Arbitrary Waveform Generator (AWFG)

Prepared by,

Kanwar Talha Bin Liaqat

Date: 20/08/2018

Table of Contents

Table of Figures

1. Introduction

The focus of this project was to develop a model of Arbitrary Waveform Generator (AWFG) based on the '***AWFG_concept***' template model which demonstrates the principle for this project. This template model was used for gaining understanding of the intended application.

This model was then expanded for generation of different waveforms and variable inputs. The final model was tested and ran on real time target platform (ZYBO Board) with CPU and FPGA code generated using Automatic Code Generation in Simulink. **Fig. 1-1** shows the flowchart of the concept model.

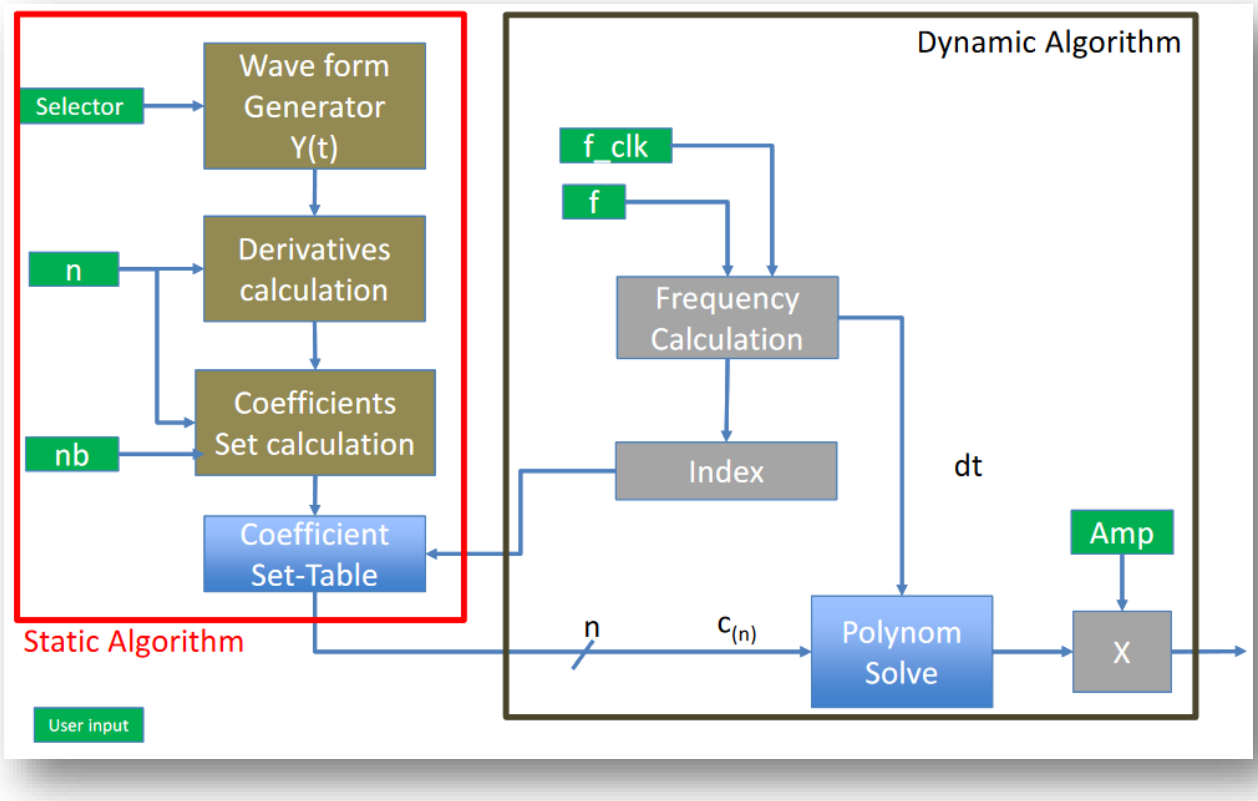

Fig. 1-1: AWFG Concept Model Flowchart

The model was divided into two parts,

- **Static Algorithm (CPU)**:
  This part is responsible for processing the inputs from the user, which are; amplitude, frequency, mode of waveform (sine, triangular, square), number of break points, polynomial order, and duty cycle. Later, based on these inputs it generates the waveforms using Taylor Series and the coefficients.

- **Dynamic Algorithm (FPGA)**:
  This part is responsible for time generation, read write operations, and calculation of polynomials.

2. Analysis of AWFG Concept Model

A reference concept model was provided to analyze and use as the basis of the final model. After exploring the model, the following understands were made:

2.1 *Main Subsystems at the Top Layer*

There were three main subsystems present at the top level of the concept model i.e. **RT_CPU**, **Precalculations**, and **FPGA_Simulation**. **Fig. 2-1** shows the top layer of AWFG concept model.

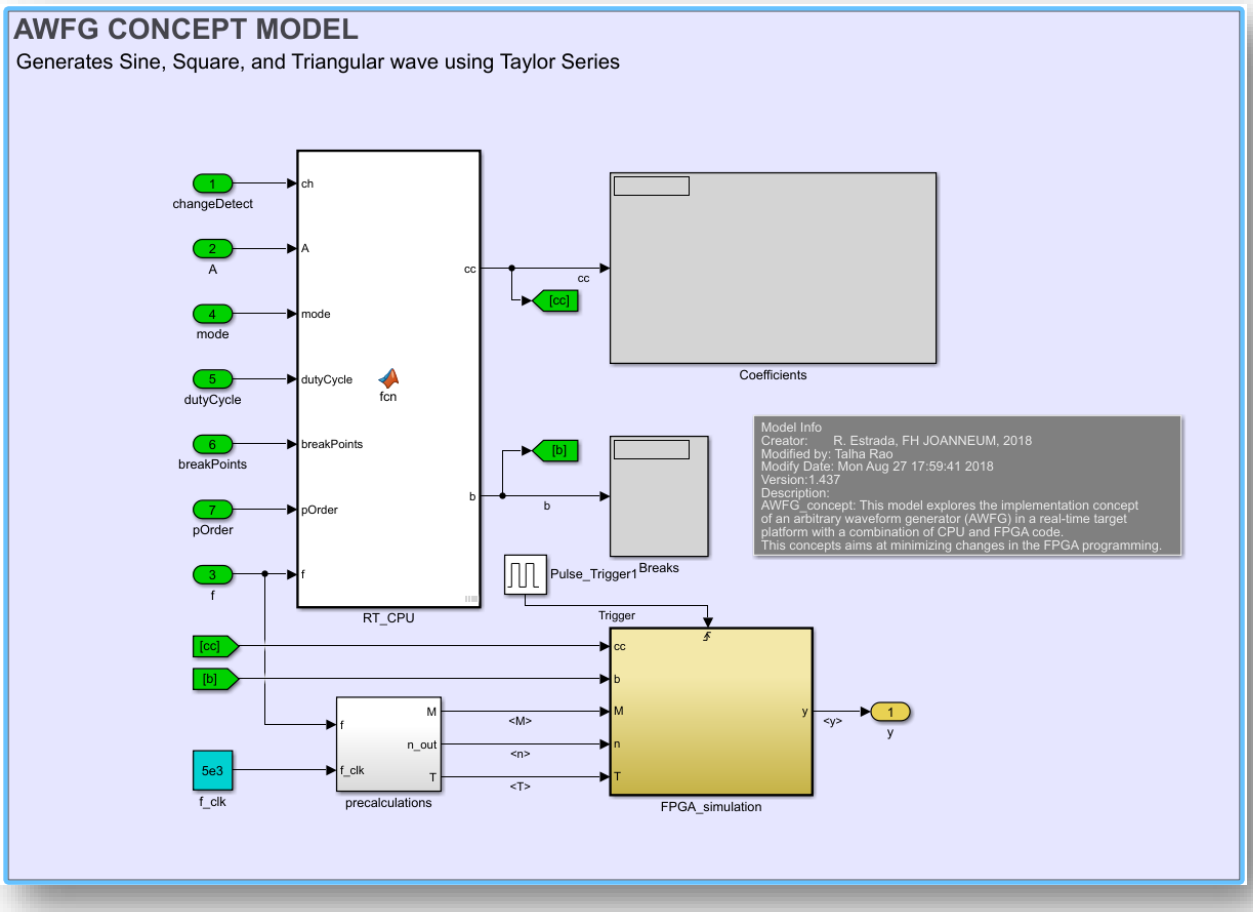

Fig. 2-1: AWFG Concept Model Top Layer

2.1.1. RT_CPU

This block is a MATLAB function block which calculates the coefficients of polynomial and breakpoints using the inputs given by the user i.e. amplitude, frequency, type of wave (mode), no. of breakpoints, polynomial order, and the duty cycle.

2.1.2. Precalculations

This block is responsible for generating the reference time, necessary for solving the polynomial, using desired frequency of the wave and the clock frequency (5000 Hz).

2.1.3. FPGA_Simulation

This block deals with the functionality that is to be implemented in FPGA. Using the reference time generated by Precalculations block is used to solve the polynomial for the desired waveform.

2.2 Computation of Coefficients only once

In the MATLAB function block RT_CPU, it was made sure that the calculation of coefficients and breakpoints is performed for every execution because initially it was not required to change the input values hence, the calculation of coefficients was only required once.

To ensure the single execution, a variable named **i0** was used which was set to **0** during initialization of the program. This variable was declared as **persistent**. The values of these variables are retained in memory between calls to the function. In the first run the value of i0 was incremented to 1. An If block was used to check the value of i0 to be less than 1 for future calls. Following code lines are used for this purpose:

```
persistent cc0 b0 i0;
% This is executed only once during Initialization
if isempty(cc0)
    cc0 = zeros(101,3);
    b0  = zeros(1,101);
    i0  = 0;  %Used to checks if the program is running for the 1st time
end
if i0<1
    i0=1;
```

2.3 Influence of ‘M’ Variable

M defines the jump size which is used for the time calculation and influences the frequency of the signal as seen in **equation 2.1**.

$$f_0 = \frac{M f_{clk}}{2^n} \qquad 2.1$$

Where, $f_\circ$ is the signal frequency. Fig. 2-2 shows how the change in value of M affects the frequency of the signal.

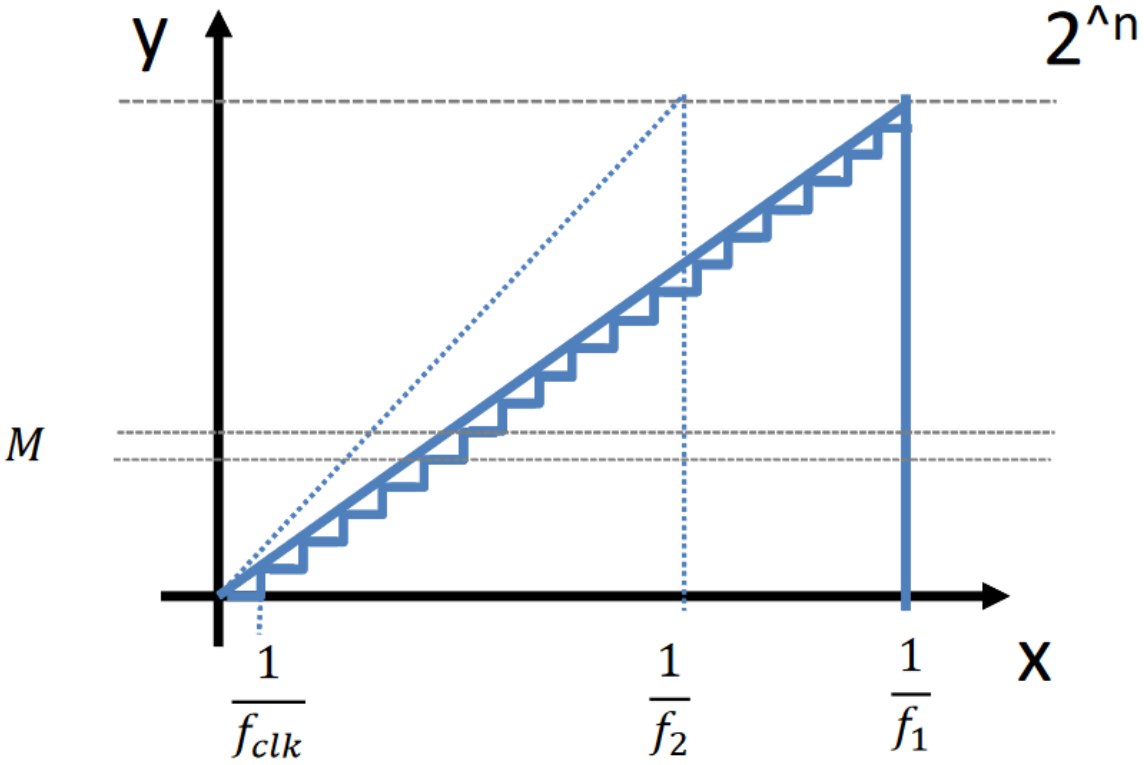

Fig. 2-2: Effect of 'M' on Signal Frequency

2.4 *Operation of for-loop in FPGA_fcn Block*

The block FPGA_fcn is responsible for solving the polynomial using the time, break points and the coefficients calculated previously according to the equation 2.2.

$$\mathrm{f}(x) = c_0 + c_1(x - x_0) + c_2(x - x_0)^2 + \cdots + c_n(x - x_0)^n \qquad 2.1$$

Where, x is the actual time and x_0 is the time where breakpoint is placed.

The for loop in the function is responsible for calculating the above equation. The for loop runs as many times as the number of coefficients. The breakpoint is selected according to the current time. In the for loop, the previous output is added to the product of next coefficient with the difference of current time and corresponding breakpoint.

```
n=size(cc,2)-1;
i=find(b>t,1);        % find closest upper break point
ii=max(i(1)-1,1);     % assign actual index for polynomial calculation

% Calculate polynomial output according to actual relative time
y=0;
for j=0:1:n
    y=y+cc(ii,j+1)*(t-b(ii))^j;
end
```

3. Development of AWFG Model (Phase 1)

After analyzing the basic model, modifications were made to the model to generate multiple waveforms with adjustable inputs. In this phase only, the static algorithm was developed.

3.1 *Addition of Input Parameters*

The MATLAB function block 'RT_CPU' was modified in such a way that the number of breakpoints, duty cycle, and the order of polynomial is given to the function as input. Which is later used to change these values in real time.

The following function arguments were introduced in the function:

```
function [cc,b]  = fcn(ch,A,mode,dutyCycle, breakPoints,pOrder, f)
```

These values were given to the model using the constant blocks in Simulink. **Fig. 3-1** shows the values given as constants to the main AWFG block.

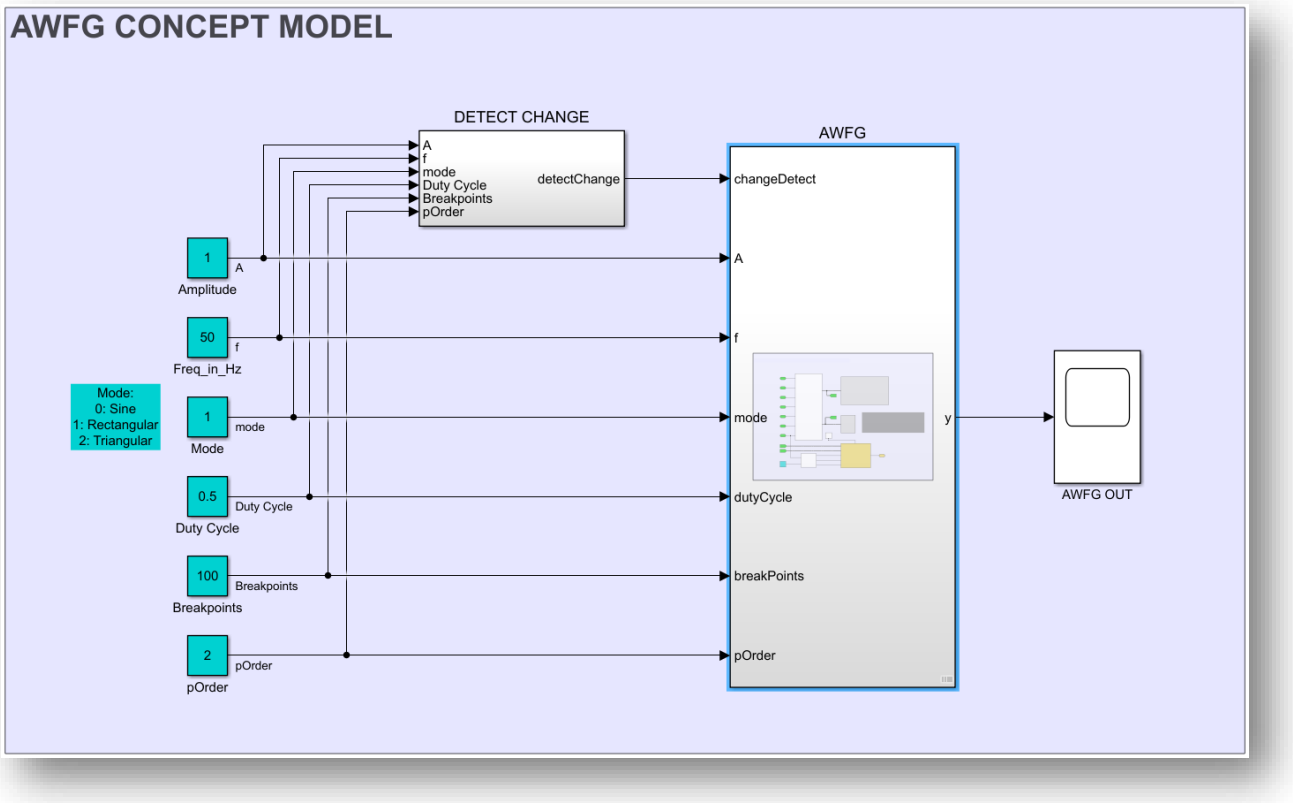

Fig. 3-1: AWFG Input Values

Now that the input parameters were subjected to change, the inputs and outputs of the block needed to be made variable in model explorer. Ranges and size of these variables were set as per the requirement. **Table 3-1** shows the upper and lower bounds of input variables.

Table 3-1: Input Parameters Range

Parameter	Min	Max
Amplitude	0	1
Mode	0	2
Number of breakpoints	4	255
Polynomial order	0	3
Duty Cycle	0	1
F_Clk	-	5e7

These ranges were set in the model explorer as shown in **Fig. 3-2**.

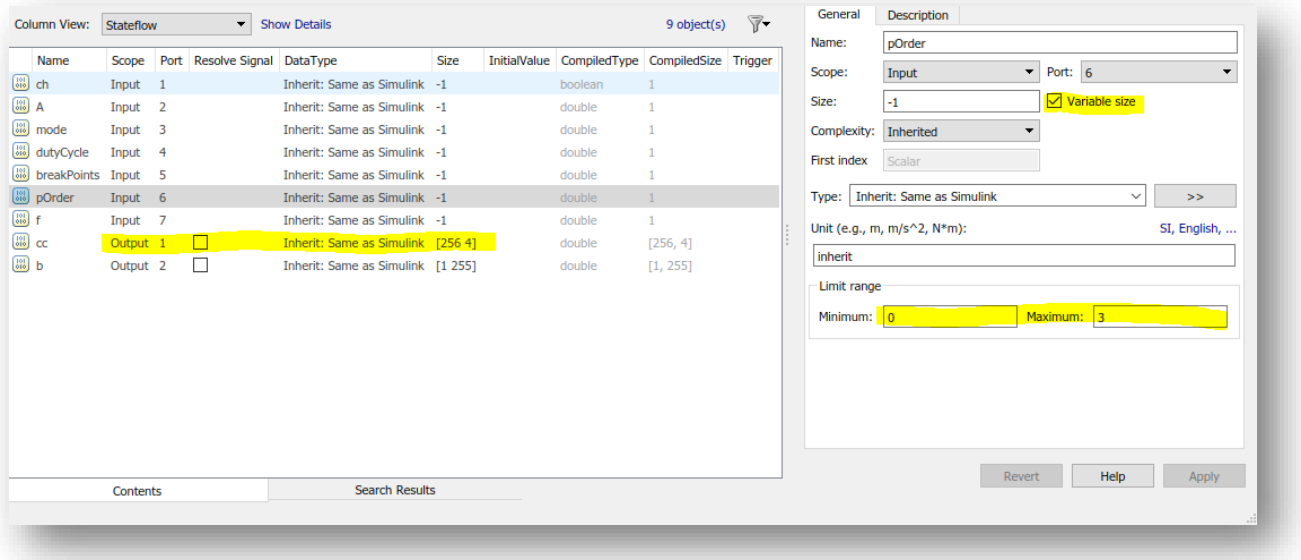

Fig. 3-2: Input Parameters Range in Model Explorer

The number of breakpoints was used in the MATLAB code for generation of output and as the number of breakpoints were subjected to change, they needed to be bounded in the code using the function assert() as follows:

```
%Bound values of breakpoints
assert(breakPoints<=256);

cc = zeros(breakPoints+1,3);          %% Creates coefficients array
b  = zeros(1,(breakPoints + 1));      %% Create breakpoints array
tf = 1/f;
y  = zeros(1 , (breakPoints + 1));
```

3.2 *Implementation of More Waveforms*

The basic model only had the functionality of generating sine wave. In the next step, the functionality of generating two more waveforms (triangular and rectangular) was implemented using case structure. The input parameter 'mode' was used as the selecting factor of the output waveform as per the requirements shown in **Table 3-2**.

Table 3-2: Selection of Waveforms

Waveform	Mode
Sine	0
Square	1
Triangular	2

In the MATLAB function block 'RT_CPU' two more cases were implemented as follows:

```
switch mode
    case 0 % Sine wave

        y  = A*sin(2*pi.*linspace(0,1,breakPoints+1));

    case 1 % Square Wave

        x= linspace(0,1,breakPoints+1);
        y = (x < dutyCycle).*(A) + (x >= dutyCycle).*(-A);
        y(breakPoints+1) = A;

    case 2 % Triangular wave

        x= linspace(0,1,breakPoints +1);
        x1= linspace(0,1,breakPoints);

        if (dutyCycle ==0 || dutyCycle == 1)     %To avoid division by 0

            y = (dutyCycle == 1).*(-A+(x1*A*2)) +...
                (dutyCycle ==0).*(-A+((1-x1)*A*2));
            y= [y y(1)];
        else

            y = (x <= dutyCycle).*(-A+(x*A*(2/dutyCycle)))+...
                (x > dutyCycle).*(-A+((1-x)*A*(2/(1-dutyCycle))));

        end

end
```

The generated waveforms are presented in **section 3.4 (Testing of Model)**.

3.3 *Detect Change*

As this model was being prepared for the final execution on a hardware target, which supports the adjustable input functionality, in an infinite loop. This functionality needed to be implemented in the model as it supported calculation of coefficients only once per execution. For this purpose, a 'Detect Change' subsystem was added to the model which could detect change in any of the input parameters and outputs a Boolean value.

The output of the Detect Change block was given to RT_CPU block where it can used as a trigger for the recalculation of coefficients, if the input is changed. **Fig. 3-3** shows the subsystem 'detectChange'. The connection of this block to RT_CPU block can be seen in **Fig. 3-1.**

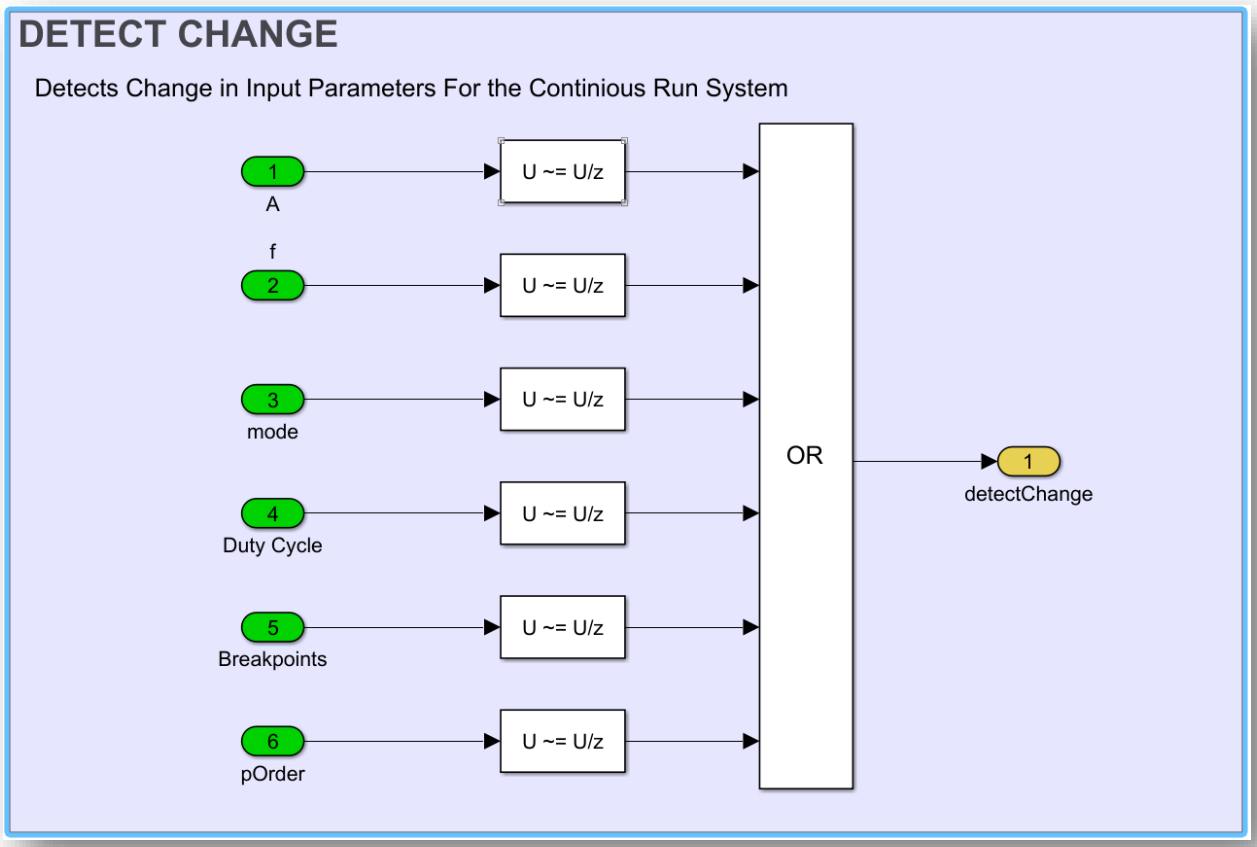

Fig. 3-3: AWFG Detect Change Block

In MATLAB function block, this Boolean value was used along with the **i0** variable with an **OR** operator as follows:

```
if (ch || i0<1)

    i0=1;
    ch = 0;
```

Every time the change is detected, the calculations are performed again and the value of **ch** (change) is set to zero.

3.4 *Model Testing*

After implementing all the requirements for phase 1, a **test harness** was created for the model with a signal builder at the input. Total nine test cases were developed in the signal builder for better analysis of the model. **Fig. 3-4** shows the Test Harness for AWFG Model.

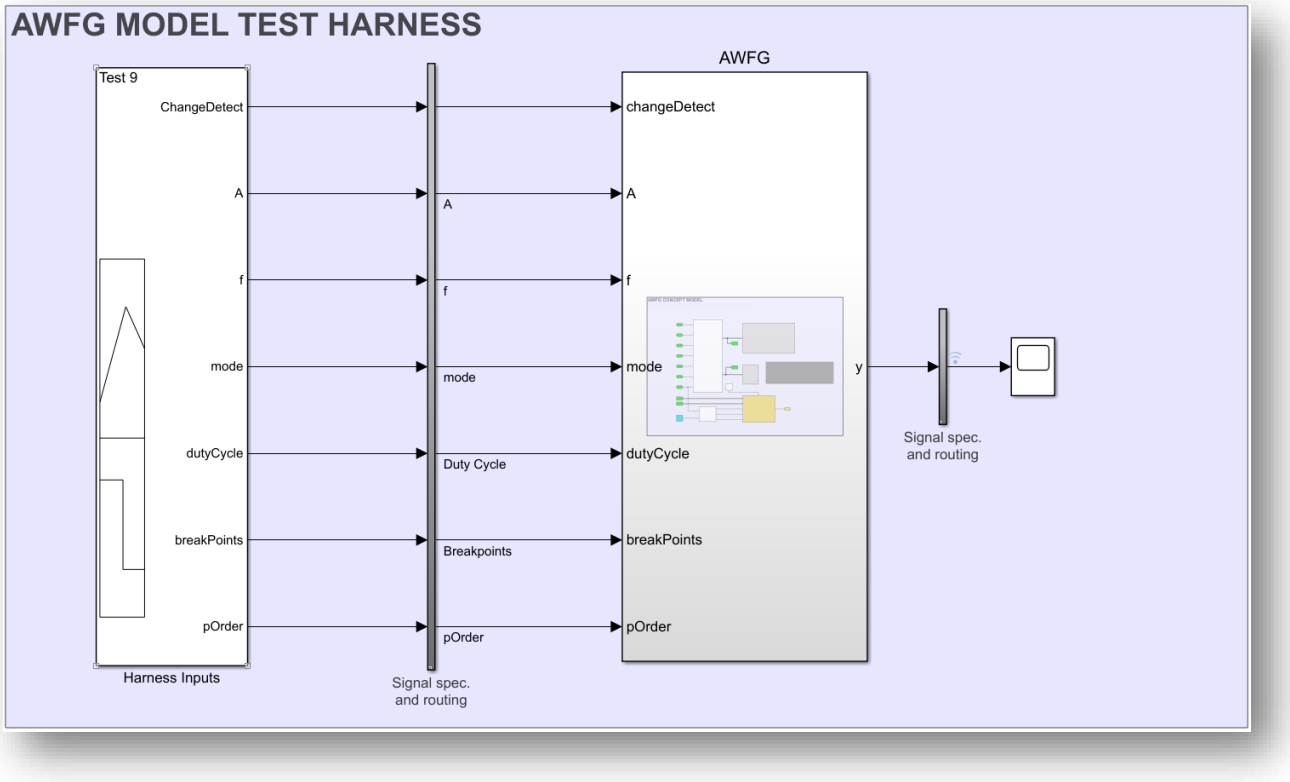

Fig. 3-4: AWFG Model Test Harness

Keeping in consideration the bounds of input signals (**Table 3-1**), nine cases were developed i.e. three cases for each type of wave. Table 3-3 shows the test cases used in signal builder.

Table 3-3: AWFG Model Test Cases

Inputs	Sine Wave			Square Wave			Triangular Wave		
	Test 1	Test 2	Test 3	Test 4	Test 5	Test 6	Test 7	Test 8	Test 9
Amplitude	0,8	0,5	1	0,8	0,5	1	0,8	0,5	1
Mode	0	0	0	1	1	1	2	2	2
Duty Cycle	0,5	0,5	0,8	0,5	0,8	0,5	0,5	0,8	0,5
Break Points	100	10	254	100	10	254	100	10	254
Order	0	0	2	0	0	2	0	0	2

These nine test cases were created in the signal builder as shown in **Fig. 3-5**.

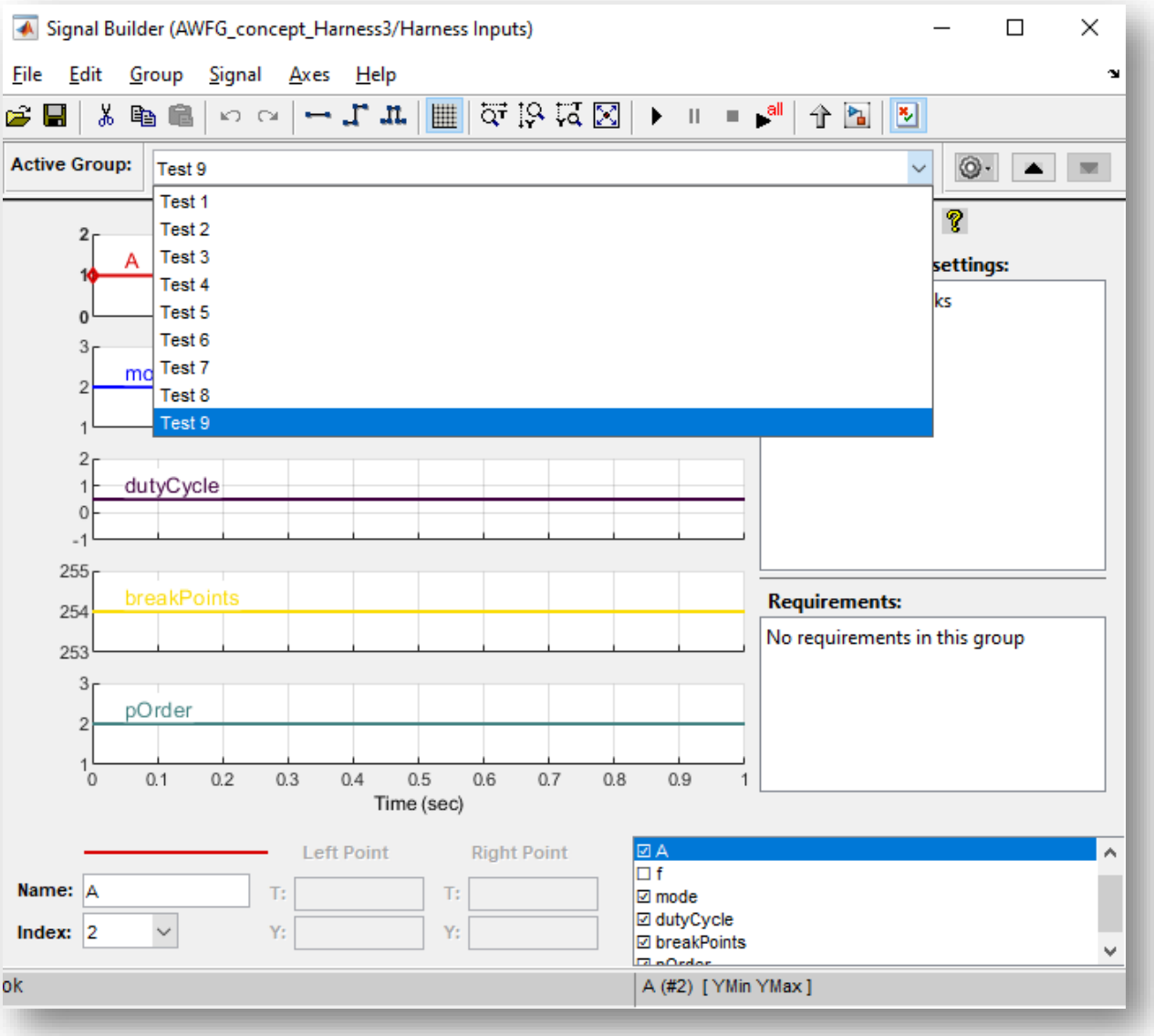

Fig. 3-5: AWFG Test Cases

These tests were performed together, and the outputs were analyzed in the **Test Manager. Fig. 3-6** to **Fig. 3-8** shows the results of these tests.

- **Sine Wave:**
  Test 1 to 3 were plotting sine wave of amplitudes **1**, **0.8**, and **0.5** with different breakpoints and polynomial order.

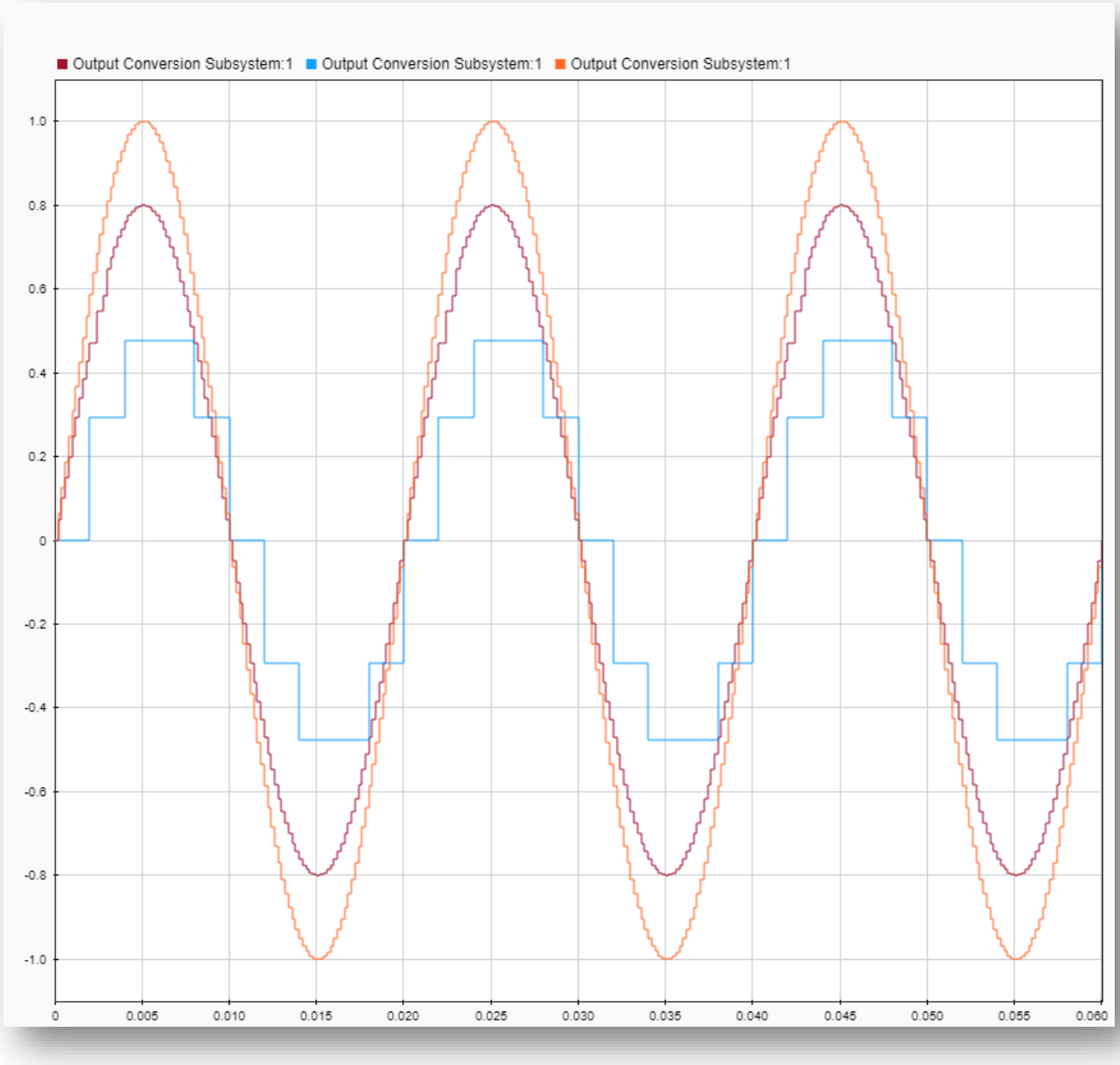

Fig. 3-6: Test Case 1 - 3 Results

- **Square Wave:**
  Test 4 to 6 were plotting square waves of amplitudes **1**, **0.8**, and **0.5** with different breakpoints and polynomial order. The **duty cycle** of wave with amplitude 0.5 was **0.8** as seen in the **Fig. 3-7**.

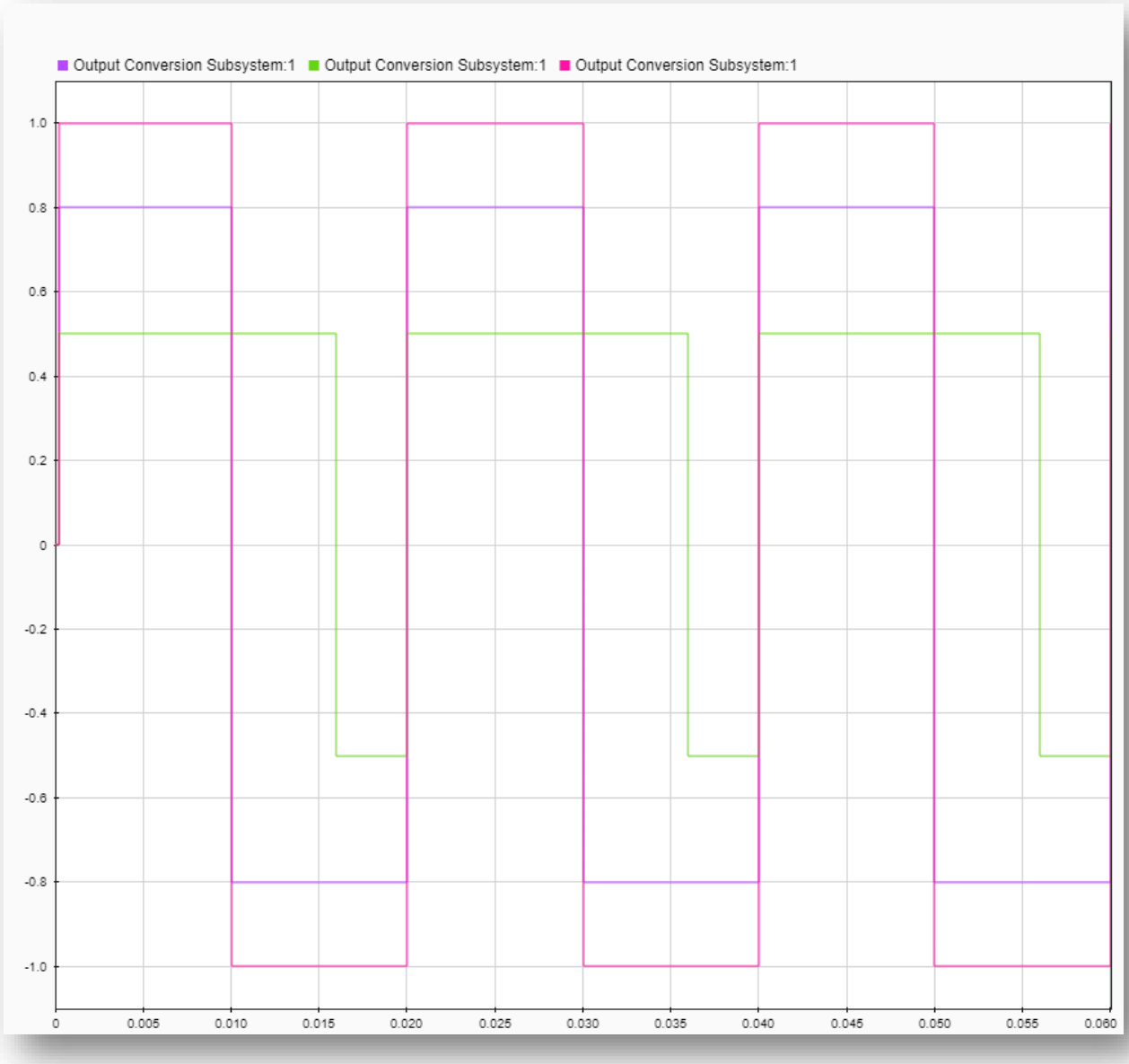

Fig. 3-7: Test Case 4 - 6 Results

- **Triangular Wave:**
  Test 7 to 9 were plotting triangular waves of amplitudes **1**, **0.8**, and **0.5** with different breakpoints and polynomial order. The **duty cycle** of wave with amplitude 0.5 was **0.8** as seen in the **Fig. 3-8**.

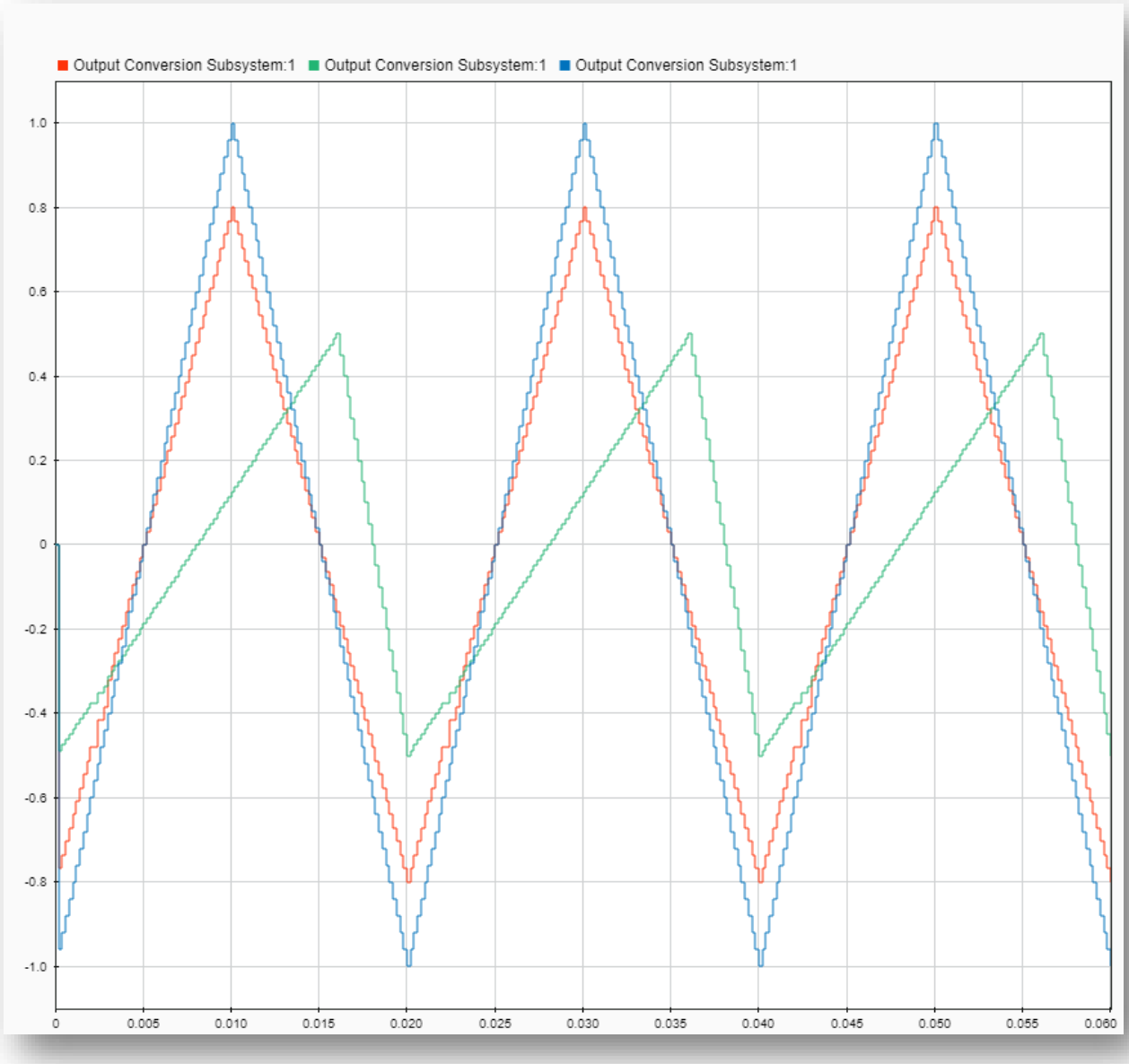

Fig. 3-8: Test Case 7 - 9 Results

The above shown test results verify the desired behavior of the model after **phase 1**. After performing these tests and few other combinations, it was observed that for **square** and **triangular** waves, using **0 order polynomials** yielded better results as both these waves have straight lines. Higher order polynomials could be used but with a higher number of breakpoints (above 200 ideally).

4. Preparation for Code Generation (Phase 2)

In the 2nd phase the model was further modified in order to make it applicable for the final ZYBO development board. These changes were implemented gradually as described in the next sections.

4.1 *Conversion to Single Precision Data*

To make the signals compatible for the use with FPGA all data signals were to be converted to single precision data (32 bits).

4.1.1 Data Type Propagation in Simulink

The Simulink engine propagates data from one block to the next along signal lines. The propagated data consists of

- Data Types
- Line Widths
- Sample Times

Propagation of the signal attributes are associated with the Inport block through a simple block diagram.

4.1.2 Modification of RT_CPU block to Work with Single Precision Data

To convert the data elements used inside the RT_CPU block, type casting to single was used "***single(x=1)***". All the data elements inside the function were casted as follows:

```
% Declaration of global variables
persistent cc0 b0 i0;
if isempty(cc0)
    cc0 = single(zeros(101,3));
    b0  = single(zeros(1,101));
    i0  = single(0);  %Used to checks if the program is running for the 1st
time
end
if (ch || i0<1)

    i0=single(1);
    ch = 0;

    %Bound values of breakpoints
    assert(breakPoints<=256);

    cc = single(zeros(breakPoints+1,3));        %% Creates coefficients array
    b  = single(zeros(1,(breakPoints + 1)));    %% Create breakpoints array
    tf = single(1/f);
    y  = single(zeros(1 , (breakPoints + 1)));
    x= single(linspace(0,1,breakPoints+1));
    x1= single(linspace(0,1,breakPoints));
```

To keep the data types consistent, all the inputs to the block were also converted to single precision type by changing the signal attribute of all the constant blocks as shown in **Fig. 4-1**.

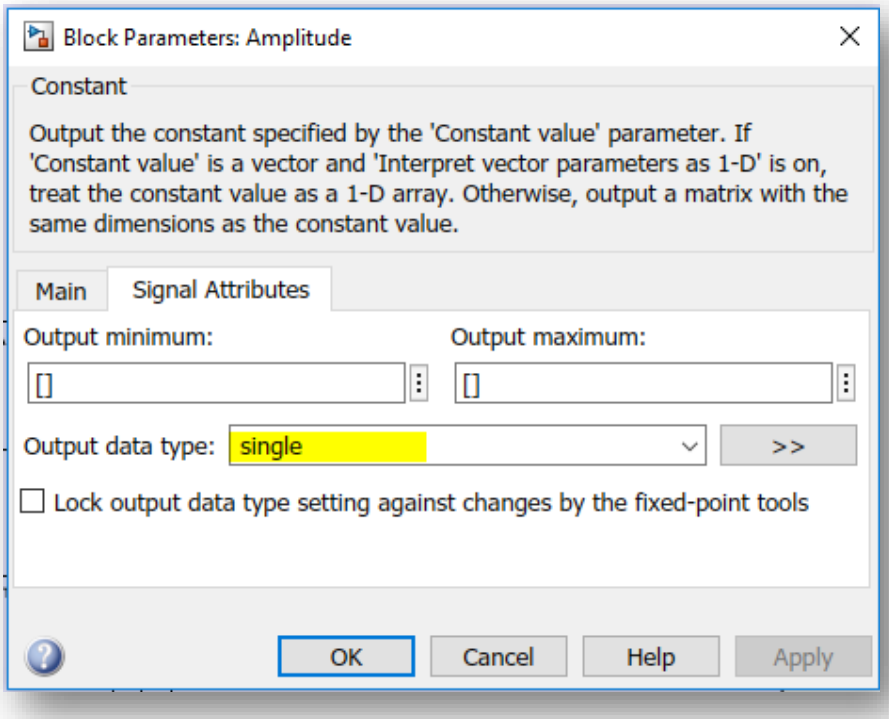

Fig. 4-1: Changing Data Type of Constant Blocks

The data type of constant blocks was verified using the model explorer as seen in **Fig. 4-2**.

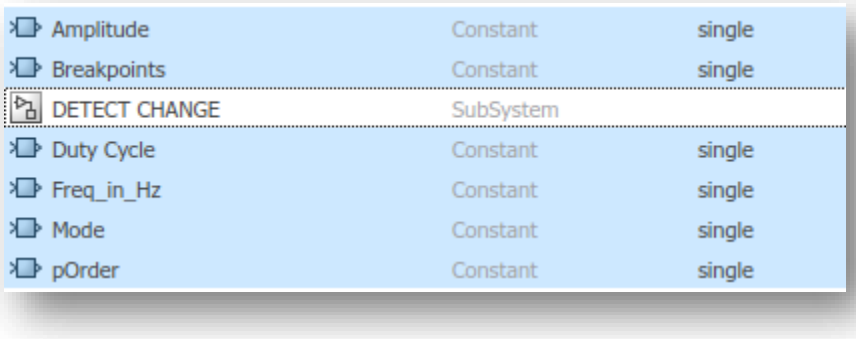

Fig. 4-2: Data Types of Constants Seen in Model Explorer

4.1.3 Model Testing and Comparison

After making these modifications, the tests performed in **section 3.4**, were performed again to verify that the model is working according to the desired behavior. All tests yielded approximately the same results as in section 3.4.

The modified model results were compared with the previous results using the Signal Analyzer and it was observed that there is difference in **amplitude** up to **6e-7**. The signal with double precision data was more accurate than the single precision data signal. Fig. 4-3 shown the comparison between the two signals.

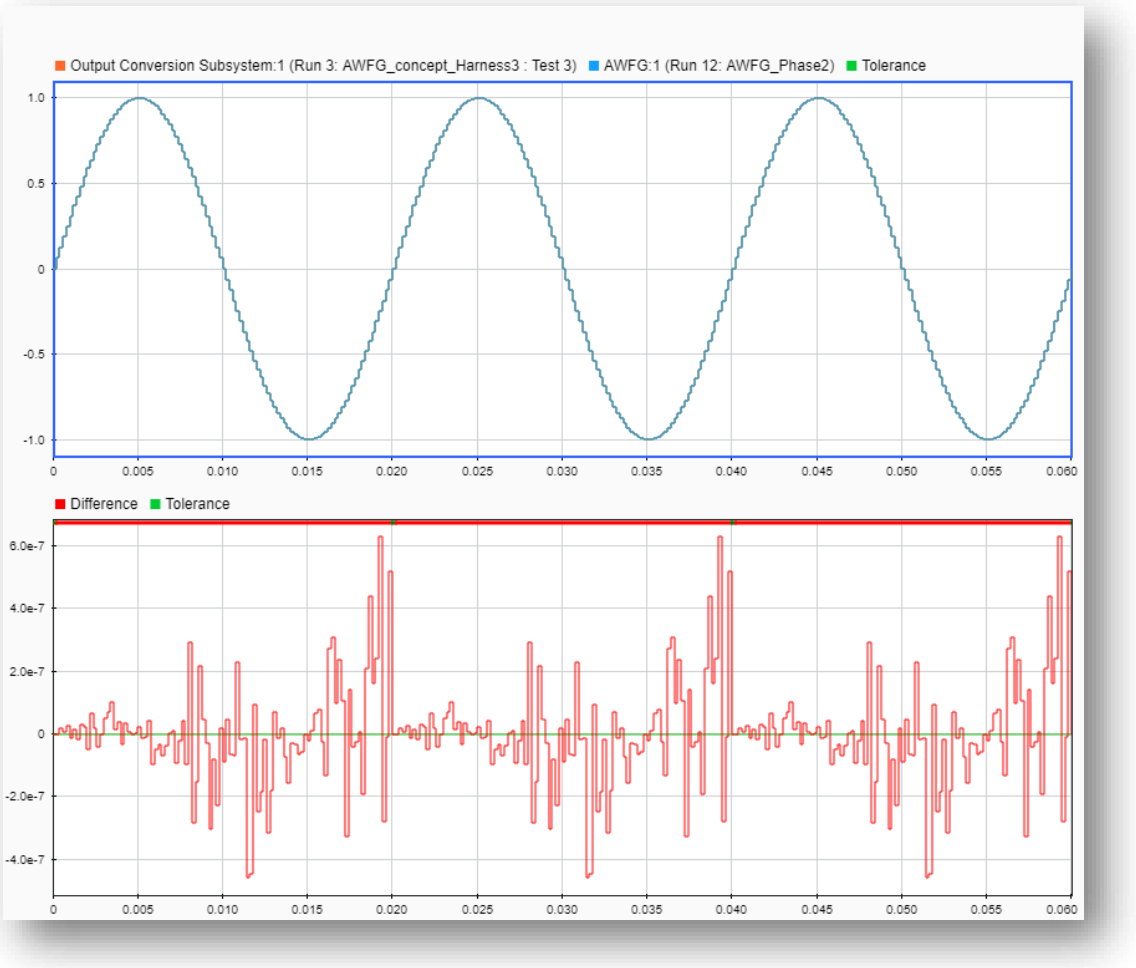

Fig. 4-3: Single Precision Data Signal vs. Double Precision Data Signal

The reason for this difference is that the double precision data type (**double**) requires **64-bits** and hence has more fractional bits (**52** least significant bits), as compared to the **single** precision data type which requires **32-bits** as has only **23** fractional bits.

4.2 *Separation of Model into Static & Dynamic Parts*

To make the clear separation of **CPU** calculations and **FPGA** calculations, the model was divided into two parts i.e. **Static** Calculations and **Dynamic** Calculations, as seen in **Fig. 4-4**.

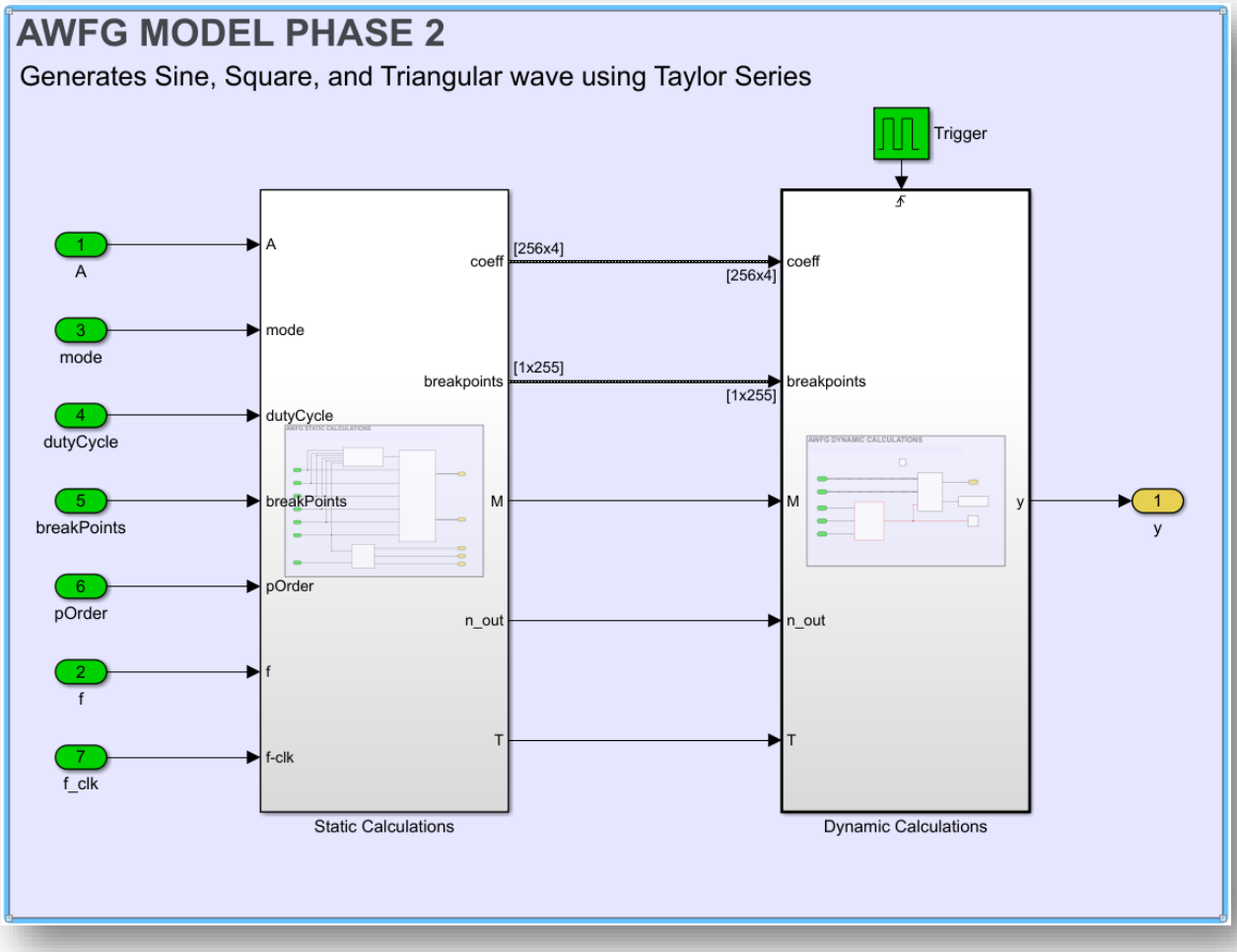

Fig. 4-4: AWFG Model Separated into Two Parts

4.2.1 Static Calculations Block (CPU)

This block is responsible for generating the coefficients and breakpoints. Since the calculations are performed only once for one wave, this block can be run on **ARM** core of ZYBO Board. **Fig. 4-5** shows the static calculations block.

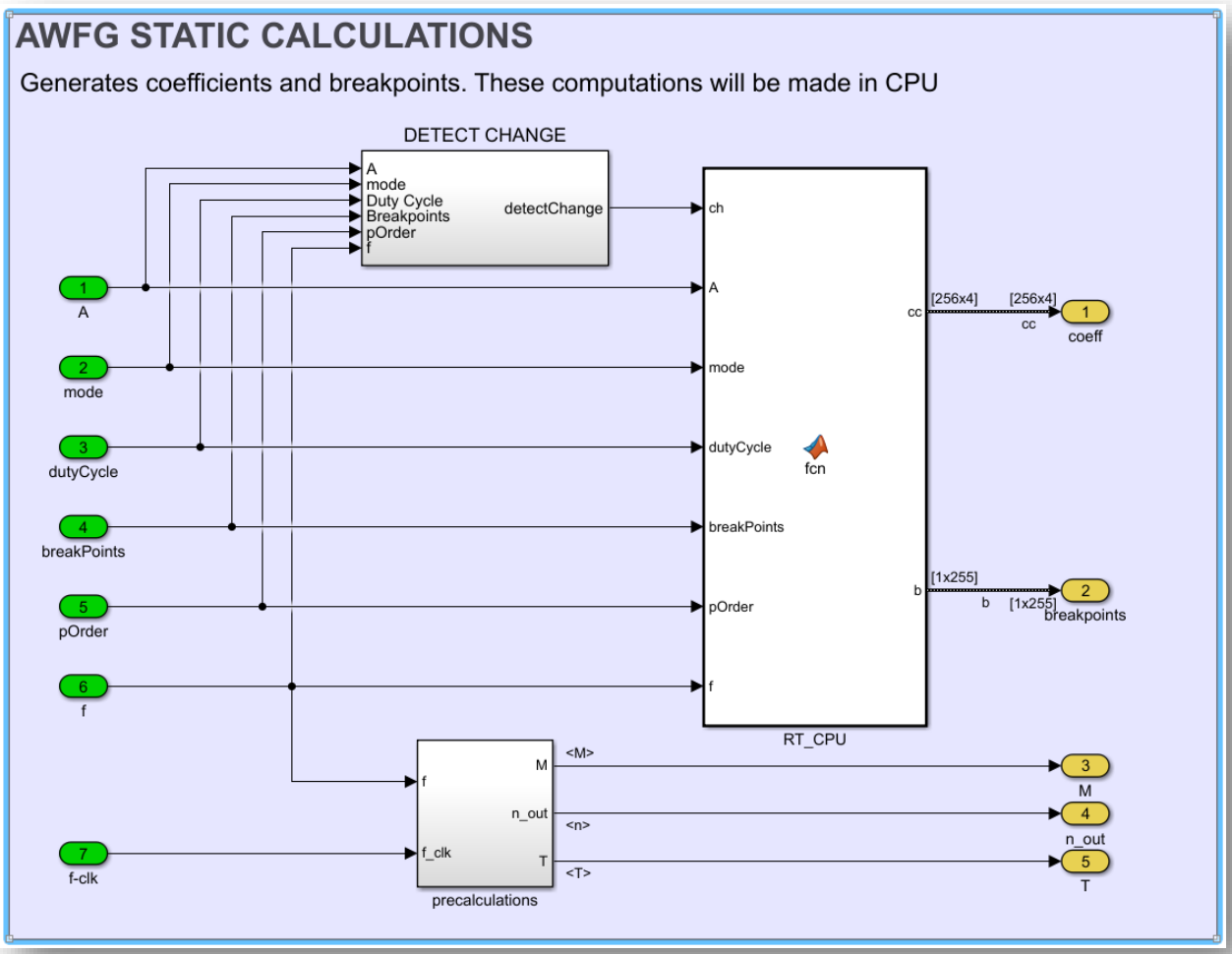

Fig. 4-5: Static Calculations Block

4.2.2 Dynamic Calculations Block (FPGA)

This block is responsible for **time generation** and solving the **polynomial**. For the accurate waveform generation, the time generation and polynomial solving needs to be done in **FPGA** to meet the real time requirements. **Fig. 4-6** shows Dynamic Calculations block.

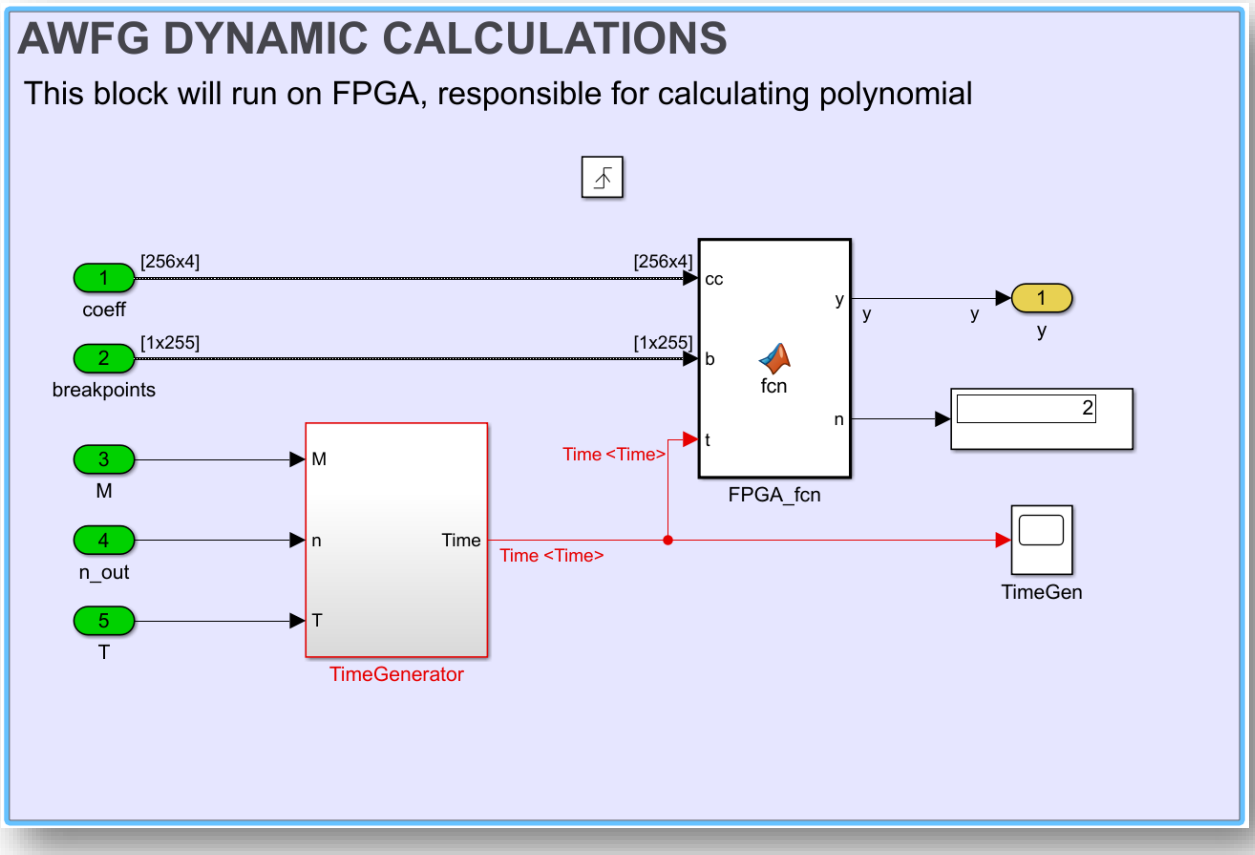

Fig. 4-6: Dynamic Calculations Block

4.3 *Generation of C-code for Static Block*

In this step the C-code for the Embedded **Real-Time Target (ERT)** of static block was generated. The system target file can be selected in the 'Configuration Parameters' panel as shown in **Fig. 4-7**.

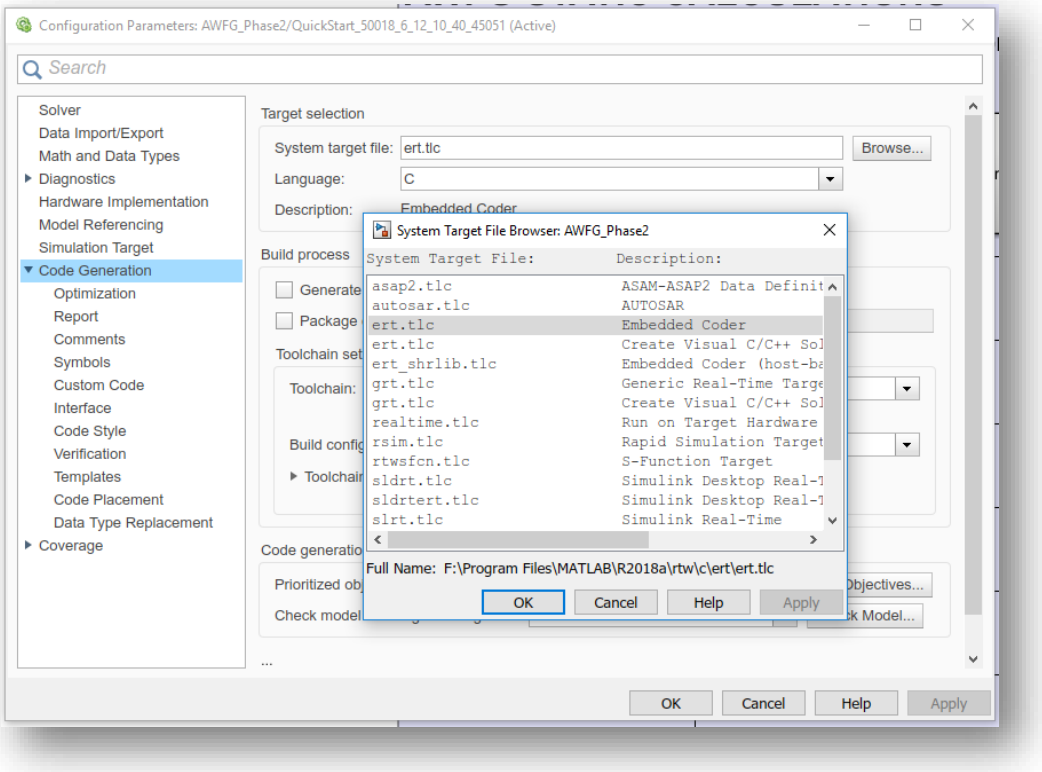

Fig. 4-7: Selection of System Target File

Code generated for ERT has production **quality** and has **reusable** unique code for each block.

As RT_CPU block needs to run on CPU, C-code was generated by right clicking on ***RT_CPU block >> C/C++ Code >> Build This Subsystem***. This generated a folder named '**RT_CPU_ert_rtw**' in the workspace which contains the generated c-code. This generated code was analysed for the better understanding.

4.3.1 Analysis of Generated C-Code

After analyzing the generated code files the following observations were made:

4.3.1.1 How the code is separated?

The C-code was separated into four different files i.e. **ert_main.c**, **RT_CPU.c**, **RT_CPU.h**, **rtwtypes.h**

- **RT_CPU.h**:
  This file contains the declaration of data structures for **block signals & states**, **inputs**, **outputs**, and **real-time model**. Two main functions are also declared in this file i.e.

'**RT_CPU_initialize**' (initialises the inputs and outputs of the model) and '**RT_CPU_step**' (called periodically based on the system sample time).

Objects of all these structures are created in this file and made available to access from outside as follows:

```
/* Block signals and states (default storage) */
extern DW rtDW;

/* External inputs (root inport signals with default storage) */
extern ExtU rtU;

/* External outputs (root outports fed by signals with default storage) */
extern ExtY rtY;

/* External outputs size (variable-sizing root outport signals with default
storage) */
extern ExtYSize rtYSize;

/* Model entry point functions */
extern void RT_CPU_initialize(void);
extern void RT_CPU_step(void);
```

- **RT_CPU.c**:
  This file has the definition of above mentioned entry point functions i.e. '**RT_CPU_initialize**' and '**RT_CPU_step**' declared in **RT_CPU.h**. The main logic of the MATLAB code is implemented in this file. To access the input, output, and the block signals, the objects of the data structures created in RT_CPU.c file i.e. **rtDW** (block signals and states), **rtU**(inputs), **rtY**(outputs), were used.

- **ert_main.c**:
  This file contains the main function for the continuous execution of the program. The RT_CPU.h file is imported in this file. The main function first initializes the I/O ports using the function **RT_CPU_initialize** and then runs in a while loop until an error is encountered.

- **rtwtypes.h**:
  This file contains all the data types that could be used in the model's generated C-code. These data types are divided into two main groups; **Fixed width word size** and **Generic** data types. **Fixed width word size** data types are bit specific as shown below:

```
/*===========================================================================*
 * Fixed width word size data types:                                         *
 *   int8_T, int16_T, int32_T     - signed 8, 16, or 32 bit integers         *
 *   uint8_T, uint16_T, uint32_T  - unsigned 8, 16, or 32 bit integers       *
 *   real32_T, real64_T           - 32 and 64 bit floating point numbers     *
 *===========================================================================*/
typedef signed char int8_T;
typedef unsigned char uint8_T;
typedef short int16_T;
typedef unsigned short uint16_T;
typedef int int32_T;
typedef unsigned int uint32_T;
typedef long long int64_T;
typedef unsigned long long uint64_T;
typedef float real32_T;
typedef double real64_T;
```

4.3.1.2 What Data Types were Used?

As all the inports were made single precision data type (32-bits), all the inputs and outputs in the code were declared with the **32-bit** data types (except one Boolean type, which is used by the signal to detect change in the inputs). All the inputs and outputs were declared in the RT_CPU.h file as a data structure for better classification. The declaration of signals was as follows:

```
/* Block signals and states (default storage) for system '<Root>' */
typedef struct {
  emxArray_real32_T_256x3 cc0;         /* '<Root>/RT_CPU' */
  emxArray_real32_T_1x255 b0;          /* '<Root>/RT_CPU' */
  real32_T i0;                         /* '<Root>/RT_CPU' */
  int32_T SFunction_DIMS2[2];          /* '<Root>/RT_CPU' */
  int32_T SFunction_DIMS3[2];          /* '<Root>/RT_CPU' */
  boolean_T cc0_not_empty;             /* '<Root>/RT_CPU' */
} DW;

/* External inputs (root inport signals with default storage) */
typedef struct {
  boolean_T ch;                        /* '<Root>/ch' */
  real32_T A;                          /* '<Root>/A' */
  real32_T mode;                       /* '<Root>/mode' */
  real32_T DutyCycle;                  /* '<Root>/dutyCycle' */
  real32_T Breakpoints;                /* '<Root>/breakPoints' */
  real32_T pOrder;                     /* '<Root>/pOrder' */
  real32_T f;                          /* '<Root>/f' */
} ExtU;

/* External outputs (root outports fed by signals with default storage) */
typedef struct {
  real32_T cc[1024];                   /* '<Root>/cc' */
  real32_T b[255];                     /* '<Root>/b' */
} ExtY;
```

```
/* External output sizes (for root outports fed by signals with variable
sizes) */
typedef struct {
  int32_T cc[2];                       /* '<Root>/cc' */
  int32_T b[2];                        /* '<Root>/b' */
} ExtYSize;
```

As seen from the above code, all the inputs and outputs and internal signals are using **fixed width** word size data types of **32-bits**, except one **Boolean** signal.

4.3.1.3 How Inputs and Outputs are Managed?

As described in the previous **section 4.3.1.2**, the inputs and outputs are classified using data structures. **ExtU** type is used to classify inputs and **ExtY** type is used for outputs.

These structures were created in RT_CPU.h file and an object to each type (**rtU**, **rtY**) was also created in the same file to be used in RT_CPU.c file for accessing the inputs and outputs. This approach makes the code more manageable and organized.

4.3.2 Validation of SIL Algorithm

For the testing of the generated code, the SIL block was created by choosing ***C/C++ Code >> Generate S-Function*** from the context menu (Make sure there are no spaces in the workspace path).

A third-party compiler (suggested by MATLAB) '**MinGW64 Compiler (C)**' was installed in MATLAB to get rid of building errors.

This generated a separate model for RT_CPU block. In the main model the MATLAB function block was replaced with the generated block. **Fig. 4-7** shows the model with replaced SIL block.

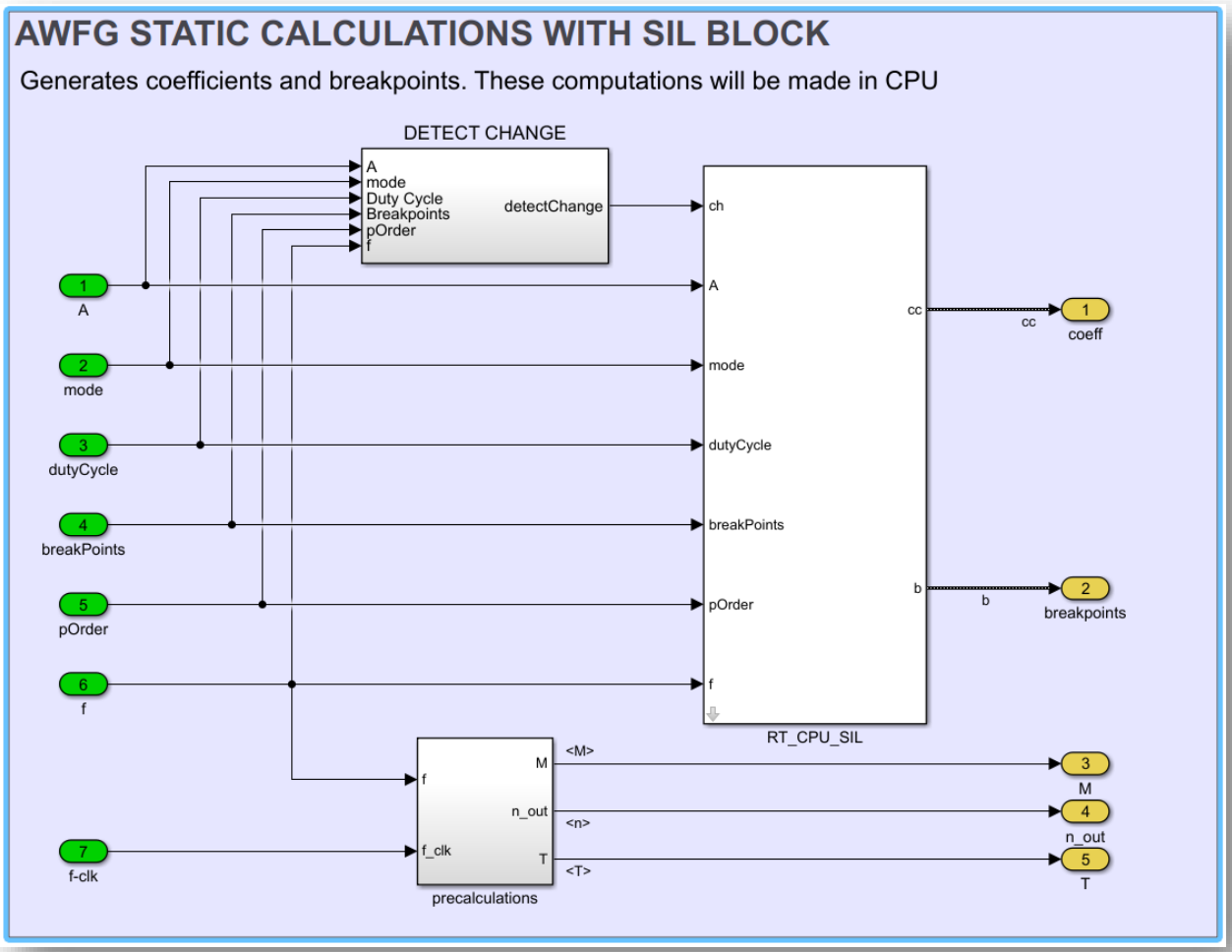

Fig. 4-8: RT_CPU SIL Model

After replacing the MATLAB function block wit SIL block, the same tests were performed on the model as developed in **section 3.4** using the new test harness for the model with SIL block. **Fig. 4-9** shows the test results for all three waves with **50% duty cycle**, **50 Hz** frequency, and amplitude **1**.

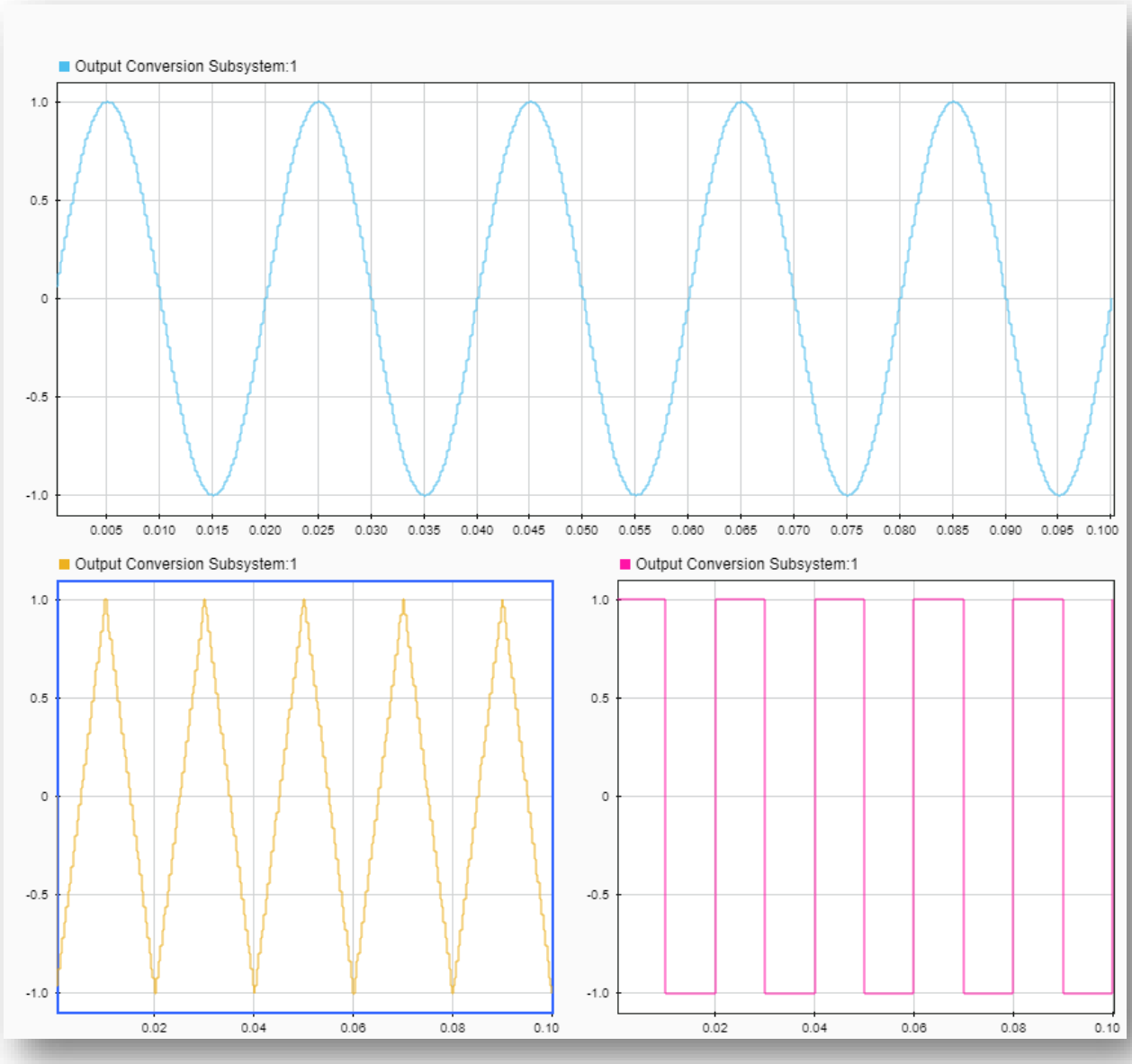

Fig. 4-9: Test Results with SIL Block; 50% Duty Cycle, 50 Hz

This SIL was only used for verification that the generated code is executable. S-functions have a particular structure useful in the context of Simulink; but not practical for embedded system programming. Hence, after the successful test results, the further modification of the model was done using the MATLAB function block.

4.4 *Development of Fixed-Point Arithmetic Model for FPGA*

In order to prepare model for its deployment on FPGA, it needed to have fixed point format for an efficient design. Some other steps were also taken for better performance.

4.4.1 Sequential transfer of coefficients

The static calculations block was further divided in such a way that the coefficients are transferred sequentially, first by splitting rows (break point) and then by elements within the row (individual coefficients), till the total number of break points is reached (128 breakpoints were chosen as default). That is required as the coefficient will be stored at corresponding RAM memory locations that the FPGA can also read.

Fig. 4-10 shows how static calculations block was introduced with another subsystem for the sequential transfer of coefficients. The input to this block was the **coefficients arra**y and a **reset signal**, that was implemented in RT_CPU block and would generated every time there's a change in input parameters.

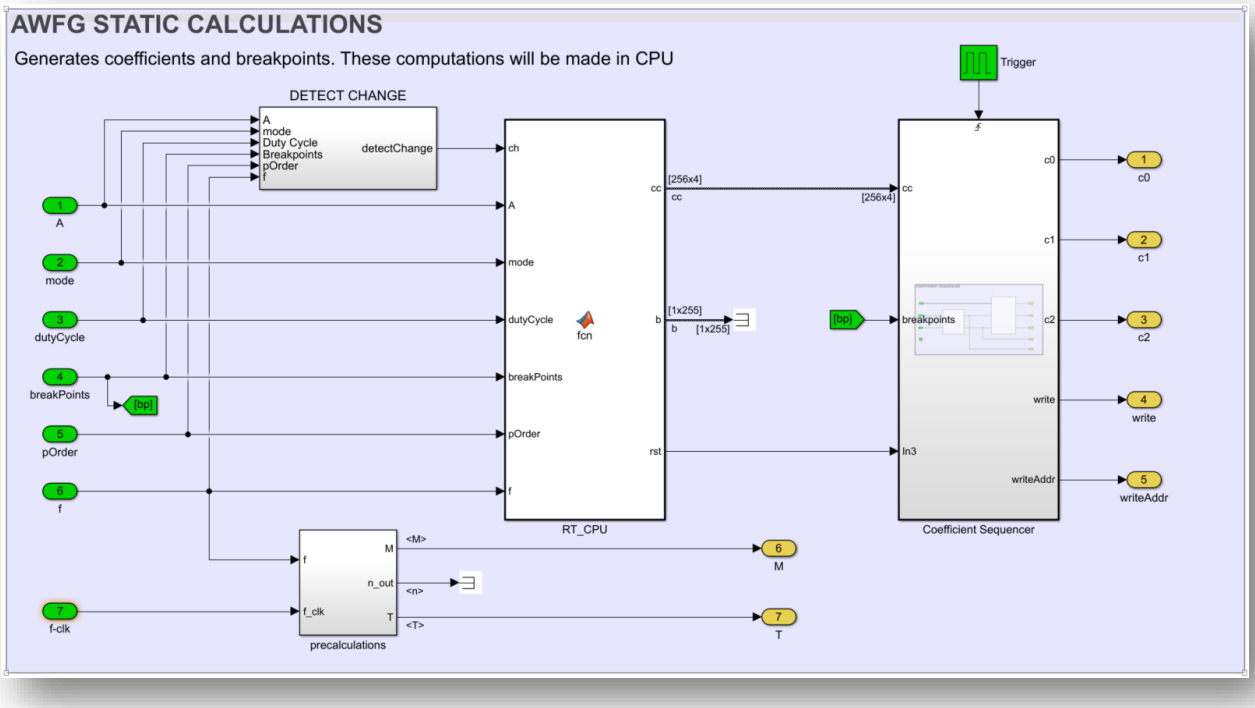

Fig. 4-10: Static Calculations Block with Coefficient Sequencer

The Coefficient Sequencer Block was further divided into two parts as shown in **Fig. 4-11**. A **Counter** block and a **Coefficient Selector** block, the details of which are provided in the next sections.

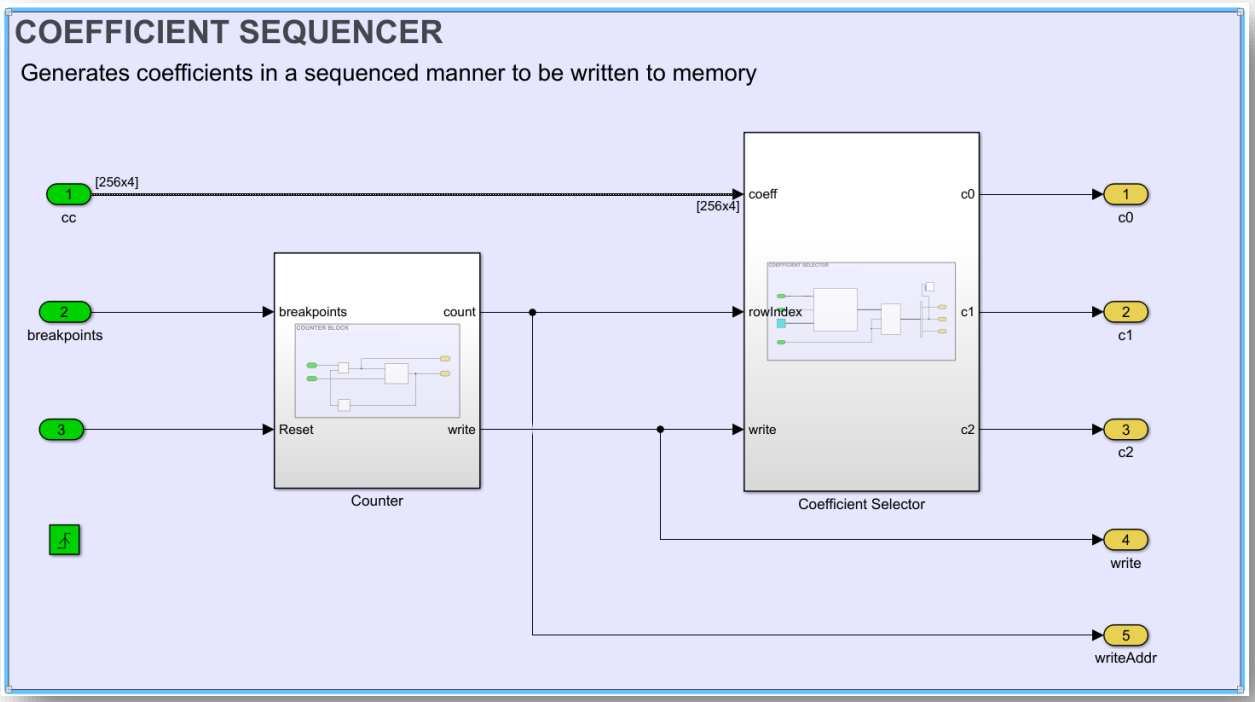

Fig. 4-11: Coefficient Sequencer Block

4.4.1.1 Counter for Sequential Transfer

A counter block was implemented which could increment sequentially, after every clock cycle, till the number of breakpoints, **128** by default. The counter was needed to split the coefficients array into rows.

The counter block checks if the counter value is less than the number of breakpoints. The counter increments if the condition is true. A **write** signal was also generated which would indicate that the counter is still incrementing, and the coefficients needs to be written to the memory. **Fig. 4-12** shows the counter block.

The starting index of he matrix was **0** and the end index was **128**.

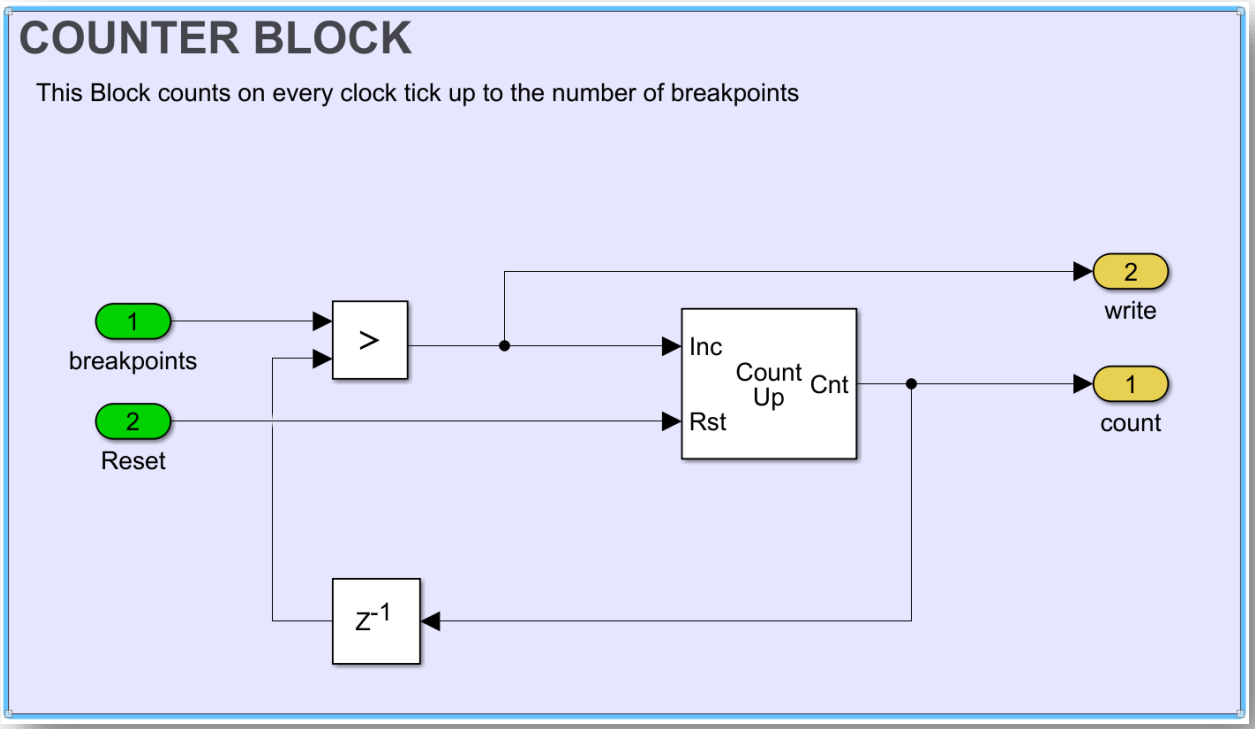

Fig. 4-12: Counter Block

4.4.1.2 Coefficient Selector Block

After generating the sequential counter for the rows, the matrix needed to be split in separate rows, using the counter value, and then in separate coefficients. A 2-D Selector block was used for spitting the rows. The starting **index of row** was given as the **counter** value, and a constant block was used to provide the **range [0:2]** for row indexes. **Write** signal was used to perform a check; if **write** is **enable**, transfer the coefficients otherwise set all coefficients to zero. The selector block was followed by a **Demux** to split the row into 3 separate coefficients. **Fig. 4-13** shows the selector block.

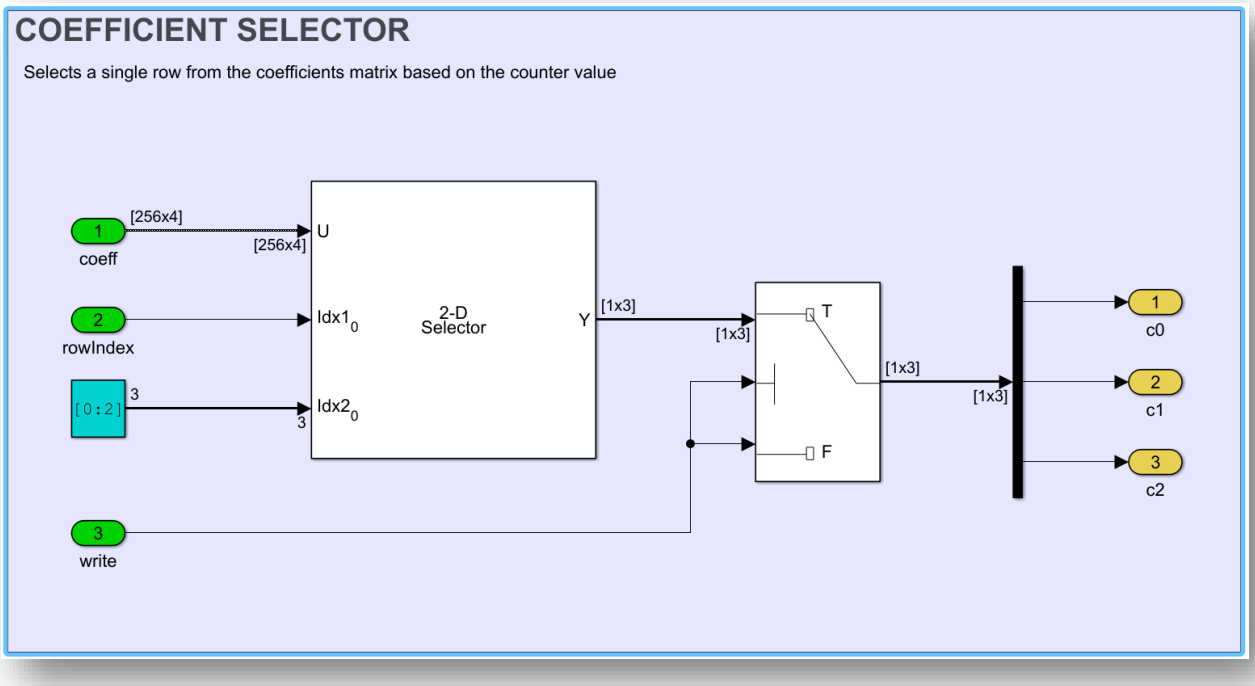

Fig. 4-13: Coefficient Selector Block

4.4.2 Scaling and Casting Data for FPGA

Before transferring data from Static to dynamic calculation block, it needed to be converted into fixed point data format as FPGAs work with fixed point data format. A 'Data Casting and Scaling' block was added in-between static and dynamic calculations block. This data was scaled up, to **2^16**, to avoid any data loss during casting of data. **Fig. 4-14** shows casting and scaling of data.

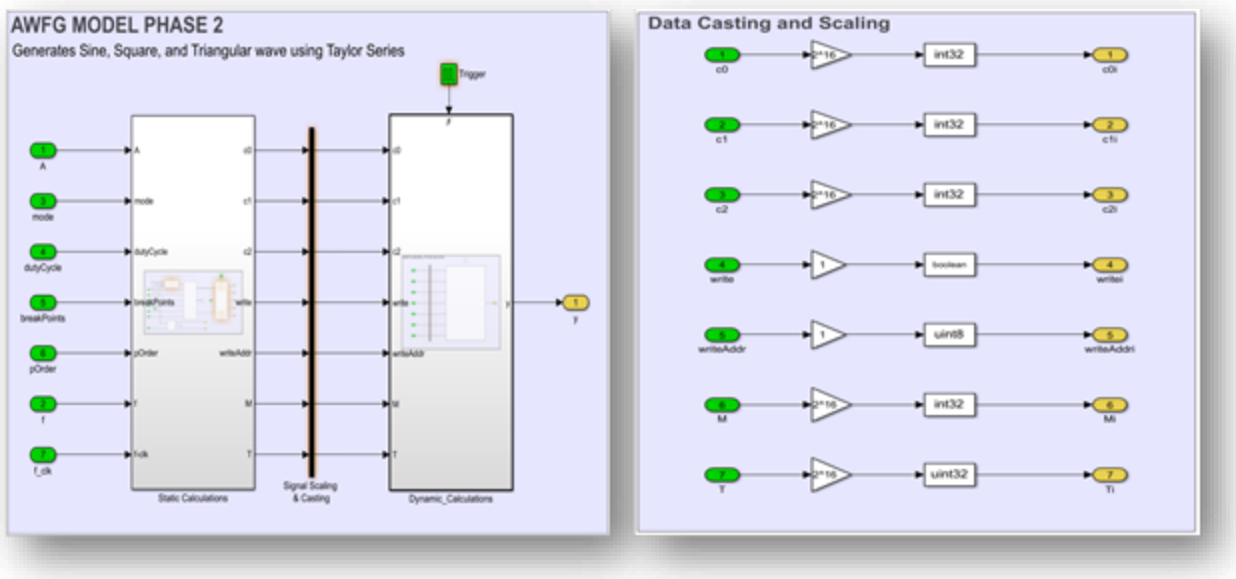

Fig. 4-14: Scaling and Casting of Data

This scaled and casted data was then scaled down in the Dynamic calculations block to get the actual values. After descaling, this data was also casted (details in the next sections). Fig. 4-15 shows the descaling of data in FPGA part.

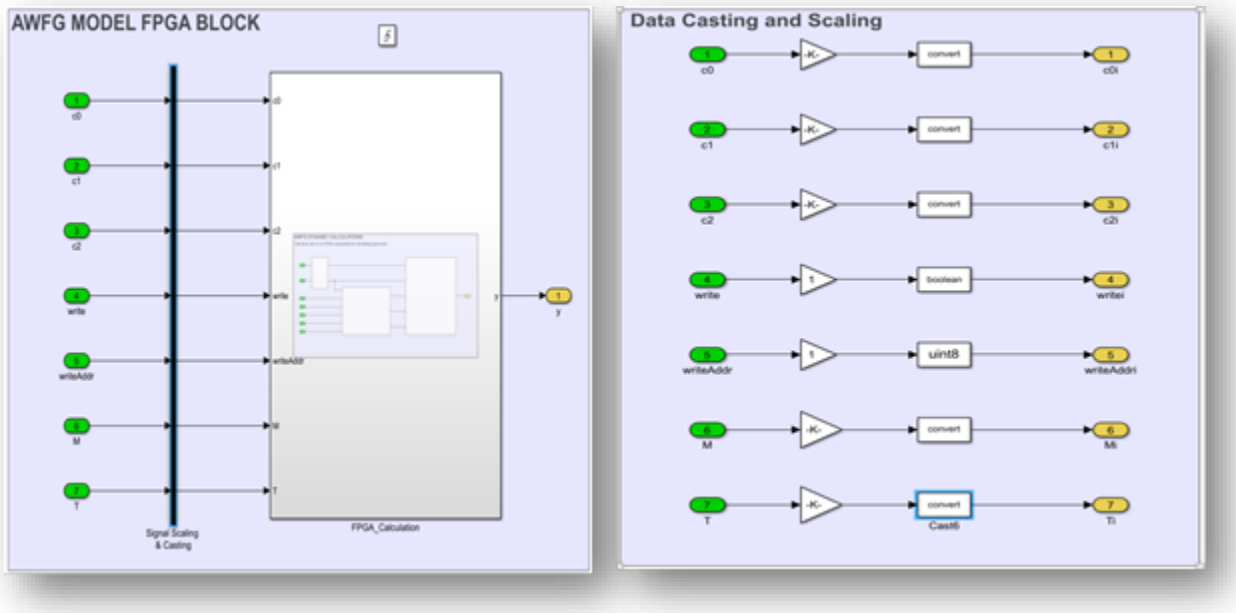

Fig. 4-15: Descaling of data in FPGA

4.4.3 Time Generator and Index Block

The calculation of polynomial in FPGA needs a time signal which was being generated by Time generator block. This block was modified according to the required precision and the number of breakpoints.

4.4.3.1 Precision and Overflow of Accumulator

Fractional numbers in fixed point format are handled differently. The decimal numbers after the decimal point are calculated with the negative powers of 2. Considering the FPGA clock and the selected frequency, the required precision bits for the range of M were calculated. **Table 4-1** shows the ranges of M.

Table 4-1: Precision Bits for Frequency Range

Frequency (Hz)	Value of M	Precision Bits
50	0.128e-3	19
1000	2.56e-3	18

The accumulator output was required to have a precision of 16 bits. However, to keep higher precision for the desired frequency range **19 precision bits** were chosen.

Since the number of breakpoints were set to 128, the resting of accumulator was done by automatically by overflow. For 128 bits 7 bits are needed. Hence the accumulator output was made 26 bits without and sign bit as shown in **Fig. 4-16**

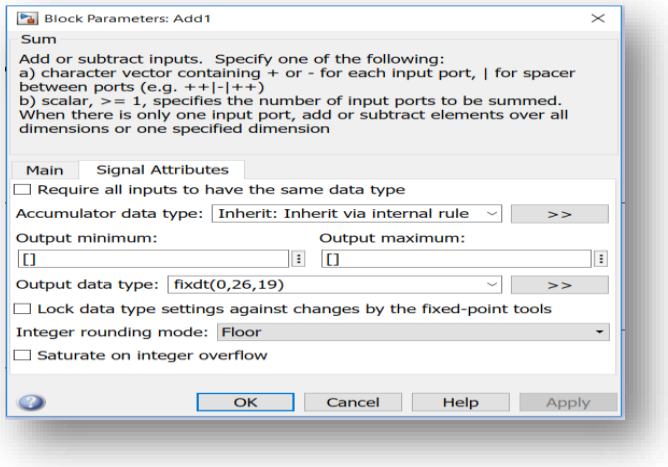

Fig. 4-16: Accumulator Data Type

Fig. 4-17 shows the Time Generator and Index block.

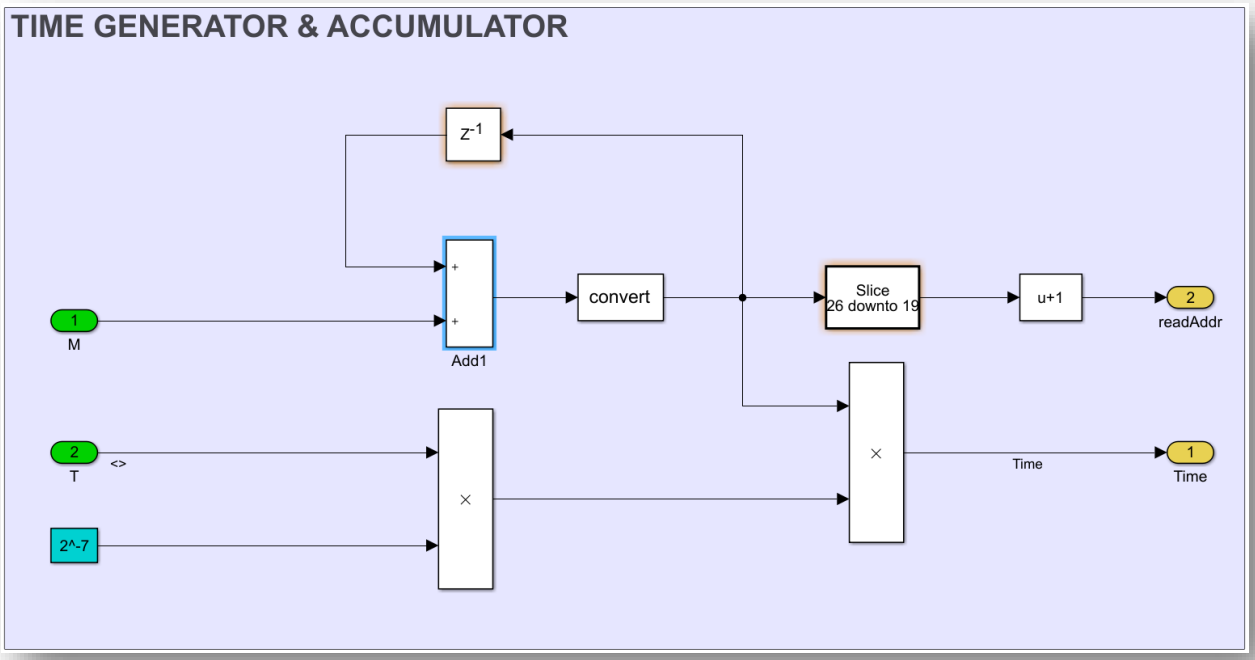

Fig. 4-17: Time Generator Block

As all the data elements were implemented with a sign bit, the output of accumulator was casted to **27-bit** data with a **sign bit** as shown in the figure above.

A **read address** signal was also generated using the MSBs of the accumulator which goes up to 128. This was done by using a **Slice block** and **26 -19** bits were sliced and given as offset of one use a bias block. This signal was used to specify the **address** on **RAM** to read from.

4.4.4 Reading and Writing Coefficients from RAM

The above mentioned sequentially generated coefficients were stored in Single-Port RAMs. For three coefficients, 3 single-port RAMs were used.

A write signal, that was generated in Counter block (**section 4.4.1.1**), was used as a selector signal for reading and writing to the RAMs. Read signal was taken from the Time generator and index block. **Fig. 4-18** shows the Write to Memory Block implementation.

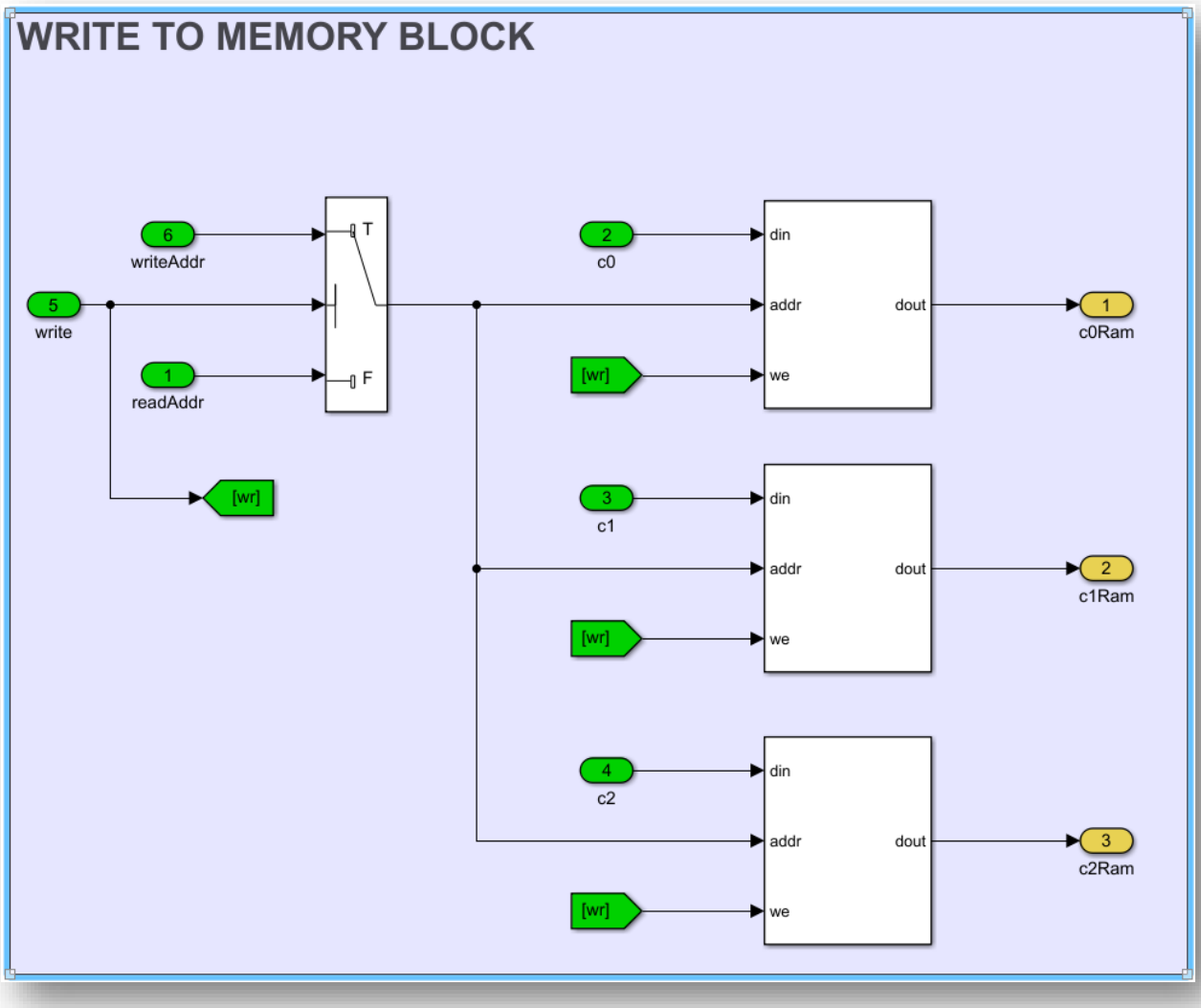

Fig. 4-18: Read/Write to RAM Block

4.4.5 Solving Polynomial

In the last step the polynomial mentioned in **equation 4.1** was solved up till 2nd order.

$$f(x) \approx c_0 + c_1(x - x_0) + c_2(x - x_0)^2 + \cdots + c_n(x - x_0)^n \qquad 4.1$$

The expression (x-x_0) was calculated first by detecting the change of the index (MSBs of the accumulator) and store the actual time value as x_0. This value was then subtracted from the actual time value forming the dt. **Fig. 4-19** shows the Time Difference calculator block.

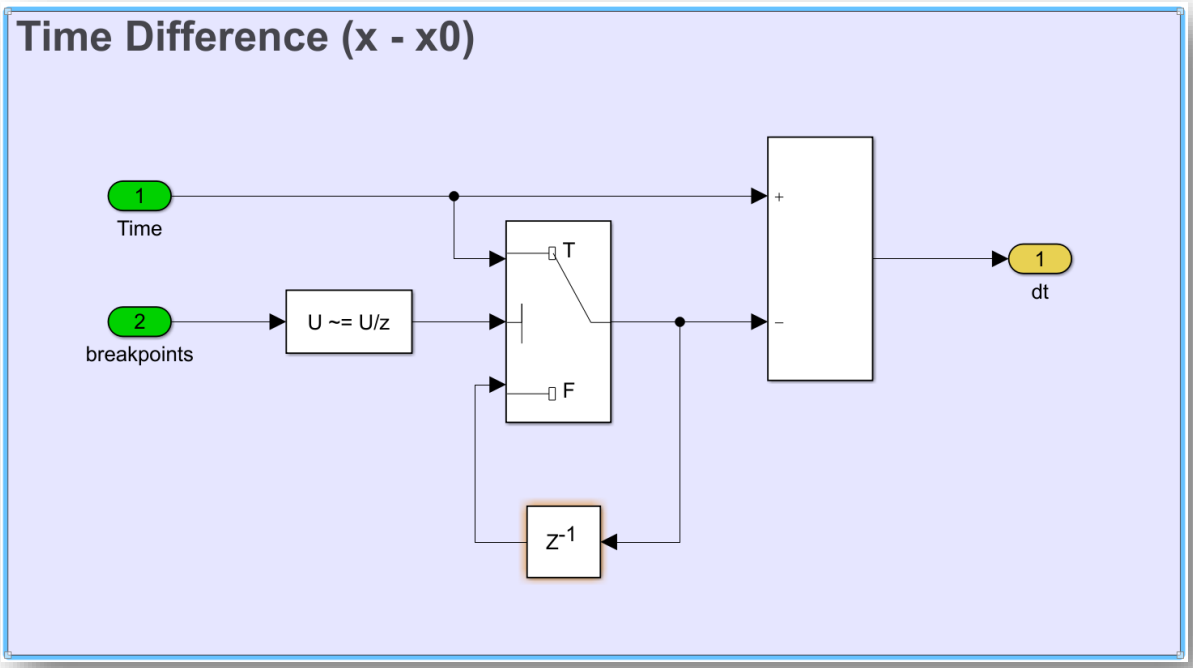

Fig. 4-19: Time Difference Calculation Block

After calculating ***dt***, the coefficients were multiplied with it using pipelining to break the combinational path. **Fig. 4-20** shows the Polynomial solver block.

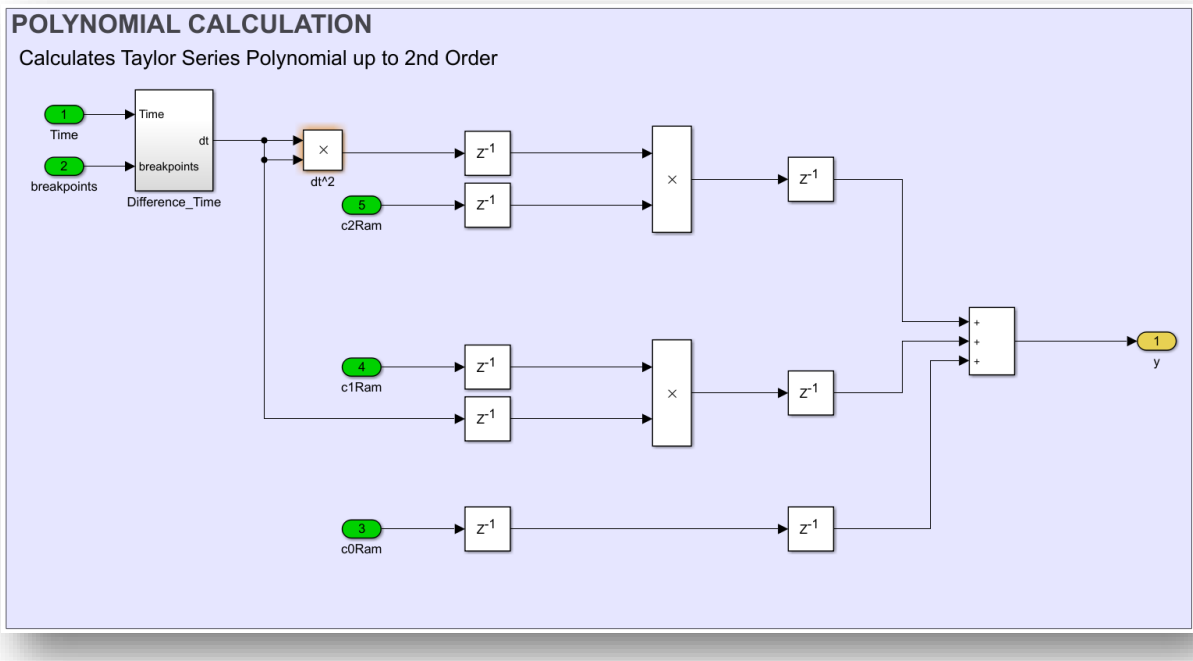

Fig. 4-20: Polynomial Calculation Block

4.5 *Generation of VHDL-code*

After connecting all the blocks in FPGA part, as shown in **Fig. 4-21**, the VHDL code was generated for the Dynamic calculation block as per the mentioned steps.

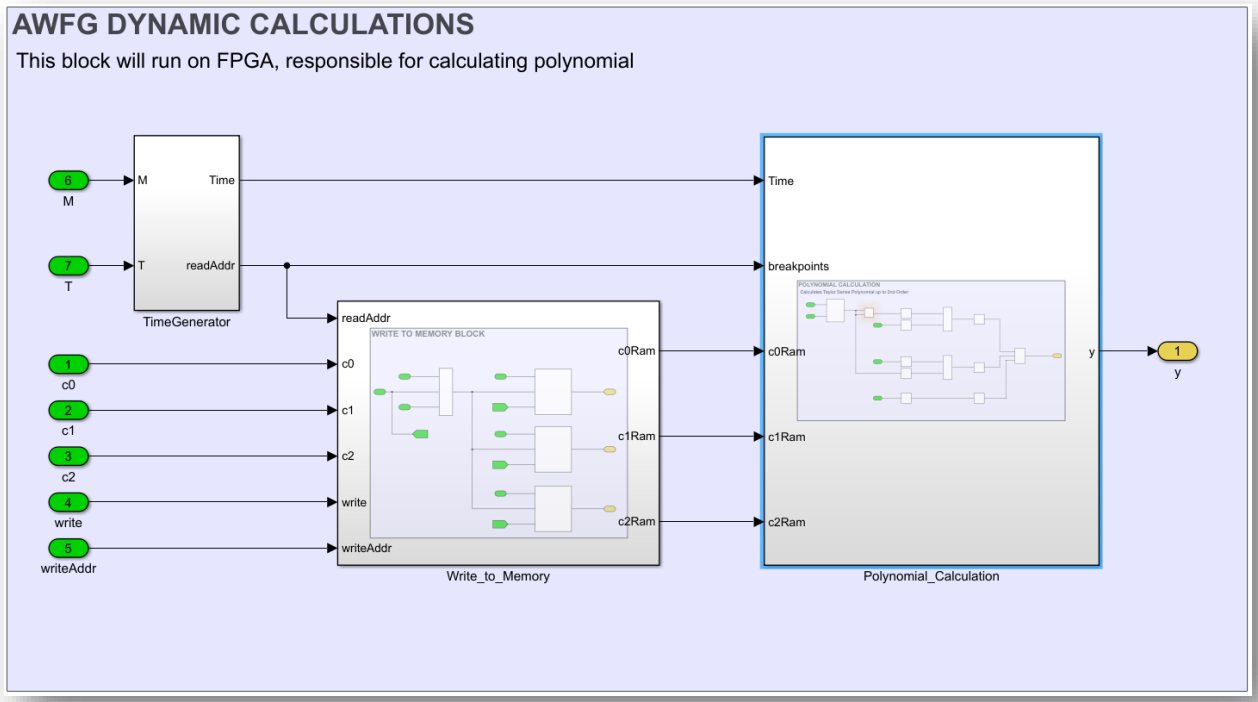

Fig. 4-21: FPGA (Dynamic) Calculations Block

Before generation, the trigger signal was removed from the blocks as VHDL coder does not support trigger signals. After the successful generation of VHDL code, the code generation report was analyzed. Fig. 4-22 shows Critical Path Details for the generated code.

Critical Path Details

Id	Propagation (ns)	Delay (ns)	Block Path
1	0.0000	0.0000	Gain6
2	0.0000	0.0000	Cast6
3	6.4270	6.4270	Product1
4	12.6970	6.2700	Product
5	13.7550	1.0580	Switch
6	15.0150	1.2600	Add
7	21.4420	6.4270	dt^2
8	21.4720	0.0300	Delay

Fig. 4-22: HDL Code Generation Critical Path Details

The report showed that the product(multiplication) blocks are producing the highest delays.

A test bench was also generated with co-simulation available. Co-Simulator lets the generated VHDL code run simultaneously on the target simulator (Modelsim).

The model was tested again after the code generation, with the same test prepared in the previous sections. All tests were passed. The VHDL code could also be simulated in Modelsim using co-simulation.

5. Integration and Testing of AWFG in ZYBO

The integration of the AWFG model created in the previous sections was done using the Simulink Model Template for ZYBO Board Development.

5.1 *Analysis of Template for ZYBO*

The template model consists of two subsystems, **CandHDL_c (PS)** and **CandHDL_hdl (PS)**. All the available peripherals on the ZYBO are connected in the framework model to the FPGA subsystem. **Fig. 5-1** shows the template model for ZYBO Board.

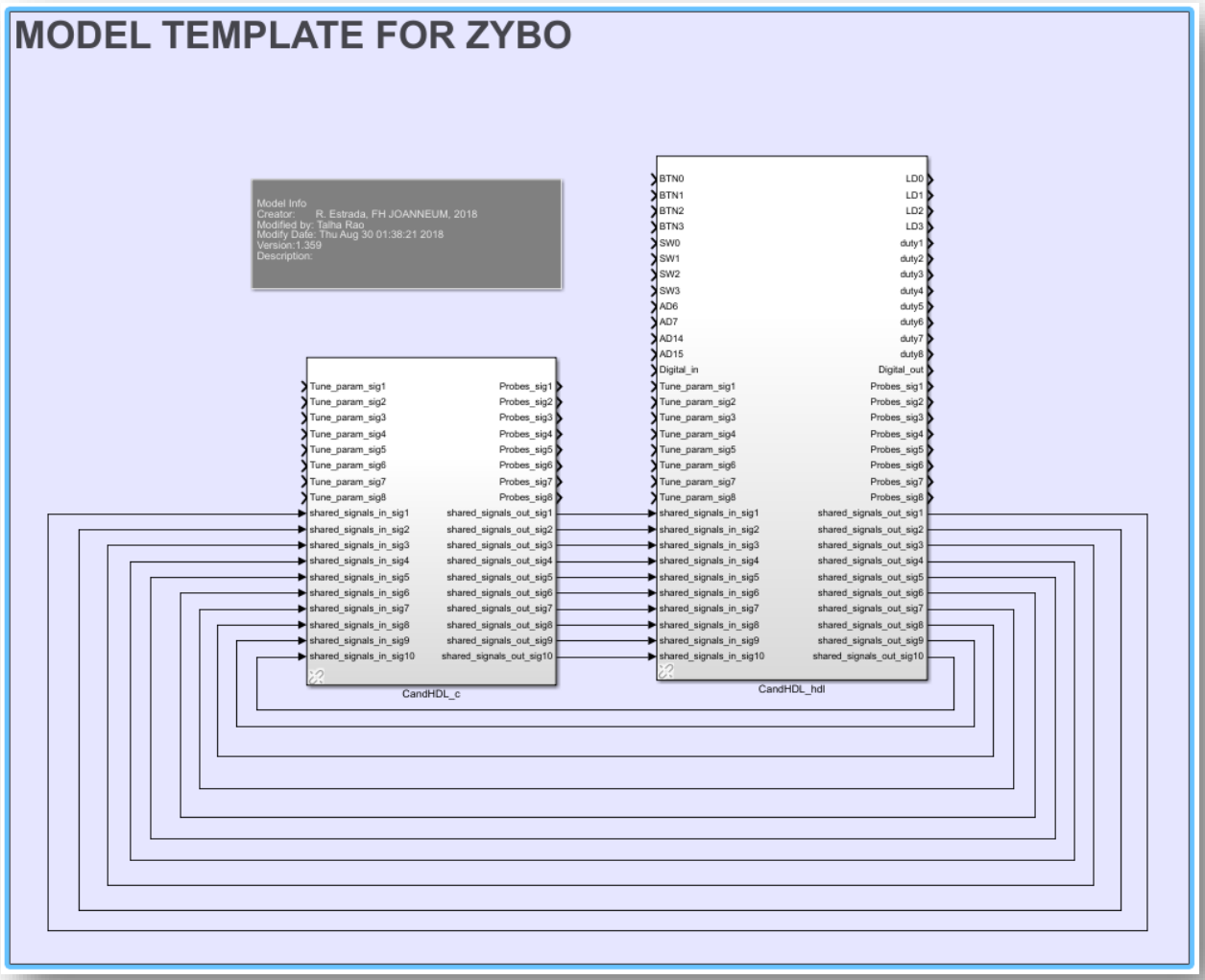

Fig. 5-1: ZYBO Board Template Model

The template model offers the connection to I/O ports, buttons, switches, LEDs and PWMs on the **ZYBO** board. The tunable parameters were also available to connect to the algorithm during runtime via UART interface. PuTTy or Hyper Terminal could be used to connect to the board and send values on runtime.

5.2 *Integration of Static Calculations Model*

After performing all the configuration steps, provided in the lab task, the Static Calculations Block was integrated in the template model's **CandHDL_c**. The required block was placed inside **cgenerate** subsystem and the inputs were connected to the tunable parameter to be able to give values on the run time. **Fig. 5-2** shows the cgenerate subsystem.

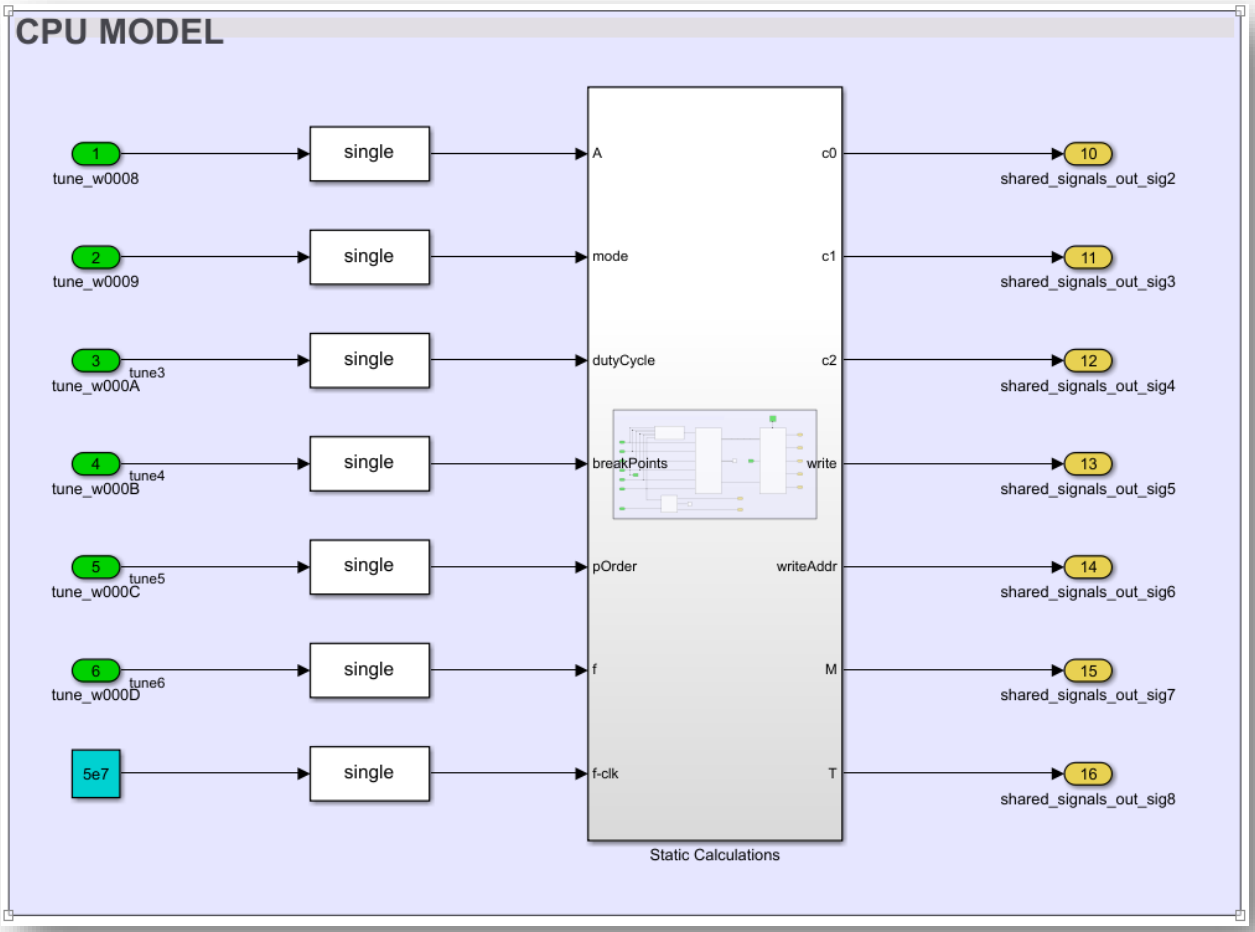

Fig. 5-2: cgenerate Sub-System

The outputs were given to the shared signals to be connected to FPGA later.

5.3 *Integration of Dynamic Calculations Model*

First, to presurve the values of the signal, the output was scaled and casted into uint16 data type, as shown in **Fig. 5-3**, which is required by the PWM output pin on Zybo Board.

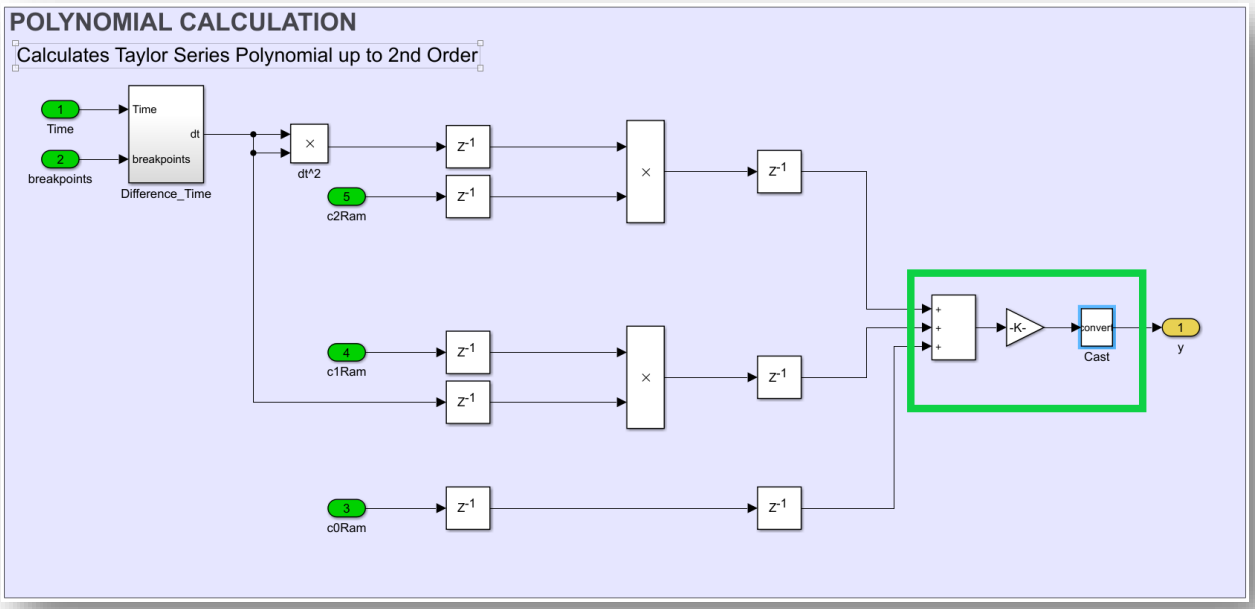

Fig. 5-3: Casting and Scaling of Output Signal

After the modifications, the FPGA block was connected to the shared in signals which were connected to the CPU block in the previous section. The output of the system was connected to the duty 1 probe which is configured with the JB1 pin on ZYBO Board. Fig. 5-4 shows the connections.

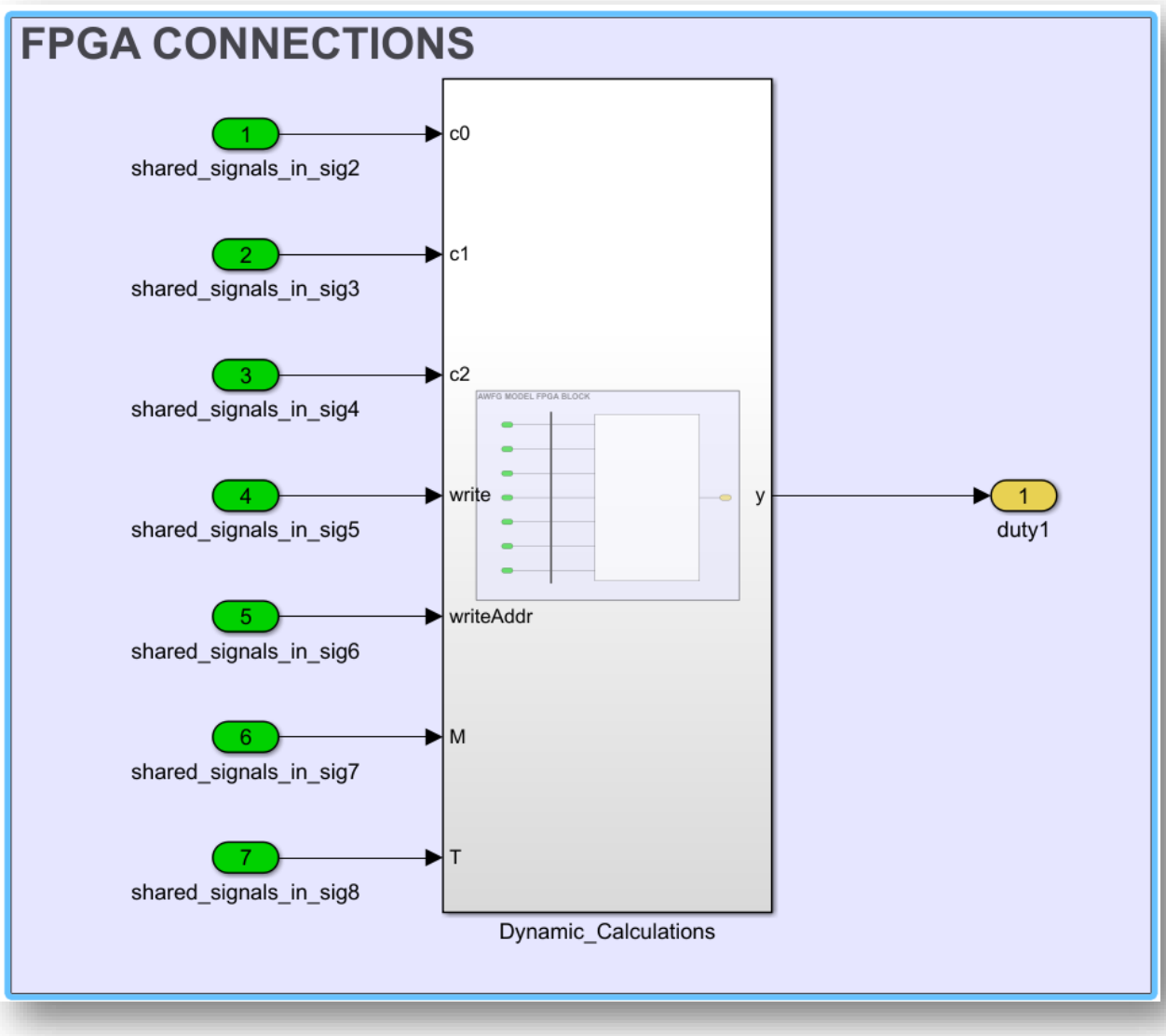

Fig. 5-4: FPGA Block Connections

5.4 *Preparation of First Order Filter*

A simple first order was designed on the bread board to reconstruct the original signal. A **150 Ω** resistance and **3.3 µF** capacitor was used to produce a cutoff frequency of **321 Hz**. **Fig. 5-5** shows the implementation of filter on the board.

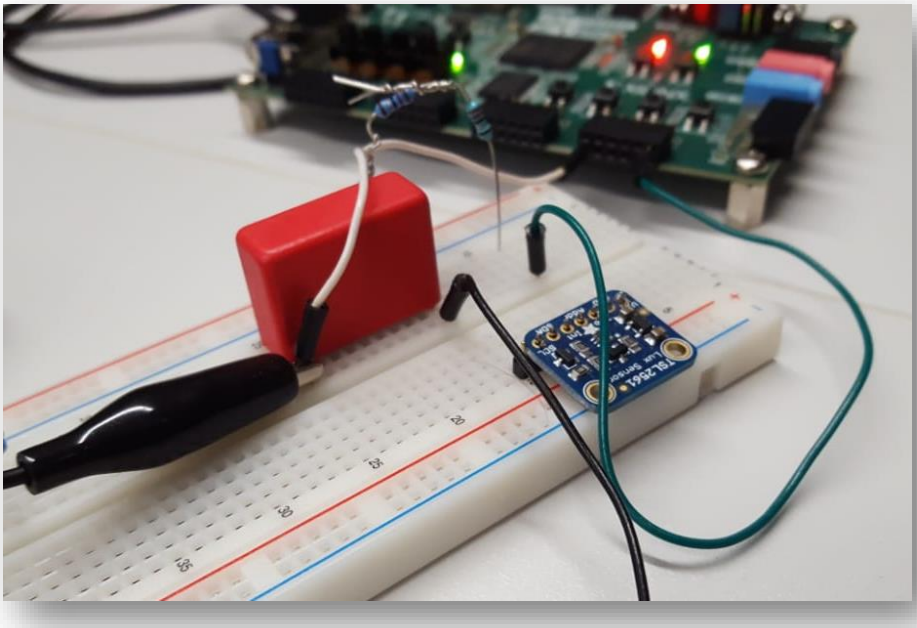

Fig. 5-5: Implementation of 1st Order Filter

5.5 *Generation and Deployment of code for ZYBO*

The ARM Core and the FPGA were programed as per the instructions provided in the manual. After deployment, the input parameters were written to tunable ports via serial connection using **PuTTy**.

5.6 *Testing and Demonstration of Results*

After the input parameters were written to the board, the output via a filter order filter was observed on oscilloscope. The figure below shows different waveform with duty cycle **50** ang **70%.**

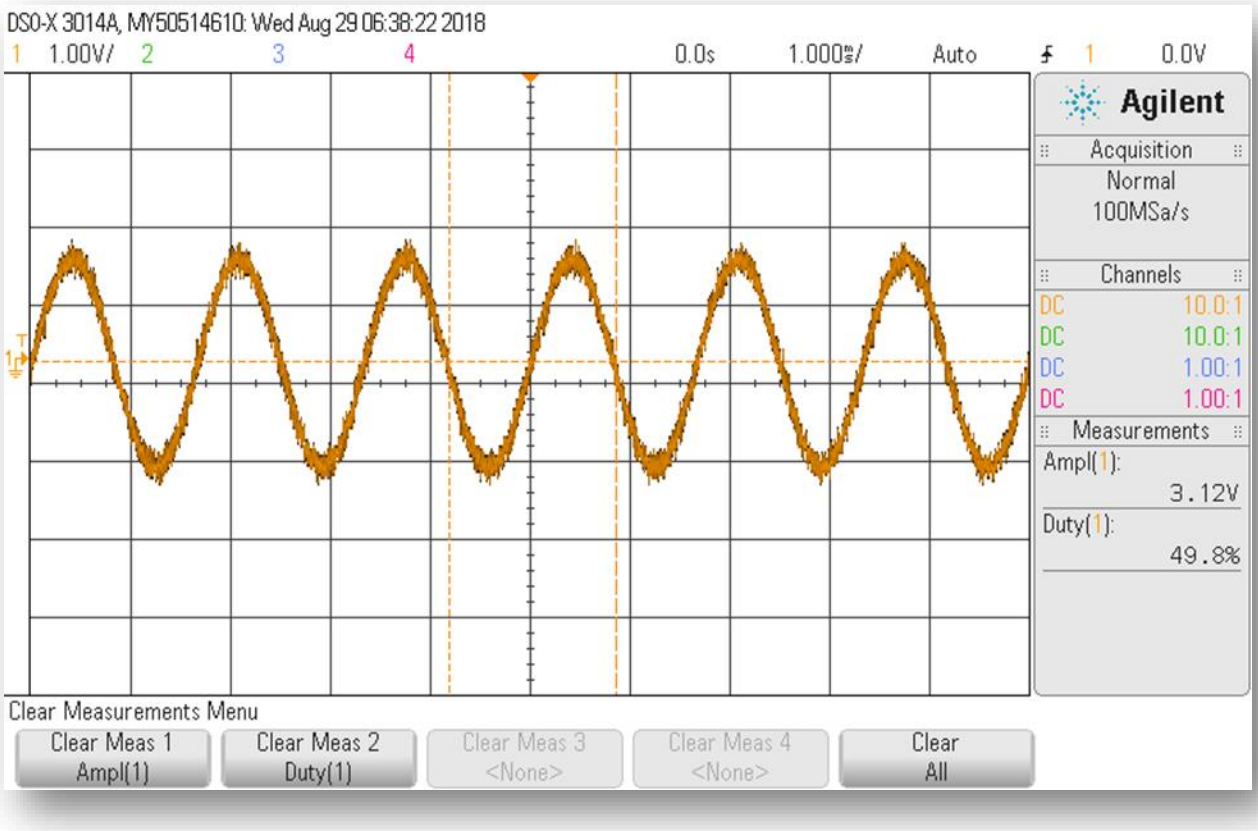

Fig. 5-6: Sine Wave Generation

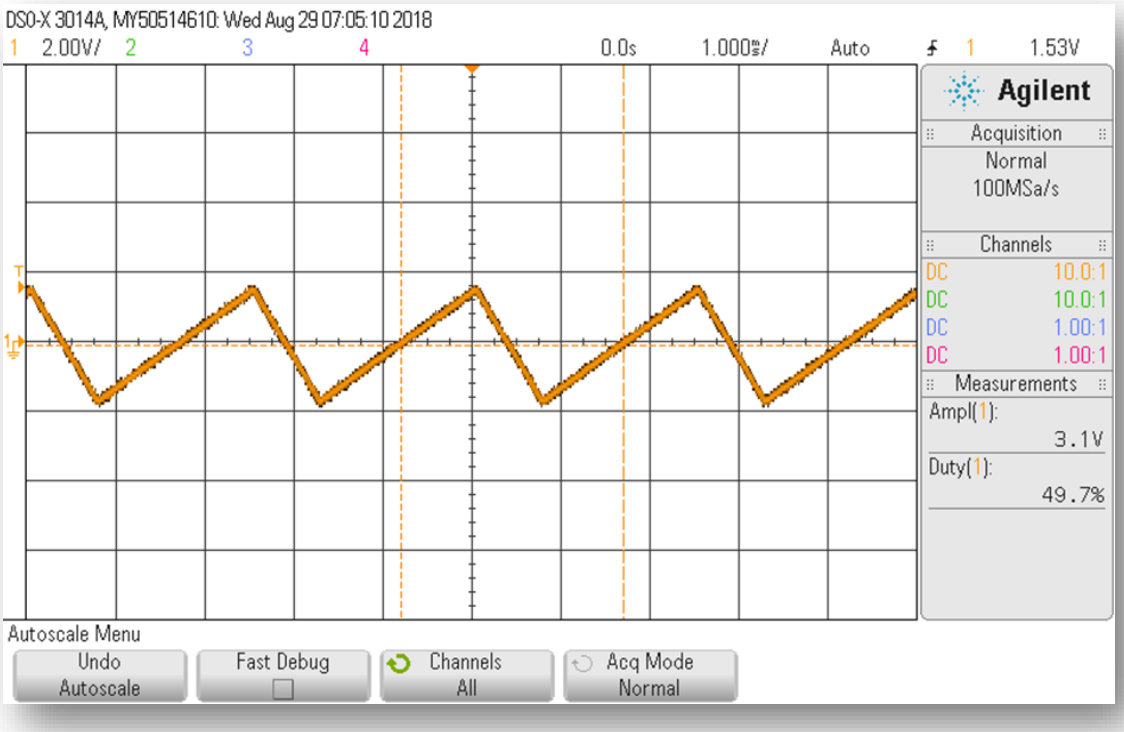

Fig. 5-7: Triangular Wave with 70% duty Cycle

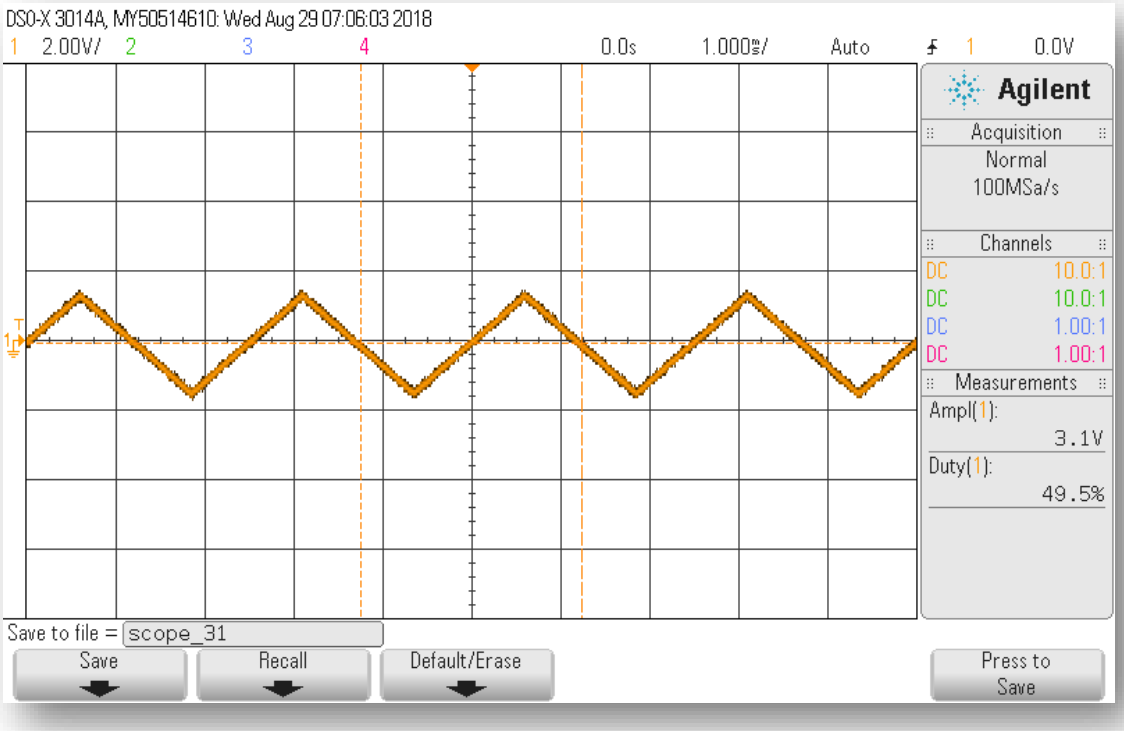

Fig. 5-8: Triangle Wave with 50% Duty Cycle

6. Conclusion

After following the step by step procedure, the AWFG model was finally executed on ZYBO Board and the results were as per the requirements.