DCT AND IDCT HARDWARE ACCELERATOR

A THESIS

Submitted by

ASWINKUMAR

for the award of the degree

of

BACHELOR OF TECHNOLOGY

IN

ENGINEERING PHYSICS

DEPARTMENT OF PHYSICS

INDIAN INSTITUTE OF TECHNOLOGY MADRAS

CHENNAI-600036

MAY 2022

THESIS CERTIFICATE

This is to certify that the thesis entitled **“DCT AND IDCT HARDWARE ACCELERATOR”** submitted by **ASWINKUMAR** to the Indian Institute of Technology, Madras for the award of the degree of **BACHELOR OF TECHNOLOGY IN ENGINEERING PHYSICS** is a bona fide record of research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Prof. V Kamakoti
Professor
Department of Computer Science and Engineering
Indian Institute of Technology Madras
Chennai – 600 036.

Place: Chennai

Date: May 2022

ACKNOWLEDGEMENTS

I would like to thank Prof. V Kamakoti for taking me on as a project student and for his time despite his busy schedule, throughout the project I have learnt a lot of new and interesting concepts about this field, I have also explored research areas around the project that I believe would be very helpful for my career.

I thank Sharan , for making time for me and patiently understanding and explaining every question that I put forward despite his busy schedule as a PhD student. His support, advice and explanation of even the most inane questions played a major role in my work.

I owe much to my family and friends for providing me with a support system and making my stay on campus a joyful one.

ABSTRACT

Keywords: Discrete Cosine Transform; Inverse discrete cosine transform; Hardware accelerator.

The core of the present work is to implement a hardware accelerator for Discrete cosine transform and inverse cosine transform for signal processing and media applications , with the growth of Information technology, the amount of information being shared between devices has grown enormously, the need for high resolution information has imparted pressure onto compression algorithms and tools to share information with the available communication infrastructure while the need for high resolution information is not going to stop , the algorithms play a major role in efficiently encoding and decoding information with minimal losses. DCT is a transform algorithm that has been widely used for several decades for it’s incredible energy compaction , this work looks onto implementing hardware level DCT and IDCT transform for energy efficient encoding and decoding.

TABLE OF CONTENTS

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

1.1. INTRODUCTION

The DCT is the most widely used transformation technique in signal processing, and by far the most widely used linear transform in data compression. Uncompressed digital media as well as lossless compression had impractically high memory and bandwidth requirements, which was significantly reduced by the highly efficient DCT lossy compression technique.

A discrete cosine transform (DCT) expresses a finite sequence of data points in terms of a sum of cosine functions oscillating at different frequencies. The DCT was first proposed by Nasir Ahmed in 1972 and is a widely used transformation technique in signal processing and data compression.

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \qquad for\ k = 0, \ldots N - 1$$

There are many DCT variants, of which four are common. The most common variant of discrete cosine transform is the type-II DCT which is shown above, which is often called simply "the DCT". This was the original DCT as first proposed by Ahmed. Its inverse, the type-III DCT, is correspondingly often called simply "the inverse DCT" or "the IDCT". The DCT, and in particular the DCT-II, is often used in signal and image processing, especially for lossy compression, because it has a **strong "energy compaction"** property, in typical applications, most of the signal information tends to be concentrated in a few low-frequency components of the DCT.

DCT compression standards are used in digital media technologies, such as digital images (such as JPEG and HEIF, where small high-frequency components can be discarded - Refer Figure 1 for Visual representation of intensity of frequency components), digital video (such as MPEG ,H.26x ,VP9 & AV1), digital audio (such as Dolby Digital, MP3 and AAC), digital television (such as SDTV, HDTV and VOD), digital radio (such as AAC+ and DAB+), and speech coding (such as AAC-LD, Siren and Opus).

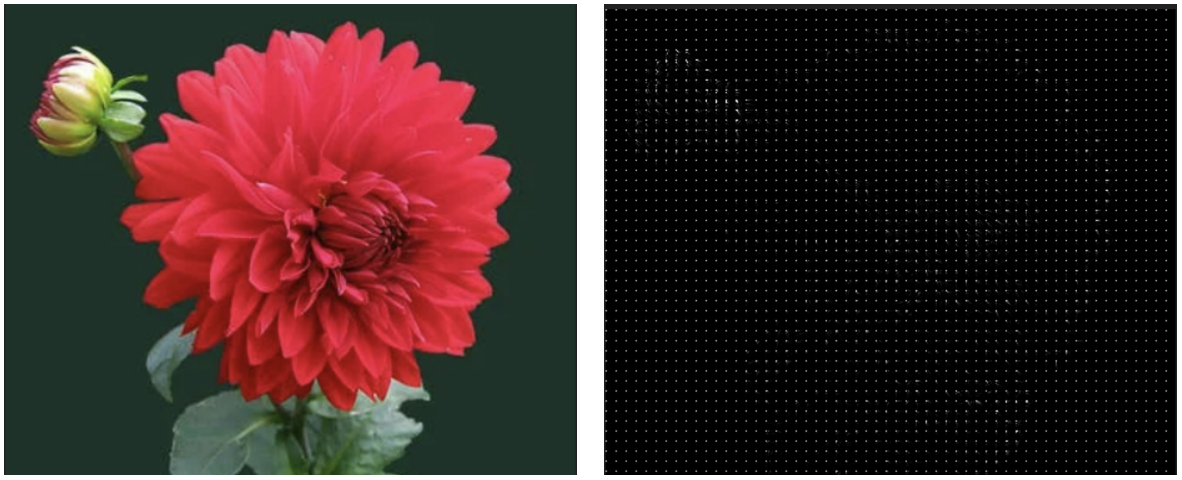


Fig. 1. Visual representation of the intensity frequency components for 8x8 DCT

1.2. OBJECTIVE OF THE WORK

The objective is to understand the various signal processing applications that utilize DCT and the implement a Fast DCT and IDCT algorithm on the hardware, though this is an approximation , it has been standardized over the years for use in practical systems., the goal is to implement a **synthesizable Integer DCT and IDCT** in **Bluespec** which would act as an accelerator for Signal processing and media applications.

1.3. SCOPE OF THE THESIS

This thesis is an attempt to present the understandings developed from reading various literature on Fast DCT and implementing the same in Bluespec.

The thesis organization is as follows…

- Introduction to DCT and it's uses
- Derivation of Fast DCT
- Implementation of DCT
- Implementation of IDCT

CHAPTER 2

HARDWARE IMPLEMENTATION

2.1. INTRODUCTION

The DCT is the most important image compression technique. It is used in image compression standards such as JPEG, and video compression standards such as H.26x, MJPEG, MPEG, DV, Theora and Daala. The DCT computation in a typical encoder takes about 21% of the time on average per block. IDCT is also the single most time consuming operation in the video decoding pipeline. Software based decoders and encoders can be accelerated for this transform making them energy efficient and suitable for mobile and edge applications with minimal changes to their code.

$$X_k = \sum_{n=0}^{N-1} x_n \cos\left[\frac{\pi}{N}\left(n + \frac{1}{2}\right)k\right] \qquad for\ k = 0, \ldots N - 1$$

The two-dimensional DCT-II of N × N blocks can be computed as a row-wise N 1D transform and then the columns respectively; in a video decoding pipeline these values are quantized and entropy coded. The result is an N × N transform coefficient array in which the (0 , 0) element (top-left) is the DC (zero-frequency) component and entries with increasing vertical and horizontal index values represent higher vertical and horizontal spatial frequencies.

As the frequency increases we find that the coefficients tend to get close to 0 saving us in size of storing the block , this is traditionally done with floating point numbers , but Advanced Video Coding (AVC) , VP9 , AV1 uses the integer DCT (IntDCT), an integer approximation of the DCT touse varied integer DCT block sizes between 4x4 and 128x128 pixels.

In the following section we will take the type-II DCT , the popular variant and derive the Integer DCT / Fast DCT which we will then implement in the following sections.

2.2. DERIVATION OF FAST DCT

The Fast Computational algorithm for DCT was introduced in 1977, 5 years after the introduction of DCT, the algorithm can be derived as follows from the DCT algorithm.

$$F(k) = \frac{2c(k)}{N} \sum_{j=0}^{N-1} f(j) \cos\left[\frac{\pi}{N}\left(j + \frac{1}{2}\right)k\right] \qquad for\ k = 0, \ldots N-1$$

$$c(k) = \frac{1}{\sqrt{2}} \quad for\ k = 0$$

$$= 1 \qquad for\ k = 1, 2, \ldots N-1$$

This can be simplified as follows

$$[F] = \frac{2}{N}[A_N][f]$$

where

$$[A_N] = c(k) \cos\left[\frac{\pi}{N}\left(j + \frac{1}{2}\right)k\right] \qquad for\ j, k = 0, \ldots N-1$$

The algorithm is based on the matrix decomposition of $[A_N]$, the matrix can be written into the following recursive form:

$$[A_N] = [P_N] \begin{bmatrix} A_{N/2} & 0 \\ 0 & R_{N/2} \end{bmatrix} [B_N]$$

where $[P_N]$ is an N X N permutation matrix that permutes the transformed vector from a bit reversed order to natural order and $[R_{N/2}]$ can be given as follows :

$$[R_{N/2}] = \left[c(k) \cos\left[\frac{\pi}{N}\left(j + \frac{1}{2}\right)(2k+1)\right]\right] \qquad for\ j, k = 0, \ldots N-1$$

$$B_N = \begin{bmatrix} I_{N/2} & \bar{I}_{N/2} \\ \bar{I}_{N/2} & -I_{N/2} \end{bmatrix}$$

$$B_N* = \begin{bmatrix} -I_{N/2} & \bar{I}_{N/2} \\ \bar{I}_{N/2} & I_{N/2} \end{bmatrix}$$

$$[I_{N/2}]\ is\ the\ identity\ matrix\ of\ order\ \frac{N}{2}$$

We can also see that $[A_N]$ has a recursive nature and from the below equation , $[A_2]$ can be further extended into higher order if **we can decompose** $[R_{N/2}]$

$$[A_2] = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$$

Decomposition of $[R_{N/2}]$ can be done as follows

$$[R_{N/2}] = [M1][M2][M3]..... [M(2log_2(N-3)]$$

where

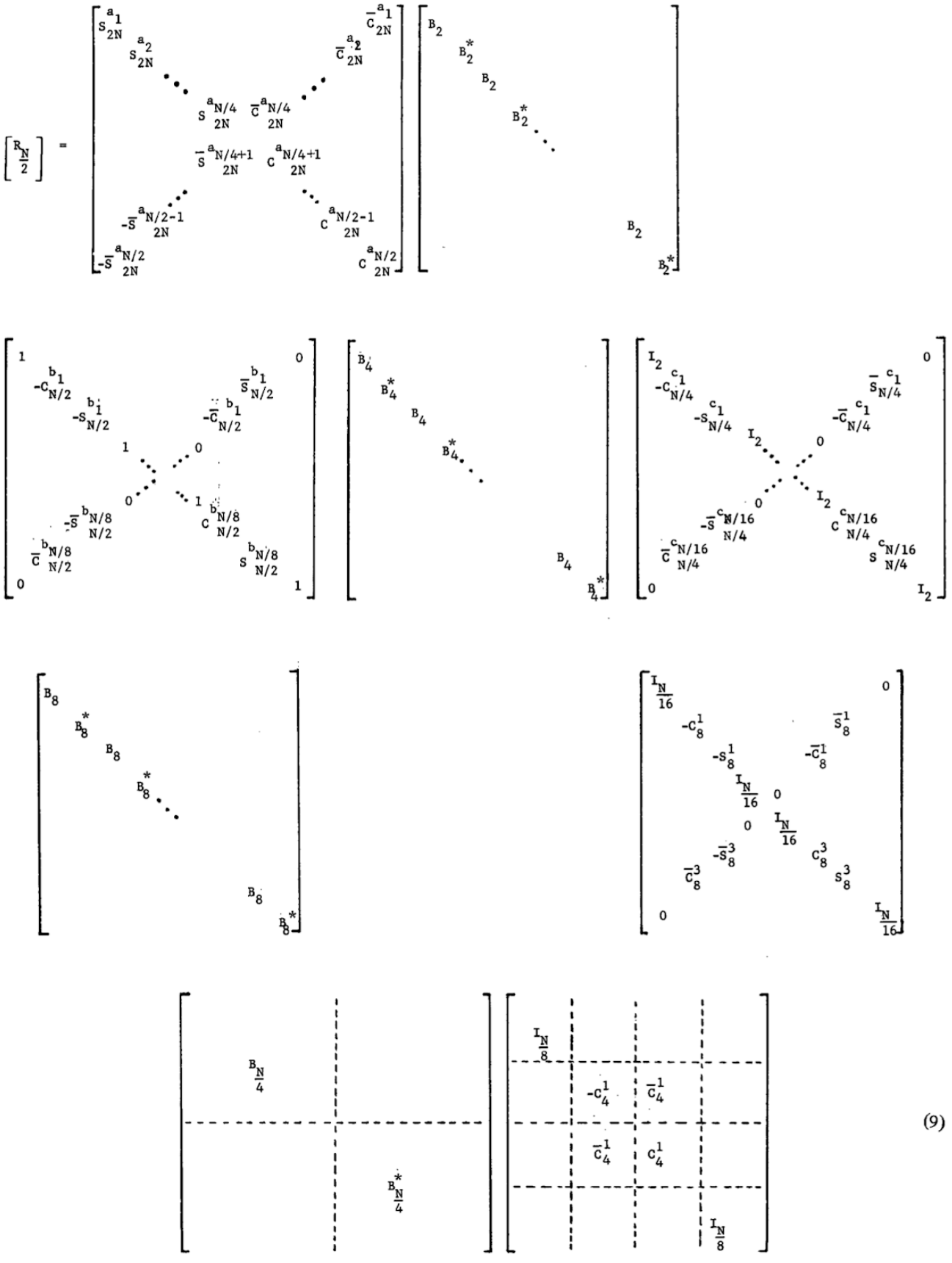
$[M1]$ is the first matrix

$[M(2log_2(N-3)]$ is the last matrix

$[M_q]$ are the remaining odd numbered matrices

$[M_p]$ are the remaining even numbered matrices

by substituting the values for the matrices we get the following :



(9)

Given that,

$$[S_i^k] = sin\frac{k\pi}{i}[I_{N/2i}]$$

$$[\bar{S}_i^k] = sin\frac{k\pi}{i}[\bar{I}_{N/2i}]$$

$$[C_i^k] = cos\frac{k\pi}{i}[I_{N/2i}]$$

$$[\bar{C}_i^k] = cos\frac{k\pi}{i}[\bar{I}_{N/2i}]$$

This allows us to build a flow-graph for our Fast DCT in a recursive manner, which can be visualized below.

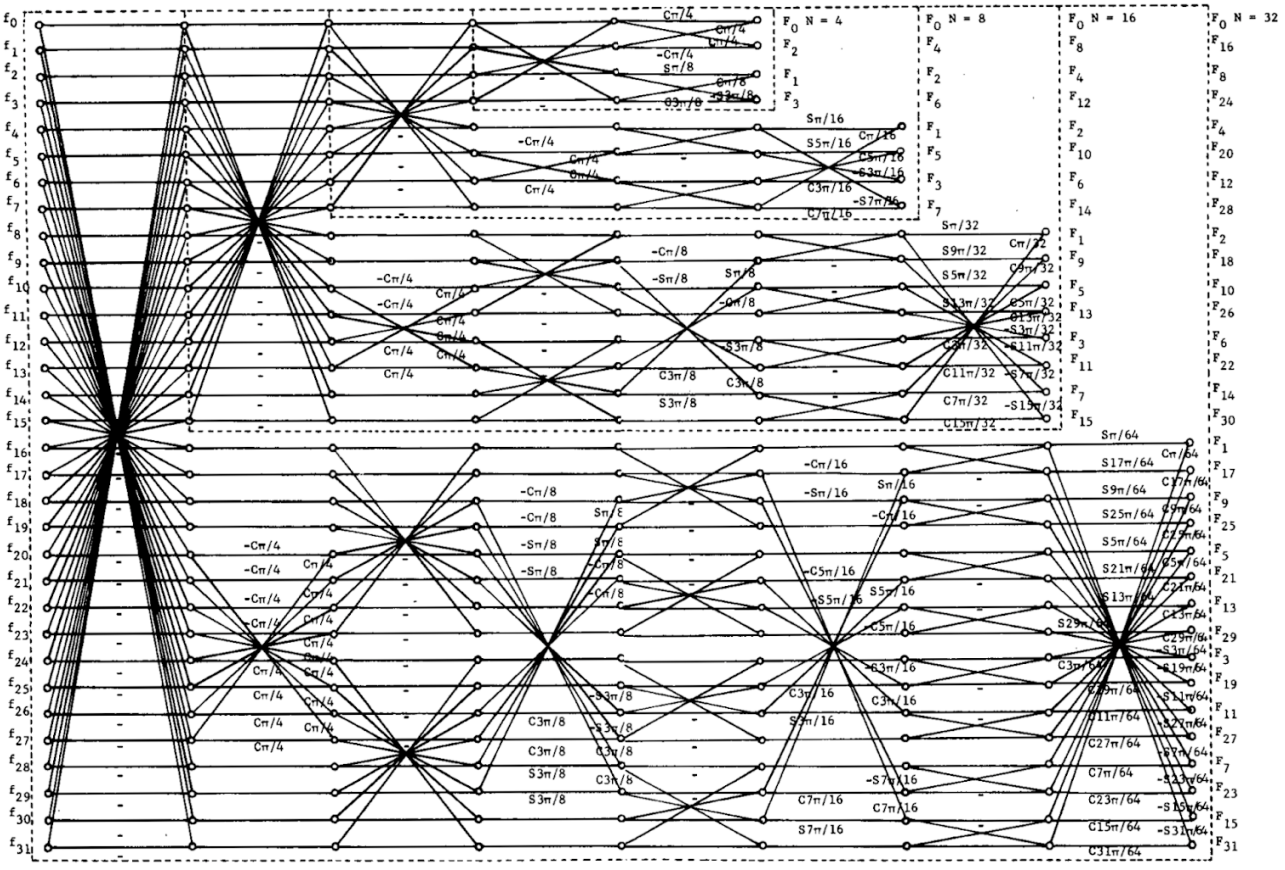


Fig 2.1 Flow Graph for DCT for N=4,8,16,32

2.3. DISCRETE COSINE TRANSFORM

The above flow-graph can now be implemented in **Bluespec**.

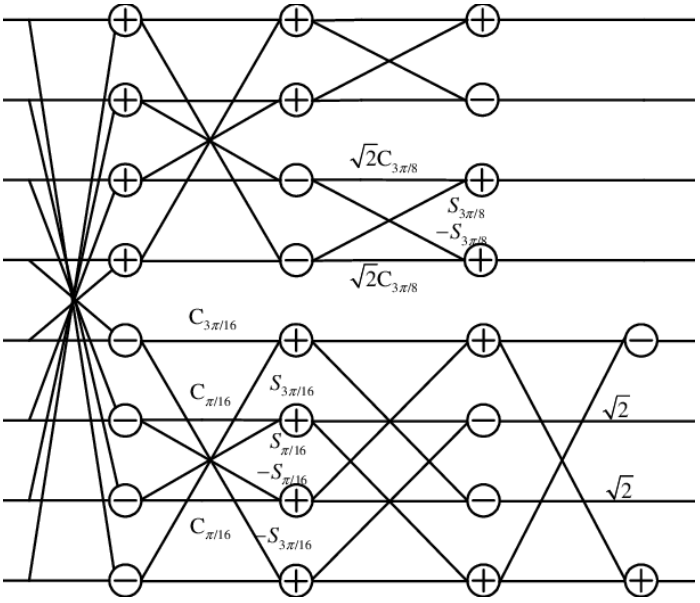


Fig 2.2 Flow graph of 8-Point DCT

The following DCT 8-point algorithm can be split into 4 stages which can then be pipelined for higher throughput , the algorithm is currently implemented as a **Finite-state machine** as rules execute every clock cycle, this FSM approach makes sure we get the expected output.

STAGE 1 :

The first stage performs the **Hadamard transform** by giving by the sums and difference of the respective elements , the **x1 vector** contains output from the first stage and takes input from the **x vector** to perform the operations.

```
// Butterfly Stage 1
        rule fly1 (stage == 1);
                x1[0] <= x[0] + x[7];
                x1[7] <= x[0] - x[7];
                x1[1] <= x[1] + x[6];
                x1[6] <= x[1] - x[6];
                x1[2] <= x[2] + x[5];
                x1[5] <= x[2] - x[5];
                x1[3] <= x[3] + x[4];
                x1[4] <= x[3] - x[4];
                stage <= 2;
        endrule: fly1
```

STAGE 2 :

In this stage both **Hadamard transform** and **Multiply-and-accumulate** operation are performed on the output from STAGE 1, this output from STAGE 2 is stored into a new vector **x2** for processing in STAGE 3.

```
// Butterfly Stage 2
	rule fly2 (stage == 2);
		x2[0] <= x1[0] + x1[3];
		x2[1] <= x1[1] + x1[2];
		x2[2] <= x1[1] - x1[2];
		x2[3] <= x1[0] - x1[3];
		x2[4] <= x1[4];
		x2[5] <= (c4*(x1[6]-x1[5]));
		x2[6] <= (c4*(x1[5]+x1[6]));
		x2[7] <= x1[7];
		stage <= 3;
	endrule: fly2
```

STAGE 3 :

The third stage is similar to STAGE 2 but we can notice << shifting operation, this is used as the sin and cosine values we use in the multiply-and-accumulate operations shifted only **x2[5]** and **x2[6]** whereas other elements were not scaled, to make sure all outputs from STAGE 3 are correctly scaled , we manually left shift the operations.

```
// Butterfly Stage 3
	rule fly3 (stage == 3);
		x3[0] <= (c4*(x2[0]+x2[1]));
		x3[1] <= (c4*(x2[0]-x2[1]));
		x3[2] <= (s2*x2[2] + c2*x2[3]);
		x3[3] <= (-c2*x2[2] + s2*x2[3]);
		x3[4] <= (x2[4] << 8) + x2[5];
		x3[5] <= (x2[4] << 8) - x2[5];
		x3[6] <= -x2[6] + (x2[7] << 8);
		x3[7] <= x2[6] + (x2[7] << 8);
		stage <= 4;
	endrule: fly3
```

STAGE 4 :

The STAGE 4 also computes the transformed output and enables the FSM to send the output back to the module which invoked the DCT module.

```
// Butterfly Stage 4
        rule fly4 (stage == 4);
                f[0] <= x3[0];
                f[1] <= (s1*x3[4]) + (c1*x3[7]);
                f[2] <= x3[2];
                f[3] <= -(s3*x3[5]) + (c3*x3[6]);
                f[4] <= x3[1];
                f[5] <= (c3*x3[5]) + (s3*x3[6]);
                f[6] <= x3[3];
                f[7] <= -(c1*x3[4]) + (s1*x3[7]);
                stage <= 5;
        endrule: fly4
```

COMMUNICATION :

The DCT module can be used using the DCTint_ifc interface.

```
interface DCTint_ifc;
        method Action fetch_Input (Vector#(8, int) signal);
        method Vector#(8, Reg#(int)) result();
endinterface
```

The **complete package and testbench** for the DCT module is available in **Appendix i**

2.4. INVERSE DISCRETE COSINE TRANSFORM

The Inverse discrete cosine transform performs the **Inverse** of Discrete cosine transform and hence the algorithm can be understood from the derivation of Discrete cosine transform as discussed in Section 2.2.

The flow graph of 32-POINT IDCT is as follows , the IDCT graph is modular , we can get output for any $n \leq N$; $N = 2^k$.

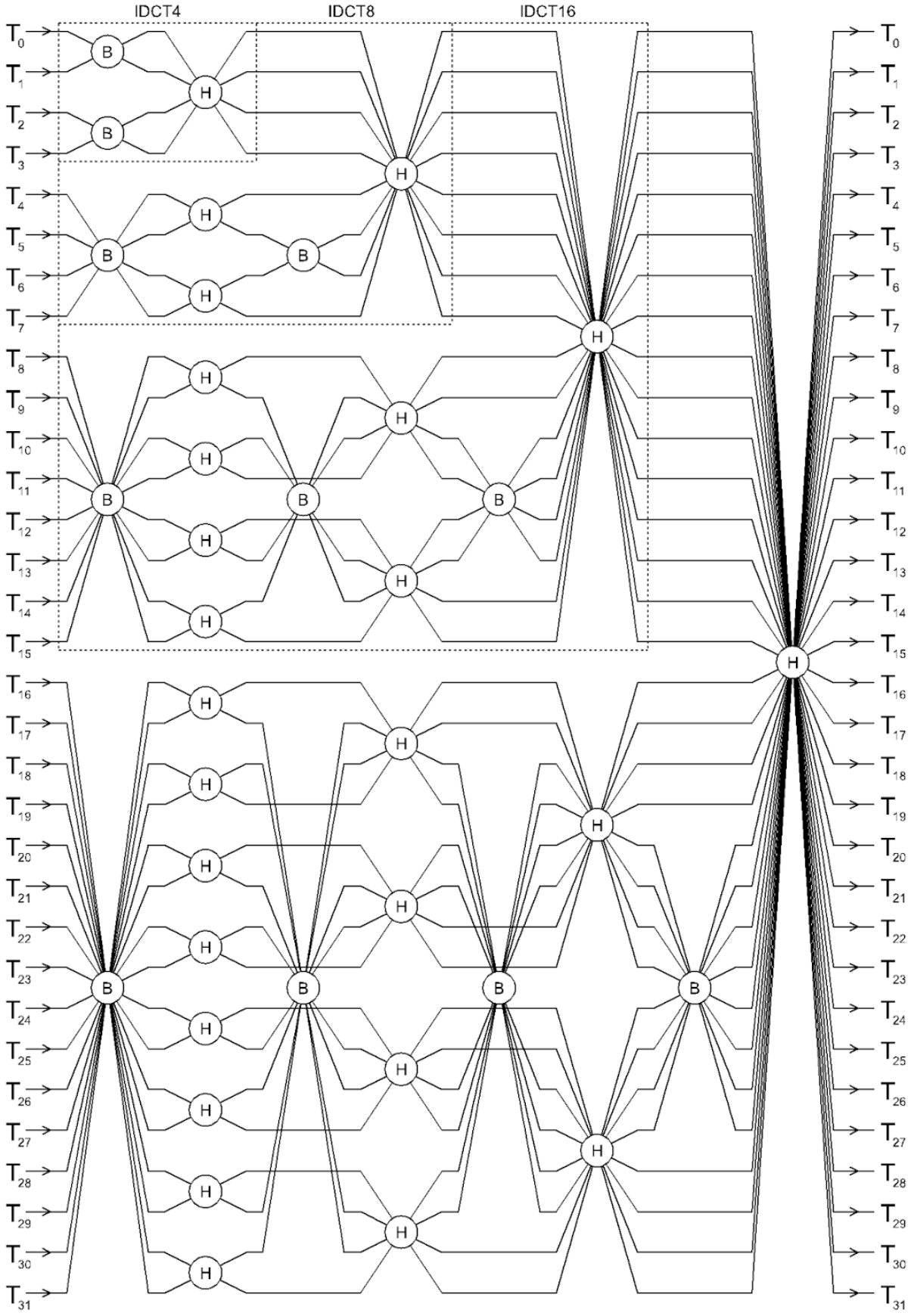
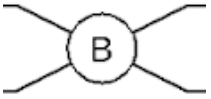


Fig 2.3 Flow graph of IDCT

BUTTERFLY B :

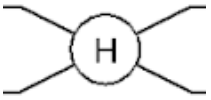


The Butterfly B performs Multiply-and-accumulate operation of the inputs with scaled cosine and sin values. These are scaled with 2^{14} and then the *round2* function is called to normalize the output after multiplication.

```
interface Ifc_butterfly;
	method Action inps(int ta, int tb, int angle, Bool flag);
	method int ta;
	method int tb;
endinterface

module mkButterfly (Ifc_butterfly);
	Reg#(int) tanew <- mkReg(0), tbnew <- mkReg(0);
	Reg#(Bool) valid_outs <- mkReg(False);

	method Action inps(int ta, int tb, int angle, Bool flag);
		int x = ta*cos64(angle) - tb*sin64(angle);
		int y = ta*sin64(angle) + tb*cos64(angle);

		if (flag) begin //flag = 1
			tanew <= round2(y, 14);
			tbnew <= round2(x, 14);
		end
		else begin
			tanew <= round2(x, 14);
			tbnew <= round2(y, 14);
		end

		valid_outs <= True;
	endmethod

	method int ta if (valid_outs);
		return tanew;
	endmethod

	method int tb if (valid_outs);
		return tbnew;
	endmethod
endmodule
```

BUTTERFLY H :

The Butterfly H performs Walsh-Hadamard transform on the inputs. This is again invoked with the Ifc_Hadamard interface for getting outputs and providing outputs from the module.




```
interface Ifc_Hadamard;
        method Action inps(int ta, int tb, Bool flag);
        method int ta;
        method int tb;
endinterface

module mkHadamard (Ifc_Hadamard);
        Reg#(int) tanew <- mkConfigReg(0), tbnew <- mkConfigReg(0);
        Reg#(Bool) valid_outs <- mkReg(False);

        method Action inps(int ta, int tb, Bool flag);
                if (flag) begin             //flag = 1
                        tanew <= tb - ta;
                        tbnew <= ta + tb;
                end
                else begin
                        tanew <= ta + tb;
                        tbnew <= ta - tb;
                end

                valid_outs <= True;
        endmethod

        method int ta if (valid_outs);
                return tanew;
        endmethod

        method int tb if (valid_outs);
                return tbnew;
        endmethod
endmodule
```

APPROXIMATE COSINE AND SINE VALUES :

The cos64 function implements the expression *round(16384 * cos(angle * pi / 64))*. The sin64 function implements the expression *round(16384 * sin(angle * pi / 64))*.

The function sin64(angle) is defined to be cos64(angle - 32).

```
function int cos64_lookup (int inp);
        case(inp) matches
                0: return 16384;
                1: return 16364;
                2: return 16305;
                3: return 16207;
                4: return 16069;
                5: return 15893;
                6: return 15679;
                7: return 15426;
                8: return 15137;
                9: return 14811;
                10: return 14449;
                11: return 14053;
                12: return 13623;
                13: return 13160;
                14: return 12665;
                15: return 12140;
                16: return 11585;
                17: return 11003;
                18: return 10394;
                19: return 9760;
                20: return 9102;
                21: return 8423;
                22: return 7723;
                23: return 7005;
                24: return 6270;
                25: return 5520;
                26: return 4756;
                27: return 3981;
                28: return 3196;
                29: return 2404;
                30: return 1606;
                31: return 804;
                32: return 0;
        endcase
endfunction
```

```
function int cos64(int inp);
        int angle2 = unpack(pack(inp) & 32'd127);

        if (( angle2 > 0) && (angle2 <= 32 ))
      return cos64_lookup(angle2);
   else if (( angle2 > 32) && (angle2 <= 64 ))
      return -1*cos64_lookup(64-angle2) ;
   else if (( angle2 > 64) && (angle2 <= 96 ))
      return -1*cos64_lookup(angle2-64);
   else
      return cos64_lookup(128-angle2);

endfunction


function int sin64(int inp);
        return cos64(inp - 32);
endfunction
```

MANIPULATION FUNCTIONS:

The function *round2* would right shift the input by 14 bits. the *brev* performs bit reversal operation on the input as discussed in [P_N] in Section 2.2.

```
function int round2(int x, Integer n);
        return (x+(1<<(n-1))) >> n;
endfunction

function int brev (Integer bits, int inp_n);
        int result = 0;
        int n = inp_n;
        for (Integer i=0; i<bits; i=i+1) begin
                result = result << 1;
                result = result | n & 1;
                n = n >> 1;
        end

        return result;
endfunction
```

IDCT :

4-POINT IDCT :

The 4-POINT IDCT uses 2 Butterfly B modules and 2 Butterfly H modules, this has been implemented in Bluespec as STAGE 1 and STAGE 7 respectively. As with the DCT , we can hardcode the modules but due to the complexity of Butterfly B , it is a good design choice to use it via the interface.

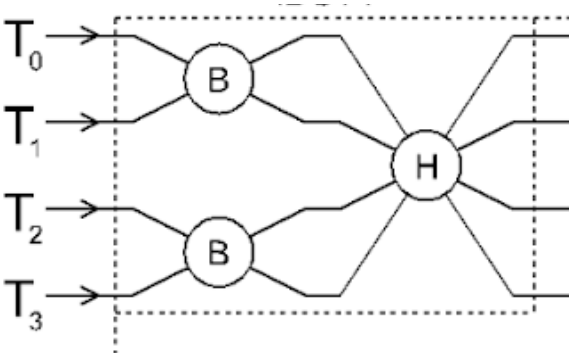


Fig. 2.4 Flow graph of 4-Point IDCT

```
module mkIDCT_4 (Ifc_IDCT#(4, 2));
        Vector#(4, Reg#(int)) t_array <- replicateM(mkConfigReg(0));
        Reg#(IDCT_4_stages) mod_stage <- mkReg(State0);
        Vector#(2, Ifc_butterfly) modB <- replicateM(mkButterfly);
        Vector#(2, Ifc_Hadamard) modH <- replicateM(mkHadamard);

        PulseWire disable_start <- mkPulseWire, updating <- mkPulseWire;

        rule rl_stg1 (mod_stage == State1);
                modB[0].inps(t_array[0], t_array[1], 16, True);
                modB[1].inps(t_array[2], t_array[3], 32 - brev(5, 2), False);
                mod_stage <= State2;
        endrule
        rule rl_stg2 (mod_stage == State2);
                t_array[0] <= modB[0].ta;
                t_array[1] <= modB[0].tb;

                t_array[2] <= modB[1].ta;
                t_array[3] <= modB[1].tb;

                mod_stage <= State3;
        endrule
        rule rl_stg7 (mod_stage == State3);
                for (int i=0; i<2; i=i+1) begin
                        modH[i].inps(t_array[i], t_array[4-1-i], False);
                end
                mod_stage <= State4;
        endrule
        rule rl_store7 (mod_stage == State4);
                for (int i=0; i<2; i=i+1) begin
                        t_array[i] <= modH[i].ta;
                        t_array[4-1-i] <= modH[i].tb;
                end

                mod_stage <= State0;
        endrule
```

```
        method Action start if((!disable_start) && (mod_stage == State0));
                mod_stage <= State1;
        endmethod

        method Vector#(4, int) t_read if((!disable_start) && (!updating) && (mod_stage
== State0));
                Vector#(4, int) temp = defaultValue;
                for (int i=0; i<4; i=i+1)
                        temp[i] = t_array[i];
                return temp;
        endmethod

        method Action t_update (Vector#(4, int) inp_vec) if((!disable_start) &&
(mod_stage == State0));
                for(int i=0; i<4; i=i+1)
                        t_array[i] <= inp_vec[i];
                updating.send();
        endmethod
endmodule
```

The **8-POINT , 16-POINT , 32-POINT IDCT** package has been attached in **Appendix ii**.

CHAPTER 3
SUMMARY AND CONCLUSION

The DCT modules and IDCT modules have been implemented in Bluespec and their results have been verified both via publicly available calculators and Python simulation of the hardware circuit.

The project will be expanded to 128-point IDCT and pipelined to fully accommodate and be usable in all the state-of-the-art Video decoders.

REFERENCES

Adrian Grange, Peter de Rivaz and Jonathan Hunt (2016) VP9 Bitstream & Decoding Process Specification.

Wen-Hsiungchen,C. Harrison Smith and S. C. Fralick (1977) A Fast Computational Algorithm forthe Discrete Cosine Transform

Martišius, Ignas & Birvinskas, D. & Jusas, Vacius & Tamosevicius, Z.. (2011). A 2-D DCT Hardware Codec based on Loeffler Algorithm. Electronics And Electrical Engineering.

Sriram Sethuraman, Cherma Rajan A., Kaustubh Patankar (2017) Analysis Of The Emerging Aomedia Av1 Video Coding Format For Ott Use-Cases

M. Corrêa et al., "AV1 and VVC Video Codecs: Overview on Complexity Reduction and Hardware Design," in IEEE Open Journal of Circuits and Systems, vol. 2, pp. 564-576, 2021, doi: 10.1109/OJCAS.2021.3107254.

Chang, Qingyun & Zhang, Youhui & Xie, Yuejian & Liu, Dong & Wang, Dongsheng. (2006). A design of H. 264 decoder with integrated hardware and software.

Tung-Chien Chen, Chung-Jr Lian and Liang-Gee Chen, "Hardware architecture design of an H.264/AVC video codec," Asia and South Pacific Conference on Design Automation, 2006., 2006, pp. 8 pp.-, doi: 10.1109/ASPDAC.2006.1594776.

Shriram, Kv & Subashri, V & Periyasamy, Sasikumar. (2010). H.264 Decoder Design using Verilog (With Simulation Results). International Journal of Computer Applications. 1. 10.5120/92-192.

Chang, Qingyun & Zhang, Youhui & Xie, Yuejian & Liu, Dong & Wang, Dongsheng. (2006). A design of H. 264 decoder with integrated hardware and software.

Desella, Raja. (2020). IMPLEMENTATION AND DEVELOPMENT OF HIGH ACCURACY H.265/HEVC STANDARD USING 45nm TECHNOLOGY. 7. 5787-5792.

APPENDIX i

DCTint.bsv

```
package DCTint;

import Vector :: *;
//import FixedPoint :: *;

import Flyint :: *;

interface DCTint_ifc;
	method Action fetch_Input (Vector#(8, int) signal);
	method Vector#(8, Reg#(int)) result();
endinterface

module mkDCTint (DCTint_ifc);

	Flyint_ifc flyint <- mkFlyint();

	Vector#(8, Reg#(int)) x <- replicateM(mkReg(0)); // Input

	Vector#(8, Reg#(int)) x1 <- replicateM(mkReg(0)); // Stage 1 Output
	Vector#(8, Reg#(int)) x2 <- replicateM(mkReg(0)); // Stage 2 Output
	Vector#(8, Reg#(int)) x3 <- replicateM(mkReg(0)); // Stage 3 Output

	Vector#(8, Reg#(int)) f <- replicateM(mkReg(0));
	Vector#(8, Reg#(int)) ft <- replicateM(mkReg(0)); // DCT Output

	Reg#(Bit#(3)) stage <- mkReg(0);  // Butterfly Stage

	// Cosine and Sine Multipliers
	let s1 = 50;  // c7
	let c1 = 251; // s7
	let s2 = 98; // c6
	let c2 = 237; // s6
	let s3 = 142; // c5
	let c3 = 213; // s5
	let c4 = 181; // s4
	/*let s1 = 799;  // c7
	let c1 = 4017; // s7
	let s2 = 1567; // c6
	let c2 = 3784; // s6
	let s3 = 2276; // c5
	let c3 = 3406; // s5
	let c4 = 2896; // s4*/

	// DCT Computation
```

```
// Butterfly Stage 1
rule fly1 (stage == 1);
        x1[0] <= x[0] + x[7];
        x1[7] <= x[0] - x[7];
        x1[1] <= x[1] + x[6];
        x1[6] <= x[1] - x[6];
        x1[2] <= x[2] + x[5];
        x1[5] <= x[2] - x[5];
        x1[3] <= x[3] + x[4];
        x1[4] <= x[3] - x[4];
        stage <= 2;
endrule: fly1

// Butterfly Stage 2
rule fly2 (stage == 2);
        x2[0] <= x1[0] + x1[3];
        x2[1] <= x1[1] + x1[2];
        x2[2] <= x1[1] - x1[2];
        x2[3] <= x1[0] - x1[3];
        x2[4] <= x1[4];
        x2[5] <= (c4*(x1[6]-x1[5]));
        x2[6] <= (c4*(x1[5]+x1[6]));
        x2[7] <= x1[7];
        stage <= 3;
endrule: fly2

// Butterfly Stage 3
rule fly3 (stage == 3);
        x3[0] <= (c4*(x2[0]+x2[1]));
        x3[1] <= (c4*(x2[0]-x2[1]));
        x3[2] <= (s2*x2[2] + c2*x2[3]);
        x3[3] <= (-c2*x2[2] + s2*x2[3]);
        x3[4] <= (x2[4] << 8) + x2[5];
        x3[5] <= (x2[4] << 8) - x2[5];
        x3[6] <= -x2[6] + (x2[7] << 8);
        x3[7] <= x2[6] + (x2[7] << 8);
        stage <= 4;
endrule: fly3

// Butterfly Stage 4
rule fly4 (stage == 4);
        f[0] <= x3[0];
        f[1] <= (s1*x3[4]) + (c1*x3[7]);
        f[2] <= x3[2];
        f[3] <= -(s3*x3[5]) + (c3*x3[6]);
        f[4] <= x3[1];
        f[5] <= (c3*x3[5]) + (s3*x3[6]);
        f[6] <= x3[3];
        f[7] <= -(c1*x3[4]) + (s1*x3[7]);
```

```
            stage <= 5;
        endrule: fly4

        method Action fetch_Input (Vector#(8, int) signal) if (stage == 0);
            for (Integer j=0; j<8; j = j+1) begin
                x[j] <= unpack(pack(signal[j]));
            end
            stage <= 1;
        endmethod

        method Vector#(8, Reg#(int)) result() if (stage == 5);
            return f;
        endmethod

endmodule: mkDCTint

endpackage
```

TbDCTint.bsv

```
package TbDCTint;

import DCTint :: *;
import Int2Fxp :: *;

import FixedPoint :: *;
import Vector :: *;

(* synthesize *)
module mkTbDCTint (Empty);
        DCTint_ifc dctint <- mkDCTint();
        Int2Fxp_ifc int2fxp <- mkInt2Fxp();
        // Input
        Vector#(8, int) sig = replicate(0); //[97, 102, 101, 98, 94, 90, 89, 91]
        sig[0] = 97;
        sig[1] = 102;
        sig[2] = 101;
        sig[3] = 98;
        sig[4] = 94;
        sig[5] = 90;
        sig[6] = 89;
        sig[7] = 91;

        Vector#(8, Reg#(int)) f0 <- replicateM(mkReg(0));
        Reg#(Bit#(3)) flag <- mkReg(0);
```

```
        FixedPoint#(32,4) out0 = 0;
        FixedPoint#(32,4) out1 = 0;
        FixedPoint#(32,4) out2 = 0;
        FixedPoint#(32,4) out3 = 0;
        FixedPoint#(32,4) out4 = 0;
        FixedPoint#(32,4) out5 = 0;
        FixedPoint#(32,4) out6 = 0;
        FixedPoint#(32,4) out7 = 0;

        rule start (flag == 0);
                dctint.fetch_Input (sig);
                flag <= 1;
        endrule

        rule transform (flag == 1);
                f0[0] <= dctint.result()[0];
                f0[1] <= dctint.result()[1];
                f0[2] <= dctint.result()[2];
                f0[3] <= dctint.result()[3];
                f0[4] <= dctint.result()[4];
                f0[5] <= dctint.result()[5];
                f0[6] <= dctint.result()[6];
                f0[7] <= dctint.result()[7];
                flag <= 2;
        endrule

        rule result (flag == 2);
                FixedPoint#(32,4) out0 = fromInt(unpack(pack(f0[0])))/512;
                FixedPoint#(32,4) out1 = fromInt(unpack(pack(f0[1])))/131072;
                FixedPoint#(32,4) out2 = fromInt(unpack(pack(f0[2])))/512;
                FixedPoint#(32,4) out3 = fromInt(unpack(pack(f0[3])))/131072;
                FixedPoint#(32,4) out4 = fromInt(unpack(pack(f0[4])))/512;
                FixedPoint#(32,4) out5 = fromInt(unpack(pack(f0[5])))/131072;
                FixedPoint#(32,4) out6 = fromInt(unpack(pack(f0[6])))/512;
                FixedPoint#(32,4) out7 = fromInt(unpack(pack(f0[7])))/131072;

                fxptWrite(4, out0); $display(", ");
                fxptWrite(4, out1); $display(", ");
                fxptWrite(4, out2); $display(", ");
                fxptWrite(4, out3); $display(", ");
                fxptWrite(4, out4); $display(", ");
                fxptWrite(4, out5); $display(", ");
                fxptWrite(4, out6); $display(", ");
                fxptWrite(4, out7); $display("\n");
                $finish;
        endrule
endmodule: mkTbDCTint

endpackage
```

APPENDIX ii

Modules.bsv

```
import ConfigReg::*;

function int cos64_lookup (int inp);
	case(inp) matches
		0: return 16384;
		1: return 16364;
		2: return 16305;
		3: return 16207;
		4: return 16069;
		5: return 15893;
		6: return 15679;
		7: return 15426;
		8: return 15137;
		9: return 14811;
		10: return 14449;
		11: return 14053;
		12: return 13623;
		13: return 13160;
		14: return 12665;
		15: return 12140;
		16: return 11585;
		17: return 11003;
		18: return 10394;
		19: return 9760;
		20: return 9102;
		21: return 8423;
		22: return 7723;
		23: return 7005;
		24: return 6270;
		25: return 5520;
		26: return 4756;
		27: return 3981;
		28: return 3196;
		29: return 2404;
		30: return 1606;
		31: return 804;
		32: return 0;
	endcase
endfunction

function int cos64(int inp);
	int angle2 = unpack(pack(inp) & 32'd127);

	if (( angle2 > 0) && (angle2 <= 32 ))
```

```
      return cos64_lookup(angle2);
   else if (( angle2 > 32) && (angle2 <= 64 ))
      return -1*cos64_lookup(64-angle2) ;
   else if (( angle2 > 64) && (angle2 <= 96 ))
      return -1*cos64_lookup(angle2-64);
   else
      return cos64_lookup(128-angle2);

endfunction


function int sin64(int inp);
        return cos64(inp - 32);
endfunction

function int round2(int x, Integer n); //Can change n to 14
        return (x+(1<<(n-1))) >> n;
endfunction

function int brev (Integer bits, int inp_n);
        int result = 0;
        int n = inp_n;
        for (Integer i=0; i<bits; i=i+1) begin
                result = result << 1;
                result = result | n & 1;
                n = n >> 1;
        end

        return result;
endfunction

interface Ifc_butterfly; ////B
        method Action inps(int ta, int tb, int angle, Bool flag);
        method int ta;
        method int tb;
endinterface

module mkButterfly (Ifc_butterfly);
        Reg#(int) tanew <- mkReg(0), tbnew <- mkReg(0);
        Reg#(Bool) valid_outs <- mkReg(False);

        method Action inps(int ta, int tb, int angle, Bool flag);
                int x = ta*cos64(angle) - tb * sin64(angle);
                int y = ta*sin64(angle) + tb*cos64(angle);

                if (flag) begin //flag = 1
                        tanew <= round2(y, 14);
                        tbnew <= round2(x, 14);
                end
```

```
                else begin
                        tanew <= round2(x, 14);
                        tbnew <= round2(y, 14);
                end

                valid_outs <= True;
        endmethod

        method int ta if (valid_outs);
                return tanew;
        endmethod

        method int tb if (valid_outs);
                return tbnew;
        endmethod
endmodule

interface Ifc_Hadamard; ///H
        method Action inps(int ta, int tb, Bool flag);
        method int ta;
        method int tb;
endinterface

module mkHadamard (Ifc_Hadamard);
        Reg#(int) tanew <- mkConfigReg(0), tbnew <- mkConfigReg(0);
        Reg#(Bool) valid_outs <- mkReg(False);

        method Action inps(int ta, int tb, Bool flag);
                if (flag) begin         //flag = 1
                        tanew <= tb - ta;
                        tbnew <= ta + tb;
                end
                else begin
                        tanew <= ta + tb;
                        tbnew <= ta - tb;
                end

                valid_outs <= True;
        endmethod

        method int ta if (valid_outs);
                return tanew;
        endmethod

        method int tb if (valid_outs);
                return tbnew;
        endmethod
endmodule
```

IDCT.bsv

```
package IDCT;

import Modules::*;
import Vector::*;
import ConfigReg::*;


interface Ifc_IDCT#(numeric type n, numeric type n1);
	method Action start;
	method Vector#(n, int) t_read;
	method Action t_update (Vector#(n, int) inp_vec);
endinterface

typedef enum {
	State0, State1, State2, State3, State4, Display
} IDCT_4_stages deriving(Bits, Eq, FShow);




module mkIDCT_4 (Ifc_IDCT#(4, 2));
	Vector#(4, Reg#(int)) t_array <- replicateM(mkConfigReg(0));/////

	Reg#(IDCT_4_stages) mod_stage <- mkReg(State0);

	Vector#(2, Ifc_butterfly) modB <- replicateM(mkButterfly);
	Vector#(2, Ifc_Hadamard) modH <- replicateM(mkHadamard);

	PulseWire disable_start <- mkPulseWire, updating <- mkPulseWire;

	rule rl_stg1 (mod_stage == State1);
		modB[0].inps(t_array[0], t_array[1], 16, True);
		modB[1].inps(t_array[2], t_array[3], 32 - brev(5, 2), False); ////
		mod_stage <= State2;
	endrule

	rule rl_stg2 (mod_stage == State2);
		t_array[0] <= modB[0].ta;
		t_array[1] <= modB[0].tb;

		t_array[2] <= modB[1].ta;
		t_array[3] <= modB[1].tb;

		mod_stage <= State3;
	endrule
```

```
	rule rl_stg7 (mod_stage == State3);
		for (int i=0; i<2; i=i+1) begin
			modH[i].inps(t_array[i], t_array[4-1-i], False);
		end
		mod_stage <= State4;
	endrule

	rule rl_store7 (mod_stage == State4);
		for (int i=0; i<2; i=i+1) begin
			t_array[i] <= modH[i].ta;
			t_array[4-1-i] <= modH[i].tb;
		end

		mod_stage <= State0;
	endrule

	method Action start if((!disable_start) && (mod_stage == State0));
		mod_stage <= State1;
	endmethod

	method Vector#(4, int) t_read if((!disable_start) && (!updating) && (mod_stage
== State0)); //////
		Vector#(4, int) temp = defaultValue;
		for (int i=0; i<4; i=i+1)
			temp[i] = t_array[i];
		return temp;
	endmethod

	method Action t_update (Vector#(4, int) inp_vec) if((!disable_start) &&
(mod_stage == State0));
		for(int i=0; i<4; i=i+1)
			t_array[i] <= inp_vec[i];
		updating.send();
	endmethod
endmodule

typedef enum {
	State0, State0x, State1, State2, State3, State4, State5, State6, State7, State8,
Display, Displayx
} IDCT_8_stages deriving(Bits, Eq, FShow);

module mkIDCT_8 (Ifc_IDCT#(8, 3));
	Vector#(4, Reg#(int)) t_array <- replicateM(mkConfigReg(0));/////

	Reg#(IDCT_8_stages) mod_stage <- mkReg(State0);

	Vector#(2, Ifc_butterfly) modB <- replicateM(mkButterfly);
	Vector#(4, Ifc_Hadamard) modH <- replicateM(mkHadamard);
```

```
        PulseWire disable_start <- mkPulseWire, updating <- mkPulseWire;

        Ifc_IDCT#(4, 2) mod_4 <- mkIDCT_4;

        rule rl_stg2 (mod_stage == State1);
                mod_4.start;

                for (int i=0; i<2; i=i+1) begin
                        modB[i].inps(t_array[4+i-4], t_array[8-1-i-4], 32-brev(5, 4+i),
False);
                end

                mod_stage <= State2;
        endrule

        rule rl_store2 (mod_stage == State2);
                for (int i=0; i<2; i=i+1) begin
                        t_array[4+i-4] <= modB[i].ta;
                        t_array[8-1-i-4] <= modB[i].tb;
                end

                mod_stage <= State3;
        endrule

        rule rl_stg3 (mod_stage == State3);
                modH[0].inps(t_array[4-4], t_array[4+1-4], False); //j=0
                modH[1].inps(t_array[4+2-4], t_array[4+2+1-4], True); //j=1

                mod_stage <= State4;
        endrule

        rule rl_store3 (mod_stage == State4);
                t_array[4-4] <= modH[0].ta;
                t_array[5-4] <= modH[0].tb;

                t_array[6-4] <= modH[1].ta;
                t_array[7-4] <= modH[1].tb;

                mod_stage <= State5;
        endrule

        rule rl_stg6 (mod_stage == State5);
                modB[0].inps(t_array[8-1-1-4], t_array[4+1-4], 16, True);

                mod_stage <= State6;
        endrule
```

```
	rule rl_store6 (mod_stage == State6);
		t_array[8-1-1-4] <= modB[0].ta;
		t_array[4+1-4] <= modB[0].tb;

		mod_stage <= State7;
	endrule

	rule rl_stg7 (mod_stage == State7);
		let z = mod_4.t_read;

		for (int i=0; i<4; i=i+1) begin
			modH[i].inps(z[i], t_array[8-1-i-4], False);
		end

		mod_stage <= State8;
	endrule

	rule rl_store7 (mod_stage == State8);
		Vector#(4, int) tempz = defaultValue;

		for(int i=0; i<4; i=i+1) begin
			tempz[i] = modH[i].ta;
			t_array[8-1-i-4] <= modH[i].tb;
		end

		mod_4.t_update(tempz);
		mod_stage <= State0;
	endrule


	method Action start if((!disable_start) && (mod_stage == State0));
		mod_stage <= State1;
	endmethod

	method Vector#(8, int) t_read if((!disable_start) && (!updating) && (mod_stage
== State0));
		Vector#(8, int) temp = defaultValue;
		let z = mod_4.t_read;

		for (int i=0; i<4; i=i+1) begin
			temp[i] = z[i];
			temp[4+i] = t_array[i];
		end
		return temp;
	endmethod
```

```
	method Action t_update (Vector#(8, int) inp_vec) if((!disable_start) &&
(mod_stage == State0));
		Vector#(4, int) temp = defaultValue;

		for(int i=0; i<4; i=i+1) begin
			temp[i] = inp_vec[i];
			t_array[i] <= inp_vec[4+i];
		end

		mod_4.t_update(temp);
		updating.send();
	endmethod
endmodule


typedef enum {
	State0, State0x, State1, State2, State3, State4, State5, State6, State7, State8, State9,
State10, State11, State12, Display, Displayx
} IDCT_16_stages deriving(Bits, Eq, FShow);

module mkIDCT_16 (Ifc_IDCT#(16, 4));
	Vector#(8, Reg#(int)) t_array <- replicateM(mkConfigReg(0));/////

	Reg#(IDCT_16_stages) mod_stage <- mkReg(State0);

	Vector#(4, Ifc_butterfly) modB <- replicateM(mkButterfly);
	Vector#(8, Ifc_Hadamard) modH <- replicateM(mkHadamard);

	PulseWire disable_start <- mkPulseWire, updating <- mkPulseWire;

	Ifc_IDCT#(8, 3) mod_8 <- mkIDCT_8;

	rule rl_stg2 (mod_stage == State1);
		mod_8.start;
		for (int i=0; i<4; i=i+1) begin
			modB[i].inps(t_array[8+i-8], t_array[16-1-i-8], 32-brev(5, 8+i),
False);
		end
		mod_stage <= State2;
	endrule

	rule rl_store2 (mod_stage == State2);
		for (int i=0; i<4; i=i+1) begin
			t_array[8+i-8] <= modB[i].ta;
			t_array[16-1-i-8] <= modB[i].tb;
		end

		mod_stage <= State3;
	endrule
```

```
	rule rl_stg3 (mod_stage == State3);
		for (int i=0; i<2; i=i+1) begin
			modH[2*i].inps(t_array[8+4*i-8], t_array[8+4*i+1-8], False); //j=0
			modH[2*i+1].inps(t_array[8+4*i+2-8], t_array[8+4*i+2+1-8],
True);
		end
		mod_stage <= State4;
	endrule

	rule rl_store3 (mod_stage == State4);
		for (int i=0; i<2; i=i+1) begin
			t_array[8+4*i-8] <= modH[2*i].ta;
			t_array[8+4*i+1-8] <= modH[2*i].tb;

			t_array[8+4*i+2-8] <= modH[2*i+1].ta;
			t_array[8+4*i+2+1-8] <= modH[2*i+1].tb;
		end

		mod_stage <= State5;
	endrule

	rule rl_stg5 (mod_stage == State5);
		modB[0].inps(t_array[16-4+2-8], t_array[8+4-3-8], 24, True);
		modB[1].inps(t_array[16-4+2-4-8], t_array[8+4-3+4-8], 24+48, True);
		mod_stage <= State6;
	endrule

	rule rl_store5 (mod_stage == State6);
		t_array[16-4+2-8] <= modB[0].ta;
		t_array[8+4-3-8] <= modB[0].tb;

		t_array[16-4+2-4-8] <= modB[1].ta;
		t_array[8+4-3+4-8] <= modB[1].tb;

		mod_stage <= State7;
	endrule

	rule rl_stg5b (mod_stage == State7);
		for (int i=0; i<2*4-7+1; i=i+1)  begin
			modH[2*i].inps(t_array[8+i-8], t_array[8+4-1-i-8], False);
			modH[2*i+1].inps(t_array[8+4+i-8], t_array[8+4-1+4-i-8], True);
		end

		mod_stage <= State8;
	endrule
```

```
rule rl_store5b (mod_stage == State8);
        for (int i=0; i<2*4-7+1; i=i+1)  begin
                t_array[8+i-8] <= modH[2*i].ta;
                t_array[8+4-1-i-8] <= modH[2*i].tb;

                t_array[8+4+i-8] <= modH[2*i+1].ta;
                t_array[8+4-1+4-i-8] <= modH[2*i+1].tb;
        end

        mod_stage <= State9;
endrule

rule rl_stg6 (mod_stage == State9);
        for (int i=0; i<2; i=i+1) begin
                modB[i].inps(t_array[16-2-1-i-8], t_array[8+2+i-8], 16, True);
        end
        mod_stage <= State10;
endrule

rule rl_store6 (mod_stage == State10);
        for (int i=0; i<2; i=i+1) begin
                t_array[16-2-1-i-8] <= modB[i].ta;
                t_array[8+2+i-8] <= modB[i].tb;
        end

        mod_stage <= State11;
endrule

rule rl_stg7 (mod_stage == State11); ////////
        let z = mod_8.t_read;

        for (int i=0; i<8; i=i+1) begin
                modH[i].inps(z[i], t_array[16-1-i-8], False);
        end

        mod_stage <= State12;
endrule

rule rl_store7 (mod_stage == State12);
        Vector#(8, int) tempz = defaultValue;

        for(int i=0; i<8; i=i+1) begin
                tempz[i] = modH[i].ta;
                t_array[16-1-i-8] <= modH[i].tb;
        end

        mod_8.t_update(tempz);
        mod_stage <= State0;
endrule
```

```
	method Action start if((!disable_start) && (mod_stage == State0));
		mod_stage <= State1;
	endmethod

	method Vector#(16, int) t_read if((!disable_start) && (!updating) && (mod_stage
== State0));
		Vector#(16, int) temp = defaultValue;
		let z = mod_8.t_read;

		for (int i=0; i<8; i=i+1) begin
			temp[i] = z[i];
			temp[8+i] = t_array[i];
		end
		return temp;
	endmethod

	method Action t_update (Vector#(16, int) inp_vec) if((!disable_start) &&
(mod_stage == State0));
		Vector#(8, int) temp = defaultValue;

		for(int i=0; i<8; i=i+1) begin
			temp[i] = inp_vec[i];
			t_array[i] <= inp_vec[8+i];
		end

		mod_8.t_update(temp);
		updating.send();
	endmethod
endmodule



typedef enum {
	State0, State0x, State1, State2, State3, State4, State5, State6, State7, State8, State9,
State10, State11, State12, State13, State14, State15, State16, Display, Displayx
} IDCT_32_stages deriving(Bits, Eq, FShow);

module mkIDCT_32 (Ifc_IDCT#(32, 5));
	Vector#(16, Reg#(int)) t_array <- replicateM(mkConfigReg(0));

	Reg#(IDCT_32_stages) mod_stage <- mkReg(State0);

	Vector#(8, Ifc_butterfly) modB <- replicateM(mkButterfly);
	Vector#(16, Ifc_Hadamard) modH <- replicateM(mkHadamard);

	PulseWire disable_start <- mkPulseWire, updating <- mkPulseWire;

	Ifc_IDCT#(16, 4) mod_16 <- mkIDCT_16;
```

```
	rule rl_stg2 (mod_stage == State1);
		mod_16.start;

		for (int i=0; i<8; i=i+1) begin
			modB[i].inps(t_array[16+i-16], t_array[32-1-i-16], 32-brev(5, 16+i),
False);
		end

		mod_stage <= State2;
	endrule

	rule rl_store2 (mod_stage == State2);
		for (int i=0; i<8; i=i+1) begin
			t_array[16+i-16] <= modB[i].ta;
			t_array[32-1-i-16] <= modB[i].tb;
		end

		mod_stage <= State3;
	endrule

	rule rl_stg3 (mod_stage == State3);
		for (int i=0; i<4; i=i+1) begin
			modH[2*i].inps(t_array[16+4*i-16], t_array[16+4*i+1-16], False);
//j=0
			modH[2*i+1].inps(t_array[16+4*i+2-16], t_array[16+4*i+2+1-16],
True); //j=1
		end
		mod_stage <= State4;
	endrule

	rule rl_store3 (mod_stage == State4);
		for (int i=0; i<4; i=i+1) begin
			t_array[16+4*i-16] <= modH[2*i].ta;
			t_array[16+4*i+1-16] <= modH[2*i].tb;

			t_array[16+4*i+2-16] <= modH[2*i+1].ta;
			t_array[16+4*i+2+1-16] <= modH[2*i+1].tb;
		end

		mod_stage <= State5;
	endrule

	rule rl_stg4 (mod_stage == State5);
		for (int i=0; i<2; i=i+1)
			for (int j=0; j<2; j=j+1)
				modB[2*i+j].inps(t_array[32-5+3-8*j-4*i-16],
t_array[16+5-4+8*j+4*i-16], 28-16*i+56*j, True);
```

```
            mod_stage <= State6;
        endrule

        rule rl_store4 (mod_stage == State6);
            for (int i=0; i<2; i=i+1)
                for (int j=0; j<2; j=j+1) begin
                    t_array[32-5+3-8*j-4*i-16] <= modB[2*i+j].ta;
                    t_array[16+5-4+8*j+4*i-16] <= modB[2*i+j].tb;
                end

            mod_stage <= State7;
        endrule

        rule rl_stg4b (mod_stage == State7);
            for (int i=0; i<2; i=i+1)
                for (int j=0; j<4; j=j+1) begin
                    Bool temp = pack(j)[0]==1;
                    modH[4*i+j].inps(t_array[16+4*j+i-16],
t_array[16+8-5+4*j-i-16], temp);
                end
            mod_stage <= State8;
        endrule

        rule rl_store4b (mod_stage == State8);
            for (int i=0; i<2; i=i+1)
                for (int j=0; j<4; j=j+1) begin
                    t_array[16+4*j+i-16] <= modH[4*i+j].ta;
                    t_array[16+8-5+4*j-i-16] <= modH[4*i+j].tb;
                end

            mod_stage <= State9;
        endrule

        rule rl_stg5 (mod_stage == State9);
            for (int i=0; i<2; i=i+1) begin
                modB[2*i].inps(t_array[32-5+2-i-16], t_array[16+5-3+i-16], 24,
True);
                modB[2*i+1].inps(t_array[32-5+2-i-8-16], t_array[16+5-3+i+8-16],
24+48, True);
            end

            mod_stage <= State10;
        endrule

        rule rl_store5 (mod_stage == State10);
            for (int i=0; i<2; i=i+1) begin
                t_array[32-5+2-i-16] <= modB[2*i].ta;
                t_array[16+5-3+i-16] <= modB[2*i+1].tb;
```

```
                t_array[32-5+2-i-8-16] <= modB[2*i+1].ta;
                t_array[16+5-3+i+8-16] <= modB[2*i+1].tb;
            end

            mod_stage <= State11;
        endrule

        rule rl_stg5b (mod_stage == State11);
            for (int i=0; i<2*5-7+1; i=i+1)  begin
                modH[2*i].inps(t_array[16+i-16], t_array[16+8-1-i-16], False);
                modH[2*i+1].inps(t_array[16+8+i-16], t_array[16+8-1+8-i-16],
True);
            end

            mod_stage <= State12;
        endrule

        rule rl_store5b (mod_stage == State12);
            for (int i=0; i<2*5-7+1; i=i+1)  begin
                t_array[16+i-16] <= modH[2*i].ta;
                t_array[16+8-1-i-16] <= modH[2*i].tb;

                t_array[16+8+i-16] <= modH[2*i+1].ta;
                t_array[16+8-1+8-i-16] <= modH[2*i+1].tb;
            end

            mod_stage <= State13;
        endrule

        rule rl_stg6 (mod_stage == State13);
            for (int i=0; i<4; i=i+1) begin
                modB[i].inps(t_array[32-4-1-i-16], t_array[16+4+i-16], 16, True);
            end
            mod_stage <= State14;
        endrule

        rule rl_store6 (mod_stage == State14);
            for (int i=0; i<4; i=i+1) begin
                t_array[32-4-1-i-16] <= modB[i].ta;
                t_array[16+4+i-16] <= modB[i].tb;
            end

            mod_stage <= State15;
        endrule

        rule rl_stg7 (mod_stage == State15);
            let z = mod_16.t_read;

            for (int i=0; i<16; i=i+1) begin
```

```
                modH[i].inps(z[i], t_array[32-1-i-16], False);
            end

            mod_stage <= State16;
        endrule

        rule rl_store7 (mod_stage == State16);
            Vector#(16, int) tempz = defaultValue;

            for(int i=0; i<16; i=i+1) begin
                tempz[i] = modH[i].ta;
                t_array[32-1-i-16] <= modH[i].tb;
            end

            mod_16.t_update(tempz);
            mod_stage <= Display;
        endrule

        rule displayer (mod_stage == Display);
            let z = mod_16.t_read;
            for (int i=0; i<16; i=i+1)
                $display("T[%d] = %d", i, z[i]);
            for (int i=0; i<16; i=i+1)
                $display("T[%d] = %d", i, t_array[i]);

            $finish;
        endrule

        method Action start if((!disable_start) && (mod_stage == State0));
            mod_stage <= State1;
        endmethod

        method Vector#(32, int) t_read if((!disable_start) && (!updating) && (mod_stage
== State0)); //////
            Vector#(32, int) temp = defaultValue;
            let z = mod_16.t_read;

            for (int i=0; i<16; i=i+1) begin
                temp[i] = z[i];
                temp[16+i] = t_array[i];
            end
            return temp;
        endmethod

        method Action t_update (Vector#(32, int) inp_vec) if((!disable_start) &&
(mod_stage == State0));
            Vector#(16, int) temp = defaultValue;

            for(int i=0; i<16; i=i+1) begin
```

```
                        temp[i] = inp_vec[i];
                        t_array[i] <= inp_vec[16+i];
                end

                mod_16.t_update(temp);
                updating.send();
        endmethod
endmodule

(*synthesize*)
module mkTestBench (Empty);
        Ifc_IDCT#(32, 5) dut <- mkIDCT_32;

        //Reg#(Vector#(32, int)) inp_vec <- mkReg(replicate(2));

        Reg#(int) stage <- mkReg(0);

        rule initialier (stage==0);
                Vector#(32, int) temp = defaultValue;

                for (int i=0; i<32; i=i+1)
                        temp[i] = i;

                dut.t_update(temp);
                stage <= 1;
        endrule

        rule starter (stage==1);
                dut.start;
                stage <= 2;
        endrule

        rule finisher (stage==2);
                let z = dut.t_read;

                for (int i=0; i<32;i=i+1) begin
                        $display("T[%d]=%d",i, z[i]);
                end
                $finish;
        endrule
endmodule
endpackage
```