI(size-1 downto 0)	I	I phase data input
iQ(size-1 downto 0)	I	Q phase data input
QA(size-1 downto 0)	I	Q phase A data input
QB(size-1 downto 0)	I	Q phase B data input
I(size-1 downto 0)	O	I phase output
Q(size-1 downto 0)	O	Q phase output

These multipliers use generic's to set the data path width. They are recognized (at lest by the synthesis tool's from Altera and Xilinx) and placed in the fixed multiplier blocks.

Parts – complex adder, multipliers	Clk	rst	IA	IB	QA	QB	I	Q	Comments
gh_complex_add.vhd	x	x	x	x	x	x	x	x	
gh_complex_ssb_mult.vhd	x	x	x	x	x	x	x		
gh_complex_mult.vhd	x	x	x	x	x	x	x	x	uses 4 multipliers
gh_complex_mult_3m.vhd	x	x	x	x	x	x	x	x	uses 3 multipliers
gh_complex_ssb_mult_2cm.vhd	x	x	x	x	x	x	x		uses 2 clock multipliers
gh_complex_mult_2cm.vhd	x	x	x	x	x	x	x	x	
gh_complex_mult_2cm_3m.vhd	x		x	x	x	x	x	x	
gh_complex_ssb_mult_2cm_xrsp.vhd	x		x	x	x	x	x		Has extra register in SUM path to increase operating frequency
gh_complex_mult_2cm_xrsp.vhd	x		x	x	x	x	x	x	
gh_complex_mult_2cm_3m_xrsp.vhd	x		x	x	x	x	x	x	

The complex adder will combine two signals – but care must be used to avoid overflow.

Parts - Scaling Multiplier	clk	Scale	iI	iQ	I	Q	Comments
gh_complex_scaling_mult.vhd	x	x	x	x	x	x	Scale is an unsigned value
gh_complex_scaling_mult_2cm.vhd	x	x	x	x	x	x	uses 2 clock multipliers

Reference

1. Richard G. Lyons, *Understanding Digital Signal Processing, Second Edition,* Prentice Hall, 2004
2. Ian Ing and Asher Hazanchuk, *Efficient FPGA Multiplier Usage in Wireless Basestations*, (a Lattice Semiconductor Corp. sponsored White Paper for the) FPGA and Structured ASIC Journal, Sept. 2007

5.9 Digital Attenuator

When creating multi-tone signals, digital attenuators will allow there amplitudes to be adjusted individually. Care must be used when adding multiple signals to avoid math overflow.

The attenuator has a ten bit control input. It has 0.125 dB resolution, total useful attenuation is limited by the dynamic range of its sixteen bit output bits. The out put is an unsigned scale factor, intended to drive one input of a multiplier (the signal to be attenuated should be a signed, and applied to the multiplier's other input).

This function is implemented two different ways, a Lookup Table, or two smaller lookup tables (17 bits wide to minimize round off errors - which will be implemented in logic), and a multiplier.

The basic equation that this function performs is:

$$Q = .5 + 65535 \times 10^{\left(\frac{atten \times .125}{20}\right)}$$

It should be noted that *atten* is a negative number.

This is the (application specific) inverse of the well known $dB = 20 \times LOG_{10}(x)$

I/O		Function
clk	I	Clock, rising edge is used
Atten(9 downto 0)	I	Asynchronous Reset, active high
Q(15 downto 0)	O	Scale output- unsigned

Parts – Digital Attenuator	clk	Atten	Q	Comments
gh_attenuation_10.vhd	x	x	x	Uses two smaller LUT's and a multiplier
gh_atten_rom_10.vhd	x	x	x	A single LUT

6 Memory

6.1 Synchronous RAM

generics	Function
size_data	Size of the data bus
size_add	Size of the address bus

I/O		Function
clk	I	Clocks, rising edge is used for A/B ports
rst	I	reset memory contents
WE	I	Write enable, active high
add	I	Address lines
D(size_data-1 DOWNTO 0)	I	Input data
Q(size_data-1 DOWNTO 0)	O	Output data

Note: The signal names on dual port RAM's have an "A_" or "B_" prefix, if both ports have that function, otherwise, no prefix is used.

The FASM (FPGA and ASIC Subset Model) memory has a Synchronous write port, but the read ports are Asynchronous.

Parts	clk	rst	WE	add	D	Q	Ports	Notes:
gh_sram_1wp_2rp.vhd	x		x	x	x	x	1 write 2 read	Minimum throughput (write to data read) 3 clocks. Recognized, by the Xilinx ISE synthesis tool, as Block RAM
gh_fasm.vhd	x		x	x	x	x	1 write 1 read	Single clock write data, Asynchronous read. Recognized, at lest by the Xilinx ISE synthesis tool, as distributed RAM
gh_fasm_1wp_2rp.vhd	x		x	x	x	x	1 write 2 read	
gh_fasm_1wp_2rp_r.vhd	x	x	x	x	x	x	1 write 2 read	
gh_sram.vhd	x		x	x	x	x	1 write 1 read	Single clock to store or read data. Minimum throughput (write to data read) 2 clocks. Recognized, at lest by the Xilinx ISE synthesis tool, as Block RAM.
gh_sram_1wp_2rp_sc.vhd	x		x	x	x	x	1 write 2 read	

6.2 FIFO's

The FIFO's are intended for applications where portability more important than performance or efficiency. It is expected that, for example, that a Xilinx CoreGen part would use fewer logic resources and run faster than using a FIFO out of this library. However, these FIFO's are pure VHDL – they are portable. The same design, without modification, can be used in Xilinx, Altera, Actel, Lattice, as well as any other FPGA families.

The control logic for the write (WR) signal needs to sample the full flag to ensure that a write can take place. When the empty flag is low, the output data word is ready, once it is used, an active read (RD) will increment the read counter for the next data word. If the empty flag is high, the read command signal will be ignored.

It should be noted that here, "synchronous" means the write and read ports use the same clock. "Asynchronous" means that the write port has one clock and the read port has a second, unrelated clock– all the control signals are synchronous with their ports clock.

FIFO generics	Function
add_width	Number of address bits used to access RAM - sets FIFO depth = 2 ^add_width
data_width	Size of the data bus

6.2.1 Synchronous FIFO

I/O		Function
clk	I	Clock, active on rising edge
rst	I	Reset, active high – resets counters, flags
srst	I	Sync Reset, active high – resets counters, flags
WR	I	Write command, advances the write counter after write
RD	I	Read command, advances read counter to read next word
D(data_width -1 DOWNTO 0)	I	Input data
Q(data_width -1 DOWNTO 0)	O	Output data
empty	O	When low, output data is valid
full	O	When low, WR is sampled for write command

File name gh_fifo_sync_sr.vhd (uses FSAM style memory)
gh_fifo_sync_rrd_sr.vhd (uses gh_sram_1wp_2rp_sc.vhd for memory- there is an extra clock delay on the output data path)

6.2.2 Asynchronous FIFO

The Asynchronous FIFO uses Gray Code Counters (style #2, as defined in reference 1.) Binary counters are used to address the memory. A binary to Gray code converter uses a second set of registers, the output of which is synchronized with the other clock domain (this insures that a maximum of one bit will be changing at the time it is sampled).

The counters have an extra bit (above what is used to address the memory) for the flag generation. If the entire count values of both counters match, the FIFO is empty. If the memory address bits match and the extra bit do not, the FIFO is full.

Optionally (parts with the suffix _wf in the name), have a ¼ Full, Half Full, and ¾ Full Flags (Gary to binary converters are used in generating these flags).

I/O		Function
clk_WR	I	Clock for write port, active on rising edge
clk_RD	I	Clock for read port, active on rising edge
rst	I	Reset, active high – resets counters, flags
srst	I	Sync Reset, active high – resets counters, flags defaults to 0 (synchronous with clk_WR, internally re-synchronized to clk_RD)
WR	I	Write command, synchronous with clk_WR, advances the write counter after write
RD	I	Read command, synchronous with clk_RD, advances read counter to read next word
D(data_width -1 DOWNTO 0)	I	Input data
Q(data_width -1 DOWNTO 0)	O	Output data
empty	O	When low, output data is valid, synchronous with clk_RD
full	O	When low, WR is sampled for write command, synchronous with clk_WR
qfull	O	Quarter Full Flag, synchronous with clk_WR
hfull	O	Half Full Flag, synchronous with clk_WR
qqqfull	O	¾ Full Flag, synchronous with clk_WR

File names gh_fifo_async_sr.vhd, gh_fifo_async_sr_wf.vhd (uses FSAM style memory)
gh_fifo_async_rrd_sr.vhd, gh_fifo_async_rrd_sr_wf.vhd (there is an extra clock delay on the output data path for these parts)

Reference

2. Clifford E. Cummings, *Simulation and Synthesis Techniques for Asynchronous FIFO Design*, Revision 1.2 (June 2005), Sunburst Design
3. Clive "Max" Maxfield, *Bebop to the Boolean Boogie, Second Edition*, Newnes 2003 – page 361

6.2.3 Asynchronous FIFO's with UART Style Flags

These two FIFO's are similar to the that are used in the VHDL 16550 UART project with the following modifications:

1. These include generics for the size of the address bus for the internal memory.
2. FSAM is *not* used for the internal memory, so that it can be instantiated in block RAM. (there is an extra clock delay in the data path output - this will never be seen on the transmit side, and will not be seen on the receive side if one of the wrappers are used.)

I/O		Function
clk_WR	I	Clock for write port, active on rising edge
clk_RD	I	Clock for read port, active on rising edge
rst	I	Reset, active high – resets counters, flags
srst	I	Sync Reset, active high – resets counters, flags defaults to 0 (synchronous with clk_WR, internally re-synchronized to clk_RD)
WR	I	Write command, synchronous with clk_WR, advances the write counter after write
RD	I	Read command, synchronous with clk_RD, advances read counter to read next word
D(data_width -1 DOWNTO 0)	I	Input data
Q(data_width -1 DOWNTO 0)	O	Output data
empty	O	When low, output data is valid, synchronous with clk_RD
q_full	O	Quarter Full Flag, synchronous with clk_RD
h_full	O	Half Full Flag, synchronous with clk_RD
a_full	O	Almost Full Flag, synchronous with clk_RD
full	O	When low, WR is sampled for write command, synchronous with clk_WR

Parts – asynchronous FIFO's with UART style flags	clk_WR	clk_RD	rst	srst	WR	RD	D	Q	empty	q_full	h_full	a_full	full	Comments
gh_fifo_async_usrf.vhd	x	x	x	x	x	x	x	x	x	x	x	x	x	With Read Flags
gh_fifo_async_uswf.vhd	x	x	x	x	x	x	x	x	x				x	With Write Flags

6.3 Four Byte Dual Port RAM

These dual port RAM modules are intended for 32 bit processor systems that need lookup tables to perform some function. The A port is for processor access (read/write) while the B port is read only.

The processor bus (A port) is setup for 32 bit write access, with byte enables so byte access is possible. For the B port, there are modules for 32 bit lookup tables, 16 bit lookup tables and 8 bit lookup tables. For the 8 and 16 bit B port modules, big endian and little endian versions are savable. (The big and little endian modules differ only in the byte/word order of the data on the B port output.)

I/O		Function
A_clk	I	Clock, rising edge is for processer access
B_clk	I	Clock, for B (read only) port
CSn	I	Chip select, active low
WE	I	Write enable, active high (for write cycles)
BE(3 downto 0)	I	Byte enable (for write cycles) BE(3) for Data bits (31 downto 24) BE(2) for Data bits (23 downto 16) BE(1) for Data bits (15 downto 8) BE(0) for Data bits (7 downto 0)
A_add(size_add-[3,2, or1] downto 0)	I	Address lines for processor bus
B_add(size_add-1 downto 0)	I	Address lines for B port
D(31 DOWNTO 0)	I	Write Data (processor bus)
A_Q(31 DOWNTO 0)	O	Read data (processor bus)
B_Q(B data size -1 DOWNTO 0)	O	Read data, B port

Parts	Notes:
gh_4byte_dpram_x32.vhd	
gh_4byte_dpram_x16_be.vhd	
gh_4byte_dpram_x16_le.vhd	
gh_4byte_dpram_x8_be.vhd	
gh_4byte_dpram_x8_le.vhd	

7 Frequency Synthesis

7.1 The DDS (also known as the NCO, or DCO)

One of the most popular uses of the Accumulator is the DDS (Direct Digital Synthesizer), which is also know as the NCO (Numerically Controlled Oscillator) and the Digitally Controlled Oscillator. The output frequency is calculated using the following equation:

$$F_{out} = F_{clk} \frac{D}{2^{size}}$$

F_{out} = Frequency out
F_{clk} = Frequency of the Clock
D = value of the input data
Size = number of bits in the accumulator

The MSB will toggle at the output frequency. The frequency resolution can be calculated by setting D = 1, and solving for the output frequency. The accuracy of output frequency is controlled by the accuracy of the Clock used.

To reduce the output phase jitter to a sensible level, the 8, 10, 12, 14, or 16 MSB's of the Accumulator used as the address for a sin lookup PROM, which can be used to drive a Digital Analog Converter – and once filtered, will produce a nice sin wave. The CORDIC (a part that also shows up in this library) can also be used to generate a sin/cos pair.

DDS Examples

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
FREQ	I	Frequency data word
PHASE	I	Phase input, adjusts the of the output
sin	O	sin output
cos	O	cos output

Parts	CLK	rst	FREQ	PHASE	sin	cos	Comments
gh_nco.vhd	x	x	x		x	x	
gh_nco_a.vhd	x	x	x		x	x	uses gh_sincos_a.vhd
gh_nco_phase.vhd	x	x	x	x	x	x	Adds a phase adjust port
gh_nco_phase_a.vhd	x	x	x	x	x	x	uses gh_sincos_a.vhd

7.1.1 NCO Style Accumulators

This is a set of accumulators designed to be part of a NCO. They include generics for the size of the accumulator (A_size) and number of bits (size) for addressing a lookup table or CORDIC. Some of them include a phase port.

Some of the parts split the output (as well as the input phase ports, if applicable) into multiple paths, which is useful in high speed DSP systems.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
srst	I	Synchronous Reset, active high
FREQ	I	Frequency data word
Phase(x)	I	Phase input, adjusts the of the output
Q(x)	O	sin output

Parts	clk	rst	srst	freq	phase	q	Comments
gh_Freq_Acc.vhd	x	x	x	x		x	
gh_Freq_Accp.vhd	x	x	x	x	x	x	
gh_Freq_Acc2.vhd	x	x	x	x		x	
gh_Freq_Acc2p.vhd	x	x	x	x	2	2	
gh_Freq_Acc4.vhd	x	x	x	x		4	
gh_Freq_Acc4p.vhd	x	x	x	x	4	4	
gh_Freq_Acc8.vhd	x	x	x	x		8	
gh_Freq_Acc8p.vhd	x	x	x	x	8	8	
gh_Freq_Acc16.vhd	x	x	x	x		16	

7.2 Sweep Generator

Here is another example of using an accumulator. When a second accumulator is added before the DDS, the frequency will change, or sweep over time.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
min_freq	I	The minimum frequency – Must be less than max_freq for proper operation NOTE: value is in 2's complement form
max_freq	I	The maximum frequency – Must be greater than min_freq for proper operation NOTE: value is in 2's complement form
freq_step	I	The frequency change every sample clock NOTE: In 2's complement – a positive number will be an up sweep, and a negative value will produce a down sweep
LOAD	I	Will load the start freq (min_freq for an up sweep, max_freq for a down sweep) and hold the phase output to 0°
sw_en	I	Sweep enable, when disabled (= 0) output frequency is zero
ping_pong	I	When active (= 1) will sweep from end point to end point, and then sweep back. When disabled (= 0), will jump to back start
phase	O	The instantaneous phase output of the NCO
sin	O	Load data w/o accumulate
cos	O	Clock enable, active high
sweep_freq	O	Output sweep frequency, to drive an NCO
sweep_end	O	End of the sweep pattern

Parts	CLK	rst	min_freq	max_freq	freq_step	LOAD	phase	sin	cos	sweep_end	sw_en	ping_pong	sweep_freq	Comments
gh_sweep_generator.vhd	x	x	x	x	x	x		x	x	x				A top level example
gh_sweep_generator_a.vhd	x	x	x	x	x	x		x	x	x				Uses version A of sin_cos part
gh_frequency_sweep.vhd	x	x	x	x	x	x	x			x				The guts
gh_frequency_sweep_wpp	x	x	x	x	x	x				x	x	x	x	Ping pong version

The design is partitioned so that it will be easy to replace the CORDIC with either a sin lookup PROM or RAM. If a RAM is used, rather than sweep a sin wave, any wave form can be sweep (a ramp, a triangle, etc.).

$$SweepRate = F_{clk}^2 \frac{D}{2^{size}}$$

Sweep Rate = Rate of frequency change (hz/sec)
F_{clk} = Frequency of the Clock
D = value of the input data
Size = 64 – merge_point, or size of accumulator (for ping pong version)

The merge point is the bit in the NCO accumulator that meets the MSB of the sweep accumulator. (Note: the merge_point bits are labeled from 32 down to 1, while the standard logic vectors are numbered from 31 down to 0). Changing the merge point to a lower value will enable a slower sweep to be generated. For example, with a 100 MHz clock, and the default merge point (24) the minimum sweep rate is about 9.09 KHz/sec. If the merge point is changed to 22, the minimum sweep rate will be about 2.27 KHz/sec. In this example, if a minimum sweep rate of less than 1 Hz/sec is required, the merge point would have to be set to 10.

The ping pong version of the sweep generator does not include the NCO or the merge point generic. Instead, it has a frequency output port (size settable with a generic) which may drive a NCO, and generics to set the size of the sweep step port and the internal accumulator.

When setting the minimum and maximum frequencies, it has to be remembered that these values are in 2's complement. A negative frequency makes since in a I/Q (quadrature)

data path where a complex multiplier is used for signal mixing. See the second simulation for an example of a sweep that goes from a "negative" frequency to a positive frequency. In other systems, the output frequency will simply be an absolute value of the frequency generated.

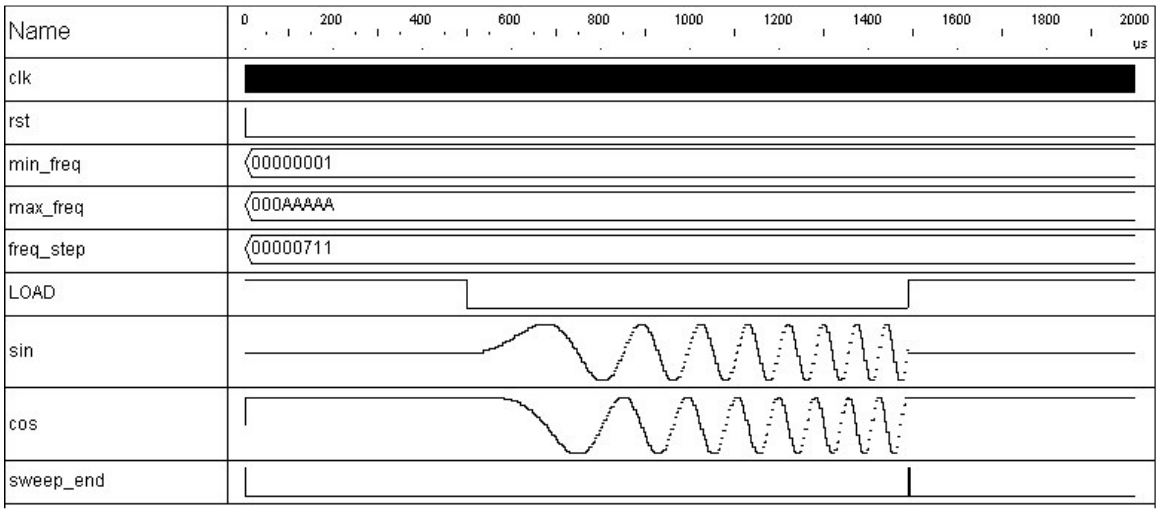
The given reference has some interesting comments on negative frequency (Section 8.4), which also includes a number of sections on quadrature systems.

Reference

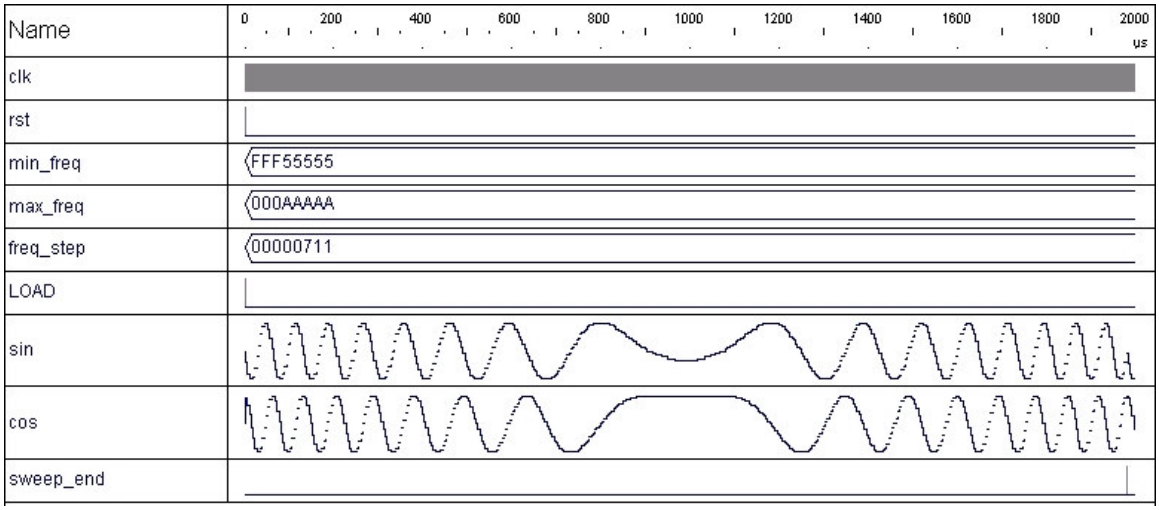
1. Richard G. Lyons, *Understanding Digital Signal Processing, Second Edition,* Prentice Hall, 2004

7.2.1 Simulation of the Sweep Generator

Here is an example of using the LOAD input for generating a pulsed sweep waveform- sometimes called a “chirp waveform”.



Here is a simulation illustrating a sweep through zero frequency. Notice that on the left half of the simulation, SIN is leading COS, while on the right half the COS is leading SIN. When driving one port of a complex mixer, one phase relationship will produce the sum of the two frequency’s, while the other phase relationship will produce the difference of the two frequency’s.



7.3 CORDIC Rotation Algorithm

In 1959, Jack E.Volder came up with a system that he called the Coordinate Rotation Digital Computer (better known as the CORDIC) rotation algorithm. It is a method for calculating trigonometry functions using only shift/adds (avoiding the need for hardware multipliers). Its most common use is polar to rectangular translation (sin and cos waveform generation) and rectangular to polar translation (to calculate magnitude and phase angle of a quadrature signal).

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
mode	I	Mode bit: 1 = rotation 0 = vectoring
x_in(size-1 downto 0)	I	X vector in
y_in(size-1 downto 0)	I	Y vector in
z_in(19 downto 0)	I	Z vector in
x_out(size-1 downto 0)	O	X vector out
y_out(size-1 downto 0)	O	Y vector out
z_out(19 downto 0)	O	Z vector out

generics	Function
size	Size of the X and Y vectors
iterations	The number of iterations the algorithm does

Parts	I/O				comments
	CLK	rst	mode	x_in y_in z_in x_out y_out z_out	
gh_cordic.vhd	x	x	x	x	A superset
gh_cordic_rotation.vhd	x	x		x	
gh_cordic_vectoring.vhd	x	x		x	
gh_cordic_rotation_28.vhd	x	x		x	Uses 28 bit atan function
gh_cordic_vectoring_28.vhd	x	x		x	Uses 28 bit atan function

7.3.1 Theory of the CORDIC

The theory of the CORDIC algorithm starts with the basic Vector Rotation Equation:

$$\begin{bmatrix} X' \\ Y' \end{bmatrix} = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \begin{bmatrix} X \\ Y \end{bmatrix}$$

For those who do not like matrix math, the equations look like this:

x' = x cosθ – y sinθ
y' = y cosθ + x sinθ

By removing the cosθ term we have:

x' = cosθ (x – y tanθ)
y' = cosθ (y + x tanθ)

The CORDIC algorithm performs the vector translation in an iterative process. Each iterative step uses successively smaller rotation angle:

$\theta_k = \tan^{-1}(2^{-k})$.

To achieve the simple shift/add in the rotation process, the magnitude of the vectors through the algorithm is not maintained (the cosθ gain factor is removed). However, since the cosine function is even, the gain is the same for both positive and negative rotation steps. The total gain depends only on the number of iterations performed.

Trigonometry shows that $\cos(\tan^{-1} x) = 1/\mathrm{sqrt}(1 + x^2)$, so the total gain becomes:

$Gn = \prod_{k=0}^{n} \sqrt{1 + 2^{-2k}}$ which becomes approximately 1.64676 for large n.

The preceding equations lead to the following equations for each iterative step:

$$x_k = x_{k-1} - \alpha y 2^{k-1}$$
$$y_k = y_{k-1} + \alpha x 2^{k-1}$$
$$\theta_k = \theta_{k-1} - \alpha \tan^{-1}(2^{k-1})$$
$$\alpha = (+1 \text{ for CW rotation}, -1 \text{ for CCW rotation})$$

The equation for θ_k is necessary to keep track of phase angle during the rotation.

The CORDIC algorithm has two basic modes:

Vector Rotation – rotates the vector (x,y) through the angle θ to create a new vector (x',y'). The sum of the iteration angles must equal the rotation angle θ:

$$\sum_{k=0}^{n} \theta - (\alpha \tan^{-1}(2^{-k})) = 0$$

In Vector Rotation, the value for α is based on the value of θ_{k-1}. To select α so that θ_k will converges towards zero, if θ_{k-1} is negative, α = -1, other wise it is +1.

Vector Translation – rotates the vector (x,y) around the circle until the y component equals zero. The output vector x is the magnitude (increased by the CORDIC gain) and the θ vector has the angle of the input vector. The sum of the y_k iterations must equal the input vector y.

$$\sum_{k=0}^{n} y_{in} - y_k = 0$$

In Vector Translation, the value for α is based on the value of y_{k-1}. If y_{k-1} is negative, α = +1, other wise it is -1. (The θ vector is not required if only the input vector magnitude is used.)

It should be pointed out that the CORDIC algorithm only works from $-\frac{\pi}{2}$ to $+\frac{\pi}{2}$, the range where the tangent function is continuous. (Well, $\tan(\frac{\pi}{2}) = \infty$, but the approximation of the algorithm is close enough to work.) To work over the full 2π range, the input has to be mapped to the range that the algorithm will accept, and the output has to be remapped to the correct quadrant.

7.3.2 Applications for the CORDIC

The gh_sincos.vhd and the gh_r_2_polar.vhd files use the generic size, which controls the data bus width and the iterations that the CORDIC uses. The A versions of these parts have increased pipelining. They will have more clock delays but will run at higher clock frequencies.

I/O for gh_sincos.vhd, gh_nsincos_28.vhd		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
add(size-1 downto 0)	I	Input Data
(n)sin(size-1 downto 0)	O	Sin wave output (28 bit atan version has negative sin output)
cos(size-1 downto 0)	O	Cos wave output

I/O for gh_r_2_polar.vhd, gh_r_2_polar_28.vhd		Function
CLK	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
x_in(size-1 downto 0)	I	X vector
y_in(size-1 downto 0)	I	Y vector
mag(size-1 downto 0)	O	Magnitude of complex vector
ang(size-1 downto 0)	O	Angle of complex vector

Note: The sin/cos signals are scaled to ≈full scale, while the gh_r_2_polar does not scale its output (the user has to take the CORDIC gain into account). [the input of the 28 bit atan version has some scaling in it, so that it will not overflow with an unscaled input]

References

1. Jack E. Volder, *The CORDIC Trigonometric Computing Technique*, IRE Transactions on Electronic Computers, September 1959.
2 Dean Groce, *The CORDIC Rotation Algorithm*, Unpublished Class paper (DSP with FPGAs), June 8, 2002
3. Xilinx Inc, CORDIC v2.0, LogiCore, March 28, 2003

7.4 Sin Cos ROM Lookup Tables

The CORDIC is one method of generating Sin/Cos wave forms. Another method is to use a lookup table. These lookup tables are set up as constants, rather than as a case statement, so that syntheses tools will implement them in block ram (at lest the tools from Altera and Xilinix).

I/O		Function
clk	I	Clock, rising edge is used
ADD(15 or 11 downto 0)	I	Frequency data word
(n)sin(15 or 11 downto 0)	O	(negative) sin output
cos(15 or 11 downto 0)	O	cos output

Parts	Comments
gh_sincos_rom_12.vhd	Full table, 1 clock delay
gh_nsincos_rom_12.vhd	Full table, 1 clock delay
gh_sincos_rom_12_2.vhd	Half size table, mapped to full pattern – 2 clock delay
gh_sincos_rom_16.vhd	Full table, 1 clock delay
gh_nsincos_rom_16.vhd	Full table, 1 clock delay
gh_sincos_rom_16_2.vhd	Half size table, mapped to full pattern – 2 clock delay
gh_sincos_rom_12_4.vhd	Quarter size table, mapped to full pattern – 3 clock delay
gh_nsincos_rom_12_4.vhd	Quarter size table, mapped to full pattern – 3 clock delay
gh_sincos_rom_14_4.vhd	Quarter size table, mapped to full pattern – 3 clock delay
gh_nsincos_rom_14_4.vhd	Quarter size table, mapped to full pattern – 3 clock delay
gh_sincos_rom_16_4.vhd	Quarter size table, mapped to full pattern – 3 clock delay
gh_nsincos_rom_16_4.vhd	Quarter size table, mapped to full pattern – 3 clock delay

8 Filters

8.1 CIC Filter

The CIC (Cascaded Integrator-Comb) Filter is a multirate filter for large changes in the sample rate.

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Asynchronous Reset, active high
D (data_in_size-1 DOWNTO 0)	I	Input Data
ND	I	Clock enable (for the differentiation section), active high
Q(data_out_size-1 DOWNTO 0)	O	Shifted bits out

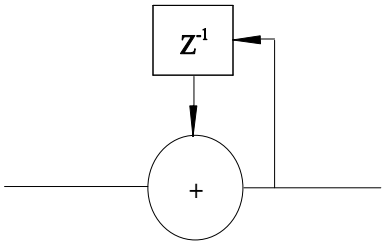
generics	Function
Data_in_size	Size of the input data bus
Data_out_size	Size of the output data bus
mode	mode 0 is decimation, mode 1 is interpolation
Stages	Number of stages (listed as N in formula's)
M	Either 1 or 2 (see theory section)

Parts	I/O					Generics					comments
	CLK	rst	D	ND	Q	data_in_size	data_out_size	mode	stages	M	
gh_CIC_filter.vhd	x	x	x	x	x	x	x	x	x	x	A superset
gh_CIC_interpolation.vhd	x	x	x	x	x	x	x		x	x	
gh_CIC_interpolation_m1.vhd	x	x	x	x	x	x	x		x		
gh_CIC_interpolation_m2.vhd	x	x	x	x	x	x	x		x		
gh_CIC_decimation_m1.vhd	x	x	x	x	x	x	x		x		
gh_CIC_decimation_m2.vhd	x	x	x	x	x	x	x		x		

The signal DCE must be high for one clock cycle every R clocks, where R the number of integration clocks for every differentiation clock.
Theory of the CIC Filter
The CIC filter is made from an equal number of two sections, the Integrator and the Differentiator (or Comb) sections. The filter is made using only registers, adders and subtracters, no multipliers are needed. The CIC filter has two forms, one used in for decimation, and the other for interpolation.

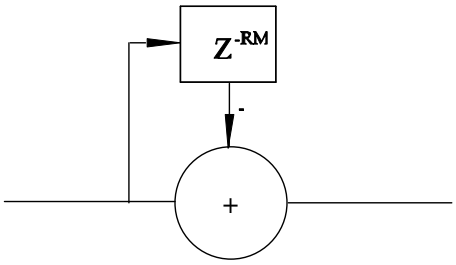
The transfer function for a single integrator section is:

$$H(z) = \frac{1}{1 - z^{-1}}$$



The transfer function for a single differentiator section is:

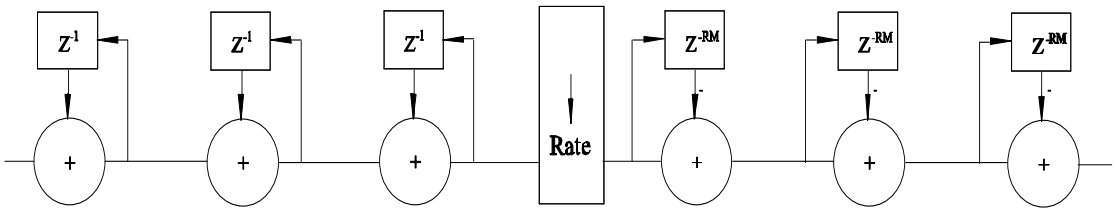
$$H(z) = 1 - z^{-RM}$$



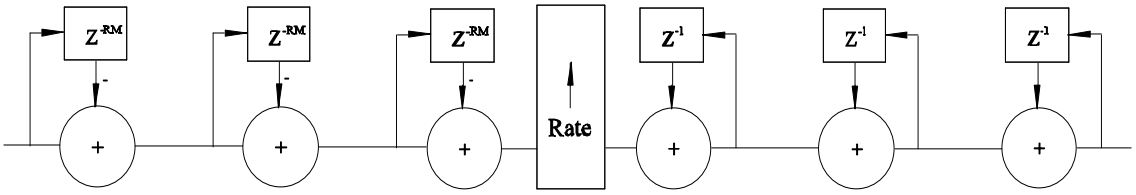
The transfer function for the CIC filters is (referenced to the higher sample rate):

$$H(z) = \frac{(1 - z^{-RM})^N}{(1 - z^{-1})^N}$$

M = number of register delays in the differentiator section.
R = data rate change between differentiator and integrator section.
N = number of stages in each of the two sections.



Three Stage Decimating CIC Filter



Three Stage Interpolation CIC Filter

In the Interpolation for of the CIC Filter, the output of the differentiator section is the input for the integrator section once every low rate sample clock, and zero's for the rest of the high speed sample clocks.

The gain for a CIC decimators is:

$$G = (RM)^N$$

For the CIC interpolator the gain is:

$$G = \begin{cases} 2 & i = 1, 2, ..., N \\ \frac{2^{2N-i}(RM)^{i-N}}{R} & i = N+1, ..., 2N \end{cases}$$

1. Matthew P. Donadio, *CIC Filter Introduction*, For Free Publication by Iowegian, July 18, 2000
2. Richard Lyons, *Understanding Cascaded Integrator-Comb Filters*, Courtesy of Embedded Systems Programming, March 31, 2005.
3. Xilinx Inc, *Cascaded Integrator-Comb (CIC) Filter V3.0*, LogiCore, March 14, 2002

8.2 Time–Varying Fractional Delay Filters

Fractional-Delay filters are a type of digital filter designed for bandlimited interpolation. Bandlimited interpolation is a technique for the evaluation a signal sample at an arbitrary point of time, even if it is located somewhere between two of the sample points.

The Fractional Delay Filter can delay a digital signal by an arbitrary time period, which can be used to align the phase of one signal with that of another. If the delay of the filter is changed over time, the output sample rate is modified, or (maintaining a constant sample rate) the output frequency can be shifted. One of the more popular applications of Time-Varying Fractional Delay Filters is for sample rate conversion.

I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Reset, active high
START	I	The TVFD sample rate (high for one clock period)
RATE(6 downto 0)	I	Instantaneous Fractional Delay (1-100 %)
L_IN(15 downto 0)	I	Left channel input data
R_IN(15 downto 0)	I	Right channel input data
coef_data(15 downto 0)	I	Filter Coefficient data from ROM
ND	O	Next data sample request
ROM_ADD ((modulo_bits + x -1) downto 0)	O	Address bus for the filter coefficient ROM
L_OUT(15 downto 0)	O	Left channel output data
R_OUT(15 downto 0)	O	Right channel output data

8.2.1 The Lagrange Interpolator

The Fractional Delay Filter is implemented using a FIR filter, setup as a Lagrange Interpolator.

The coefficients of the Lagrange interpolator are given by the following equation:

$$h(n) = \prod_{k=0,k\neq n}^{N} \frac{D-k}{n-k} \quad \text{for n = 0, 1, 2, …, } N$$

D = filter delay – see below for recommended rage of D (here $3.00 \leq D \leq 3.99$)
N = order of filter (this implantation uses N = 8)

Lagrange interpolators have a number of desirable characteristics:

1. Accurate model of the desired fractional delay
2. A lowpass filter with an almost flat magnitude response (the error gets bigger as the frequency increases)

3 The amplitude of the signal is never overestimated (magnitude gain ≤ 1) when the delay meets the following constraint:

$$\left(\frac{N-1}{2}\right) \le D \le \left(\frac{N+1}{2}\right) \text{ when N is odd}$$

$$\left(\frac{N}{2}-1\right) \le D \le \left(\frac{N}{2}+1\right) \text{ when N is even}$$

8.2.2 Time–Varying Control

The input data is stored in a circular buffer. With each new delay step eight consecutive data samples are multiplied with the corresponding filter coefficients for the desired fractional delay and added together. When the fractional delay step crosses an integer boundary, a new sample is loaded into the buffer.

8.2.3 TVFD Application Notes

The filter coefficients are stored in the file gh_tvfd_coef_prom.vhd. If the synthesis tool does not recognize the structure, and place it into a PROM (or RAM with an initialization file), it will consume a lot of resources.

The generics make it easy to modify the filter without editing the file. For example, if a 200 point Fractional Delay filter is used in place of the shown 100 point, a new coefficient ROM file is needed- set the modulo_bits generic to 8, and the modulo_count generic to 200. Bingo, you're done!!

There are now two versions of the TVFD filter:

gh_tvfd_filter.vhd - 16 bit data path width, 1 to 100% fractional delay range
gh_tvfd_filter_w.vhd – generics for data path width, 1 to 200% fractional delay range – other than the gh_MAC_ld.vhd, self contained

References

1. Vesa Valimaki, *Discrete-Time Modeling of Acoustic Tubes Using Fractional Delay Filers*, Dissertation for Doctor of Technology, Helsinki University of Technology, December 1995.
2. V. Valimaki and T. I. Laakso, *Principles of Fractional Delay Filters*, IEEE International Conference on Acoustics, Speech, and Signal Processing, Istanbul, Turkey, 5-9 June 2000
3. Siddharth Mathur, *Variable-Length Vocal Tract modeling for Speech Synthesis,* Master Thesis at The University of Arizona, 2003

.

8.3 A single MAC FIR Filter

The FIR Filter part of the TVFD filter was removed to make this part.

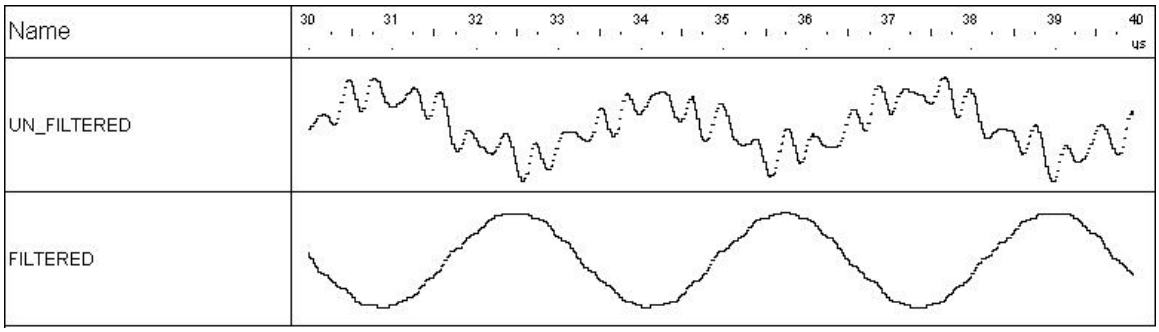
I/O		Function
CLK	I	Clock, rising edge is used
rst	I	Reset, active high
sample	I	The Filters sample rate (high for one clock period)
D_IN(15 downto 0)	I	Input data
coef_data(15 downto 0)	I	Filter Coefficient data from ROM (expects a two clock delay)
ROM_ADD(x-1)	O	Address bus for the filter coefficient ROM
D_OUT(15 downto 0)	O	Left channel output data

File name: gh_FIR_filter.vhd
gh_FIR_coef_prom.vhd (an example set of coefficients)
gh_FIR_filter_fg.vhd A version of the filter with generics

The FIR Filter has the generic x, which sets the order of the filter:
Filter order = 2^x

Zero's can be used in the coefficient PROM to get a filter order less than 2^x. The input "sample" must have a period at lest 2^x times greater than the period for the input "CLK".

Here is a simulation of the FIR Filter. Note: a CIC interpolation filter was used in the test circuit to increase the number of samples in the plot.



8.4 Symmetrical, parallel FIR Filters

These FIR Filters use a transposed parallel structure, giving them a data rate equal to the clock rate (unless the clock enable is used to slow the data rate). They make use of symmetry, so that only half as many multipliers are needed.

To help minimize round off, fractional bits are available (settable with a generic). These are bits in the adder chain, below those that become the output data. Using some of the "extra" bits out of the multipliers will minimize round off errors.

The filters include an over flow limiter at the output – this will limit overflow from ringing in a step response, but may not stop overflow if the coefficients are too large.

The filter coefficients are input as one large vector. The use of the configuration registers (see section 2.9) will make them easily to modify with software. The first (and last) coefficients in the data path are bits 15 down to 0. The top 16 bits are used by the center tap(s).

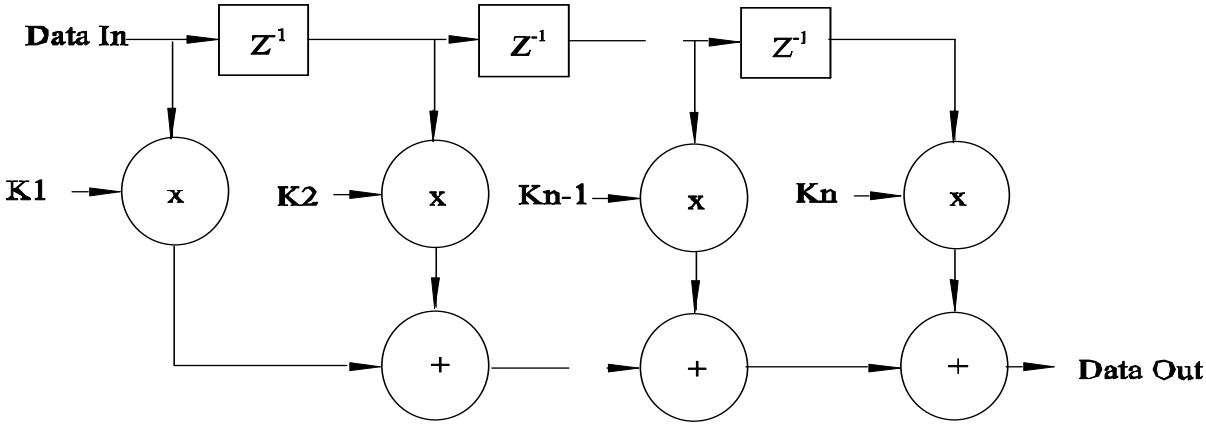
These filters have the generics "d_size", "coef_size" and "half_tap_size" (which sets the number of filter taps [which is 2 * half_size]).

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Reset, active high
ce	I	Clock enable
D (15 downto 0)	I	Input data (signed)
coef(16 * half_tap_size -1 downto 0)	I	Filter Coefficient data (signed)
Q(15 downto 0)	O	output data

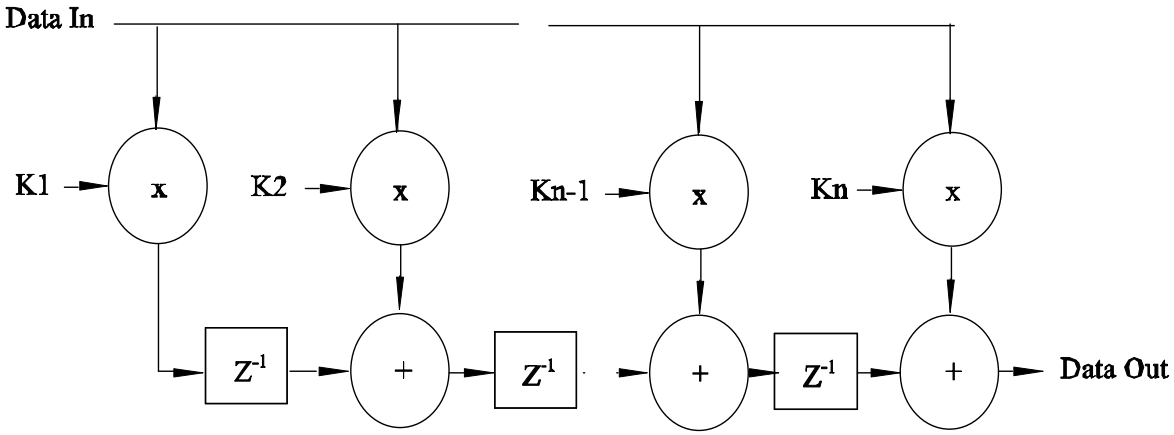
generics	Function
d_size	number of bits in data path
coef_size	number of bits in each coefficients words
fract_bits	fractional bits – used to minimize round off errors
half_tap_size	half the number of taps (for odd order filters, number of taps = 2 * half_tap_size + 1)

Parts	comments
gh_FIR pfilter.vhd	even number of taps, positive symmetry
gh_FIR pfilter_ns.vhd	even number of taps, negative symmetry
gh_FIR pfilter_ot.vhd	odd number of taps, positive symmetry
gh_FIR pfilter_ot_ns.vhd	odd number of taps, negative symmetry - Center tap is always zero, to maintain negative symmetry
gh_fir_pfilter_nsc.vhd	A non symmetrical version - uses a multiplier for every tap The top coefficients create the first outputs

8.4.1 FIR Filter Architecture



Standard FIR Filter Architecture



Transposed FIR Filter Architecture

I should be noted that for a symmetrical FIR Filter, Kn = K1, Kn-1 = K2… and for negative symmetry, Kn = -K1, Kn-1 = -K2…

8.5 FIR Filters Without Multipliers

Since multiplication and division of powers of two is simply a data bit shift. This makes coefficients of 0.5, 0.25, 0.125, 0.0625 etc easy to use. Additional coefficients, such as 0.75 are the summation of different shifted numbers (0.75 = 0.5 + 0.25)

With enough shift and adds, any desired coefficient can be calculated. However, too many shifts and adds will inspire most just to use a filter with multipliers.

The compensation filters are high pass filters, where the change in gain (from DC to Nyquist) is part of the file name. A close examination of the interpolation filter will show its limitations – it is only useful for narrow bandwidth signals (- 3 dB point is at about 15% of the sample rate, while at 44% of the sample rate has a gain -30 dB. -80 dB is achieved by 49.53% of sample rate).

I/O		Function
clk	I	Clock, rising edge is used
rst	I	Reset, active high
ce	I	Clock enable
D (size-1 downto 0)	I	Input data (signed)
Q(size-1 downto 0)	O	output data

Parts	comments
gh_filter_compensation_2dB.vhd	filter described on page 407, 408 of Richard G. Lyons's, *Understanding Digital Signal Processing* Note: has a gain greater than 1 coefficients = [-0.0625 1.125 -0.0625]
gh_filter_compensation_4dB.vhd	coefficients = [-0.09375 0.8125 -0.09375]
gh_filter_compensation_6dB.vhd	coefficients = [-0.125 0.75 -0.125]
gh_filter_AB_interpolation.vhd	coefficients = [0.03125 0.5 0.9375 0.5 0.03125]

Reference

1. Richard G. Lyons, *Understanding Digital Signal Processing, Second Edition,* Prentice Hall, 2004

9 VMEbus [VXIbus] Interface Modules

The VMEbus, in use for over 26 years, is ancient in computer years. Yet, it still finds use in harsh and mission-critical environments. It is an open architecture and custom cards are easy to design for it. Although the buss has had a number of upgrades over the years, backwards compatibility has been rigorously defended along the way.

The VXIbus is the VMEbus Extensions for Instrumentation. The VXIbus has additional requirements above and beyond what the VMEbus requires. But, standard VMEbus cards may be used in VXIbus systems.

The VMEbus slave modules can be used to interface the VMEbus with the 4 byte configuration registers, control registers, and/or dual port ram. The MUX, required if multiple blocks are to be read by the VMEbus, is left as an exercise to the user.

9.1 VME Slave Modules

I/O		Function
clk	I	Clock, rising edge is used
RESn	I	VME SYSRESET* signal, active low
CRDSn	I	Card select, decode of MSB address bits
WRITEn	I	VME signal
IACKn	I	VME signal
ASn	I	VME signal
AM(5 downto 0)	I	VME signal
LWORDn	I	VME signal
DS0n	I	VME signal
DS1n	I	VME signal
Vadd(add_size-1 downto 0)	I	VME signal
LD_IN(31 downto 0)	I	Local Data bus data In
L_ACK	I	Local acknowledge signal (if low, will add wait states until driven high)
VD(31 downto 0)	I/O	VME Data Bus
BRDSLn	O	Local Board Select, active low
rst	O	local reset, active high
WR	O	local Write strobe, active high
DTACKn	O	VME signal
VD_ENn	O	VME Data buffer output enable
VD_DIR	O	VME Data buffer Direction control
BE(3 downto 0)	O	Local Byte Enables BE(3) for Data bits (31 downto 24) BE(2) for Data bits (23 downto 16) BE(1) for Data bits (15 downto 8) BE(0) for Data bits (7 downto 0)
LA(add_size-1 downto 0)	O	Local Address bus
LD_OUT(31 downto 0)	O	Local Data Out

additional I/O for modules with interrupts		Function
IACK_INn	I	VME signal
g_IRQ[A,B,C,D]	I	generate interrupt, active rising edge
IRQ_L[A,B,C,D] (2 downto 0)	I	Interrupt level, from 1 to 6 (does not support level 7)
IRQ_V[A,B,C,D] (7 downto 0)	I	Interrupt Vector
IACK_OUTn	O	VME signal
IRQn(6 downto 1)	O	VME Interrupt signals (can only generate interrupts on levels 1 through 6)

Parts	comments
gh_vme_slave_a16_d16.vhd	allows word, byte access
gh_vme_slave_a24_d16.vhd	allows word, byte access
gh_vme_slave_a32.vhd	allows long word, word, byte access
gh_vme_slave_a32_wi1.vhd	allows long word, word, byte access
gh_vme_slave_a32_wi4.vhd	allows long word, word, byte access

Design Notes:

1. Data transfers must be aligned (i.e. Long Words must be on long word [32 bit] boundaries, word transfers must be on word [16 bit] boundaries).
2. The upper address lines are not on the Slave modules – it is expected that they will be compared (outside the module) with the board select dip switches and the active low of the compare to drive the Card Select (CRDSn) module pin. The number of lower address bits that the Slave Module uses is selectable with generics – by default, it is expected that the upper eight address lines will be used for the card select decode.
3. Supervisory data Access and Non-Privileged Data Access are the address modifiers accepted for a data transfer.
4. Block Transfers are not supported.
5. The drive for the Open Collector outputs (DTACKn, IRQn[6-1] are set up to drive the output enable of a tri-state buffer (such as the 74ABT125) witch will act as an Open Collector output. Using 74ABT125 buffers makes the modification to use Rescinding DTACKn (as recommended in the VXIbus, for example) easier.
6. The designer must remember to take into account the delay of any buffers between the FPGA/ASIC and the VMEbus when verifying the bus timing on read cycles. If the DTACKn buffer and the Data buffers have the same delay, the read cycle will have a timing margin of one clock period.

References

1. Motorola, *The VMEbus Specification, Revision C.1,* October 1985
2. VXIbus Consortium, *VMEbus Extensions for Instrumentation System Specification,* Inc., Revision 3.0 November 24, 2003
3. Secretariat VMEbus International Trade Association, *American National Standard for VME64 [ANSI/VITA 1-1994],* April 10, 1995
4. Secretariat VMEbus International Trade Association, *American National Standard for VME64 Extensions [ANSI/VITA 1.1-1997],* October 7, 1998

9.2 VME Chip Select Modules

The Slave Modules will expand the single local chip select from a VMEbus Slave module, and with the local address lines, generate up to 20 chip selects – the address range for each of them are set with generics.

I/O		Function
CRDSn	I	Local Card Select, active low
Ladd	I	Local Address Bus (size varies with part)
CSn (19 downto 0)	O	Local Chip Selects (20), active low

Parts	comments
gh_vme_cs20lw_28a.vhd	uses long word addressing, for 28 (byte) address lines
gh_vme_cs20lw_24a.vhd	uses long word addressing, for 24 (byte) address lines
gh_vme_cs20w_20a.vhd	uses word addressing, for 20 (byte) address lines
gh_vme_cs20w_16a.vhd	uses word addressing, for 16 (byte) address lines
gh_vme_cs20w_12a.vhd	uses word addressing, for 12 (byte) address lines

9.3 VME Read Modules

The VME read Modules are basically a custom mux to aid designing the read side of the interface. The Chip Select input bus is easy to interface with the Chip Select Module, and each of the data inputs have a generic to set the input data size - the leading, unused data bits will be set to zero.

I/O		Function
CSn (x downto 0)	I	Local Chip Selects (20), active low
RDx(CSx_dsize-1 downto 0)	I	Local Address Bus (size varies with part)
DATA_o(31 or 15 downto 0)	O	Local Chip Selects (20), active low

x depends on which module is used (number of words - 1)

Parts	comments
gh_vme_read_20lw.vhd	expects 20 data words, each 32 bits (max)
gh_vme_read_10lw.vhd	expects 10 data words, each 32 bits (max)
gh_vme_read_5lw.vhd	expects 5 data words, each 32 bits (max)
gh_vme_read_20w.vhd	expects 20 data words, each 16 bits (max)
gh_vme_read_10w.vhd	expects 10 data words, each 16 bits (max)
gh_vme_read_5w.vhd	expects 5 data words, each 16 bits (max)

10 Library Notes

It seems egotistical to add gh_ to the name of the parts in this library. However, would it be less egotistical to think that this library will be the only used in a design?

As noted by Jiri Gaisler (of Gaisler Research), in his paper "A Dual-Use Open Source VHDL IP library"

> A common, and often challenging, design tasks during SOC development is to integrate a number of third-party IP cores into a single design… Other issues include resolving of name clashes… each IP vendor is assigned a unique library name.

While integrating multiple libraries into a signal project can be problematic, it is hoped that the gh_ prefix is unique, making it easy for parts in this library to included in your next project.