Implementation of the Asynchronous Pulse Blanker

Grant Hampson

June 26, 2002

Introduction

This document describes an Altera FPGA design and simulation of the Asynchronous Pulse Blanker (APB) component of the IIP Radiometer RFI processor described in [1] and more recently in [2]. A generalized block diagram of the APB is shown in Figure 1. There are three significant blocks to the APB. The FIFO provides the delay for pre-detection blanking as well as a processing latency buffer. The APB processor is responsible for calculating the mean and variance of the input data and to determine if blanking is required. The interface provides the ability to program the APB using a micro-controller. The controller is responsible for the sequencing of the APB processor. It would be desirable if the APB could process 100% of the input bandwidth (100 MSPS.)

This document is broken into three sections. The first section describes the APB processor. Secondly, the Rabbit control interface is described. The controller state machines are illustrated in Section 3. Several appendices list source and simulation code.

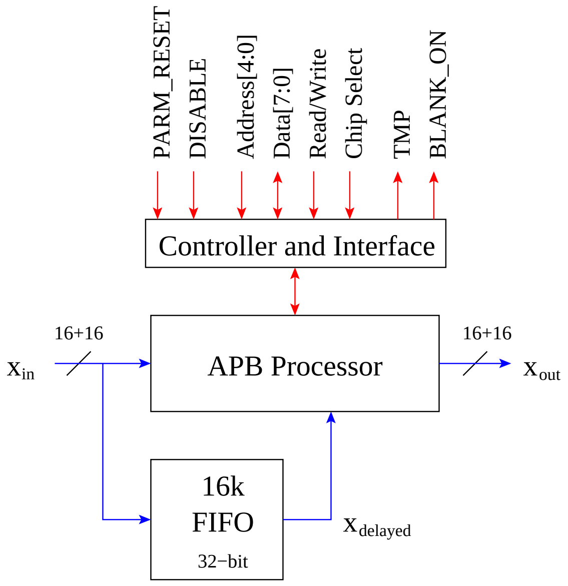


Figure 1: Block diagram of the APB. Complex data input width (N) can be from 8 to 16-bits. Output data has identical resolution to the input data. The processor works on input data and upon detection of interference will zero the output. A microprocessor interface is provided control and monitor the APB.

1 APB Processor

The APB processing component has been previously described using pseudo code in [2]. An interface diagram of the APB processor is shown in Figure 2, where two parameters are introduced. The first is N, the width of the processed data path, and L, the datapath width of β^2, μ_{mean} and μ_{var}. When `load_parm` is asserted the values of `mean_reset` and `var_reset` are loaded and processor calculates the mean and (if the `update_enable` is asserted) the variance of x. The mean (`meanx`) and variance (`varx`) are updated when `clock_enable` is asserted. If a pulse is detected the `blank` output is asserted.

Note that the variables μ_{mean} and μ_{var} are implemented as constants to reduce size and increase speed of the processor. Table 1 lists the corresponding bit widths and time constants. The larger the bit width the slower the output varies with time. Refer to [3] for more information on exponentially weighted filters.

The size of the processor is mainly effected by the input bit width N. The processor is implemented using fixed point arithmetic (since these libraries are freely available from Altera.) A block diagram of the APB processor is shown in Figure 3, for which the source code is listed in Appendix B.

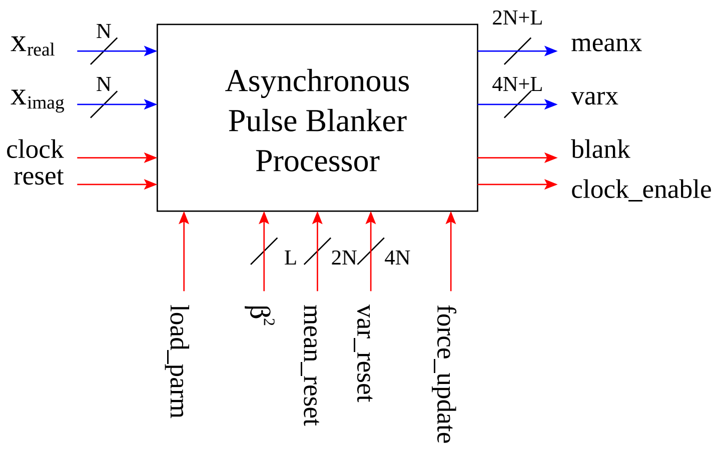


Figure 2: An interface block diagram of the APB processor.

Table 1: Maximum time constant values for a given bit width L. For example, when L=12 then $\mu_{mean} = \mu_{var} = \frac{4095}{4096} = 0.999756$

L-bits	$(2^L) - 1/2^L$	L-bits	$(2^L) - 1/2^L$
1	0.500000	9	0.998047
2	0.750000	10	0.999023
3	0.875000	11	0.999512
4	0.937500	12	0.999756
5	0.968750	13	0.999878
6	0.984375	14	0.999939
7	0.992188	15	0.999969
8	0.996094	16	0.999985

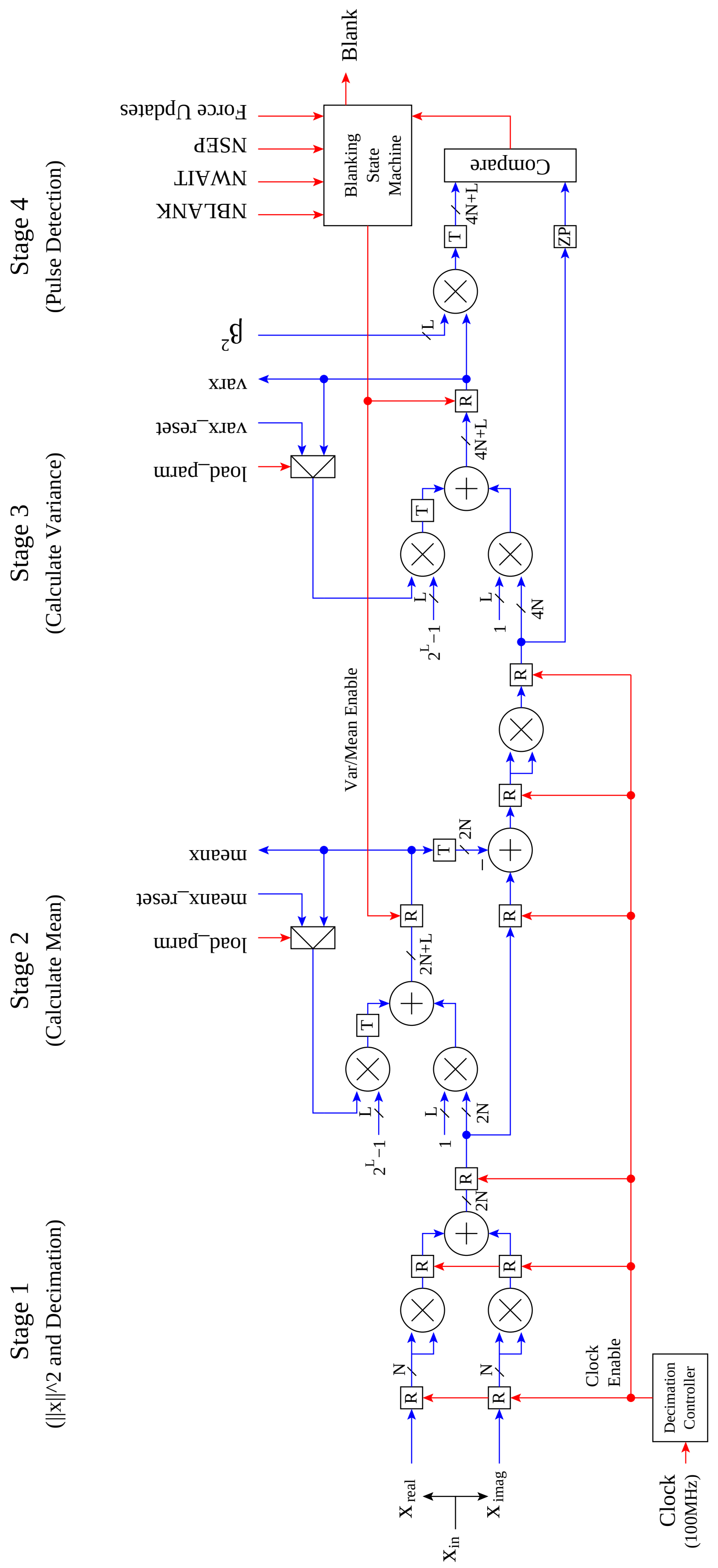


Figure 3: A computational block diagram of the APB processor. Note that ’R’ represents a register delay, ’T’ represents a truncation, and ’ZP’ a zero pad of L-MSBs. All stages operated at 100MHz, however registers are only updated according to clock enable generated by the decimator controller. All arithmetic operations are fixed point.

The mean and variance output data paths are very wide ($2N + L$ and $4N + L$, respectively) due to two squaring operations in the processor. Squaring operations are inherently dynamic range hungry operations. For this fixed point arithmetic processor it is critical that the dynamic range is preserved and no intermediate truncations occur.

An implementation using floating point arithmetic (e.g., Matlab with 64-bit double precision) will have no difficulties. However, reducing the number of bits (since we have 13 parallel multiply/add/compare operations) in the floating point word will have catastrophic results. The reason is that while floating point numbers have sufficient dynamic range (large exponents) it is also important to have a large number of significant digits in the mantissa. Consider adding a very small number (e.g. $(1 - \mu_{mean}) \times ||x||^2$) and very large number (e.g. $\mu_{mean} \times m_x$). If during decimal point alignment there are not enough significant digits the very small number is considered to be zero, which is not the desired result.

Consequently, the processor is implemented using fixed point arithmetic with truncation only occurring in places where it is non-critical. An implementation in floating arithmetic would require libraries to be created for data conversion, addition, multiplication, and comparison which wouldn't be cost effective and computationally advantageous.

1.1 APB Processor Implementation Results

Due to the large bit growth in the processor it is unfortunate that the processor will not be able to process 100% of the input data. Consequently, a decimation controller is introduced to sub-sample the input data stream (i.e., not decimation in the strict sense.) Current compilations of the design indicate that one in four samples, or 25%, of the input data can be processed. Measurements of radar pulses [4] indicate that the radar pulse lengths are 100 clock cycles (at 100MHz) long. Processing 25% of the data should not present any problems. It takes $7 \times 4 = 28$ clock cycles from input data to pulse detection (which can be accounted for in the blanking timing registers.)

The slowest part of the design is the final compare between the variance and the incoming data. For example, if $N = L = 12$ bits then the final compare is 60-bits wide, which takes considerable time to compute. Table 2 lists a few input data widths and the corresponding processor size and speeds for $L = 12 \equiv \mu_{mean} = \mu_{var} = 0.99975$.

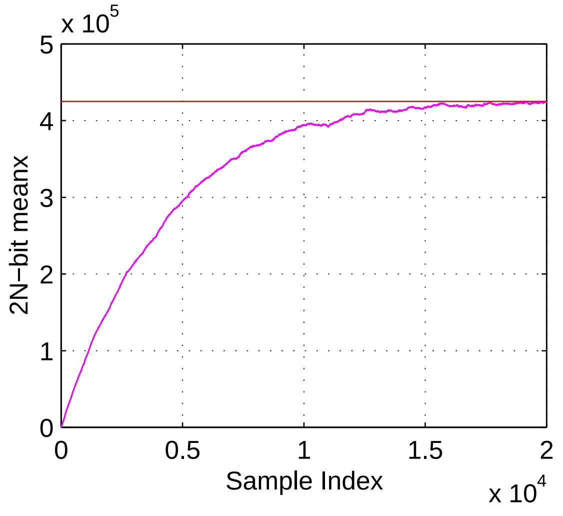
The processor size and speed is not effected greatly by the parameter L. For example, when N=12 and L=16 the resulting size is 2918LE (70%) and runs at 26MHz. Both the mean and the variance contain a fractional part which is L-bits wide.

Table 2: Various compilations results for the APB processor. The target FPGA is a $150 Altera EP20KE100QC240-1. This is the fastest FPGA in its size (4160 logic elements.)

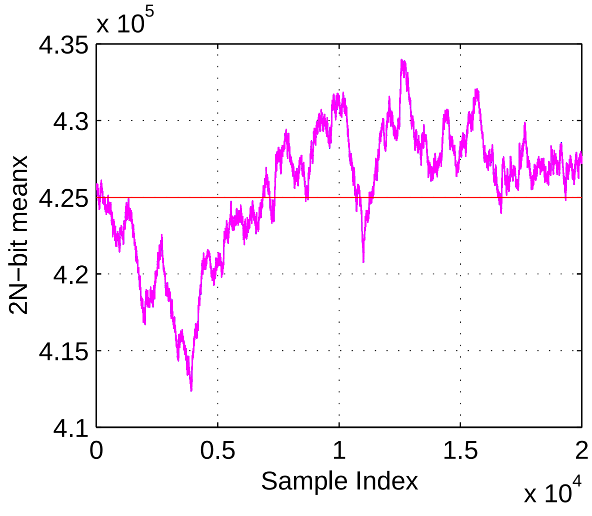
N-bits	Logic Elements	% of 4160	Max. Speed (MHz)
8	1480	35%	29.5
10	1946	46%	27.1
12	2483	59%	28.5
14	3061	74%	23.2
16	4042	97%	20.0

1.2 APB Processor Simulation

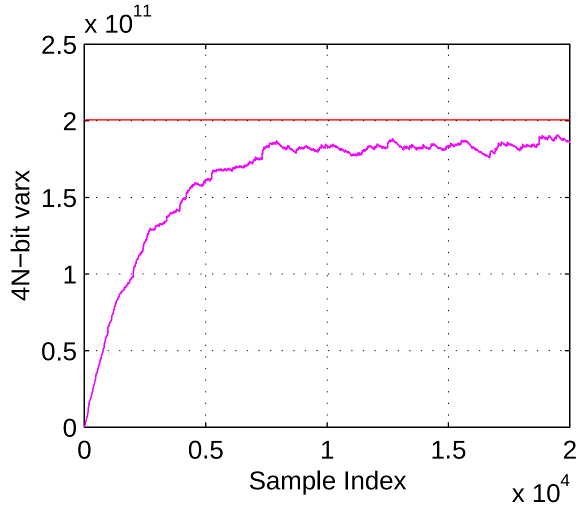
The APB processor was first implemented in Matlab to check implementation details - both in floating point and fixed point arithmetic. Appendix B lists the source code to the fixed point version of the code. Figure 4 illustrates some results from the Matlab source for different initial conditions. The exponential weighting factor used here is $L = 12 \equiv 0.99975$ which results in convergence after 20000 snapshots, or $800\mu s$ (assuming a decimation of 4.) It is interesting to note the amount of variation in the result after convergence. Table 3 lists the percentage of variation for both the mean and variance. The variance has a greater degree of variation, which could be possibly due to the fact that it is proportional to x^4 (the mean is proportional to x^2.) To achieve less than 1% variation L should be set to 15 or 16.



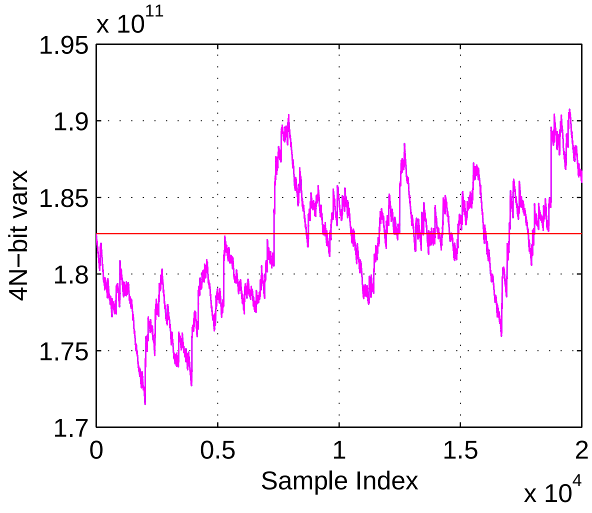
(a) Mean with 0 start



(b) Converged Mean



(c) Variance with 0 start



(d) Converged Variance

Figure 4: Convergence curves for different starting conditions. (a&c) start from zero, and (b&d) have the initial mean and variance set to the Matlab calculated value. (L=12)

Table 3: Convergence oscillations for various values of L. Percentage variation is calculated as is 100 × (max(m_x) − min(m_x))/mean(x). Statistics are based on N=12.

L-bits	mu_{mean}, μ_{var}	% variation in mean	% variation in variance
9	0.99804	16%	62%
10	0.99902	13%	35%
11	0.99951	7%	17%
12	0.99975	4%	13%
13	0.99987	2%	6%
14	0.99993	1%	3%
15	0.99996	0.5%	1.0%
16	0.99998	0.3%	0.9%

Given that the output mean and variance data paths are $2N + L$ and $4N + L$ bits wide it is of interest to see the performance (accuracy) of the algorithm over this range. Figure 5 illustrates the output of algorithm, as compared to the Matlab computed mean and variance. This graph is plotted as a function of the input bits occupied in x. For large values of x the mean and variance is very accurate, however for small x the mean and variance approach zero.

It was expected that the L-fractional bits would be sufficient to represent smaller means and variances - but for some reason this is not the case and the values equal zero. This will require more research to understand this fault, however it is assumed that some RFI and background noise will be always present and consequently a number of bits will be toggling. Additionally, the output of the Digital IF processor [5] is 16-bits wide and it will be possible to process 12-bits of this (connected to MSBs) in the current FPGA size (this can be expanded to a $250 FPGA.) Experimentation will be required to determine if these assumptions are valid.

If the variance is zero then $\beta^2\times$ variance will also equal zero. Consequently, x^2 will be larger than zero and the blanking output will remain on. This is not the desired result. Additional logic could be implemented to detect this *quiet* scenario.

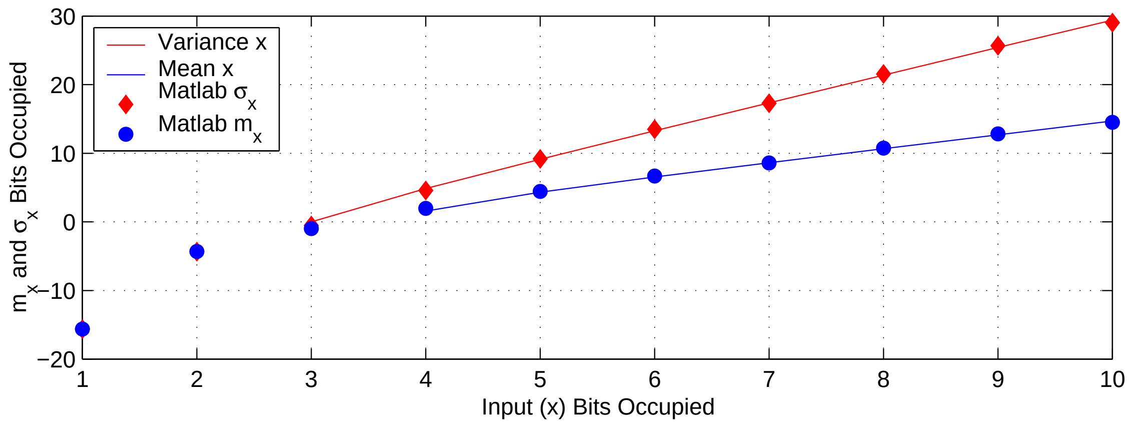


Figure 5: Mean and variance dynamic ranges. Results from both Matlab (floating point) and from the fixed-point algorithm.

The final test of the APB processor is to apply Gaussian noise and measure the percentage of time the processor is blanking for various values of β. The results of this experiment are listed in Table 4, where the same data set was used for each value of β. In this experiment the variance is always updated.

Table 4: Percentage of a Gaussian input blanked for various values of β. Parameters are $N = L = 12$, 20000 samples after convergence, always updating variance.

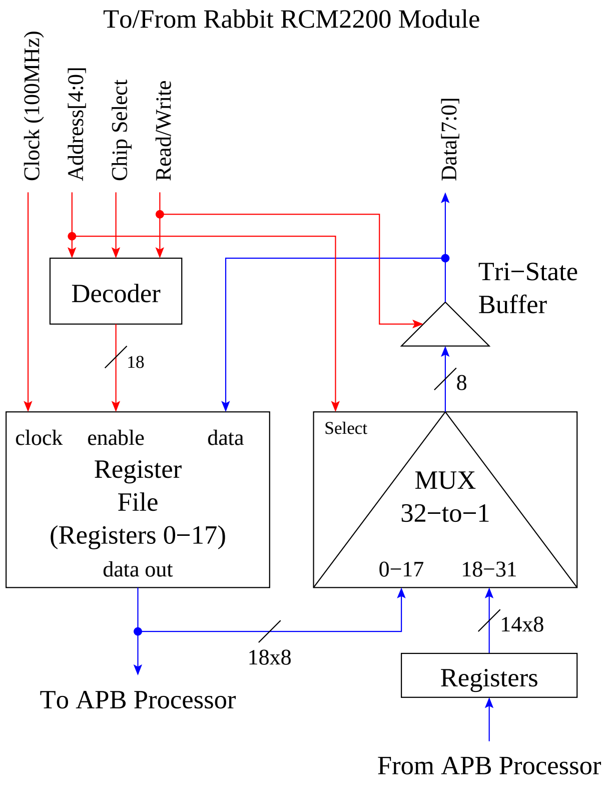
β	% blanked
1	13.4%
2	4.84%
3	1.91%
4	0.74%
5	0.33%
6	0.13%
7	0.05%
8	0.01%

2 APB Interface

The APB processor requires a certain degree of control which is achieved by implementing a microprocessor interface containing many read/writable registers. The microprocessor which will interface to the processor has been chosen to be a Rabbit RCM2200 controller [6]. The list of interface registers is listing in Table 5. Some of the registers can be written to and read back for verification. N and L will determine the size of some registers, for which there is a maximum size. If the register is larger than the maximum size then

Table 5: Interface register allocation table and descriptions. Each register is 8-bits wide as the Rabbit is an 8-bit micro-controller. Registers which are wider than 8-bits occupy more than one register.

Address	Name	R/W	Width (bits)	Max Bits	Description
0	Control Write	R/W	8	8	Control Register
1-2	Beta Squared	R/W	L	16	Blanking threshold
3-5	Mean Reset	R/W	2N	24	Initial mean
6-11	Variance Reset	R/W	4N	48	Initial variance
12-13	NWAIT	R/W	16	16	Cycles before blanking
14-15	NBLANK	R/W	16	16	Cycles to blank
16-17	NSEP	R/W	16	16	Cycles between blanks
18	Control Read	R	8	8	Feedback Register
19-23	Mean x	R	2N+L	40	Current Mean of x
24-31	Variance x	R	4N+L	64	Current Variance of x



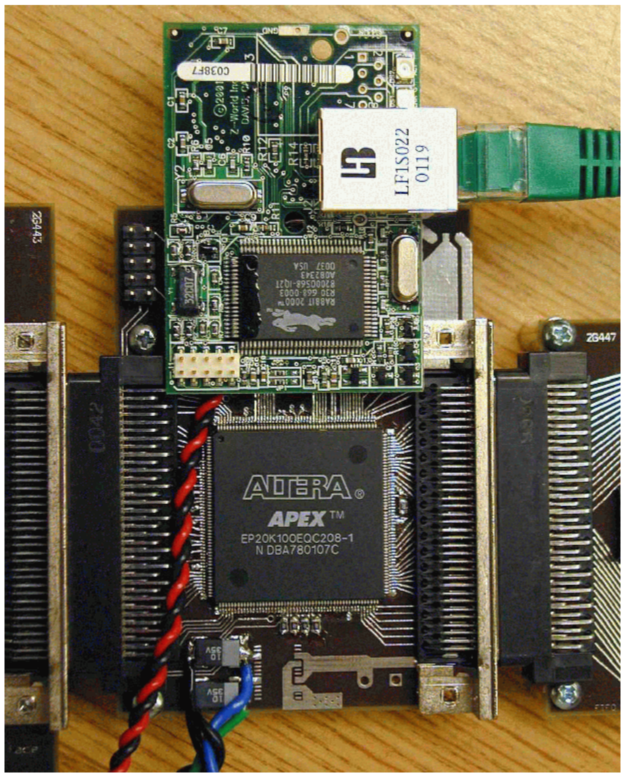


Figure 6: (top) A block diagram of the APB processor Rabbit interface. 18 control registers (which are read/writable) control the processor. 14 registers from the processor provide feedback (these registers are updated using a bit in the Control Write register.) (bottom) Photograph of the Rabbit processor and the FPGA on which the preliminary pulse blanker is implemented.

only MSBs will be read/written. Figure 6 illustrates the hardware required to implement the register bank. The hardware is limited to a register file and a large multiplexer. The number of logic elements required to implement the design is approximately 466. The source code for the controller is listed in Appendix C. Table 6 lists preliminary control register allocations. Given that the control interface is relatively low bandwidth, not many control bits are required to control the APB.

Table 6: Interface control register descriptions.

Register	Control Read	Control Write
bit-0	APB_TMP	APB_MODE_0
bit-1		APB_MODE_1
bit-2		FORCE_UPDATE

The APB is controlled via the Rabbit processor. The Rabbit processor is equipped with a TCP/IP ethernet connection which allows control to originate from any connection on the network. A TCP/IP packet has the possibility for a payload of 2048bytes, which is far greater than the 32-bytes required for the APB.

Currently, it is possible to transmit a TCP/IP packet from a LAB Windows CVI client, to the Rabbit which is configured as a server. The Rabbit has an IP address of 128.146.90.109 using port 7. When a packet arrives at this port the Rabbit software simply writes the 32-bytes to all the APB registers. The building blocks required for the ethernet based control are shown in Figure 7.

The software could be improved to only accept data from particular clients so as to improve security. Additionally, the server could reply with data read from the output registers of the APB if required.

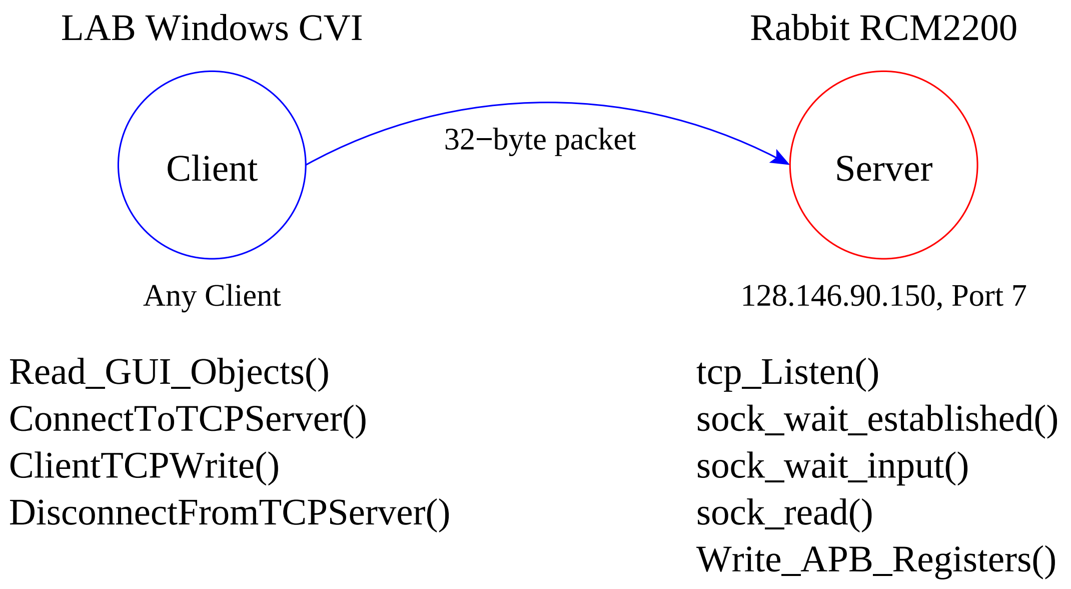


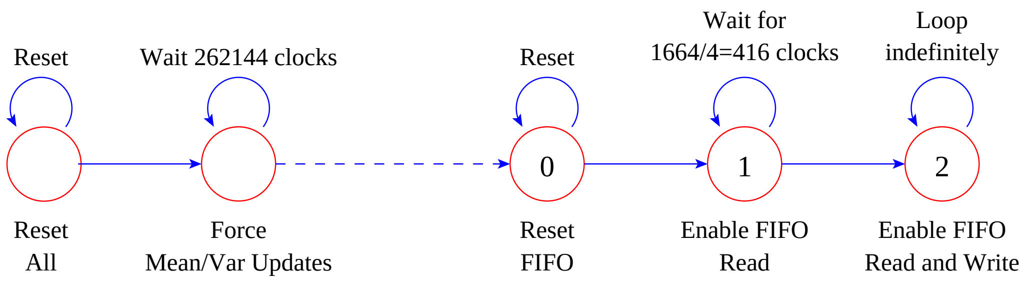
Figure 7: Remote TCP/IP control mechanism for a LAB Windows CVI client and a Rabbit server. The fundamental TCP/IP functions used are also listed.

3 APB Controllers

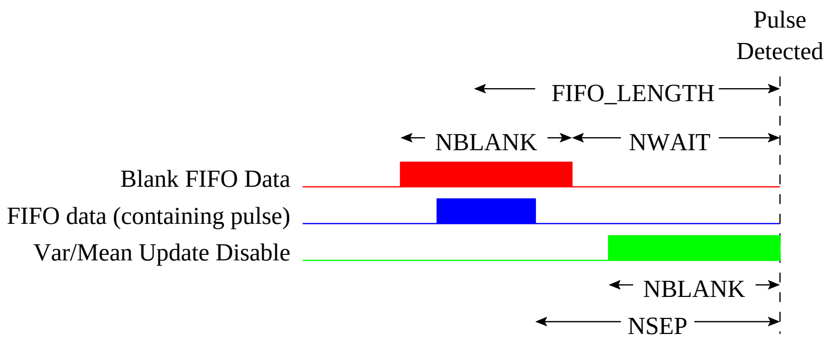
There are two fundamental controllers in the pulse blanking engine. The first is to fill the FIFO to a known length during initialization so that `NWAIT` has a reference (`NWAIT<FIFO_LENGTH`.) The `FIFO_LENGTH` determines the amount of data preceding the pulse that can be blanked. The FIFO is also required to account for processing latency, for which this equals $7 \times 4 = 28$ clock cycles when the decimation is 4.

The second state machine is a `BTR`, or blanking-timing-register [2] which has been shown previously in Figure 3. In the preliminary implementation of this state machine there will only be a single BTR as the radar pulses expected are significantly far apart [4].

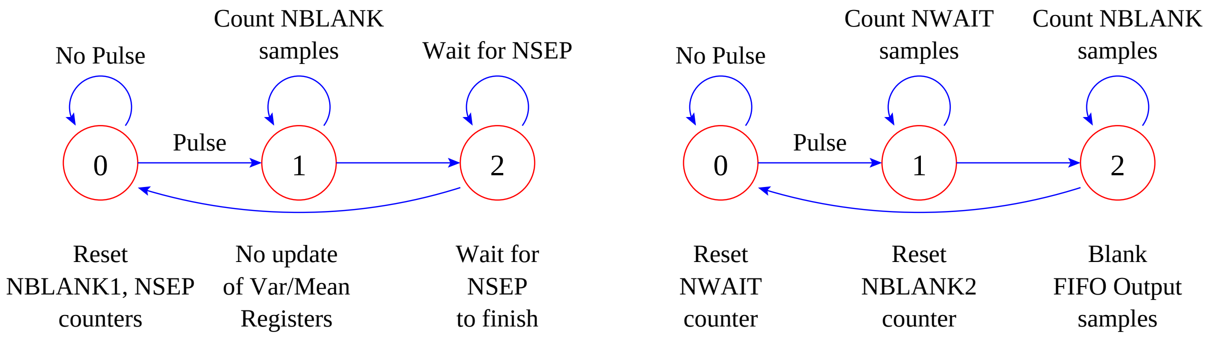
Figure 8 illustrates these two preliminary state machines. These state machines operate on the decimated clock rate and hence `NWAIT`, `NBLANK` and `NSEP` are divided by the decimation factor. The counters which control the state machines are not shown. Refer to Appendix A for the source code to the state machines.



(a) State machine to initialize mean/var and fill FIFO



(b) Definition of blanking parameters



(c) State machines to implement a single BTR

Figure 8: State machines for filling the FIFO and FIFO Blanking.

4 APB Initial Test Results

The preliminary test APB has been implemented using a single EP20K100EQC208-1 Altera FPGA ($150) with an internal FIFO length of 1024 samples, as shown previously in Figure 6. This design will be modified to include a larger FIFO and possibly a larger FPGA so that a larger processor can be implemented. The test design uses the parameters of $N = L = 12$. See Appendix D for the top level code which sets the size of the APB engine. Some initial results from this engine will be presented. Further tests will be presented in a separate report.

Currently a modified RF front end for the digital processing chain has been put together and it is possible to receive a pulsed Radar signal from a nearby location [4].

Figure 9(a) illustrates the LAB Windows CVI interface for the radiometer. It is possible to control the parameters of the APB with the GUI objects in the top left corner. There are four modes of operation:

1. Pass data directly from the Digital IF to the output,
2. Output 16-bits of the variance, 15-bits of the mean and the 1-bit blank signal,
3. Output data which has been blanked by the APB.
4. Output 16-bits real FIFO unblanked data, 15-bits imaginary FIFO unblanked data, and the 1-bit blank signal.

Each mode is capable of testing the processor in different ways. Mode 1 is used to bypass the processor and results are not relevant here. Mode 2 and mode 4 are shown in Figure 9. Mode 2 is used to see the variation of the mean and variance as a function of time. It is possible to read the full precision result using the Rabbit interface. The variation of the mean and variance is 7% and 17% which is close to the expected values (see Table 3). Also note that the mean and variance are not updated during blanking (the yellow trace.) It is possible to force updates always using the `Force Updates` GUI object.

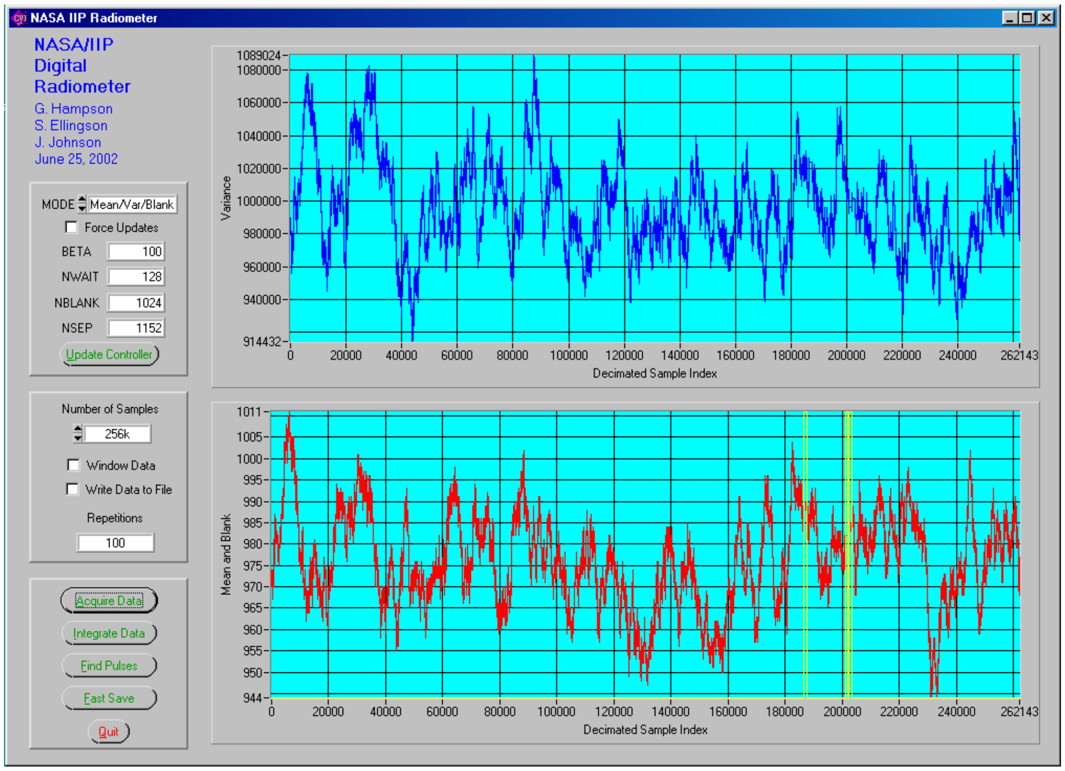
Mode 3 and 4 are very similar, except that one has useful debug purposes. In Mode 4 the blank signal is also output so that it is possible to see the signal the APB processor wants to blank (with the loss of one bit of resolution.) Figure 9(b) shows a radar pulse (and a reflection) which has results in a peak in the spectrum at -50MHz. It is interesting to alter the parameters `BETA`, `NWAIT`, `NBLANK` and `NSEP` to see the effect on what part of the time sequence is blanked.

There are three other features of the software which should also be noted:

Integrate Data Integrates spectrums as data is acquired.

Find Pulses A routine which keeps on grabbing data and looking for pulses in the data. When found it writes data to disk and displays the pulse in the time and frequency domains.

Fast Save Writes data to disk in a binary format as fast as possible. No displaying of data.

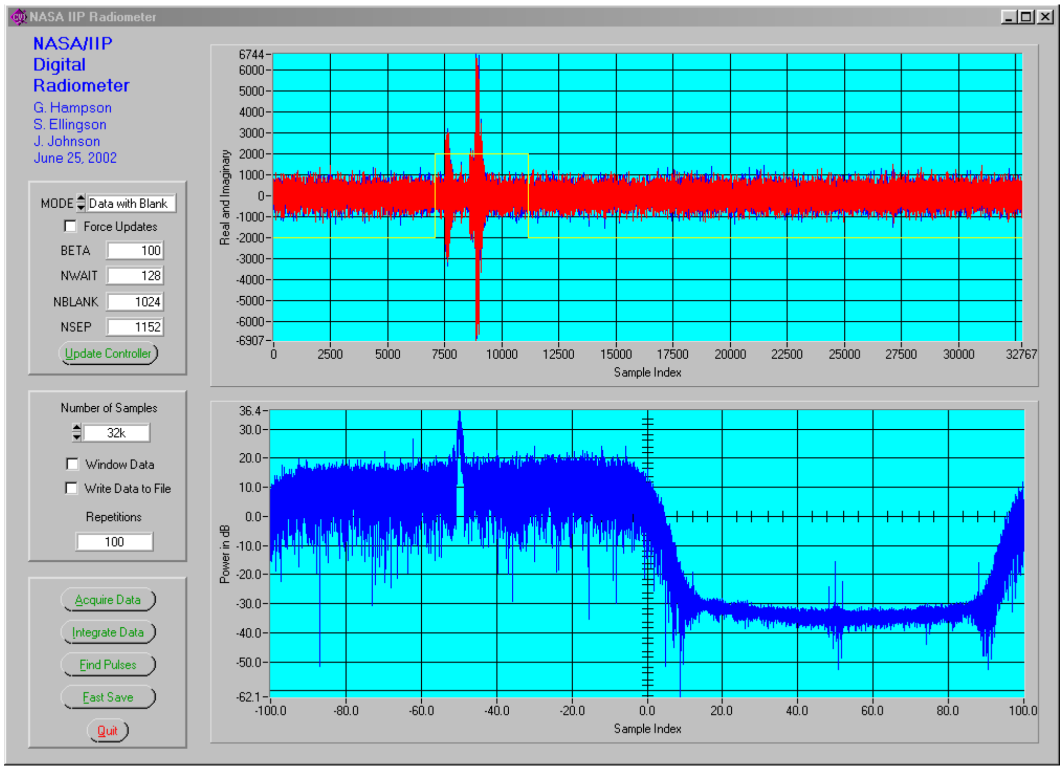

NASA IIP Radiometer

NASA/IIP Digital Radiometer

G. Hampson
S. Ellingson
J. Johnson
June 25, 2002

MODE Mean/Var/Blank
Force Updates
BETA 100
NWAIT 128
NBLANK 1024
NSEP 1152

Number of Samples 256k
Window Data
Write Data to File
Repetitions 100


NASA IIP Radiometer

NASA/IIP Digital Radiometer

G. Hampson
S. Ellingson
J. Johnson
June 25, 2002

MODE Data with Blank
Force Updates
BETA 100
NWAIT 128
NBLANK 1024
NSEP 1152

Number of Samples 32k
Window Data
Write Data to File
Repetitions 100

Figure 9: Initial test results of the pulse blanking system. The top screen capture illustrates the variation of the mean and variance with time. The bottom screen capture illustrates a typical Radar pulse arriving with the blank signal shown in yellow. All the pulse blanker parameters can be set through the interface.

Summary and Conclusions

This report has presented a possible implementation of an asynchronous pulse blanker using an Altera FPGA. The processor can be built from free standard Altera library functions. Simulations of the processor indicate that approximately 25% of the input data can be processed. The number of required FPGA logic elements depends on input width and the exponential weighting factor. For $N = L = 12$ the processor will occupy approximately 77% of a $150 FPGA (EP20K100EQC208-1).

Through simulation and implementation of the algorithm a deeper understanding of the processes involved was obtained. Much thought has been devoted to understanding the effects of fixed and floating point arithmetic on this particular processor. The processor is implemented using fixed point arithmetic with careful attention given to truncation.

The APB consists of three main parts; the mean and variance processor, a Rabbit interface and finally the controlling state machines. Each of these components has been illustrated in this document. The details of these parts will evolve when it is implemented and tested with real data. A TCP/IP control mechanism has also been implemented and tested.

Initial implementation results of the processor have also been presented to illustrate some of the hardware/software possibilities. A future document will go into greater detail.

References

[1] S. W. Ellingson, “Design Concept for the IIP Radiometer RFI Processor,” January 23 2002. http://esl.eng.ohio-state.edu/rfse/iip/rfiproc1.pdf.

[2] S. W. Ellingson, “Functional Design of the Asynchronous Pulse Blanker,” March 20 2002. http://esl.eng.ohio-state.edu/rfse/iip/fdapb.pdf.

[3] M. Tham, “Dealing with Measurement Noise,” 2002. http://lorien.ncl.ac.uk/ming/filter/filter.htm.

[4] S. W. Ellingson, “Preliminary RFI Survey for IIP,” June 11 2002. http://esl.eng.ohio-state.edu/rfse/iip/rfi020610.pdf.

[5] G. A. Hampson, “An FPGA Implementation of the Digital IF Processor,” March 7 2002. http://esl.eng.ohio-state.edu/rfse/iip/digitalif.pdf.

[6] *RCM2200 RabbitCore*, Rabbit Semiconductor. http://www.rabbitsemiconductor.com/products/rcm2200/index.html.

Appendix A: APB Processor AHDL Code

```
-- Asynchronous Pulse Blanker Engine
-- Grant Hampson 21 June 2002

INCLUDE "lpm_mult.inc"; -- basic processing elements
INCLUDE "lpm_add_sub.inc";

PARAMETERS (N = 12,   -- data widths (input and intermediate)
            L = 12);  -- fractional data path width

SUBDESIGN apbproc
(
   clock,                     -- clock/control bits
   reset,                     -- global reset
   real[15..0],               -- Imaginary data bus in (i)
   imag[15..0],               -- Real data bus in (q)
   beta[L-1..0],              -- blanking threshold
   nblank[15..0],             -- samples to blank
   nwait[15..0],              -- samples to wait before blanking
   nsep[15..0],               -- separation of blank triggers
   force_updates              -- forces updates of mean/var
      : INPUT;

   meanx[(N+N+L-1)..0],       -- mean of x (2N+L-bits)
   varx[(N+N+N+N+L-1)..0],    -- var of x (4N+L-bits)
   blank_fifo,                -- blanking output for fifo
   clock_enable               -- clock enable
      : OUTPUT;
)

VARIABLE
   sm_decimation[1..0]          : DFF; -- Decimation statemachine

   real_reg[(N-1)..0],                 -- Register for Real input
   imag_reg[(N-1)..0],                 -- Register for Imaginary input
   multa_reg[(N+N-1)..0],              -- Register for i^2
   multb_reg[(N+N-1)..0],              -- Register for q^2
   adda_reg[(N+N-1)..0],               -- Register for i^2 + q^2
   meanx_reg[(N+N+L-1)..0],            -- Register for mean
   delayx[(N+N-1)..0],                 -- Delays x^2 one clock cycle
   addd_reg[(N+N-1)..0],               -- Registers for x-mx
   multe_reg[(N+N+N+N-1)..0],          -- Register for (x-mx)^2
   varx_reg[(N+N+N+N+L-1)..0],         -- Register for variance
   sm_disable[1..0],                   -- State machine to disable mean/var updates
   sm_blank[1..0],                     -- State machine for blanking FIFO
   cntr_disable[15..0],                -- counts samples to disable engine
   cntr_nsep[15..0],                   -- counts samples between triggers
   cntr_nwait[15..0],                  -- counts samples before blanking
   cntr_nblank[15..0],                 -- counts blanked samples
   blankfifo_reg               : DFFE; -- blanking output register

   pulse_detected,                     -- goes high when pulse detected
   meanvar_enable,                     -- clock enable for mean and variance registers
   blank_enable,                       -- disables update of mean and variance registers
   disable_cnt,                        -- goes high when disable count finished
```

```
nsep_cnt,                           -- goes high when nsep count finished
nwait_cnt,                          -- goes high when nwait count finished
nblank_cnt,                         -- goes high when nblank count finished
multe_ext[(N+N+N+N+L-1)..0] : NODE; -- Node for comparison

multa, multb : lpm_mult WITH(LPM_WIDTHA = N,   -- real^2 and imag^2
                             LPM_WIDTHB = N,
                             LPM_WIDTHP = N+N,
                             LPM_WIDTHS = N+N,
                             LPM_REPRESENTATION = "SIGNED");

adda : lpm_add_sub WITH(LPM_WIDTH = N+N,       -- real^2 + imag^2 adder
                        LPM_REPRESENTATION = "UNSIGNED",
                        LPM_DIRECTION = "ADD");

multc : lpm_mult WITH(LPM_WIDTHA = N+N,        -- (1-mumean)*x
                      LPM_WIDTHB = L,
                      LPM_WIDTHP = N+N+L,
                      LPM_WIDTHS = N+N+L,
                      LPM_REPRESENTATION = "UNSIGNED");

multd : lpm_mult WITH(LPM_WIDTHA = N+N+L,      -- mumean*meanx
                      LPM_WIDTHB = L,
                      LPM_WIDTHP = N+N+L+L,
                      LPM_WIDTHS = N+N+L,
                      LPM_REPRESENTATION = "UNSIGNED");

addc : lpm_add_sub WITH(LPM_WIDTH = N+N+L,     -- mean update
                        LPM_REPRESENTATION = "UNSIGNED",
                        LPM_DIRECTION = "ADD");

addd : lpm_add_sub WITH(LPM_WIDTH = N+N,       -- remove mean
                        LPM_REPRESENTATION = "SIGNED",
                        LPM_DIRECTION = "SUB");

multe : lpm_mult WITH(LPM_WIDTHA = N+N,        -- (x-meanx)^2
                      LPM_WIDTHB = N+N,
                      LPM_WIDTHP = N+N+N+N,
                      LPM_WIDTHS = N+N+N+N,
                      LPM_REPRESENTATION = "SIGNED");

multf : lpm_mult WITH(LPM_WIDTHA = N+N+N+N,    -- (1-mumean)*x
                      LPM_WIDTHB = L,
                      LPM_WIDTHP = N+N+N+N+L,
                      LPM_WIDTHS = N+N+N+N+L,
                      LPM_REPRESENTATION = "UNSIGNED");

multg : lpm_mult WITH(LPM_WIDTHA = N+N+N+N+L,  -- mumean*meanx
                      LPM_WIDTHB = L,
                      LPM_WIDTHP = N+N+N+N+L+L,
                      LPM_WIDTHS = N+N+L,
                      LPM_REPRESENTATION = "UNSIGNED");

addf : lpm_add_sub WITH(LPM_WIDTH = N+N+N+N+L, -- mean update
                        LPM_REPRESENTATION = "UNSIGNED",
                        LPM_DIRECTION = "ADD");
```

```
   multh : lpm_mult WITH(LPM_WIDTHA = L,          -- beta * varx
                         LPM_WIDTHB = N+N+N+N,
                         LPM_WIDTHP = N+N+N+N+L,
                         LPM_WIDTHS = N+N+N+N+L,
                         LPM_REPRESENTATION = "SIGNED");

BEGIN
---------------------  STAGE 1: Calculate i^2 + q^2  ---------------------

   TABLE                                          -- Decimation State Machine
      sm_decimation[].q => sm_decimation[].d;
                 B"00" => B"01";
                 B"01" => B"10";
                 B"10" => B"11";
                 B"11" => B"00";
   END TABLE;
   sm_decimation[].clk = clock;                   -- 100MHz clock
   clock_enable = (sm_decimation[] == B"00");     -- Clock enable 1 in 4 clocks

   real_reg[].d = real[15..(16-N)];               -- Connection of Inputs
   imag_reg[].d = imag[15..(16-N)];
   real_reg[].clk = clock;                        -- 100 MHz clock
   imag_reg[].clk = clock;
   real_reg[].ena = clock_enable;
   imag_reg[].ena = clock_enable;

   multa.dataa[] = real_reg[];                    -- real(x)
   multa.datab[] = real_reg[];
   multa_reg[].d = multa.result[];                -- result is i^2
   multa_reg[].clk = clock;
   multa_reg[].ena = clock_enable;

   multb.dataa[] = imag_reg[];                    -- imag(x)
   multb.datab[] = imag_reg[];
   multb_reg[].d = multb.result[];                -- result is q^2
   multb_reg[].clk = clock;
   multb_reg[].ena = clock_enable;

   adda.dataa[] = multa_reg[];                    -- i^2
   adda.datab[] = multb_reg[];                    -- q^2
   adda_reg[].d = adda.result[];                  -- N-bit result is i^2 + q^2
   adda_reg[].clk = clock;
   adda_reg[].ena = clock_enable;                 -- x^2 1 in 4 clock cycles

------------------  STAGE 2: (1-mumean)x + mumean*meanx  -----------------

   multc.dataa[] = adda_reg[];                    -- x^2 (adder A register)
   multc.datab[0] = VCC;                          -- 1-mu_mean (4096-4095=B"000...001")
   multc.datab[(L-1)..1] = GND;
   multd.dataa[] = meanx_reg[];                   -- meanx
   multd.datab[] = VCC;                           -- mu_mean (4095=B"111...111")
   addc.dataa[] = multc.result[];                 -- x^2 by (1-mumean)
   addc.datab[] = multd.result[(N+N+L+L-1)..L];   -- mu_mean by meanx

   meanx_reg[].d = addc.result[];                 -- update meanx register
   meanx_reg[].clk = clock;
   meanx_reg[].clrn = reset;                      -- startup mean value is 0
   meanx_reg[].ena = clock_enable AND meanvar_enable;
   meanx[] = meanx_reg[];                         -- to controller/IO registers

--------------  STAGE 3: (1-muvar)(x^2-meanx) + muvar*varx  --------------
   delayx[].d = adda_reg[];                       -- x^2 decimated
   delayx[].clk = clock;                          -- delay one clock cycle
   delayx[].ena = clock_enable;

   addd.dataa[] = delayx[];                       -- x^2 delayed one clock cycle
   addd.datab[] = meanx_reg[(N+N+L-1)..L];        -- subtract meanx (remove fraction)
   addd_reg[].d = addd.result[];
   addd_reg[].clk = clock;
   addd_reg[].ena = clock_enable;
   multe.dataa[] = addd_reg[];                    -- x^2-meanx
   multe.datab[] = addd_reg[];                    -- input is squared
   multe_reg[].d = multe.result[];                -- 4N bit register
   multe_reg[].clk = clock;
   multe_reg[].ena = clock_enable;

   multf.dataa[] = multe_reg[];                   -- (x^2-meanx)^2
   multf.datab[0] = VCC;                          -- 1-mu_var (4096-4095=B"000...001")
   multf.datab[(L-1)..1] = GND;
   multg.dataa[] = varx_reg[];                    -- varx
   multg.datab[] = VCC;                           -- mu_var (4095=B"111...111")
   addf.dataa[] = multf.result[];                 -- (x^2-meax) by (1-muvar)
   addf.datab[] = multg.result[(N+N+N+N+L+L-1)..L];-- mu_var by varx
   varx_reg[].d = addf.result[];                  -- update varx register
   varx_reg[].clk = clock;
   varx_reg[].clrn = reset;                       -- startup value is 0
   varx_reg[].ena = clock_enable AND meanvar_enable;
   varx[] = varx_reg[];                           -- to controller/IO registers

-----------------------  STAGE 4: Pulse Detection ------------------------

   multh.dataa[] = beta[];                        -- beta
   multh.datab[] = varx_reg[(N+N+N+N+L-1)..L];    -- varx
   multe_ext[(N+N+N+N+L-1)..(N+N+N+N)] = GND;     -- extend L-MSBs with zeros
   multe_ext[(N+N+N+N-1)..0] = multe_reg[];       -- LSBs are (x-mx)^2

   if  multe_ext[] < multh.result[] then          -- pulse detection
     pulse_detected = B"0";                       -- no pulse, update variance
   else
     pulse_detected = B"1";                       -- pulse detected, no update of mean/var
   end if;

--------------------  STAGE 5: Blanking State Machines --------------------

   cntr_disable[].clk = clock;                    -- disable pulse_detected counter
   cntr_disable[].ena = clock_enable;
   cntr_disable[].d = cntr_disable[].q + 1;
   cntr_disable[].clrn = !(sm_disable[].q == B"00");
   disable_cnt = cntr_disable[].q == nblank[];

   cntr_nsep[].clk = clock;                       -- separation counter
   cntr_nsep[].ena = clock_enable;
   cntr_nsep[].d = cntr_nsep[].q + 1;
   cntr_nsep[].clrn = !(sm_disable[].q == B"00");
   nsep_cnt = cntr_nsep[].q == nsep[];

   cntr_nwait[].clk = clock;                      -- FIFO wait counter
   cntr_nwait[].ena = clock_enable;
   cntr_nwait[].d = cntr_nwait[].q + 1;
   cntr_nwait[].clrn = !(sm_blank[].q == B"00");
   nwait_cnt = cntr_nwait[].q == nwait[];

   cntr_nblank[].clk = clock;                    -- blanking counter
   cntr_nblank[].ena = clock_enable;
   cntr_nblank[].d = cntr_nblank[].q + 1;
   cntr_nblank[].clrn = !(sm_blank[].q == B"01");
   nblank_cnt = cntr_nblank[].q == nblank[];

   TABLE                                         -- Disable blanking State Machine
   sm_disable[].q, pulse_detected, disable_cnt, nsep_cnt => sm_disable[].d, blank_enable;
            B"00",          B"0",        B"X",      B"X" =>         B"00",        B"1";
            B"00",          B"1",        B"X",      B"X" =>         B"01",        B"0";
            B"01",          B"X",        B"0",      B"X" =>         B"01",        B"0";
            B"01",          B"X",        B"1",      B"X" =>         B"10",        B"0";
            B"10",          B"X",        B"X",      B"0" =>         B"10",        B"1";
            B"10",          B"X",        B"X",      B"1" =>         B"00",        B"1";
   END TABLE;
   sm_disable[].clk = clock;
   sm_disable[].ena = clock_enable;
   meanvar_enable =  force_updates OR blank_enable;

   TABLE-- Blanking State Machine
      sm_blank[].q, pulse_detected, nwait_cnt, nblank_cnt => sm_blank[].d, blankfifo_reg.d;
             B"00",          B"0",      B"X",       B"X" =>       B"00",            B"0";
             B"00",          B"1",      B"X",       B"X" =>       B"01",            B"0";
             B"01",          B"X",      B"0",       B"X" =>       B"01",            B"0";
             B"01",          B"X",      B"1",       B"X" =>       B"10",            B"0";
             B"10",          B"X",      B"X",       B"0" =>       B"10",            B"1";
             B"10",          B"X",      B"X",       B"1" =>       B"00",            B"1";
   END TABLE;
   sm_blank[].clk = clock;
   sm_blank[].ena = clock_enable;
   blankfifo_reg.clk = clock;
   blankfifo_reg.ena = clock_enable;
   blank_fifo = blankfifo_reg.q;
END;
```

Appendix B: Matlab APB Processor Code

```
% APB Processor Matlab Test Code
% Grant Hampson June 10 2002

clear all

N = 12;       % input data x resolution
D = 4;        % decimation factor
L = 12;       % time constant mumean resolution
S = 40000;    % initial number of x samples
s = (1:S);    % sample index
B = 0;        % blank inhibit (0=always update, 1=selective update)

x = randn(S,1) + sqrt(-1)*randn(S,1);                % generate example input
maxx = max([max(abs(real(x))) max(abs(imag(x)))]);  % find largest value
x = fix((x.'/maxx) * (2^(N-1)-1));                  % quantise to N-bits
x2 = (real(x).*real(x)) + (imag(x).*imag(x));       % square and add
x2d = x2(1:D:S);                                    % decimate by skipping samples
SD = S/D;                                           % reduced number of samples
sd = (1:SD);                                        % decimated sample index

figure(1)
subplot(221),plot(s,real(x),'b',s,imag(x),'r'),grid
ylim([-1*2^(N-1) 2^(N-1)])
xlabel('Sample Index'), ylabel('N-bit Input Samples')
title('Complex Input x')

subplot(222),plot(s,x2,'b'),grid
xlabel('Sample Index'), ylabel('2N-bit x^2')
title(['x^2, Matlab Mean=' num2str(mean(x2))])

subplot(223),plot(sd,x2d,'b'),grid
xlabel('Decimated Sample Index'), ylabel('2N-bit x^2')
title(['Decimated x^2 (D=' int2str(D) ...
      '), Matlab Mean=' int2str(mean(x2d))])

mumean = ((2^L)-1)/(2^L);         % time constant for mean
mumean = fix(mumean*2^L);         % integer L-bit (max 2^L-1)
onemumean = (2^L-mumean)/(2^L);   % fractional L-bit (min 1/2^L)
mumean = mumean/(2^L);            % convert to fractional L-bit
meanx(1) = 0*mean(x2d);           % initial mean value
for i = 2:SD
   meanx(i) = onemumean*x2d(i) + fix(mumean*meanx(i-1)*(2^L))/(2^L);
end
meanxfrac = meanx;
meanx = fix(meanx); % truncate fractionional part (2N-bit integer left)

subplot(224),plot(sd,meanx,'m'),hold on, grid
plot(sd,meanxfrac,'b')
plot([1 S],[mean(x2) mean(x2)],'r'),hold off
xlabel('Sample Index'), ylabel('2N-bit meanx')
title(['Computed Mean of x^2, \mu_{mean}=' int2str(mumean*(2^L-1))])

xmm = x2d-meanx;   % 2N-bit subtract the mean
xmm2 = xmm.^2;     % square the result 4N-bit result
```

```
figure(2)
subplot(221),plot(sd,xmm,'b'),grid
xlabel('Sample Index'), ylabel('2N-bit x^2-meanx')
title('x^2 - meanx')

subplot(222),plot(sd,xmm2,'b'),grid
xlabel('Sample Index'), ylabel('2N-bit (x^2-meanx)^2')
title(['Matlab Mean is ' int2str(mean(xmm2))])

muvar = ((2^L)-1)/(2^L);        % time constant for variance
muvar = fix(muvar*2^L);         % integer L-bit (max 2^L-2)
onemuvar = (2^L-muvar)/(2^L);   % fractional L-bit
muvar = muvar/(2^L);            % convert to fractional L-bit
beta2 = 1^2;                    % beta is L-bit
varx(1) = 0*mean(xmm2);         % initial variance value
for i = 2:SD
   varx_temp = onemuvar*xmm2(i) + fix(muvar*varx(i-1)*(2^L))/(2^L);
   bv(i) = beta2*(varx_temp);   % beta squared by variance

   if B
      if xmm2(i) < bv(i)
         pd(i) = 0;             % no pulse detected
         varx(i) = varx_temp;   % update variance
      else
         pd(i) = 1;             % pulse detected
         varx(i) = varx(i-1);   % no update of variance
      end
   else
      if xmm2(i) < bv(i)
         pd(i) = 0;             % no pulse detected
      else
         pd(i) = 1;             % pulse detected
      end
      varx(i) = varx_temp;      % always update variance
   end
end
varxfrac = varx;
varx = fix(varx); % truncate L-bit fraction  (4N-bit integer remaining)

subplot(212),plot(sd,varx,'m'),hold on, grid
plot(sd,varxfrac,'b');
plot([1 SD],[mean(xmm2) mean(xmm2)],'r'),hold off
xlabel('Sample Index'), ylabel('4N-bit varx')
title(['Computed Mean of xmm2, \mu_{var}=' int2str(muvar*(2^L-1))])

figure(3)
subplot(211), plot(sd,xmm2,'b')
hold on,plot(sd,bv,'r'), grid, hold off
xlabel('Sample Index'), ylabel('x^2 and \beta\sigma^2')
title(['x-meanx (blue), \beta\sigma^2 (red), \beta=' int2str(sqrt(beta2))])

subplot(212),plot(sd,pd),grid
xlabel('Sample Index'), ylabel('Blank on/off')
title(['Pulse Detection, \beta=' int2str(sqrt(beta2)) ...
      ', Pon/S=' num2str(100*length(find(pd(SD-1000:SD)==1))/1000) '%'])
```

Appendix C: APB Interface AHDL Code

```
-- APB Processor Interface to Rabbit Controller
-- Grant Hampson June 14 2002

INCLUDE "lpm_decode.inc";
INCLUDE "lpm_mux.inc";

SUBDESIGN rabbitinterface
(
   clock,                      -- clock/control bits
   reset,                      -- global reset
   address[4..0],              -- address from Rabbit
   chipselect,                 -- chip select active low
   read_write,                 -- read high, write low
   get_meanvar,                -- updates mean and variance registers
   fromapb_proc[13..0][7..0]   -- data from processor

      : INPUT;

   toapb_proc[17..0][7..0]     -- data to processor

      : OUTPUT;

   data[7..0]                  -- Rabbit bidirectional data bus

      : BIDIR;
)

VARIABLE
   addr_dec : lpm_decode WITH(LPM_WIDTH = 5,     -- decodes address for register file
                              LPM_DECODES = 18);

   regfile[17..0][7..0] : DFFE;                  -- Register file of 18x8-bit registers
   proc_reg[13..0][7..0] : DFFE;                 -- register variables from processor

   data_mux : lpm_mux WITH(LPM_WIDTH = 8,        -- multiplexes registers/data out
                           LPM_SIZE = 32,
                           LPM_WIDTHS = 5);

BEGIN
   addr_dec.data[] = address[4..0];                  -- connect address decode
   addr_dec.enable = !(chipselect OR read_write);

   regfile[][].clk = clock;
   regfile[][].clrn = reset;
   FOR i IN 0 TO 17 GENERATE
      regfile[i][7..0].ena = addr_dec.eq[i];         -- connect each clock enable
      regfile[i][7..0].d = data[7..0];
   END GENERATE;
   toapb_proc[][] = regfile[][];

   proc_reg[][].clk = clock;                         -- register incoming signals
   proc_reg[][].d = fromapb_proc[][];
   proc_reg[][].ena = get_meanvar;
   data_mux.sel[] = address[4..0];                   -- 32-to-1 multiplexer
   FOR i IN 0 TO 17 GENERATE
      data_mux.data[i][7..0] = regfile[i][7..0];     -- registers
   END GENERATE;
   FOR i IN 18 TO 31 GENERATE
      data_mux.data[i][7..0] = proc_reg[i-18][7..0]; -- processor data
   END GENERATE;

   data[0] = TRI(data_mux.result[0], read_write AND (!chipselect)); -- tri-state bus
   data[1] = TRI(data_mux.result[1], read_write AND (!chipselect));
   data[2] = TRI(data_mux.result[2], read_write AND (!chipselect));
   data[3] = TRI(data_mux.result[3], read_write AND (!chipselect));
   data[4] = TRI(data_mux.result[4], read_write AND (!chipselect));
   data[5] = TRI(data_mux.result[5], read_write AND (!chipselect));
   data[6] = TRI(data_mux.result[6], read_write AND (!chipselect));
   data[7] = TRI(data_mux.result[7], read_write AND (!chipselect));
END;
```

Appendix D: APB Top Level AHDL Code

```
-- Pulse Blanker (top level)
-- Grant Hampson 21 June 2002

CONSTANT MODE = 3;   -- Mode=1: Raw Data, Mode=2: Mean/Var output,
                     -- Mode=3: Blanked data, Mode 4:put blank into data

INCLUDE "lpm_fifo.inc";   -- used for internal FPGA FIFO

FUNCTION apbproc (clock, reset, real[15..0], imag[15..0], beta[(L-1)..0],
                  nblank[15..0], nwait[15..0], nsep[15..0], force_updates)
   WITH (N, L)
   RETURNS (meanx[(N+N+L-1)..0], varx[(N+N+N+N+L-1)..0], blank_fifo, clock_enable);

FUNCTION rabbitinterface (clock, reset, address[4..0], chipselect, read_write,
                          get_meanvar, fromapb_proc[13..0][7..0])
   RETURNS (toapb_proc[17..0][7..0], data[7..0]);

SUBDESIGN PulseBlanker
(
   control1in,        -- clocks from general connector
   control2in,
   realin[15..0],     -- data inputs from general connector
   imagin[15..0],
   addr[3..0],        -- address from rabbit interface
   pe7,               -- chip select
   iord,              -- read
   iowr,              -- write
   porta[7..0]        -- port a - byte wide input
      :INPUT;

   control1out,       -- control lines to general connector
   control2out,
   realout[15..0],    -- data outputs for general connector
   imagout[15..0],
   blank
      :OUTPUT;

   data[7..0]         -- rabbit data bus
      :BIDIR;
)

VARIABLE
   realreg[15..0], imagreg[15..0],        -- output registers
   cntr_force[15..0],                     -- 256144 sample startup time
   cntr_fifolength[9..0],                 -- counter for fifolength
   sm_force,                              -- intialisation state machine
   sm_fillfifo[1..0]             : DFF;  -- fill fifo state machine

   force_length,                          -- signal for counted samples
   fifolength                    : NODE; -- signal for fifo full

   apbprocessor : apbproc WITH (N=12, L=12);

   iointerface : rabbitinterface;
   data_fifo : lpm_fifo WITH (LPM_WIDTH = 32,
                              LPM_NUMWORDS = 1024,
                              LPM_WIDTHU = 10,
                              LPM_SHOWAHEAD = "OFF");

BEGIN
   iointerface.clock = control2in;
   iointerface.reset = VCC;
   iointerface.address[3..0] = addr[3..0];
   iointerface.address[4] = porta[0];
   iointerface.chipselect = pe7;
   iointerface.read_write = iowr;
   data[] = iointerface.data[];
   iointerface.get_meanvar = apbprocessor.clock_enable AND porta[1];
   iointerface.fromapb_proc[0][7..0] = GND;                        -- Reg 18
   iointerface.fromapb_proc[1][7..0] = apbprocessor.meanx[7..0];   -- Reg 19
   iointerface.fromapb_proc[2][7..0] = apbprocessor.meanx[15..8];  -- Reg 20
   iointerface.fromapb_proc[3][7..0] = apbprocessor.meanx[23..16]; -- Reg 21
   iointerface.fromapb_proc[4][7..0] = apbprocessor.meanx[31..24]; -- Reg 22
   iointerface.fromapb_proc[5][3..0] = apbprocessor.meanx[35..32]; -- Reg 23
   iointerface.fromapb_proc[5][7..4] = GND;                        -- Reg 23
   iointerface.fromapb_proc[6][7..0] = apbprocessor.varx[7..0];    -- Reg 24
   iointerface.fromapb_proc[7][7..0] = apbprocessor.varx[15..8];   -- Reg 25
   iointerface.fromapb_proc[8][7..0] = apbprocessor.varx[23..16];  -- Reg 26
   iointerface.fromapb_proc[9][7..0] = apbprocessor.varx[31..24];  -- Reg 27
   iointerface.fromapb_proc[10][7..0] = apbprocessor.varx[39..32]; -- Reg 28
   iointerface.fromapb_proc[11][7..0] = apbprocessor.varx[47..40]; -- Reg 29
   iointerface.fromapb_proc[12][7..0] = apbprocessor.varx[55..48]; -- Reg 30
   iointerface.fromapb_proc[13][3..0] = apbprocessor.varx[59..56]; -- Reg 31
   iointerface.fromapb_proc[13][7..4] = GND;                       -- Reg 31

   apbprocessor.clock = control2in;    -- connections to APB processor
   apbprocessor.reset = VCC;
   apbprocessor.real[15..0] = realin[15..0];
   apbprocessor.imag[15..0] = imagin[15..0];
   apbprocessor.beta[11..8] = iointerface.toapb_proc[2][3..0];
   apbprocessor.beta[7..0] = iointerface.toapb_proc[1][7..0];
   apbprocessor.force_updates = (!sm_force.q) OR iointerface.toapb_proc[0][2];
   apbprocessor.nwait[15..8]  = iointerface.toapb_proc[13][7..0];
   apbprocessor.nwait[7..0]   = iointerface.toapb_proc[12][7..0];
   apbprocessor.nblank[15..8] = iointerface.toapb_proc[15][7..0];
   apbprocessor.nblank[7..0]  = iointerface.toapb_proc[14][7..0];
   apbprocessor.nsep[15..8]   = iointerface.toapb_proc[17][7..0];
   apbprocessor.nsep[7..0]    = iointerface.toapb_proc[16][7..0];

   cntr_force[].clk = control2in;   -- counter for APB initialisation
   cntr_force[].d = cntr_force[].q + 1;
   force_length = cntr_force[].q == B"1111111111111111";
   TABLE                                    -- Force updates in APB
      sm_force.q, force_length => sm_force.d;
           B"0",          B"0" =>       B"0";   -- wait for forcelength samples
           B"0",          B"1" =>       B"1";   -- finished mean/var initialisation
           B"1",          B"X" =>       B"1";   -- loop indefinitely
   END TABLE;
   sm_force.clk = control2in;
   data_fifo.clock = control2in;        -- Connection of data FIFO
   data_fifo.data[15..0] = realin[];
   data_fifo.data[31..16] = imagin[];
   data_fifo.aclr = !sm_force.q;

   cntr_fifolength[].clk = control2in;  -- counter for filling FIFO
   cntr_fifolength[].d = cntr_fifolength[].q + 1;
   cntr_fifolength[].clrn = sm_force.q; -- resets until after initilisation
   fifolength = cntr_fifolength[].q == B"1111111111";
   TABLE
      sm_fillfifo[].q, fifolength => sm_fillfifo[].d, data_fifo.wrreq, data_fifo.rdreq;
               B"00",       B"X" => B"01",           B"0",       B"0"; -- reset FIFO
               B"01",       B"0" => B"01",           B"1",       B"0"; -- not filled
               B"01",       B"1" => B"10",           B"1",       B"0"; -- FIFO filled
               B"10",       B"X" => B"10",           B"1",       B"1"; -- loop
   END TABLE;
   sm_fillfifo[].clk = control2in;
   sm_fillfifo[].clrn = sm_force.q;

   realreg[].clk = control2in; -- output registers
   imagreg[].clk = control2in;
   realout[] = realreg[];      -- connect to FPGA outputs
   imagout[] = imagreg[];
   control2out = GND;          -- unused output

   case (iointerface.toapb_proc[0][1..0]) is
      when B"00" =>
         realreg[].d = realin[];      -- output = input data
         imagreg[].d = imagin[];
         control1out = control2in;    -- 100MHz output clock
         blank = GND;                 -- unused output
      when B"01" =>
         realreg[15].d = apbprocessor.blank_fifo;       -- APB processor information
         realreg[14..0].d = apbprocessor.meanx[26..12];
         imagreg[15..0].d = apbprocessor.varx[35..20];
         control1out = apbprocessor.clock_enable;       -- lower data rate clock
         blank = apbprocessor.blank_fifo;
      when B"10" =>
         if apbprocessor.blank_fifo == B"1" then
            realreg[].d = GND;                          -- Blank the output data
            imagreg[].d = GND;
         else
            realreg[].d = data_fifo.q[15..0];           -- else send FIFO data
            imagreg[].d = data_fifo.q[31..16];
         end if;
         control1out = control2in;                      -- 100MHz output clock
         blank = apbprocessor.blank_fifo;
      when B"11" =>
         realreg[0].d = blank;                         -- put out blank in signal
         realreg[15..1].d = data_fifo.q[15..1];        -- don't blank any data
         imagreg[].d = data_fifo.q[31..16];
         control1out = control2in;                     -- 100MHz output clock
         blank = apbprocessor.blank_fifo;
   end case;
END;
```