EE577b VLSI Design Project
A Design of a Turbo Decoder Chip

Pornchai Pawawongsak

CENG 00-05

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213-740-4481)
May 2000

EE577B VLSI Design Project

A Design of Turbo Decoder Chip

Spring 2000

By Pornchai Pawawongsak

Contents

Contents

Summary of Upgrades/Fixes from Phase I and II

1) The test bench now reads two input files, one for system bit information (ck0.dat) and the other for parity bit (ck1.dat). The test bench also compares the final output against the given test vector file (hard_dec_.dat).
2) Add 2 more pipeline stages in structural RTL design.
3) Add tri-state gates to block the data flow going into the completion stage.
4) The final output now is read out at double the clock rate to avoid the conflict of sharing the deinterleaver with SISO2.
5) The RAM used as buffers in Input_buff now takes only one input address and does an internal decoding to produce two addresses. This is done just to match the layout implementation.
6) Other minor bug fixes.

Summary of Incompleteness

1) Behavioral RTL design works with any even number of block lengths. Structural RTL design works only with a block length of 2^B-2, where B is an integer. So, simulation for test vector 1024 for structural RTL is not provided.
2) The speed and area estimation is also done at a block length of 1022.
3) In my design, the behavioral RTL design is an unpipelined version of the structural RTL design. Therefore, the output of structural RTL design is a delayed version of the behavioral one.

   However, during the F_B state while the completion process is not running, the soft information output do not exactly match because the implementation of the two designs is slightly different. The behavioral one holds the completion process by not specifying the output for this case, so the output is automatically latched. The structural one holds the completion process by blocking the input of the first completion stage, so the output is held at some garbage value. The soft information output during this period is not used so it does not affect the correctness of the calculation in any way. I find no use going back to make the result of the behavioral design match "exactly" that of the structural one.
4) I did not break everything down to their lowest level (adder, mux, decoder, …) in the Verilog codes because it will be too messy and does not help in visualizing the design. I broke it down enough to see the architecture of the design. Everything should be synthesizable except ROMs and RAMs. The speed and area analysis in section 4, however, goes down to the lowest level.
5) The speed and area estimation is crude.
6) The detailed description of the algorithm implemented is not included. The reader is referred to [5] and any other additional references about Turbo codes.
7) The explanation about why F&B values can be implemented using on seven bits is not included.

Section 1 An Overview of Turbo Decoder

Introduction

Turbo codes are a new class of error correcting codes that have been proved to provide a considerably better performance than existing traditional coding schemes. Originated in early 90's, Turbo codes now is a center of many designers and researchers' attention. The main application of turbo codes is in the area of communications especially mobile communications.

Three outstanding characteristics of Turbo codes are

1) Use of "soft" information: Starting from detecting the signal from the channel (at demodulators), a turbo decoder expects soft information about the symbols detected, not hard decision (1 or 0) as most decoding schemes use. In other words, a Turbo decoder keeps the statistics or probability information of how much likely a received symbol is close to "0" or "1" and take that information into account in determining a hard decision later. This is a reason why Turbo codes have a chance to perform better than traditional codes.

2) Iterative processing: Turbo decoding is a scheme that takes soft information inputs and produce soft information outputs. This operation can be performed repeatedly, meaning taking the output of previous iteration and feeding it back as input again, until the soft information outputs converge to good values.

3) Use of interleaver: A characteristic of good error control code is to add redundant bits to the uncoded sequence such that the coded sequence has information about the original sequence spread out equally over a long period of coded sequence. Turbo codes have this property because the coder works on both a normal sequence and an interleaved sequence. The longer the interleaver, the better protection performance it can achieve.

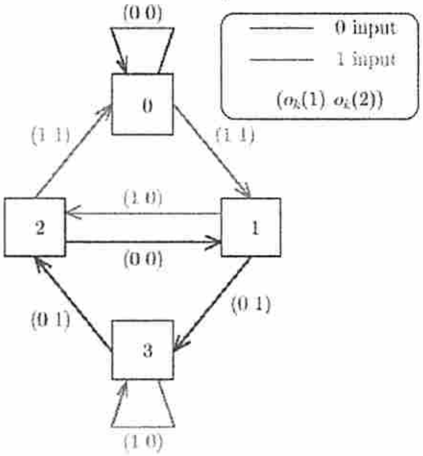
How do Turbo codes work?

Let's look at some more details of the operation of Turbo coding and decoding. The picture in the next page shows a top-level block diagram of a Turbo coder and decoder inside a transmitter and receiver respectively.

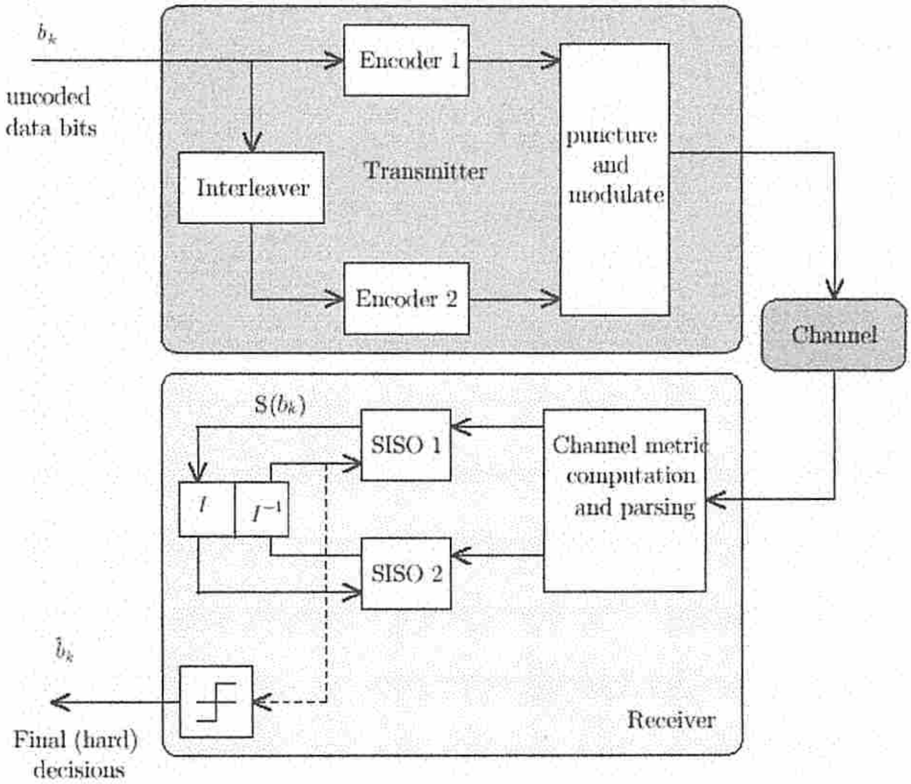
At the coder side,

1) The normal uncoded sequence Enter Encoder 1 while the interleaved uncoded sequence enters Encoder 2.

2) Both encoders work the same way. It is just a 4-state finite state machine generating a "parity" bit for each received bit. The state diagram is shown below.



The two number at each transition (i, j) means i is the received bit and j is the generated parity bit.



3) Now we have four sequences, the original sequence (system1), the parity from the original sequence (parity1), the interleaved sequence (system2), and the parity form the interleaved sequence (parity2). We can choose how much information we want to send depending on how much bandwidth we have and how much reliable we want. The more information we send, the more protection we have against channel noises. For example, we could send

a) a sequence of …, system1 , parity1, system1, parity2, system1, parity1, … This is the most popular choice people use and gives a coding rate of 0.5 (original sequence rate is 0.5 that of transmitted sequence).
b) a sequence of …, system1, parity1, parity2, system1, parity1, parity2, … This gives a rate of 1/3.
c) a sequence of …, system1 system1, parity1, system1, system1, parity1, … This gives a rate of 2/3.

In other words, any rule is possible and, interestingly, the decoder can work with any of them.

3) The sequence is modulated and transmitted over the channel.

At the decoder,

1) The demodulator receives the signal (analog) and translates it into a sequence of soft information. Soft information can be represented using a multi-bit digital signal.

2) Now the received sequence has to be demultiplexed to SISO1 and SISO2. SISO is a soft-input soft-output processing block that systematically tries to find a better estimate of the soft information of system bits. The detail operation of SISOs is omitted here. SISO1 expects to work on system1 and parity1 bits, while SISO2 expects to work on system2 and parity2.

Depending on what has been sent, the soft information from the channel gets distributed to the corresponding SISO. For the bits that have not been sent, the SISO will just take the soft information of the bits to be zero meaning equally likely between "1" and "0".

3) First, SISO1 starts working on the sequence and produces a sequence of soft information output, which will be passed through the interleaver (same as being used at the transmitter) to SISO2. After SISO1 completes its first iteration, SISO2 starts working the same way and produces the soft information output, which will be passed through the deinterleaver back to SISO1.

4) Both SISOs take turn processing the sequence and exchange the soft information until the soft information sequence converges to acceptable range. The hard decision will then be made from the final soft information sequence.

Section 2 Behavioral RTL Design

The Turbo Decoding Algorithm to be Implemented

This project implements the Turbo decoding algorithm as described in class [5] which has the following characteristics:

1. Radix-2 based: has 8 transitions to be determined in each time step
2. Min-sum: approximate the logarithmic term in the soft information calculation to be zero.
3. Fixed-interval: each soft information output is produced from the data of the whole block.
4. Normalization of soft information for input being zero to be zero

The verilog code for this project can work with any even number of the block length and any number of bit-widths for soft information and F/B matrices. I choose the bit-width for F/B matrices to be 7. The number of iterations is fixed to be 10.

Variable Mapping between Algorithm Description and Verilog Code

To avoid confusion in reading the code, it is useful to summarize the variables used in the algorithm description in [5] and the corresponding variables (or arrays) used in the Verilog code. Note that the M, F, and B matrices, which have two dimensions (time and state), are mapped to one-dimensional arrays in the verilog code and MATLAB code.

Name	Variable name in algorithm description	Variable name in the code
Channel soft information	$SI(C_k(1)=1)$, $SI(C_k(2)=1)$ (system bit, parity bit)	zf1, zf2 for forward ACS zb1, zb2 for backward ACS
Transition matrix	$M(S_k, O_k(1), S_{k+1})$ S_k = state at time k S_k+1 = state at time k+1 $O_k(1)$ = input at time k	Although there are 8 values of M for each time step, two of them are always zero and half of them are always identical to each other, so we actually need to calculate and store only 3 values at each time step. M[k] : M(0, 1, 1) and M(2,1,0) M[k+1] : M(1, 0, 3) and M(3,0,2) M[k+2] : M(1, 1, 2) and M(3,1,3)
Forward ACS matrix, Backward ACS matrix	$F_{k-1}(S_k)$, $B_k(S_k)$	F[4*k] : $F_{k-1}(S_k=0)$ F[4*k+1] : $F_{k-1}(S_k=1)$ F[4*k+2] : $F_{k-1}(S_k=2)$ F[4*k+3] : $F_{k-1}(S_k=3)$ B[4*j] : $B_k(S_k=0)$ B[4*j+1] : $B_k(S_k=1)$ B[4*j+2] : $B_k(S_k=1)$ B[4*j+3] : $B_k(S_k=3)$

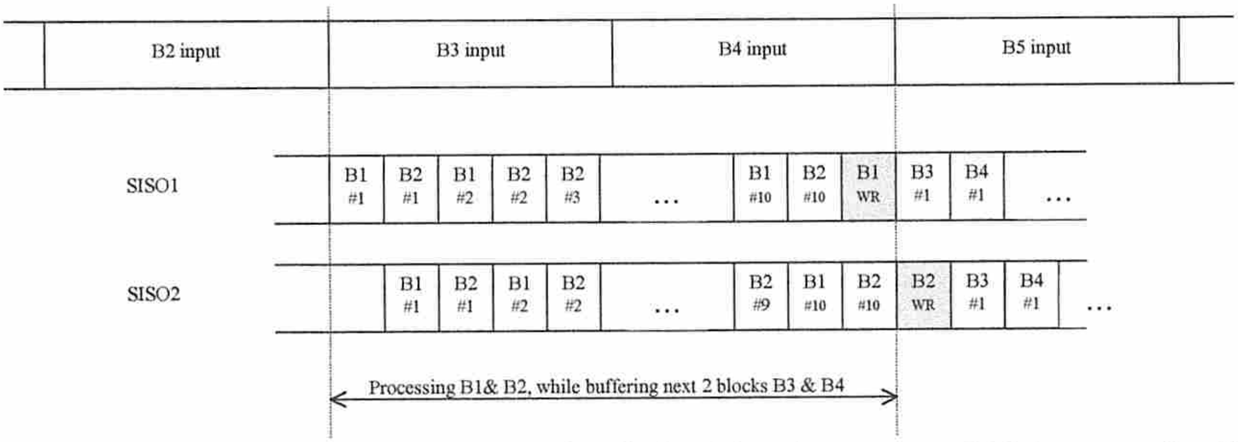
Name	Variable name in algorithm description	Variable name in the code
Soft information input	$SI(B_k(1)=1)$	SIf for forward computation (also use SI[k] to store received SIf) SIb for backward computation (also use SI[j] to store received SIb)
Soft information output	$SO(B_k(1)=1)$	SOf for forward computation SOb for backward computation

Scheduling of Computation at the block level

Each block of the input signals needs to be processed ten times in SISO1 and SISO2. Due to data dependency, the processing needs to be sequential, that is, SISO1 iteration 1 produces the results for SISO2 iteration 1, SISO2 iteration 1 produces the results for SISO1 iteration 2, and so on. Thus, to gain a major throughput increment through parallelism, pipelining across blocks of data has to be implemented with the cost of using more hardware resources and introducing more output delay.

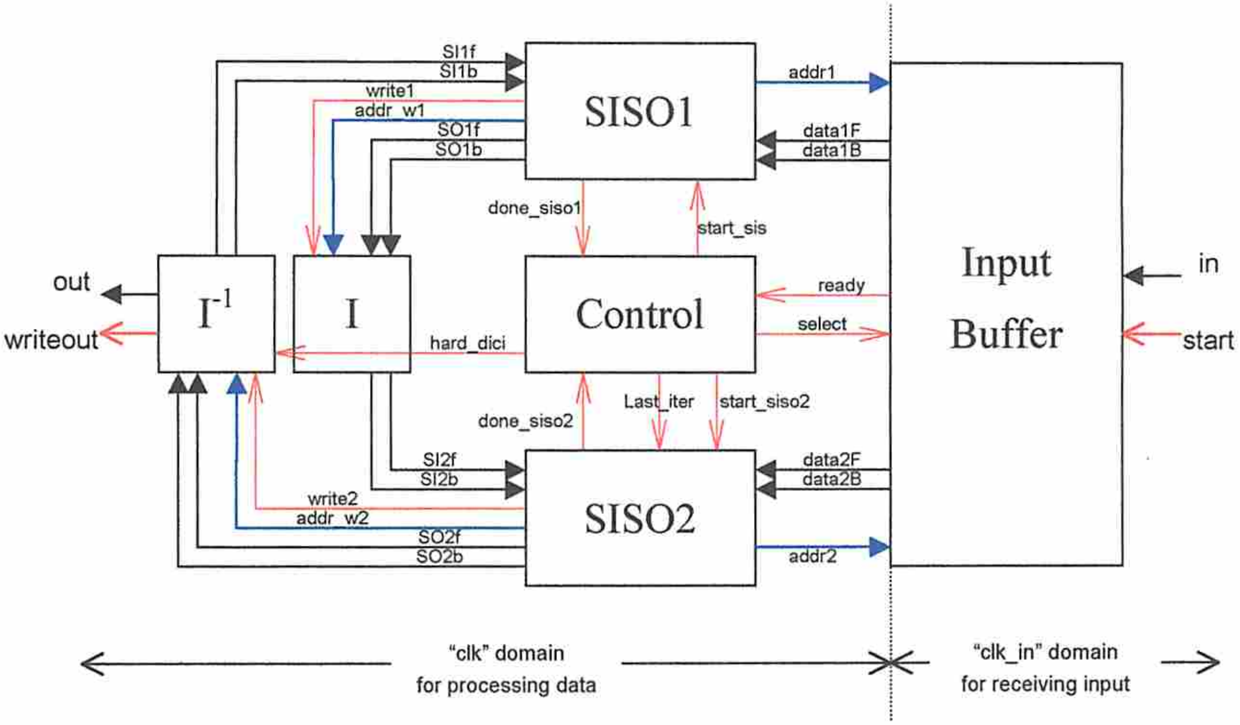
In this project, I use one level of pipelining across blocks by using separate hardware for SISO1 and SISO2, and process two blocks of data at the same time. This approximately increases the throughput by a factor of two over the case where single hardware is alternately used to do SISO1 and SISO2 calculation.

The rough timing diagram below show the input signal and the time it gets processed in the SISO1 and SISO2 block. "Bi" denotes data block i. During the time data B1 and B2 get processed in SISO1 and SISO2, B3 and B4 get buffered in the input buffer.



"WR" denotes the period of generating the hard decision output, which happens after 20 SISO iterations. The hard decision generation is actually implemented in the interleaver outside the SISO1 and SISO2. The WR period written in the diagram is just to help visualizing the scheduling. Currently the processing of SISO1 and SISO2 is stalled during the WR periods to simplify the design but further improvement is possible to start processing new two blocks while generating the hard decision of current blocks.

Top Level of the Design



The turbo decoder chip is designed to have 6 main functional blocks. Each block is described in structural RTL style in separate verilog files and combined structurally in "Turbo.v", which described the exact connections and signal names as shown in the figure above. Please be referred to section 3 of this report for all the verilog files in this design. Before going into details about the design of each block, here I will explain briefly the function of them.

1) Input buffer (described in Input_buff.v)

The 4-bit input signal, "in", is assumed to come in sequentially and alternatingly between the soft information of the system bits and the soft information of the parity bits, that is, SI(C_k (1)=1), SI(C_k(2)=1), SI(C_{k+2}(1)=1), SI(C_{k+2}(2)=1), ... The input buffer samples the input at the positive edge of clk_in signal, the input sample clock. Upon receiving the input, it combines system bit information and parity bit information to make an 8-bit data and store the data at every other clk_in cycle. Since the SISOs want to process 2 blocks of data at the same time, the input buffer needs to have 4 blocks of data buffer. While two blocks are used to buffer new inputs, the other two keep old data for SISO processing. Every time new two blocks of data have been received, the input buffer generates the "ready" pulse to tell the "control" block to start processing the new received data.

The reading process is done asynchronously; it responds directly to the address signals provided by the SISO's. Each data signal (data1F, data1B, data2F, and data2B) is 8-bit wide, which comprises system bit information in the 4 MSBs and parity bit information in the 4 LSBs.

2) SISO1 and SISO2 (described in SISO1.v and SISO2.v)

A SISO receives its inputs from the input buffer and from the other SISO through the interleaver/deinterleaver. It, in tern, produces soft information outputs to the other SISO. It is controlled to start processing in each iteration by the "control" block and each iteration works on the whole block of data. The detailed operation of SISOs will be explained later.

SISO1 and SISO2 have two slight differences. First, the calculation of the transition matrix is different and customized for each one of them. Second, SISO2 can be controlled to avoid subtracting the soft information input from its soft information output. This is particularly useful for the last iteration of SISO2 computation before making the final hard decision.

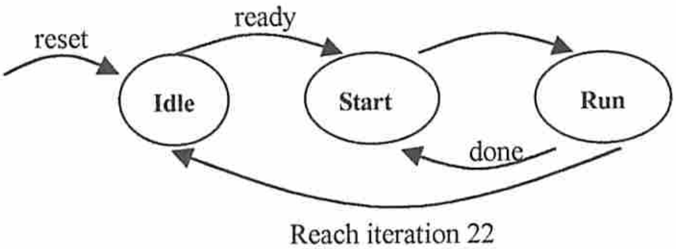
3) Interleaver and Deinterleaver (described in Interleave.v and Deinterleave.v)

The design of both is very similar to each other. The interleaver coefficient is stored in a file named "int6.dat" (or int1024.dat for block length 1024 case) and this acts like a look-up table to translate a normal address to the interleaved address. The SISOs always provides the interleaver/deinterleaver with normal addresses. In behavioural RTL design, the interleaver is implemented by storing data using interleaved addresses and reading it back using normal addresses, while the deinterleaver stores data using normal addresses and reads bak using interleaved addresses. So, we need only one coefficient file here for both interleaver and deinterleaver.

The deinterleaver has a built-in hard-decision function because the hard decision is made after the final iteration of SISO2 and the sequence has to be deinterleaved back to a normal sequence. The final hard decision is basically the sign bit of the data stored in the deinterleaver. The deinterleaver has an internal counter that counts up at double the clock rate. So, the output is read out as a 1-bit sequence and finished at the middle of the iteration leaving the deinterleaver available to be written by SISO2 in the second half of the current iteration. A control signal named "write_out" is also generated during the final output sequcence. Together, they can be easily used to write to an external buffer (not implemented in this design).

4) Control (described in Control.v)

The central control block is an FSM to generate the control pulses to synchronize the operation of the input buffer, SISO1, SISO2, and hard-decision making function in the interleaver. It also generates "select" signal to control the input buffer to alternate the two blocks of input data back and forth for SISO1 and SISO2 at every iteration. Please look at figure 4 of the simulation results to see the detailed waveform of the control signals.



The design of this control block is very simple. It has an internal counter named "iter" to count the number of iterations. The operation is divided into 3 states

1) Idle: It waits for the ready signal from the input buffer to start processing the next 2 blocks of input.
2) Start: This state lasts for only 1 clock cycle and generates control pulses for SISO1, SISO2, and deinterleaver based on the current number of iteration.

Iteration 1:	only SISO1 running
Iteration 2 – 20 :	SISO1 and SISO2 running, last iteration processing for 1st block.
Iteration 21:	only SISO2 running, last iteration procesing for 2nd block, the final decision of 1st block generated from deinterleaver
Iteration 22:	the final decision of 2nd block generated from deinterleaver

After generating the control pulses for iteration 22 , the controller goes immediately to Idle state without having to wait for this iteration to complete. Only the interleaver is active in iteration 22, so SISO1 can start processing the new set of input data during this period. without interfering with the old results. In summary the time to process two blocks of data is equal to 21 iterations plus two extra clock cycle to let the controller go into interation 21 and back to idle state.

Note on clock signals

There are two clock signals in this design: "clk_in" is the input sampling clock and "clk" is the processing clock. To be able to process the input data stream in real-time, the procession clock rate must be high enough relative to the input clock. That is

$$\text{Time to process 2 blocks of data} \leq \text{Time to receive new 2 blocks of data}$$

$$\text{21 iteration time} + \text{2 extra cycles} \leq \text{Time to receive new 2 blocks of data}$$

$$(21(N + 2 + pl) + 2)\, T_{clk} \leq 4(N + 2)\, T_{clk\_in}$$

$$\frac{f_{clk}}{f_{clk\_in}} = \frac{T_{clk\_in}}{T_{clk}} \geq \frac{21(N+2+pl)+2}{2(N+2)}$$

where N is the block length (excluding tail bits)

pl is the pipeline overhead cycles. Although there is no pipeline here for behavioral RTL design, pl = 1 because we lose 1 cycle due to the hand-shake between SISOs and the main controller.

T_{clk} and T_{clk_in} are clock periods

f_{clk} and f_{clk_in} are clock rates.

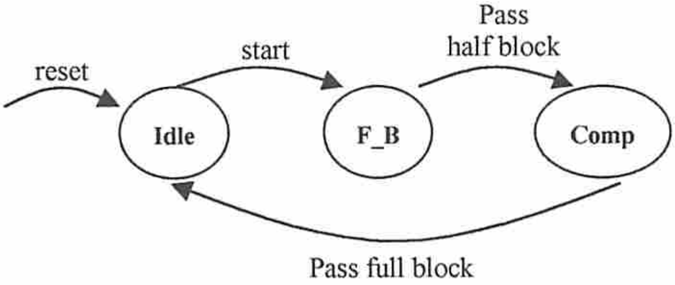
For N=6, the ratio is about 5.97 and for N=1024, the ratio is about 5.25.

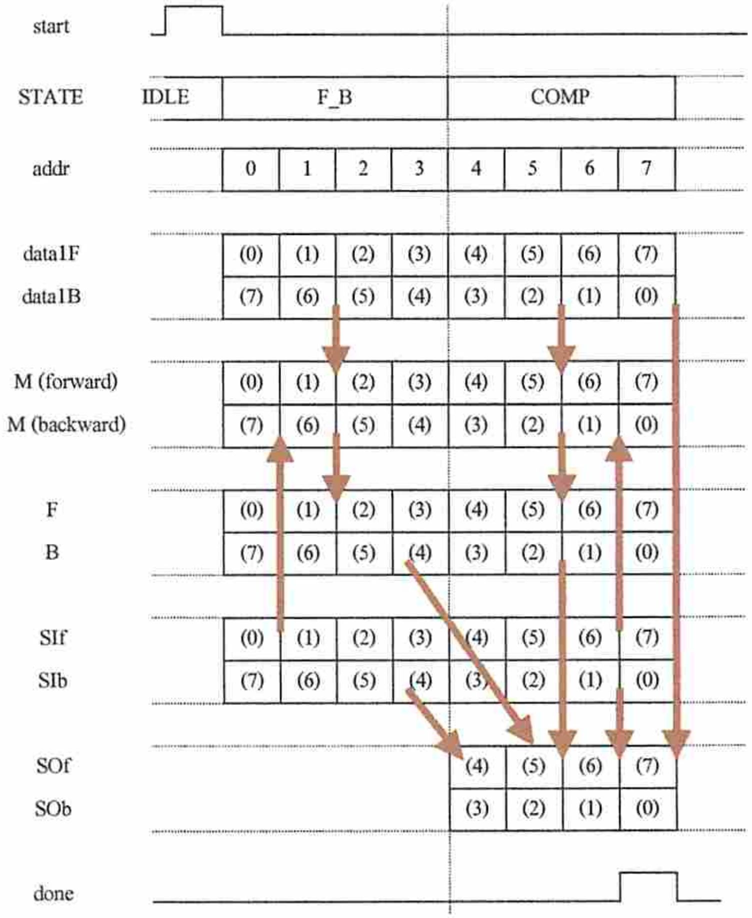
Scheduling of Computation in SISO

There is a local control block inside each SISO (described in SISO_ctrl.v) which is composed of an address counter and a state machine. The operation of the state machince is divided into 3 states.

1) Idle: It waits for a start pulse from the main controller.
2) F_B: The SISO calculates M, F, and B matrices using inputs from the input buffer and soft information input from the interleaver/deinterleaver.

3) COMP: The SISO continues to calculate M, F, and B matrices and at the same time does the completion process to produce soft information output along the way.





(i) denotes data pointed by address i.

The rough timing diagram above shows the computation scheduling within one SISO iteration assuming the block length (including tail bits) is 8. The red bold arrows in the diagram show the flow of data meaning what calculation takes what input. Vertical arrows mean that data dependency happens within the same state and computation can be implemented purely in combinational logic or with sample-level pipelining to increase the throughput. Oblique arrows mean that data dependency happens across the state and need internal buffers to store the data between states. I will use LIFO (Last In First Out) buffers for this purpose.

In state F_B, the SISO only produces Fs and Bs to be used in the next state. In state COMP, the operation gets very complicated and needs further explanation. There are soft information outputs to be calculated in the forward and backward operation, which take a lot of inputs. For example, the forward completion process will produce SOf for time k=4, 5, 6, 7 and take the following inputs:

- Soft information inputs stored in LIFO from state F_B (backward path)
- Bs stored in LIFO from state F_B (backward path)
- Ms being recalculated in this state (forward path)
- Fs being calculated in this state (forward path)

There are some interesting design decisions having been made here.

1) Ms are calculated twice, once in state F_B and again in state COMP. We could have used Ms calculated in the forward path of state F_B in the backward path of state COMP but this will require a large buffer. Given that Ms are very easy to calculate and there is no resource conflict, the cost of recalculating Ms should be less than storing them.

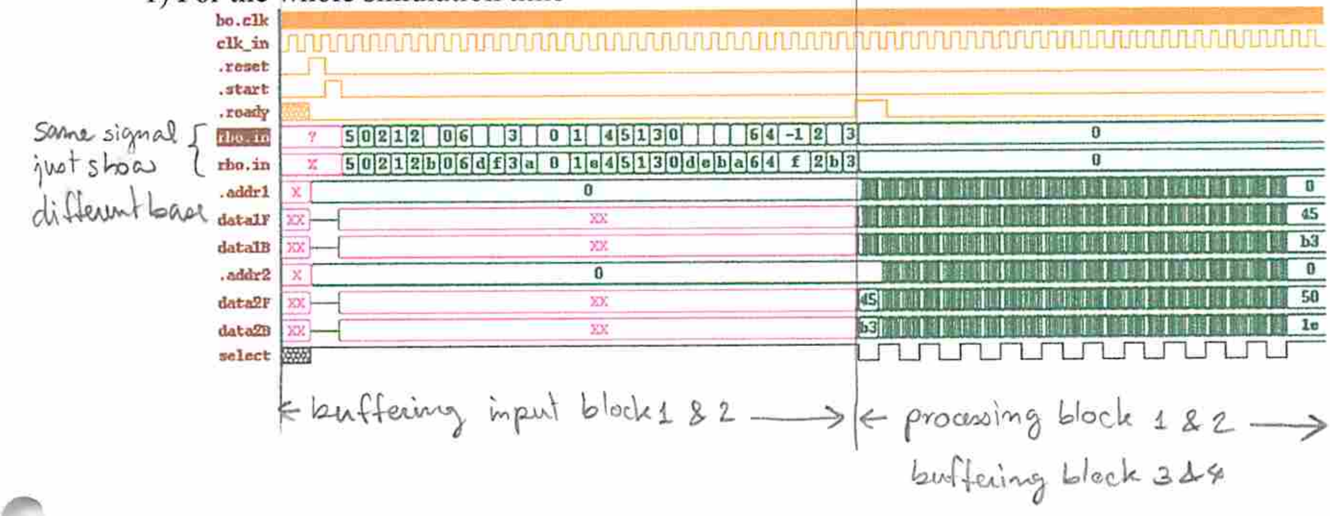
2) Soft information inputs are stored in state F_B to be used in state COMP so that the interleaver/deinterleaver are read only in state F_B and written in state COMP. This makes the design of the interleaver/deinterleaver simpler and with less storage required but comes with a cost of soft information LIFO buffers inside SISOs.

Simulation Results of Behavioral RTL Design

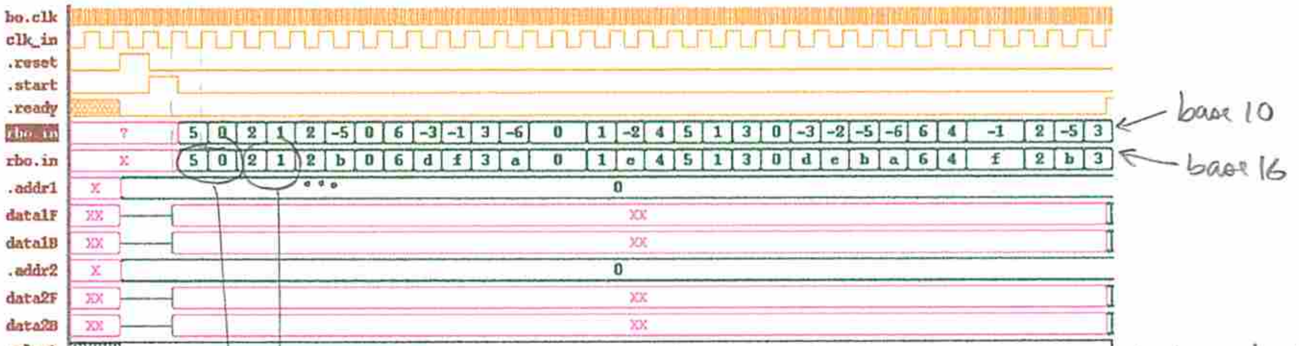
The results in 1) to 10) are from the case where the block length is equal to 6. Two blocks of input data are generated from MATLAB in section 7 and are put a file named "vec61.dat" (system bit information) and "vec62.dat" (parity bit information). The input files are read by the test bench file "Testturbo.v" to be used in the simulation.

At the input buffer

1) For the whole simulation time

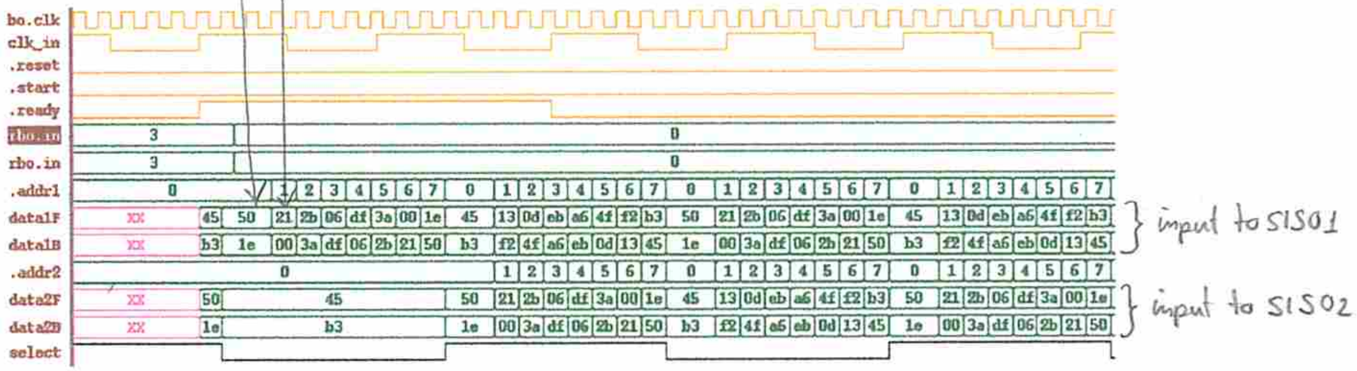


2) During the first half of the simulation while the input buffer stores the first two blocks of data.



System bit and parity bit information are combined during storing and read out together as 8-bit information

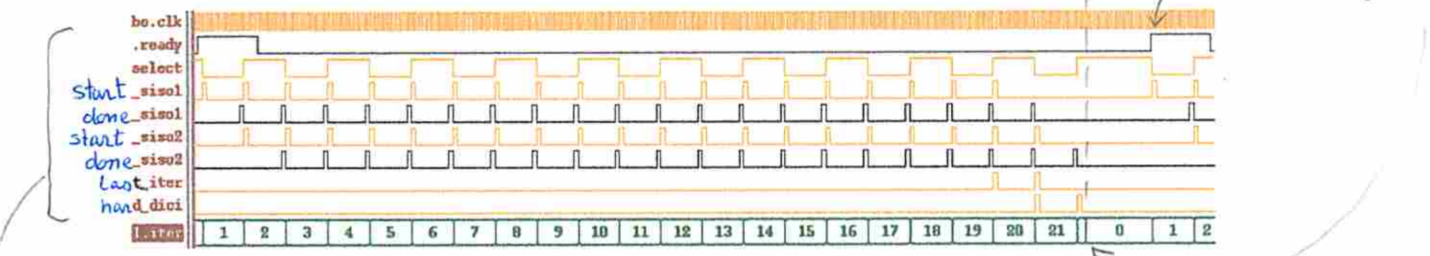
3) During the start of the second half of the simulation while the buffered data is read out to processing



All the next simulation results are grapped from the second half of the simulation, the processing period.

4) The control signals from the control block during the entire processing period

Control signal from Input buff to start processing next 2 bl.

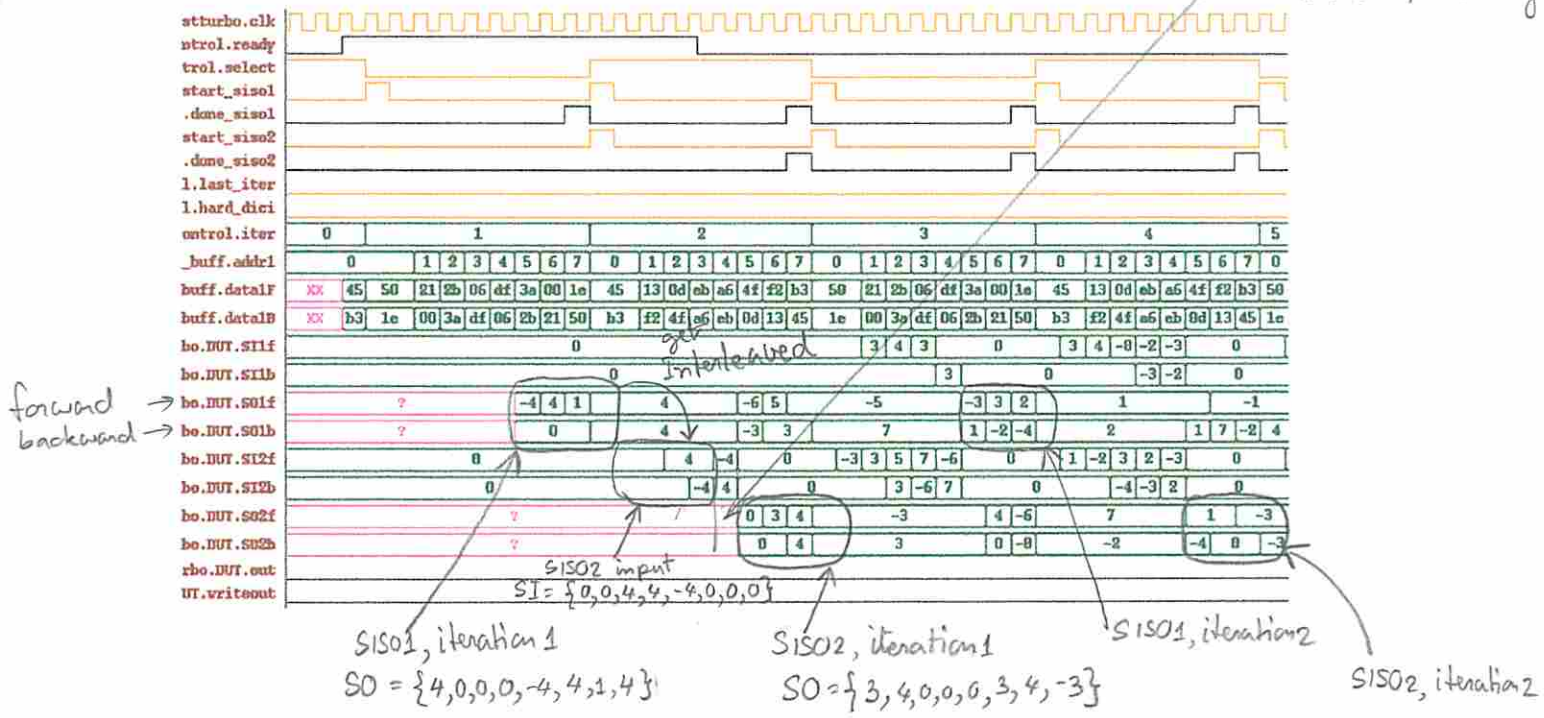


Can start as soon as here. (the last iter. at SISO2 is done)

black are status signal from SISOs → Control.

Yellow are control signal from Control → SISOs

5) The soft information input and output of SISOs during iteration 1-4



1 cycle delay here from in → out due to behav. style coding.

get interleaved

forward →

backward →

SISO2 input
SI = {0,0,4,4,-4,0,0,0}

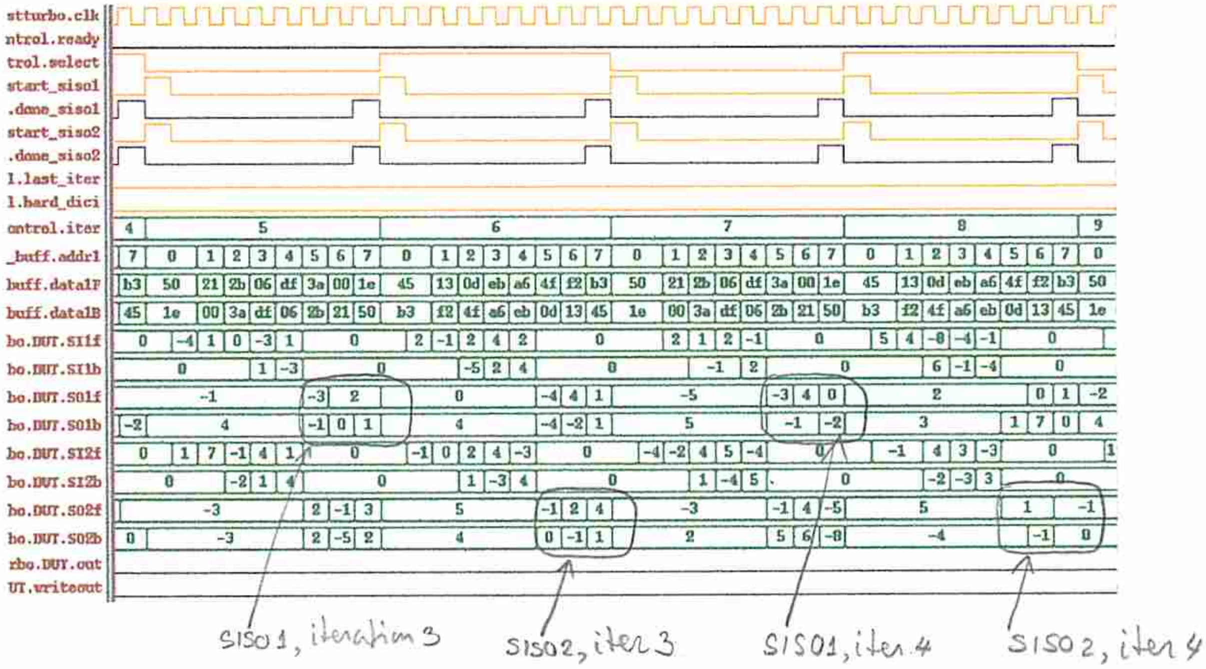
SISO1, iteration 1
SO = {4,0,0,0,-4,4,1,4}

SISO2, iteration 1
SO = {3,4,0,0,0,3,4,-3}

SISO1, iteration 2

SISO2, iteration 2

6) The soft information input and output of SISOs during iteration 5-8



SISO1, iteration 3

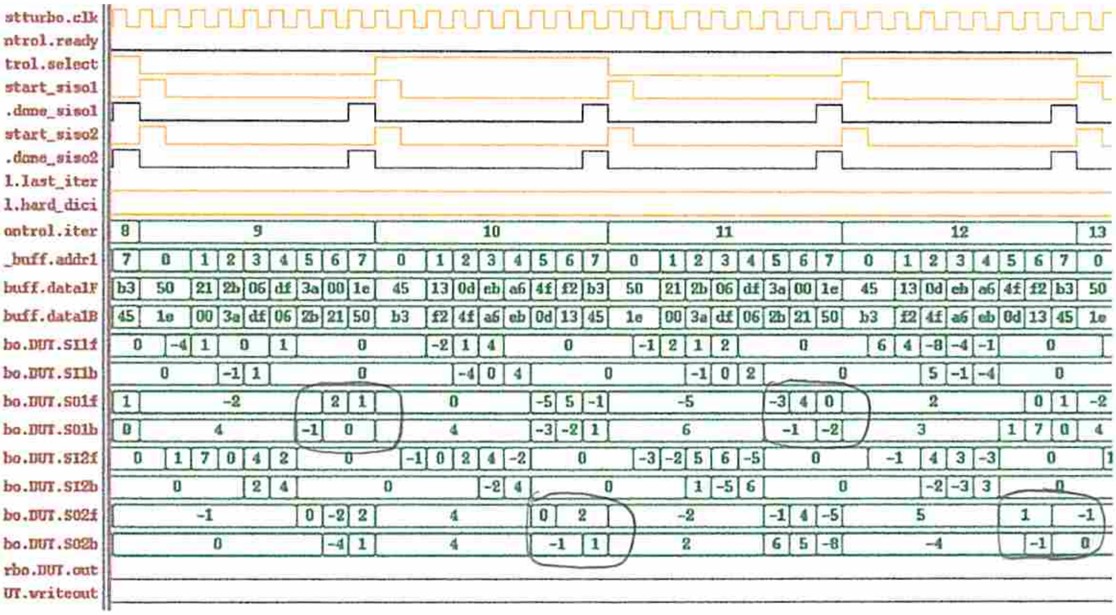
SISO2, iter 3

SISO1, iter 4

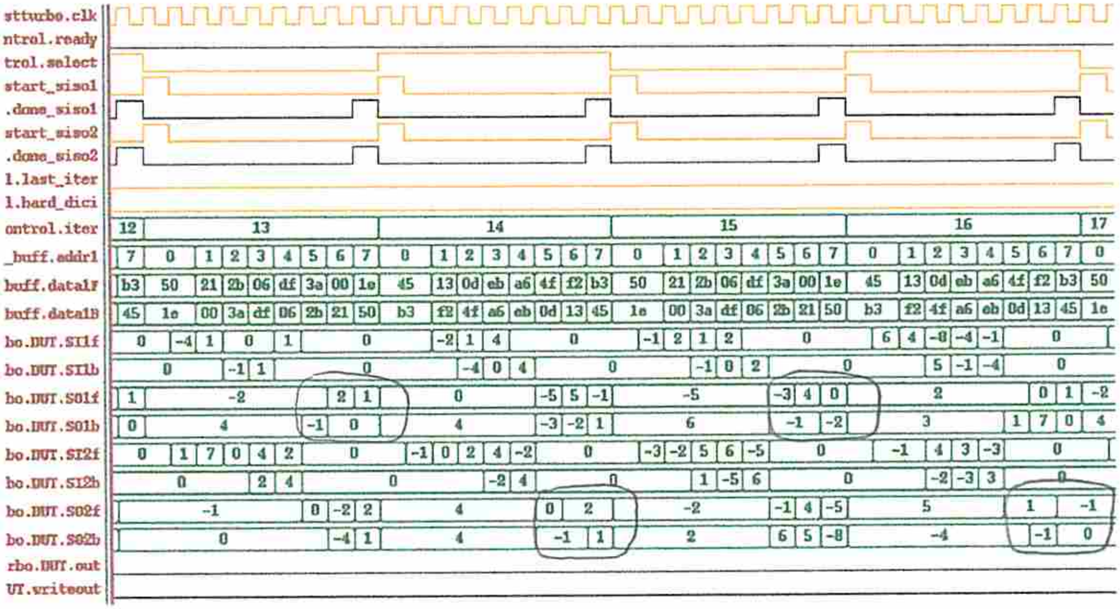
SISO2, iter 4

Only the outputs for block 1 are pointed out here.
The outputs for block 2 are just at the other SISO.
(My iteration 1 ≡ test vector's iteration 0)

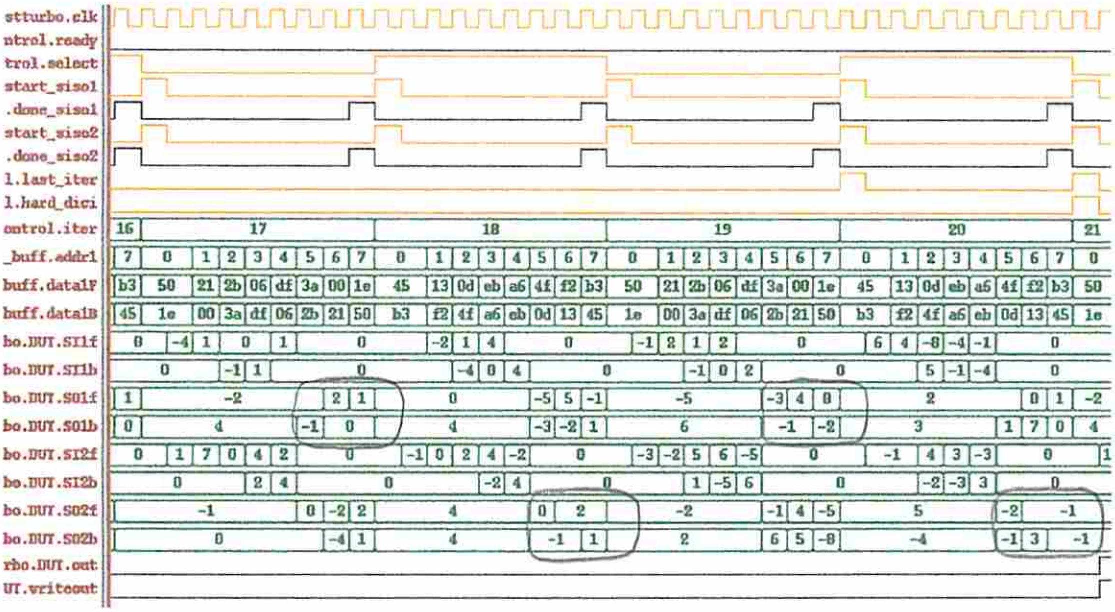
7) The soft information input and output of SISOs during iteration 9-12



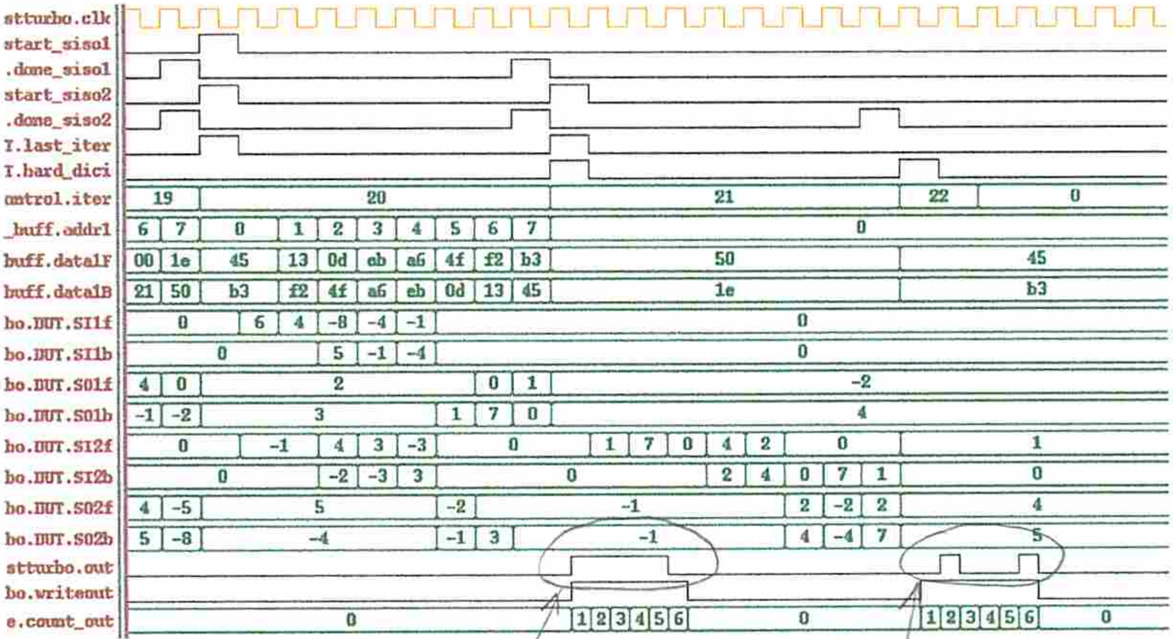
8) The soft information input and output of SISOs during iteration 13-16



9) The soft information input and output of SISOs during iteration 16-20



10) The soft information input and output of SISOs during iteration 20-22

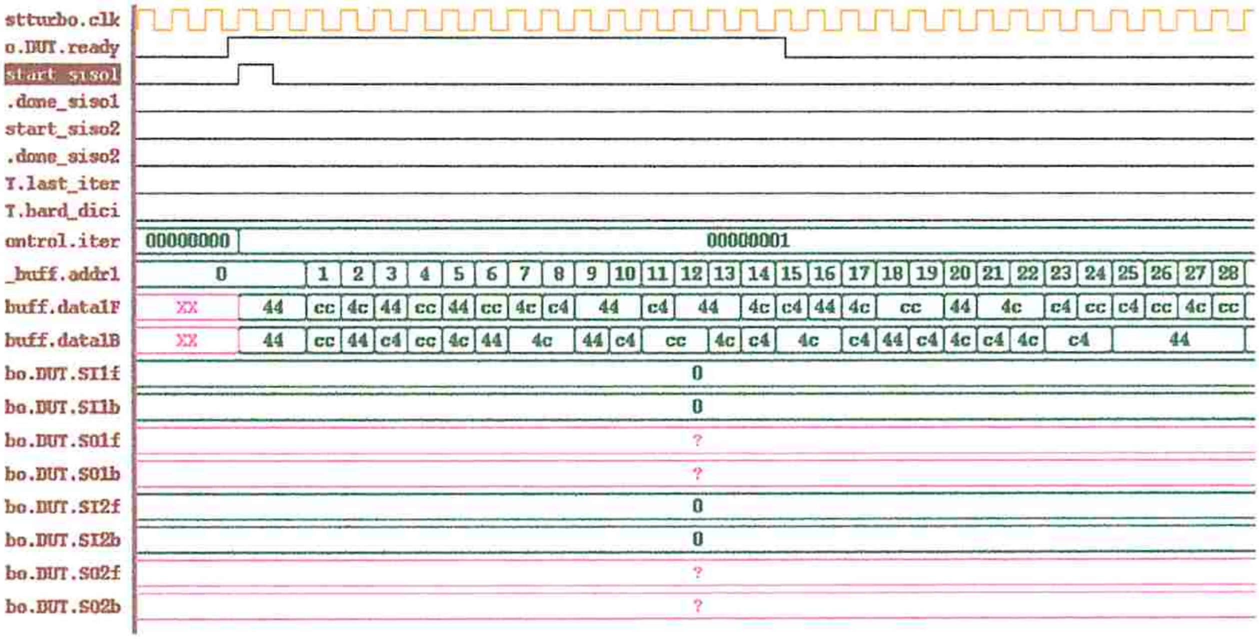


final output of block 1
= { 1, 1, 1, 1, 1, 0 }

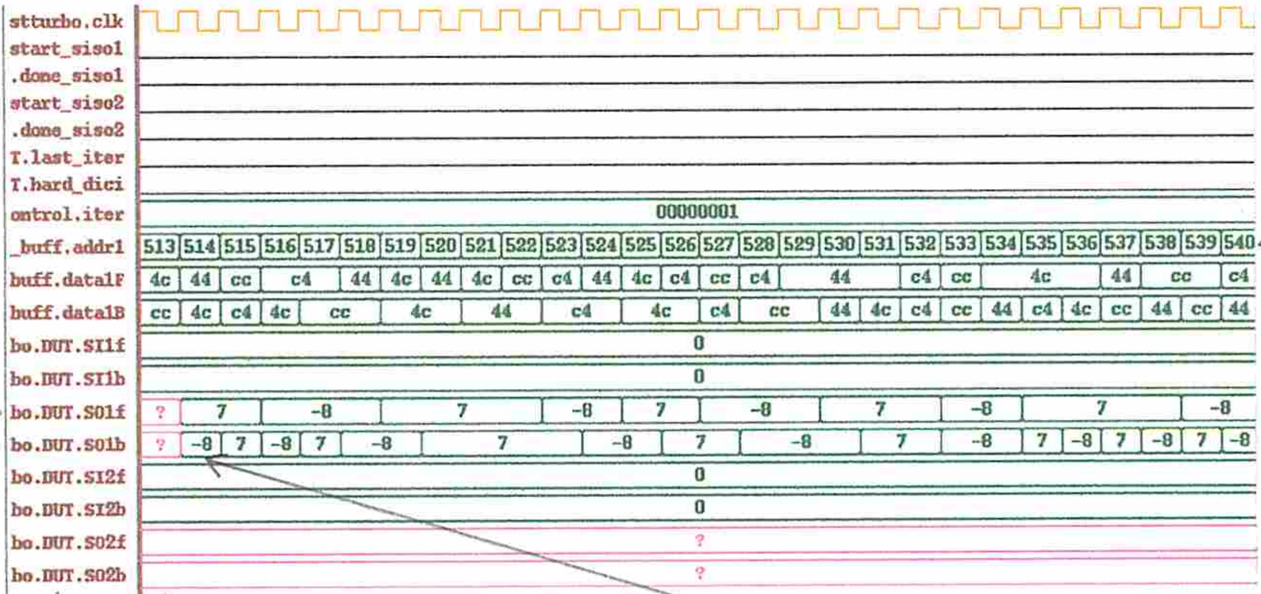
final output of block2
= { 0, 1, 0, 0, 0, 1 }

The results in 11) to 16) are from the case where the block length is equal to 1024. The first block of data is from the noisefree case and the second block is from the 1dB-noise case provided in the class web page. **

11) Block length 1024 : At the start of processing, SISO1 takes inputs from the input buffer and starts processing iteration 1.


stturbo.clk
o.DUT.ready
start_siso1
.done_siso1
start_siso2
.done_siso2
T.last_iter
T.hard_dici
ontrol.iter 00000000 00000001
_buff.addr1 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
buff.data1F XX 44 cc 4c 44 cc 44 cc 4c c4 44 c4 44 4c c4 44 4c cc 44 4c c4 cc c4 cc 4c cc
buff.data1B XX 44 cc 44 c4 cc 4c 44 4c 44 c4 cc 4c c4 4c c4 44 c4 4c c4 4c c4 44
bo.DUT.SI1f 0
bo.DUT.SI1b 0
bo.DUT.SO1f ?
bo.DUT.SO1b ?
bo.DUT.SI2f 0
bo.DUT.SI2b 0
bo.DUT.SO2f ?
bo.DUT.SO2b ?

12) Block length 1024 : At the middle of iteration 1, SISO1 starts giving out the soft information outputs


stturbo.clk
start_siso1
.done_siso1
start_siso2
.done_siso2
T.last_iter
T.hard_dici
ontrol.iter 00000001
_buff.addr1 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540
buff.data1F 4c 44 cc c4 44 4c 44 4c cc c4 44 4c c4 cc c4 44 c4 cc 4c 44 cc c4
buff.data1B cc 4c c4 4c cc 4c 44 c4 4c c4 cc 44 4c c4 cc 44 c4 4c cc 44 cc 44
bo.DUT.SI1f 0
bo.DUT.SI1b 0
bo.DUT.SO1f ? 7 -8 7 -8 7 -8 7 -8 7 -8
bo.DUT.SO1b ? -8 7 -8 7 -8 7 -8 7 -8 7 -8 7 -8 7 -8 7 -8
bo.DUT.SI2f 0
bo.DUT.SI2b 0
bo.DUT.SO2f ?
bo.DUT.SO2b ?

≡ k (for forward)

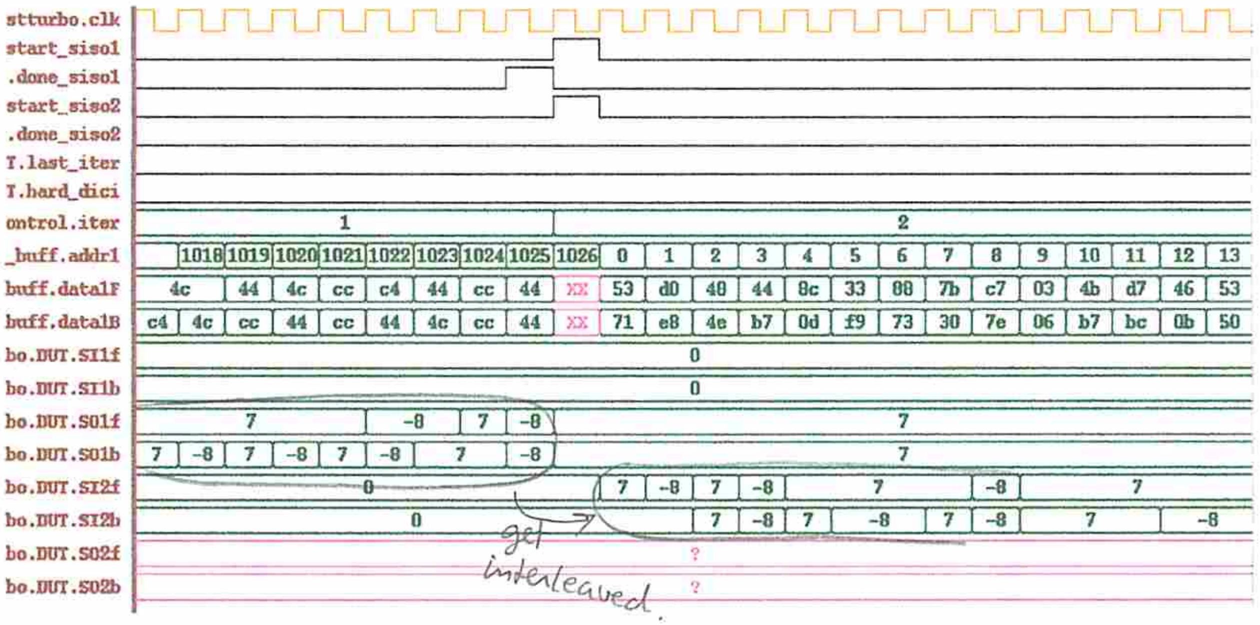
. . .

. . .

Block 1, iteration 1:
SO output of the forward calculation
start at the middle down to the end
(k = 514, 515, 516, . . . , 1025)

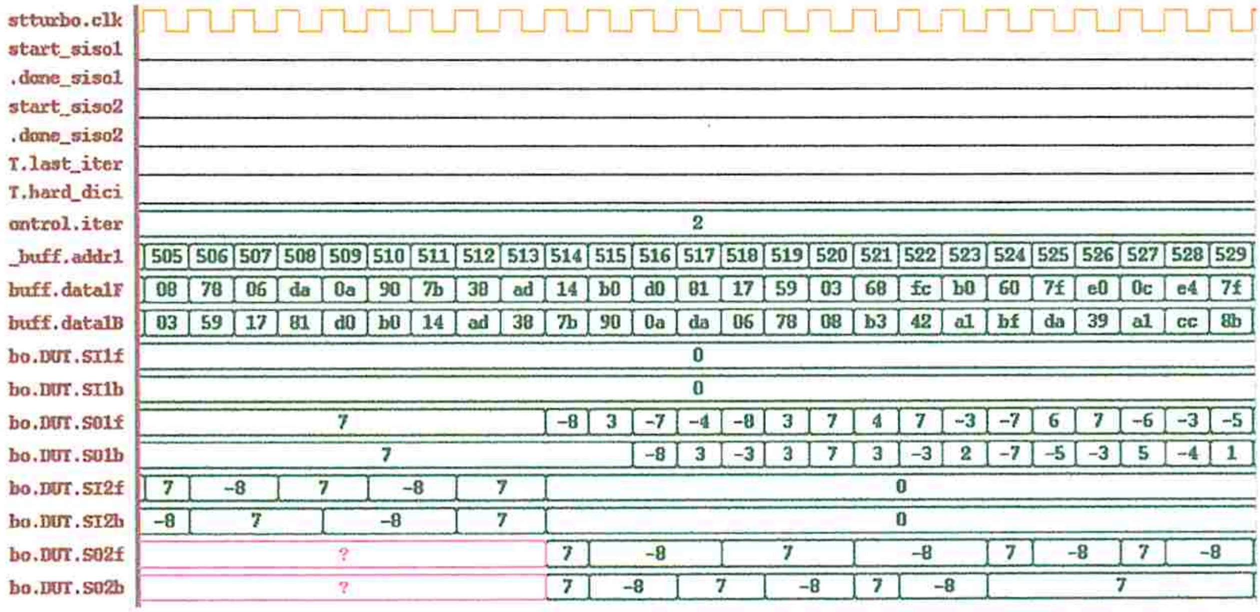
SO output of the backward calcu.
start at the middle to the beginning
(k = 513, 512, 511, . . . , 0)

13) Block length 1024 : At the end of iteration 1 and start of iteration 2. SISO1 starts working on data block 2, and SIO2 starts for block 1.


stturbo.clk
start_siso1
.done_siso1
start_siso2
.done_siso2
T.last_iter
T.hard_dici
ontrol.iter 1 2
_buff.addr1 1018 1019 1020 1021 1022 1023 1024 1025 1026 0 1 2 3 4 5 6 7 8 9 10 11 12 13
buff.data1F 4c 44 4c cc c4 44 cc 44 XX 53 d0 48 44 8c 33 88 7b c7 03 4b d7 46 53
buff.data1B c4 4c cc 44 cc 44 4c cc 44 XX 71 e8 4e b7 0d f9 73 30 7e 06 b7 bc 0b 50
bo.DUT.SI1f 0
bo.DUT.SI1b 0
bo.DUT.SO1f 7 -8 7 -8 7
bo.DUT.SO1b 7 -8 7 -8 7 -8 7 -8 7
bo.DUT.SI2f 0 7 -8 7 -8 7 -8 7
bo.DUT.SI2b 0 7 -8 7 -8 7 -8 7 -8
bo.DUT.SO2f ?
bo.DUT.SO2b ?
get interleaved.

input to SISO2

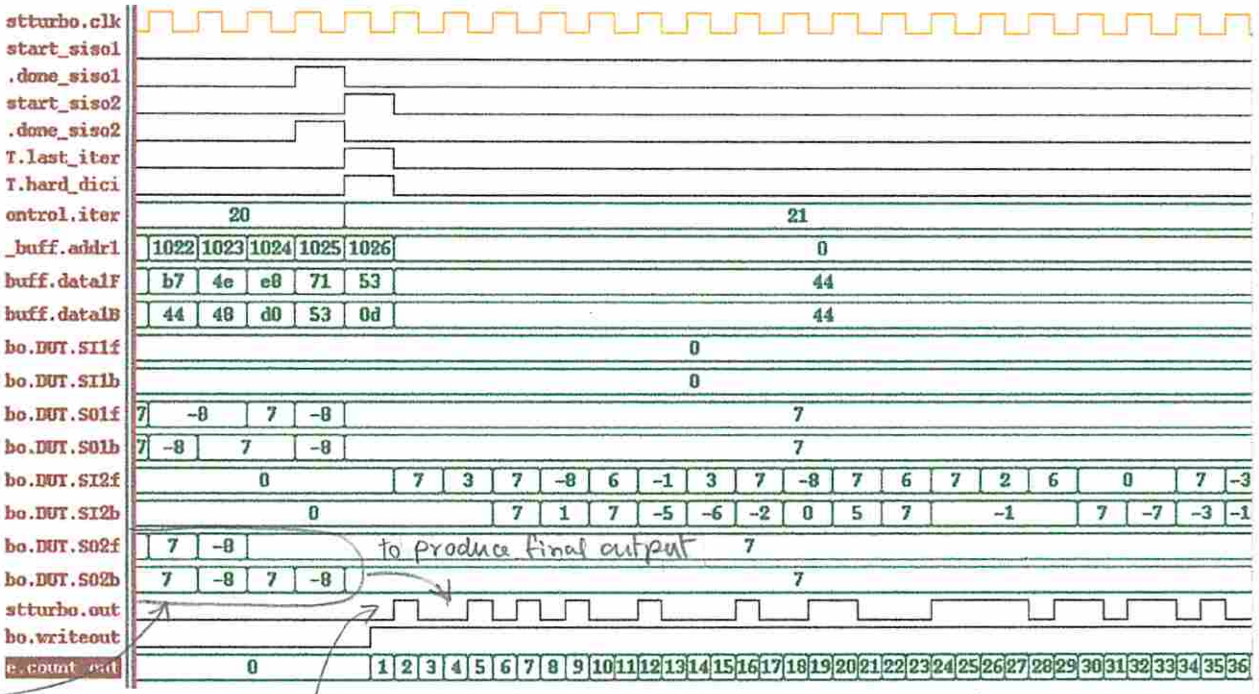
14) Block length 1024 : At the middle of iteration 2. Both SISOs start giving soft information outputs.


stturbo.clk
start_siso1
.done_siso1
start_siso2
.done_siso2
T.last_iter
T.hard_dici
ontrol.iter 2
_buff.addr1 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529
buff.data1F 08 78 06 da 0a 90 7b 38 ad 14 b0 d0 81 17 59 03 68 fc b0 60 7f e0 0c e4 7f
buff.data1B 03 59 17 81 d0 b0 14 ad 38 7b 90 0a da 06 78 08 b3 42 a1 bf da 39 a1 cc 8b
bo.DUT.SI1f 0
bo.DUT.SI1b 0
bo.DUT.SO1f 7 -8 3 -7 -4 -8 3 7 4 7 -3 -7 6 7 -6 -3 -5
bo.DUT.SO1b 7 -8 3 -3 3 7 3 -3 2 -7 -5 -3 5 -4 1
bo.DUT.SI2f 7 -8 7 -8 7 0
bo.DUT.SI2b -8 7 -8 7 0
bo.DUT.SO2f ? 7 -8 7 -8 7 -8 7 -8
bo.DUT.SO2b ? 7 -8 7 -8 7 -8 7

output of block 2, iteration 1 at SISO1

output of block1, iteration 2 at SISO 2

15) Block length 1024 : At the end of iteration 20, SISO2 has just finished processing its last iteration for data block 1 and the deinterleaver starts giving out the final output. Note that the final output is sent out at at double the clock rate.

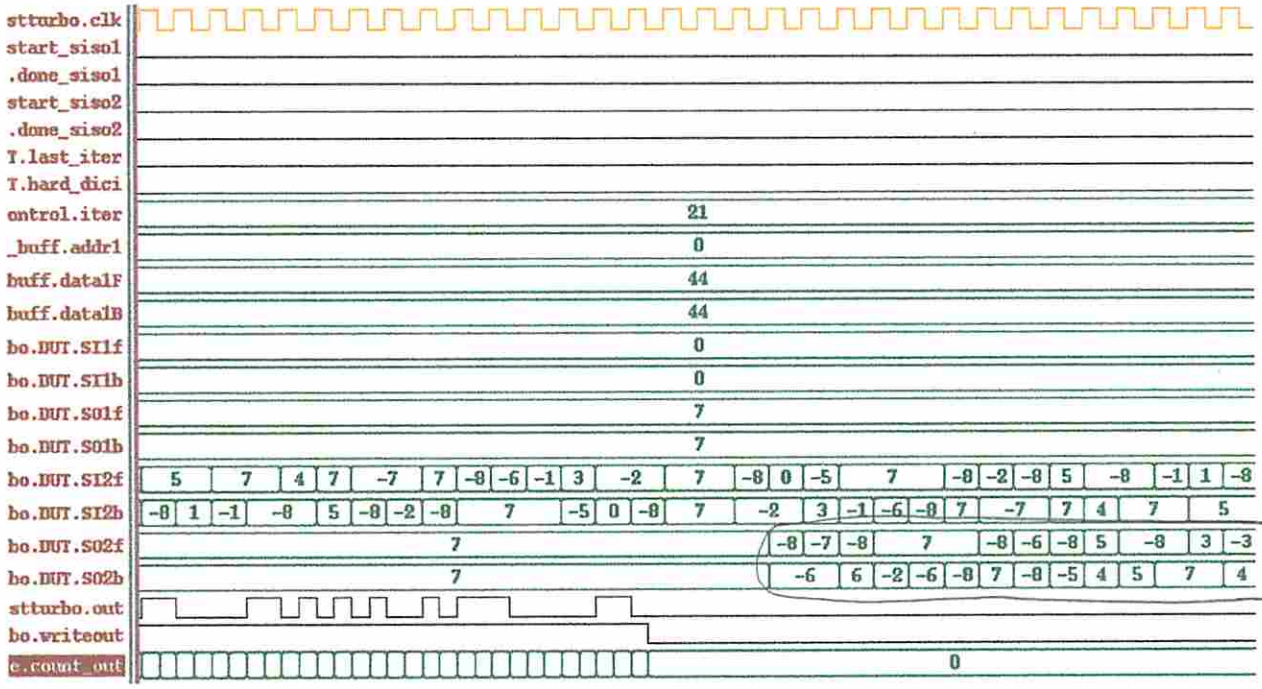


= k+1

output of block 1, iteration 10 at SISO2 (the last iter.)

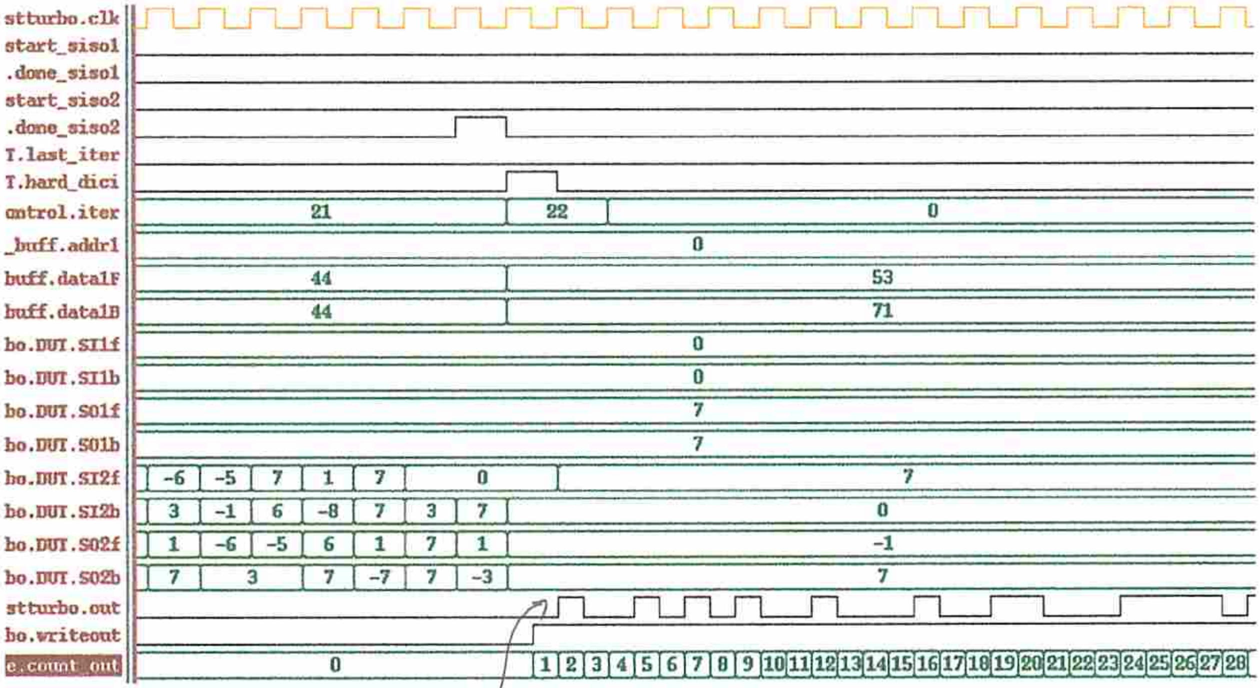
final output for block 1 (noise free case)

16) Block length 1024 : At the middle of iteration 20, the deinterleaver has just finished giving out the final output for block 1.



output of block 2, iteration 10 at SISO2

17) Block length 1024 : At the end of iteration 21, SISO2 has just finished processing its last iteration for data block 2 and the deinterleaver starts giving out the final output for block 2.



final output for block 2 (1-dB noise case)

18) Compilation report

```
Compiling source file "Testturbo.v"
Compiling included source file "Turbo.v"
Compiling included source file "SISO1.v"
Continuing compilation of source file "Turbo.v"
Compiling included source file "SISO2.v"
Continuing compilation of source file "Turbo.v"
Compiling included source file "Control.v"
Continuing compilation of source file "Turbo.v"
Compiling included source file "Input_buff.v"
Continuing compilation of source file "Turbo.v"
Compiling included source file "Interleave.v"
Continuing compilation of source file "Turbo.v"
Compiling included source file "Deinterleave.v"
Continuing compilation of source file "Turbo.v"
Continuing compilation of source file "Testturbo.v"
Highest level modules:
Testturbo

Block          0 : Start checking final decisions
Block          0 : Complete with #errors =          0.

Block          1 : Start checking final decisions
Error at time k =          71, Simulation output = 0
Error at time k =         173, Simulation output = 0
Error at time k =         427, Simulation output = 0
Error at time k =         597, Simulation output = 0
Error at time k =         643, Simulation output = 0
Error at time k =         655, Simulation output = 0
Error at time k =         706, Simulation output = 0
Error at time k =         780, Simulation output = 0
Error at time k =        1000, Simulation output = 0
```

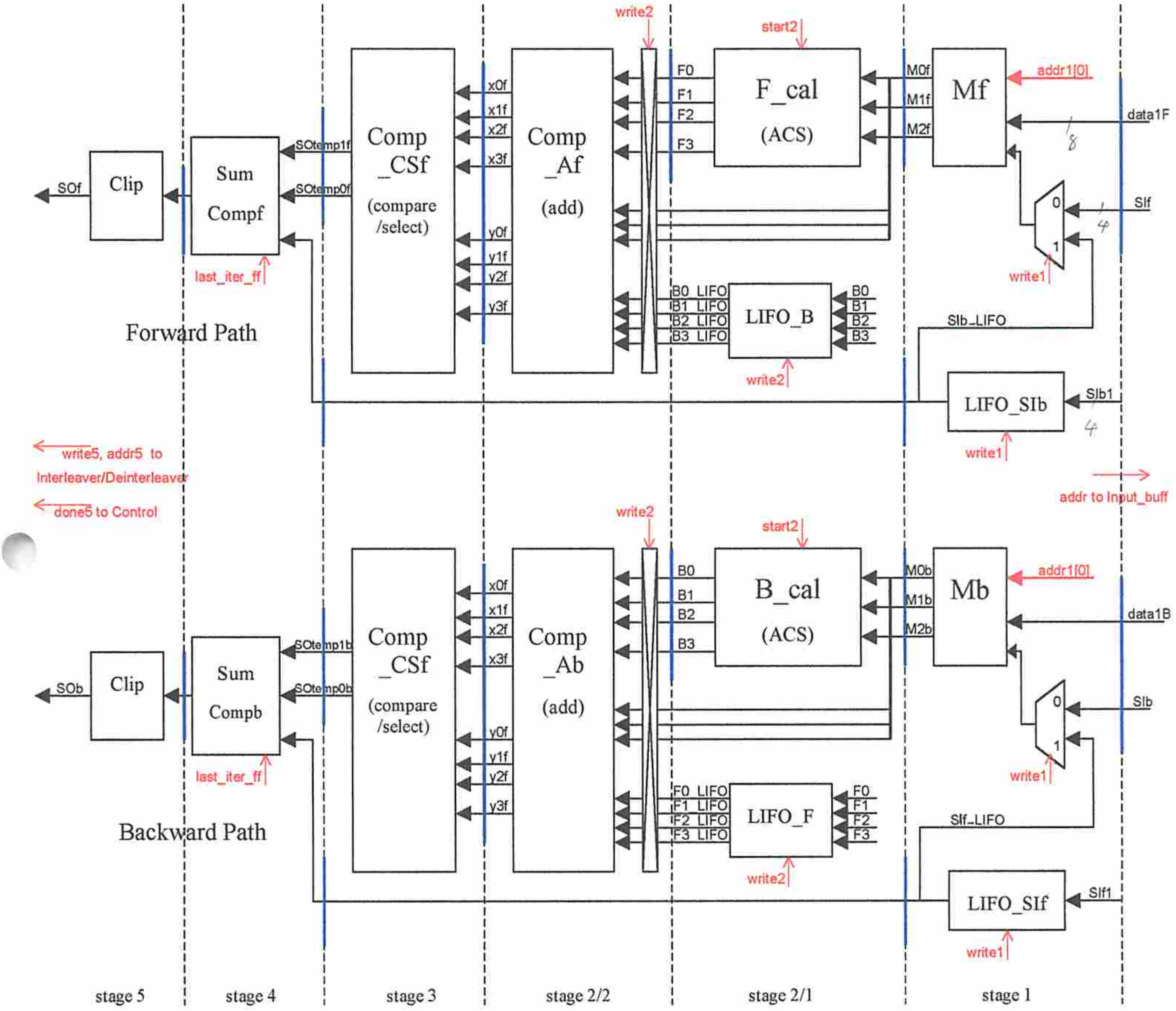
```
Block           1 : Complete with #errors =          9.

L28 "Testturbo.v": $finish at simulation time 700100
0 simulation events (use +profile or +listcounts option to count)
CPU time: 3.5 secs to compile + 0.8 secs to link + 285.2 secs in simulation
```

The are 9 samples of the simulation result of block 2 (1db-noise case) that do not match the given test vector. For all mismatched points, the simulation gives 0 while the test vector gives 1. I have done my own Matlab simulation and had the same results as this Verilog simulation, so I think it is the given test vector that is wrong. (My Matlab code uses full precision for F and B matrix calculation so this also proves that a wordlength of 7 for storing Fs and Bs is enough.)

Section 3 Structural RTL Design

SISO Data Path



(Blue vertical lines denote pipeline registers. Red denotes control signals.)

(write = 1 during state COMP)

Explanation of SISO Design

1) The SISO block is subdivided into 6 stages of calculation which will introduce a input-to-output delay of 5 clock cycles. There is no delay between stage 2/1 and 2/2 although there are registers for storing Fs and Bs. This is because we use the initial values of Fs and Bs (0, Inf, Inf, Inf) for the first step in the completion process. These initial values can be made available immediately at the same time when Ms are ready from stage 1, so we save on using pipeline registers for Ms and F_LIFO between stage 2/1 and 2/2, and also save 1 delay cycle.

The blue vertical lines denote pipeline registers which are actually implemented implicitly inside the blocks generating the output signals. (The blocks produce output in an always process sensitive only to the clock signal). The registers inside LIFO buffers are not counted as pipeline registers.

2) The ACS processing inside F_cal and B_cal is known to be unpipelinable due to the feed back of Fs and Bs inside this block. The number of pipeline stages as well as the amount of processing in each stage are designed such that the delay in each stage is approximately balanced and less than that of F_cal and B_cal block. This means the maximum clock frequncy will be limited by F_cal and B_cal. The details of each block can be found in Verilog codes in section 6 and will be discussed later in this section for estimating area and speed.

3) The local controller in SISOs is the same as the case of behavioral RTL design. The control signals either from the SISO controller (i.e. write and addr) or from the main controller (i.e. start) are delayed by going through serial registers. The delayed control signals are used in the pipelined data path with the amount of delay corresponding the the pipeline stage.

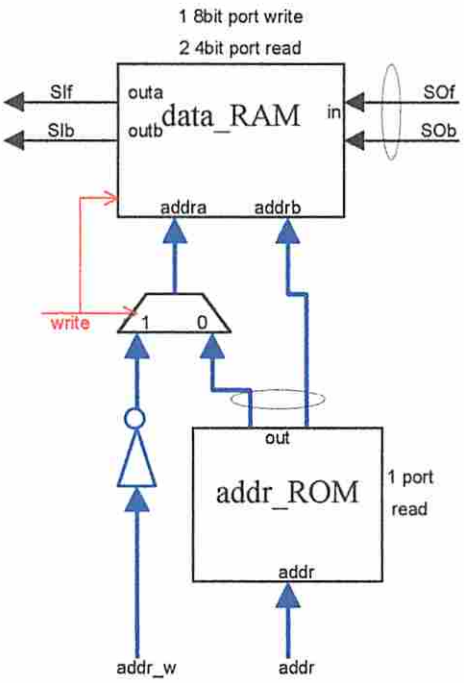
For example, "write" is generated from the local controller, passing through a register and becoming "write1" to control components in stage 1. "write1" passes another register and becomes "write2" to control components in stage 2, and so on as needed. By using pipelined control signals, we simplify the controller design at a cost of using more pipeline registers.

4) LSB of addr signals from the counter, addr[0],is used to differentiate between odd and even time steps in M calculation.

5) The thin block right before the completion process (Comp_A block) is a tri-state gate and controlled by "write2" signal. They block the signal during the first half of each iteration (F_B state), where completion calculation is not needed. They do not contribute to the calculation of output signal and do not affect the correctness of the output, but are provided just to save energy in real implementation.

Interleaver

The interleaver is implemented by using a 2-port-read RAM and an address-translation ROM. SISO1 writes to interleaver by supplying address addr_w, and data SOf and SOb. These forward and backward data (SOf and SOb) are written together through an 8-bit write port and go into two adjacent locations addressed by addra. The reading process, however, happens in an interleaved order (can be thought of as a random order) and need two address ports and two data read ports for this purpose. SISO2, who wants to read the data, supplies the interleaver with address addr and this gets translated by addr_ROM to be two addresses pointed to the needed locations in the RAM.



The trick to enable us to use only 1-port-write RAM instead of 2-port-write, which will save us 2 extra bit lines later, is how to pre-manipulate the coefficients stored in addr_ROM so that it will point to the correct locations during the read process. Let's look at an example of block length N=6 and interleaver coefficeint {3, 2, 5, 0, 4, 1}. In my design, I need to store the 2 dummy tail bits also and this make the block length virtually 8 and the coefficients {3, 2, 5, 0, 4, 1, 6, 7}. Consider the fact that a pair of data will be written at the same time in order {0,7}, {1,6}, {2,5}, and {3,4}, if I use an 8-bit write port, two data in a pair have to be combined and written to adjacent locations. Thus, the stored data will be in order {0, 7, 1, 6, 2, 5, 3, 4}.

The original coefficients {3, 2, 5, 0, 4, 1, 6, 7} is meant to be used with a normal sequence of stored data {0, 1, 2, 3, 4, 5, 6, 7}. Now that we have stored the data in a non-normal sequence, that is {0, 7, 1, 6, 2, 5, 3, 4}, we will have to derive a new set of coefficients to retrive the data.

Let D = {0, 7, 1, 6, 2, 5, 3, 4} and I_{old} = {3, 2, 5, 0, 4, 1, 6, 7}. The new coefficient, I_{new}, will be

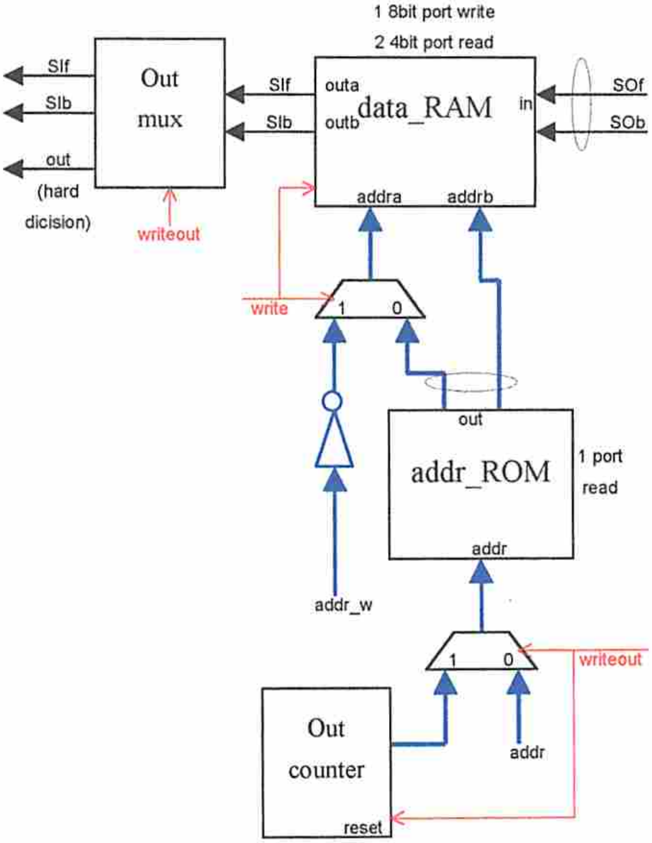
$$I_{new}(i) = k \text{ where } D(k) = I_{old}(i)$$

For example, $I_{new}(0) = 6$ because $D(6) = 3 = I_{old}(0)$. Following this equation, we will get I_{new} = {6, 4, 5, 0, 7, 2, 3, 1}.

We have avoided using a two-port write RAM by noticing the writing pattern. Similarly, we can also avoid using a two-port-read addr_ROM by noticing the fact that the soft information is always read in pairs of order {0,7}, {1,6}, {2,5}, and {3,4}. So, we can store the coefficients for k={0,7} together as a 6-bit coefficient and so on. Thus, the stored coefficients will be {(6,1), (4,3), (5,2), (0,7)} = {110001, 100011, 101010, 000111}. I have written a Matlab program, gencoeff.m in section 7, to calculate these new coefficeints for any even number of block length.

Deinterleaver

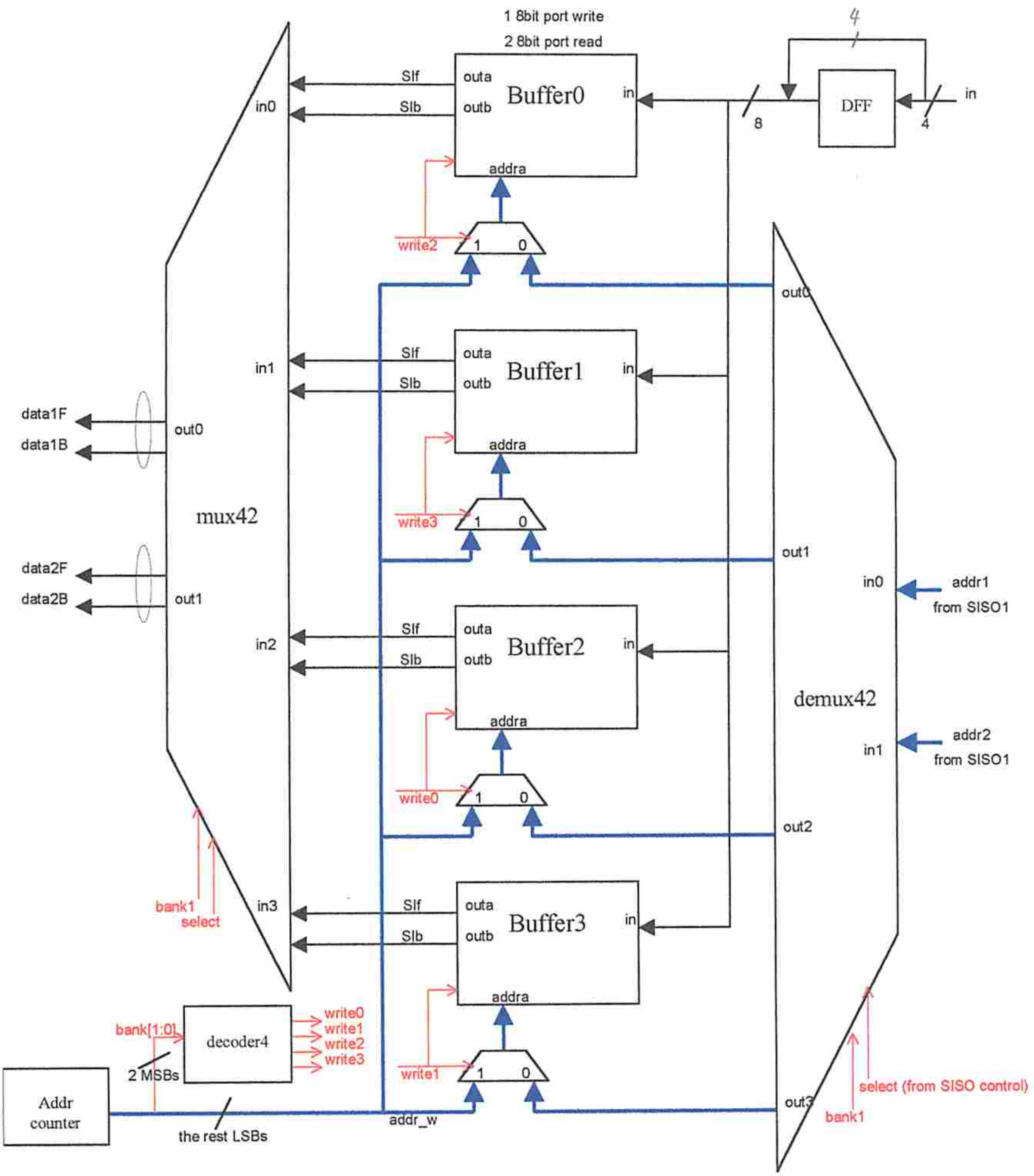
The deinterleaver is designed the same way as the interleaver except that there are some added components to accommodate reading out the final decision output. The final decision is basically the sign bit (MSB) of the stored information. We just have to read it out in a deinterleaved order. A counter is used to generate a normal sequence and a special multiplexer is used at the output to select the output. The first half of the output will come from SIf port and the later from SIb port. A special control signal named "writeout", which is active during the final output period, is generated from the control pulse, hard_dici, from the main controller.



Input Buffer

We need 4 blocks of input buffers to be able to process the signal in real-time, each block with a size of (N+2) x 8 bit. While two buffers are used to keep the incoming data, the other two will provide the old received data to SISOs. They have to switch roles at every two blocks of data. A 4-bit input register is used to combine system bit and parity bit information into an 8-bit data word.

The address counter is used to generate the write address, addr_w, and control signal, bank[1:0], to select which bank is being read or written. bank[1:0] are just the two MSBs from the counter. The mux42 and demux42 are special multiplexers used to multiplex data and address bus.


1 8bit port write
2 8bit port read
4
DFF
in
4
8
Buffer0
outa
outb
in
addra
Slf
Slb
in0
write2
1
0
out0
Buffer1
outa
outb
in
addra
Slf
Slb
in1
write3
1
0
out1
data1F
data1B
out0
mux42
data2F
data2B
out1
in0
addr1
from SISO1
Buffer2
outa
outb
in
addra
Slf
Slb
in2
demux42
write0
1
0
in1
addr2
from SISO1
out2
bank1
select
in3
Buffer3
outa
outb
in
addra
Slf
Slb
bank[1:0]
decoder4
write0
write1
write2
write3
2 MSBs
write1
1
0
Addr
counter
the rest LSBs
addr_w
out3
select (from SISO control)
bank1

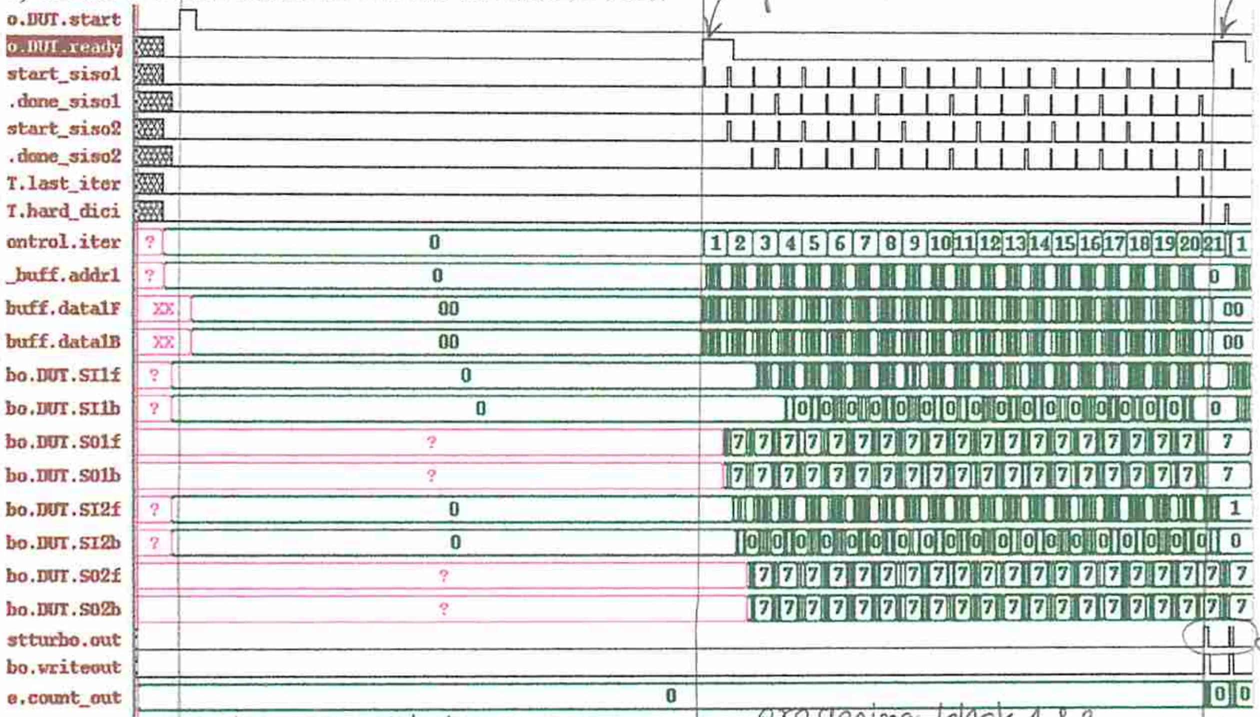
The timing of the control signals and the operations happenning are as follows:

bank1	Select	Operations
0	0	Buffer0 is read to SISO1 Buffer1 is read to SISO2 Buffer2 or Buffer3 is written, selected by bank0
0	1	Buffer0 read to SISO2 Buffer1 read to SISO1 Buffer2 or Buffer3 is written, selected by bank0
1	0	Buffer2 read to SISO1 Buffer3 read to SISO2 Buffer0 or Buffer1 is written, selected by bank0
1	1	Buffer2 read to SISO2 Buffer3 read to SISO1 Buffer0 or Buffer1 is written, selected by bank0

Simulation Results

The result are for the case where the block length is equal to 6. The results at the input buffer and main controller are the same as the behavioral RTL case (figure 1 to 4 in Behavioral RTL simulation results).

1) Simulation results for the whole simulation time.



signal to start processing block 1 & 2

signal to start processing block 3 & 4

SI, SO of SISO1

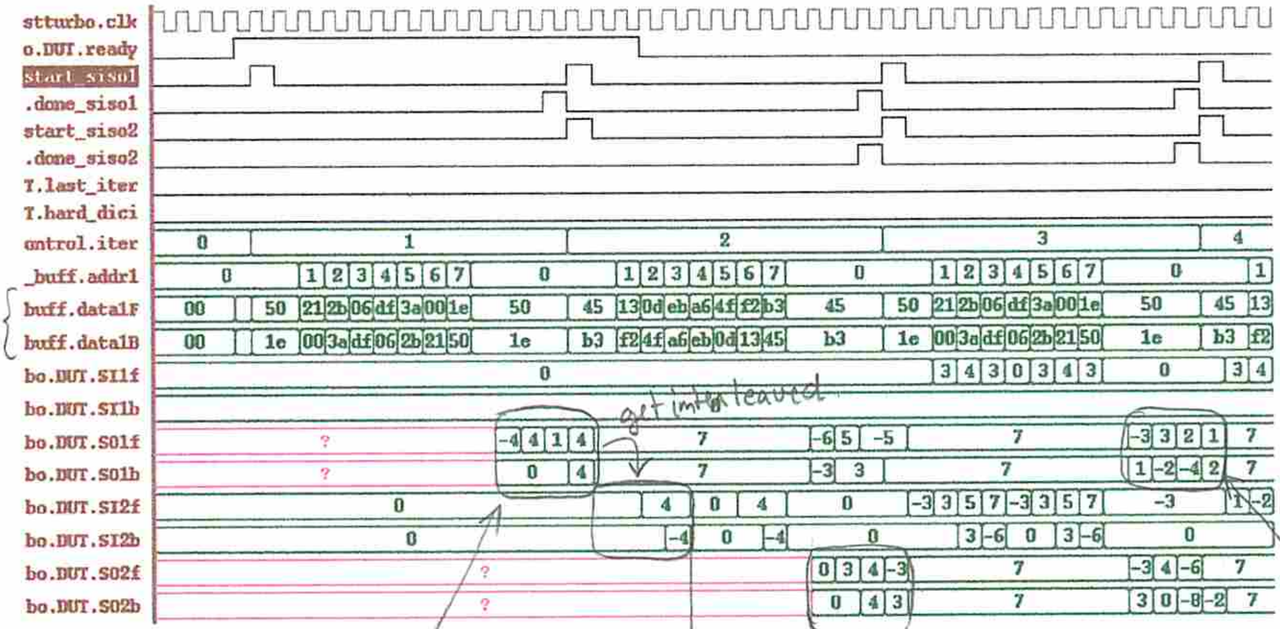
SI, SO of SISO2

← buffering block 1 & 2 →

← processing block 1 & 2 buffering block 3 & 4 →

← processing block 3&4, and so on →

2) The soft information input and output of SISOs during iteration 1-3



channel input to SISO1

Note: Only outputs for block1 are pointed out here.

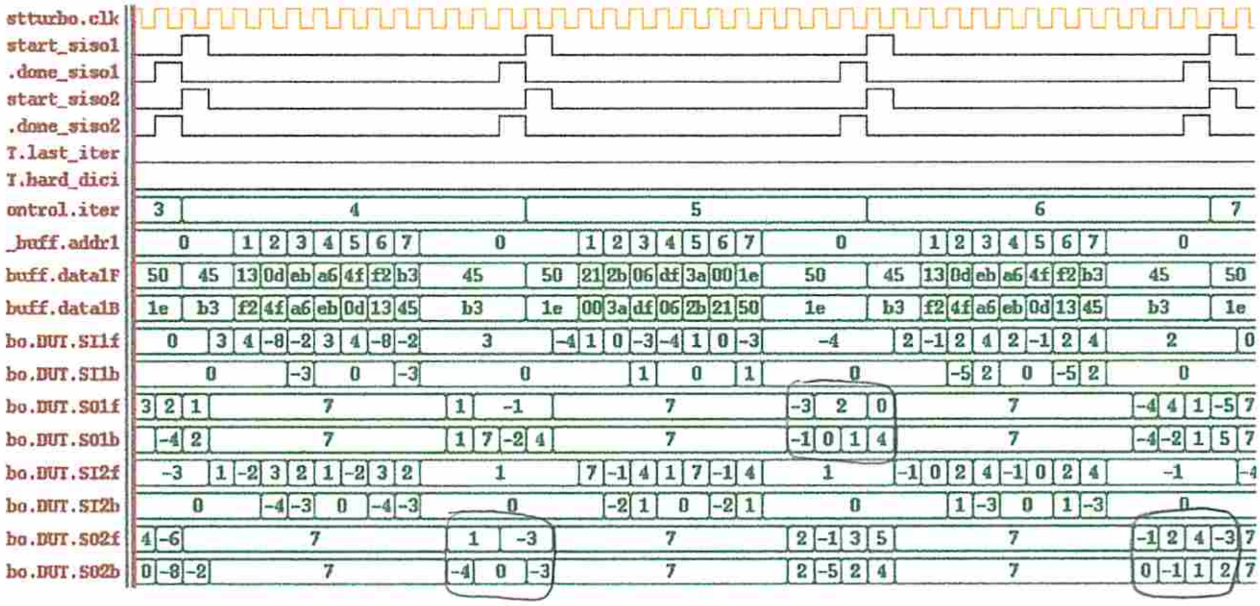
Soft information output SISO1, iteration 1 SO = {4,0,0,0,-4,4,1,4}

5 cycle delay from in → out due to pipelining

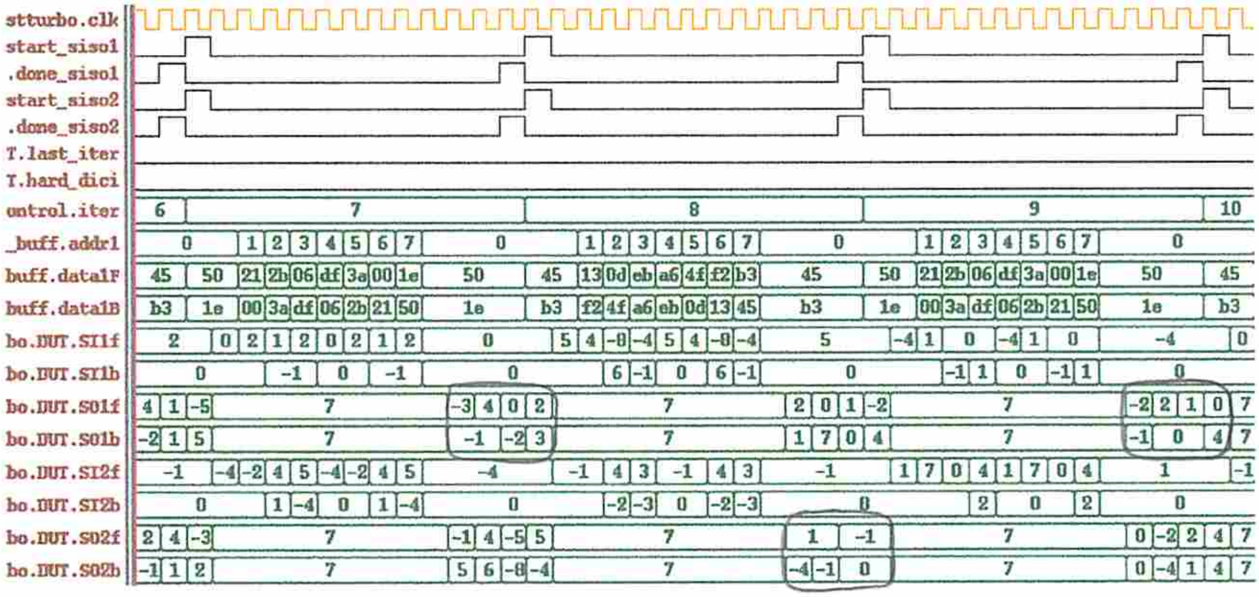
SISO2, iter1 SO = {3,4,0,0,0,3,4,-3}

SISO1, iter2 SO = {2,-4,-2,1,-3,3,2,

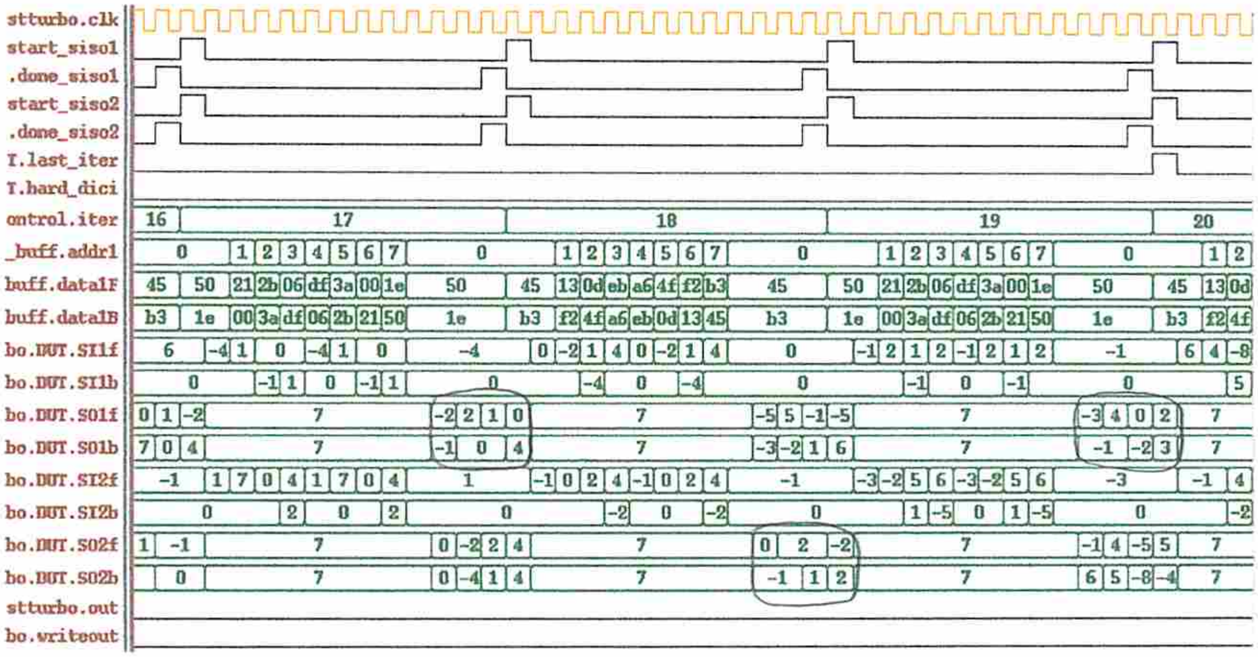
3) The soft information input and output of SISOs during iteration 4-6


stturbo.clk
start_siso1
.done_siso1
start_siso2
.done_siso2
T.last_iter
T.hard_dici
ontrol.iter 3 4 5 6 7
_buff.addr1 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0
buff.data1F 50 45 13 0d eb a6 4f f2 b3 45 50 21 2b 06 df 3a 00 1e 50 45 13 0d eb a6 4f f2 b3 45 50
buff.data1B 1e b3 f2 4f a6 eb 0d 13 45 b3 1e 00 3a df 06 2b 21 50 1e b3 f2 4f a6 eb 0d 13 45 b3 1e
bo.DUT.SI1f 0 3 4 -8 -2 3 4 -8 -2 3 -4 1 0 -3 -4 1 0 -3 -4 2 -1 2 4 2 -1 2 4 2 0
bo.DUT.SI1b 0 -3 0 -3 0 1 0 1 0 -5 2 0 -5 2 0
bo.DUT.SO1f 3 2 1 7 1 -1 7 -3 2 0 7 -4 4 1 -5 7
bo.DUT.SO1b -4 2 7 1 7 -2 4 7 -1 0 1 4 7 -4 -2 1 5 7
bo.DUT.SI2f -3 1 -2 3 2 1 -2 3 2 1 7 -1 4 1 7 -1 4 1 -1 0 2 4 -1 0 2 4 -1 -4
bo.DUT.SI2b 0 -4 -3 0 -4 -3 0 -2 1 0 -2 1 0 1 -3 0 1 -3 0
bo.DUT.SO2f 4 -6 7 1 -3 7 2 -1 3 5 7 -1 2 4 -3 7
bo.DUT.SO2b 0 -8 -2 7 -4 0 -3 7 2 -5 2 4 7 0 -1 1 2 7

4) The soft information input and output of SISOs during iteration 7-9

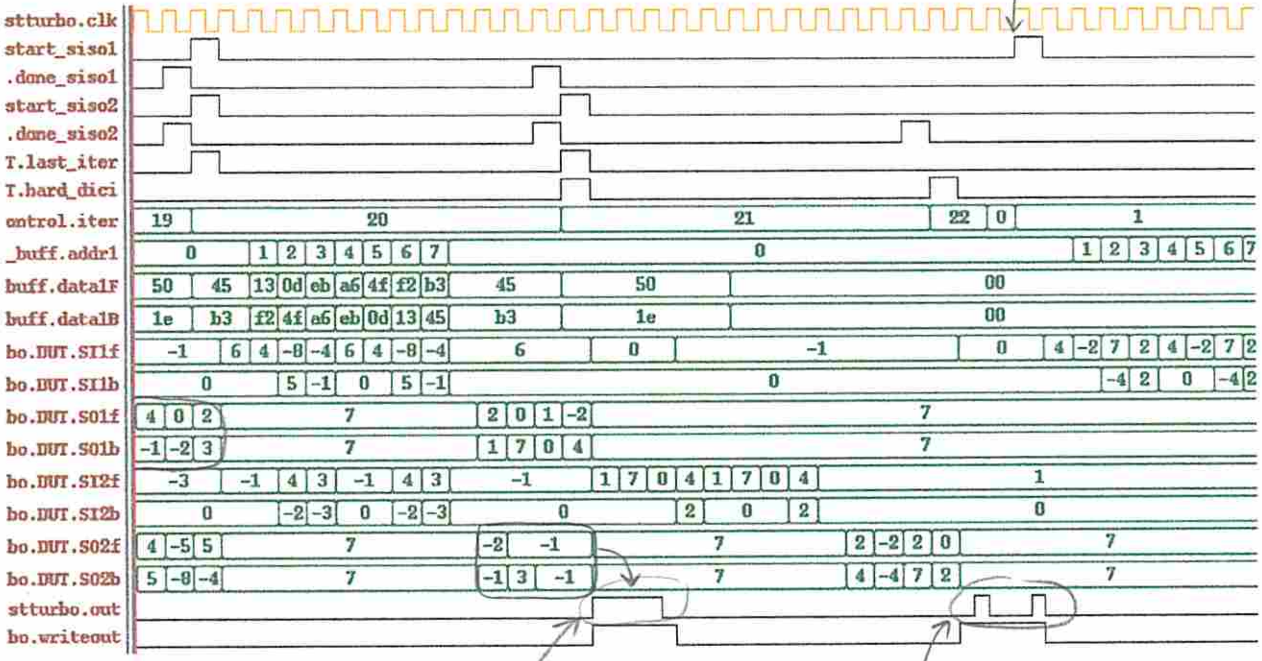

stturbo.clk
start_siso1
.done_siso1
start_siso2
.done_siso2
T.last_iter
T.hard_dici
ontrol.iter 6 7 8 9 10
_buff.addr1 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0
buff.data1F 45 50 21 2b 06 df 3a 00 1e 50 45 13 0d eb a6 4f f2 b3 45 50 21 2b 06 df 3a 00 1e 50 45
buff.data1B b3 1e 00 3a df 06 2b 21 50 1e b3 f2 4f a6 eb 0d 13 45 b3 1e 00 3a df 06 2b 21 50 1e b3
bo.DUT.SI1f 2 0 2 1 2 0 2 1 2 0 5 4 -8 -4 5 4 -8 -4 5 -4 1 0 -4 1 0 -4 0
bo.DUT.SI1b 0 -1 0 -1 0 6 -1 0 6 -1 0 -1 1 0 -1 1 0
bo.DUT.SO1f 4 1 -5 7 -3 4 0 2 7 2 0 1 -2 7 -2 2 1 0 7
bo.DUT.SO1b -2 1 5 7 -1 -2 3 7 1 7 0 4 7 -1 0 4 7
bo.DUT.SI2f -1 -4 -2 4 5 -4 -2 4 5 -4 -1 4 3 -1 4 3 -1 1 7 0 4 1 7 0 4 1 -1
bo.DUT.SI2b 0 1 -4 0 1 -4 0 -2 -3 0 -2 -3 0 2 0 2 0
bo.DUT.SO2f 2 4 -3 7 -1 4 -5 5 7 1 -1 7 0 -2 2 4 7
bo.DUT.SO2b -1 1 2 7 5 6 -8 -4 7 -4 -1 0 7 0 -4 1 4 7

5) The soft information input and output of SISOs during iteration 17-29



6) The soft information input and output of SISOs during iteration 20-22

start SISO1 to process block 3 & 4



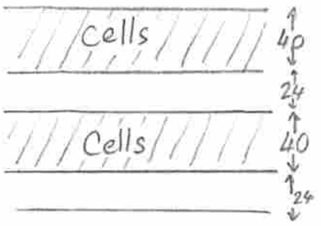
final output of block 1

final output of block 2

Section 4 Timing and Area Estimation

Layout and Color Plan

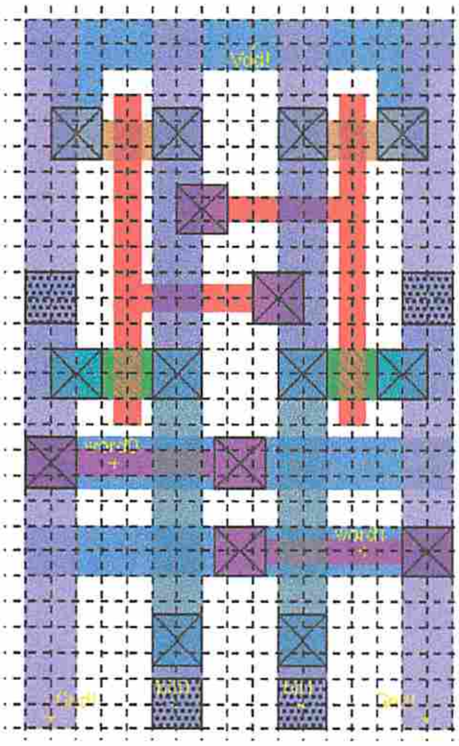
- Standard cell height = 40 λ
- Routing space between cell rows = 24 λ (three 4-bit-wide horizontal channels) or 17 λ (two 4-bit-wide horizontal channels)
- Routing within standard cells : metal1, poly, and metal2 (if really neccesary)
- Routing within blocks : metal1 for horizontal and metal2 for vertical
- Global routing : metal3, metal4

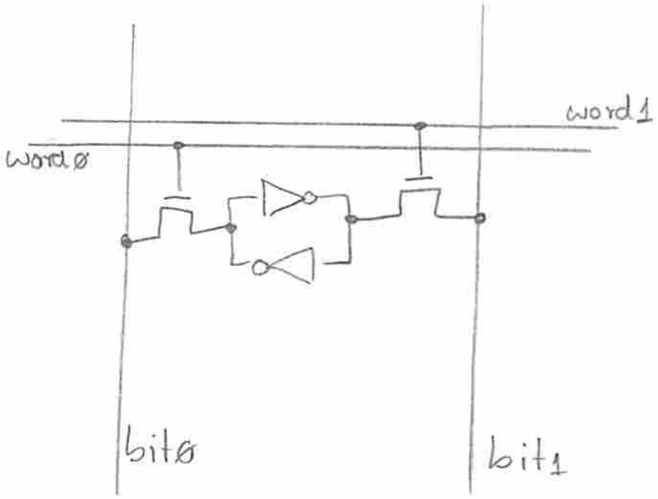


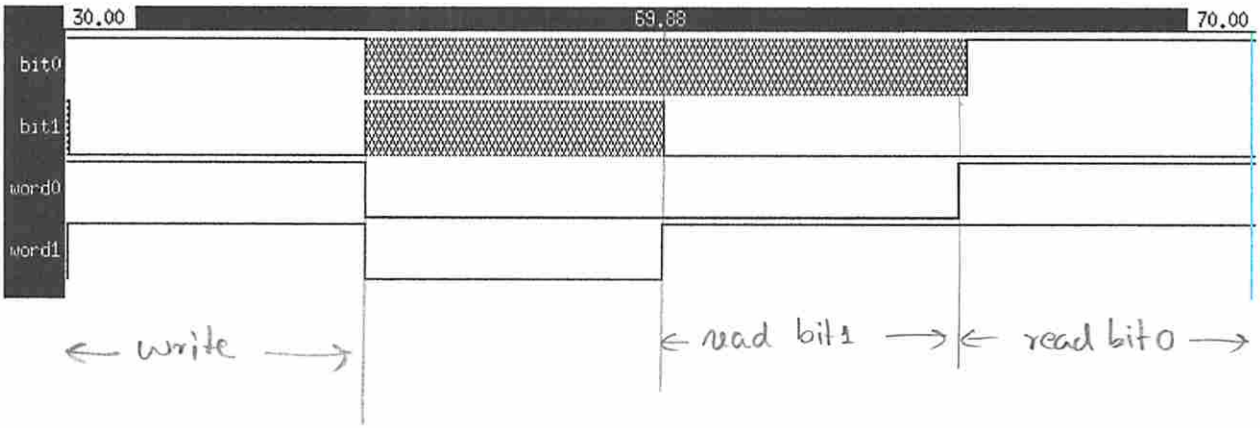
Layout of Basic Blocks (grid = 2λ)

2-port RAM 1 bit cell

size = 34 x 56
delay 0.8 ns (when output goes low.)

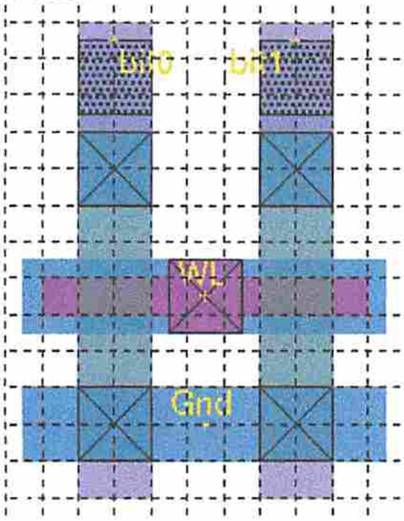


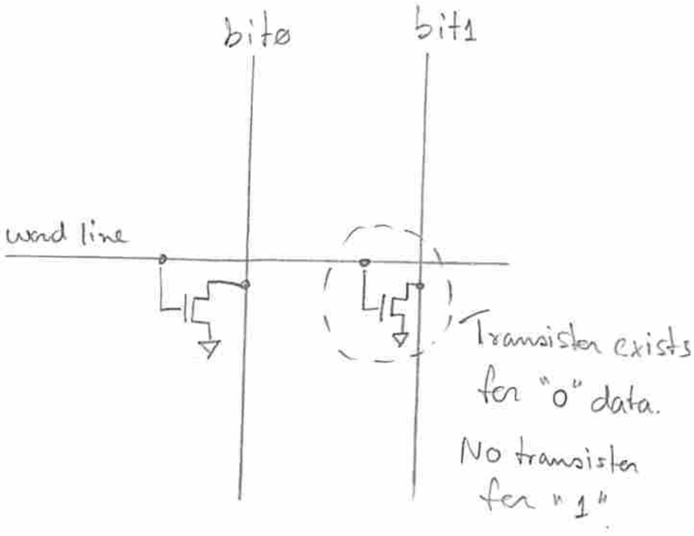




ROM 2-bit cell

size = 20 x 26





Transistor exists for "0" data.

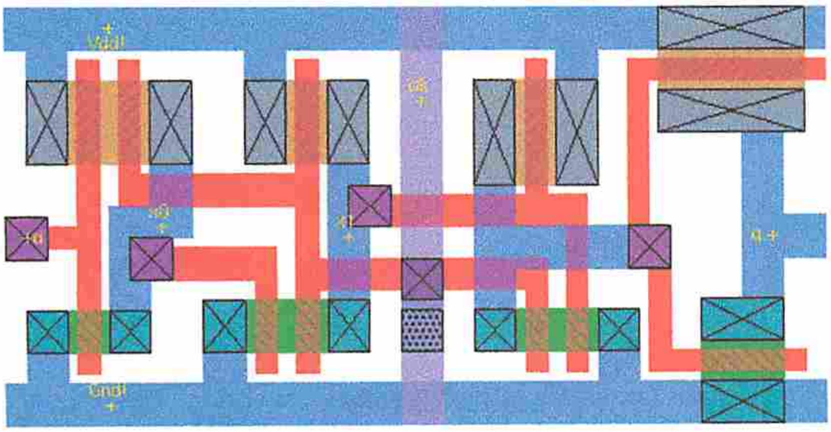
No transistor for "1".

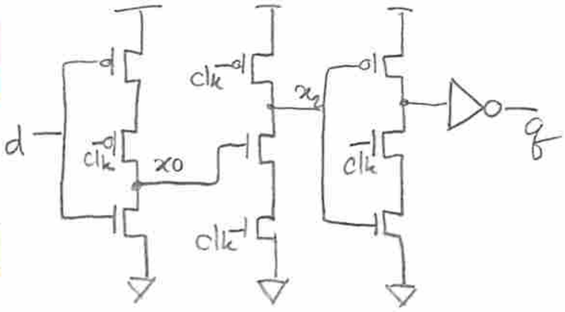
DFF

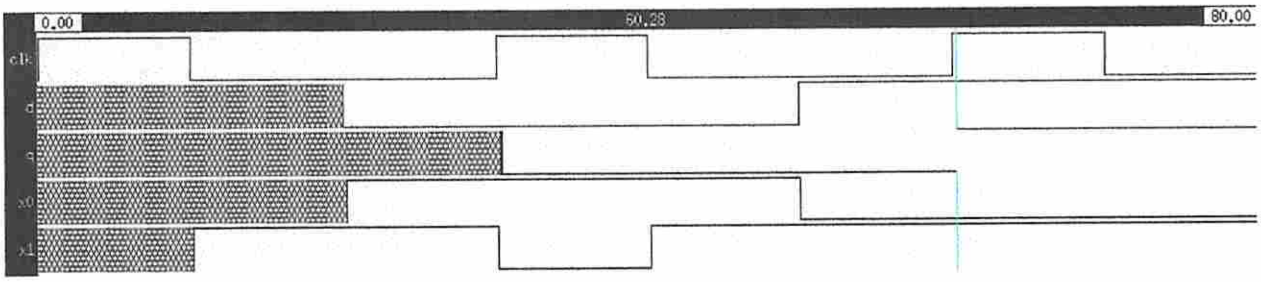
size 80 x 40

Worst-case clk-to-q delay $t_q = 0.26$ ns

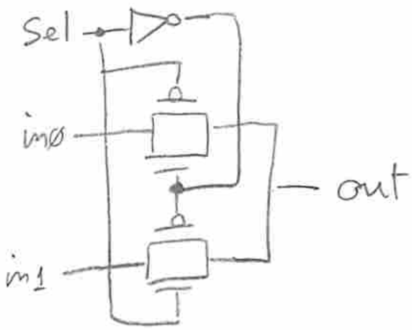
Worst-case setup time $t_s = 0.18$ ns

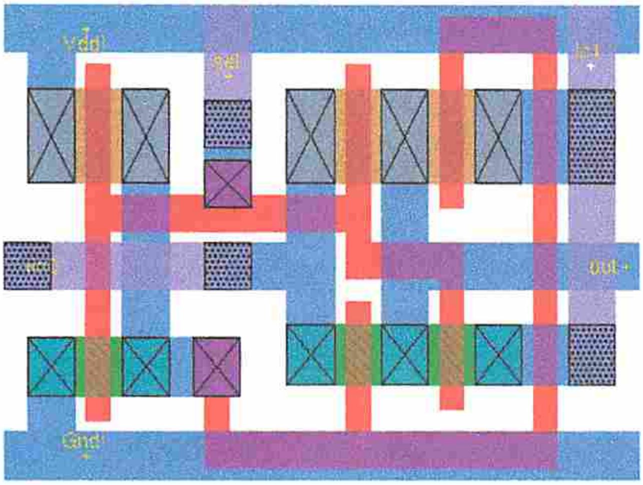


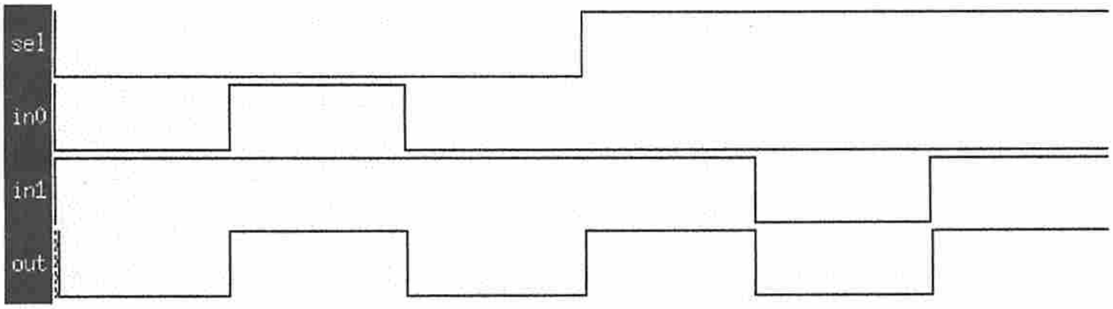




Mux2 size 54 x 40
worst-case delay tpass = 0.03 ns
sel-to-out delay = 0.25 ns

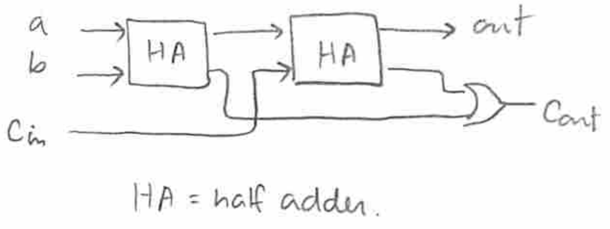


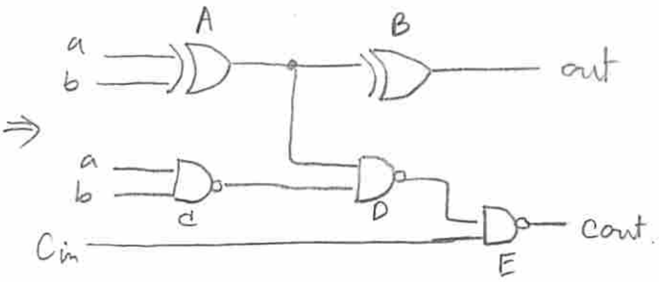




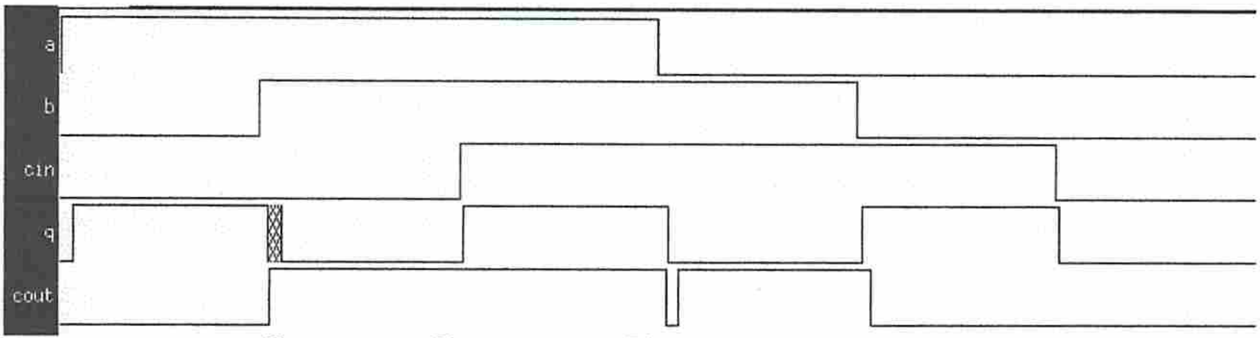
1-bit Full Adder

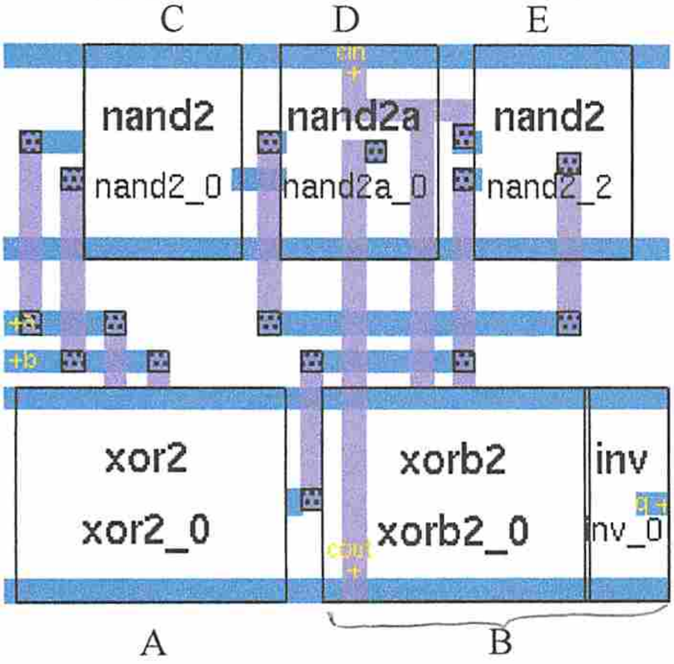
A half adder is simply an XOR gate (for sum) and an AND gate (for carry). Two half adder and an OR gate can form a 1-bit full adder as in the picture. The path for generating the carry can be transformed to use 3 2-input NAND gates. The XOR gate is built based on pass transistors, which uses only 6 transistors but has a poor driving capability. To increase its driving capability, the second XOR gate is changed to be an inverted XOR gate plus an invertor, which acts as an output buffer.





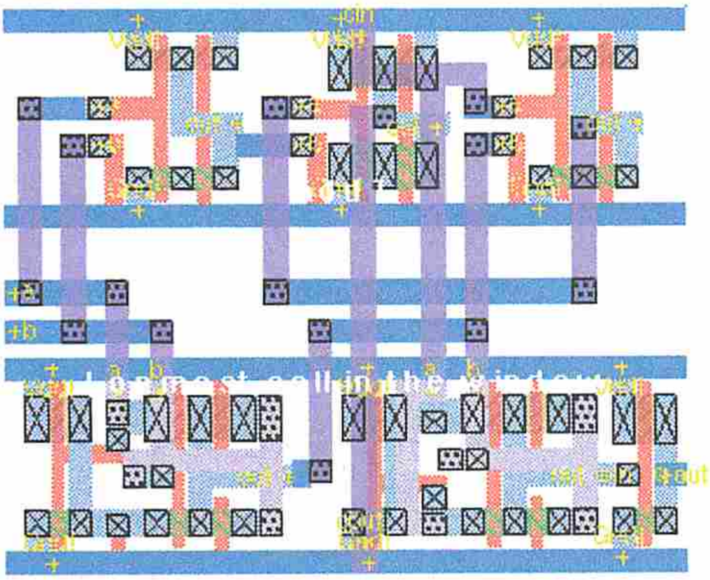
size (126 x 104)
worst case delay = 1.1 ns (approx.)

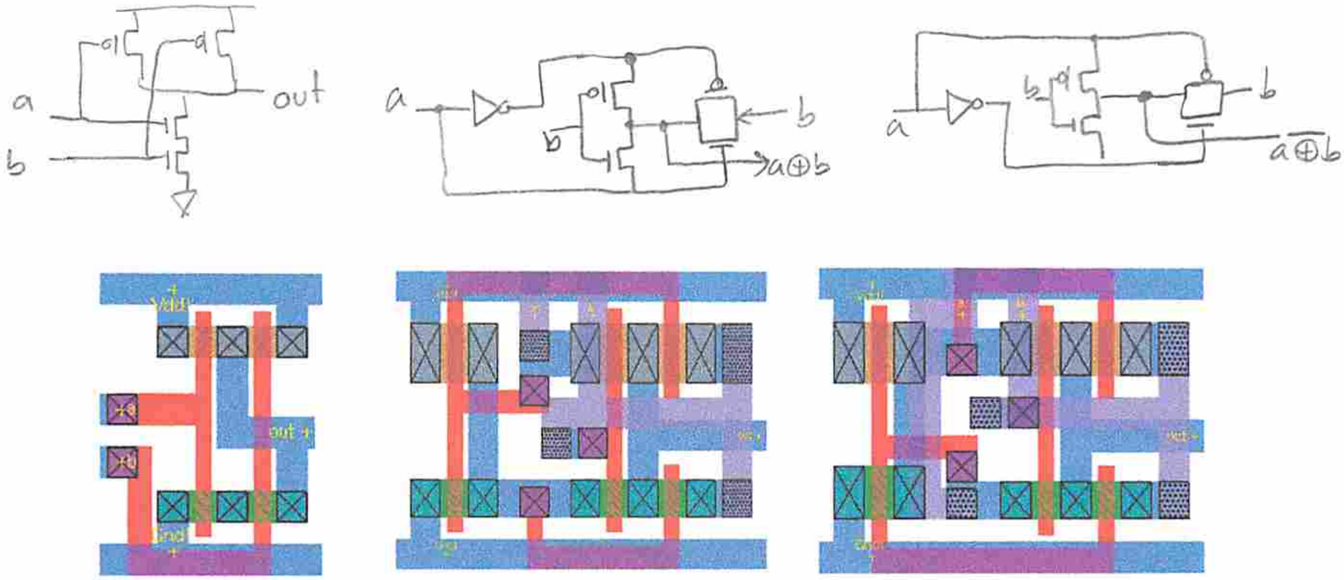




NAND E uses bigger transistors to increase its output driving capability.

XOR B is implemented with an inverted XOR + an invertor as an output buffer.

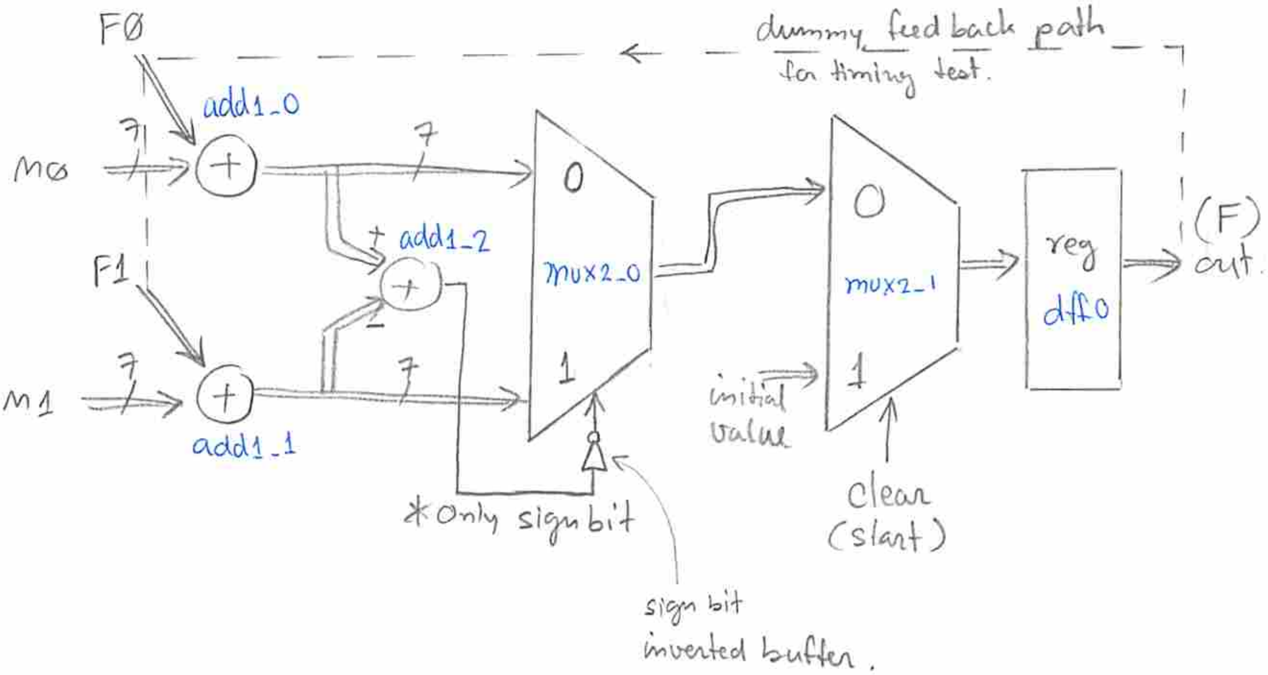




nand2 (30 x 40) xor2 (52x40) xor2b (52x40)

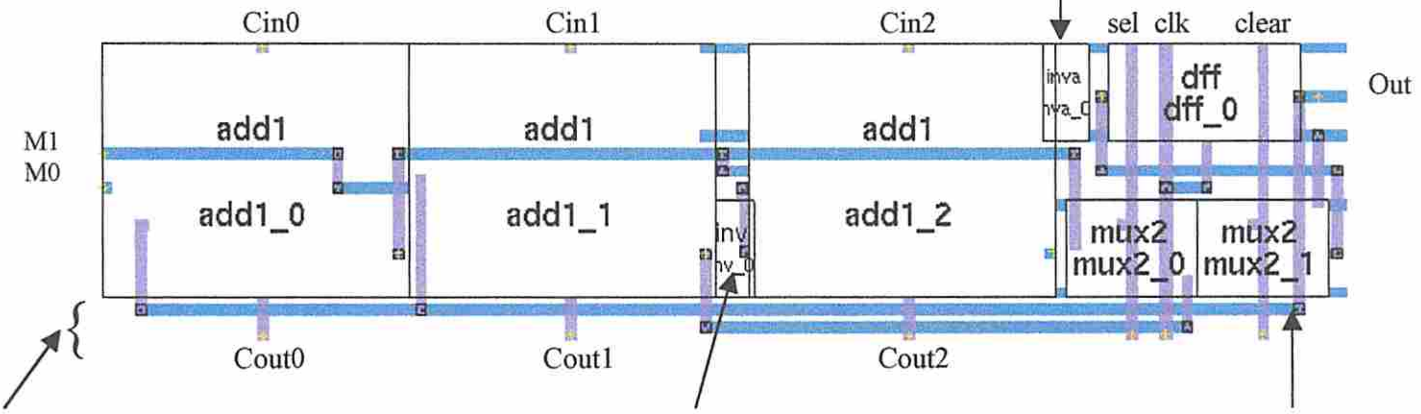
SISO stage 2: F & B Calculation (F_cal and B_cal)

Compared to other SISO stages as we will see from their schematic later, this stage obviously has the maximum calculation path. Assuming that the delay in the input buffer stage is also less than this stage, we can find the maximum running freqency by determining the delay of this path.



1- Bit ACS unit + register size = 512 x 128

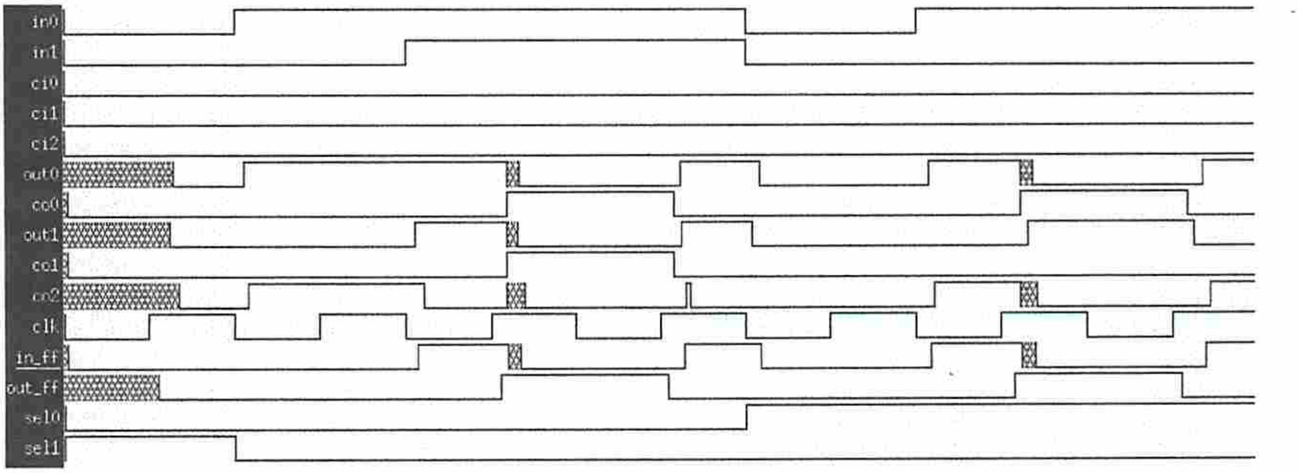
Sign bit inverted buffer, left floating, used only at the MSB unit. (pmos 16x2, nmos 8x2)



3 available routing channels. One is already used in ACS; the other two are for feedback paths.

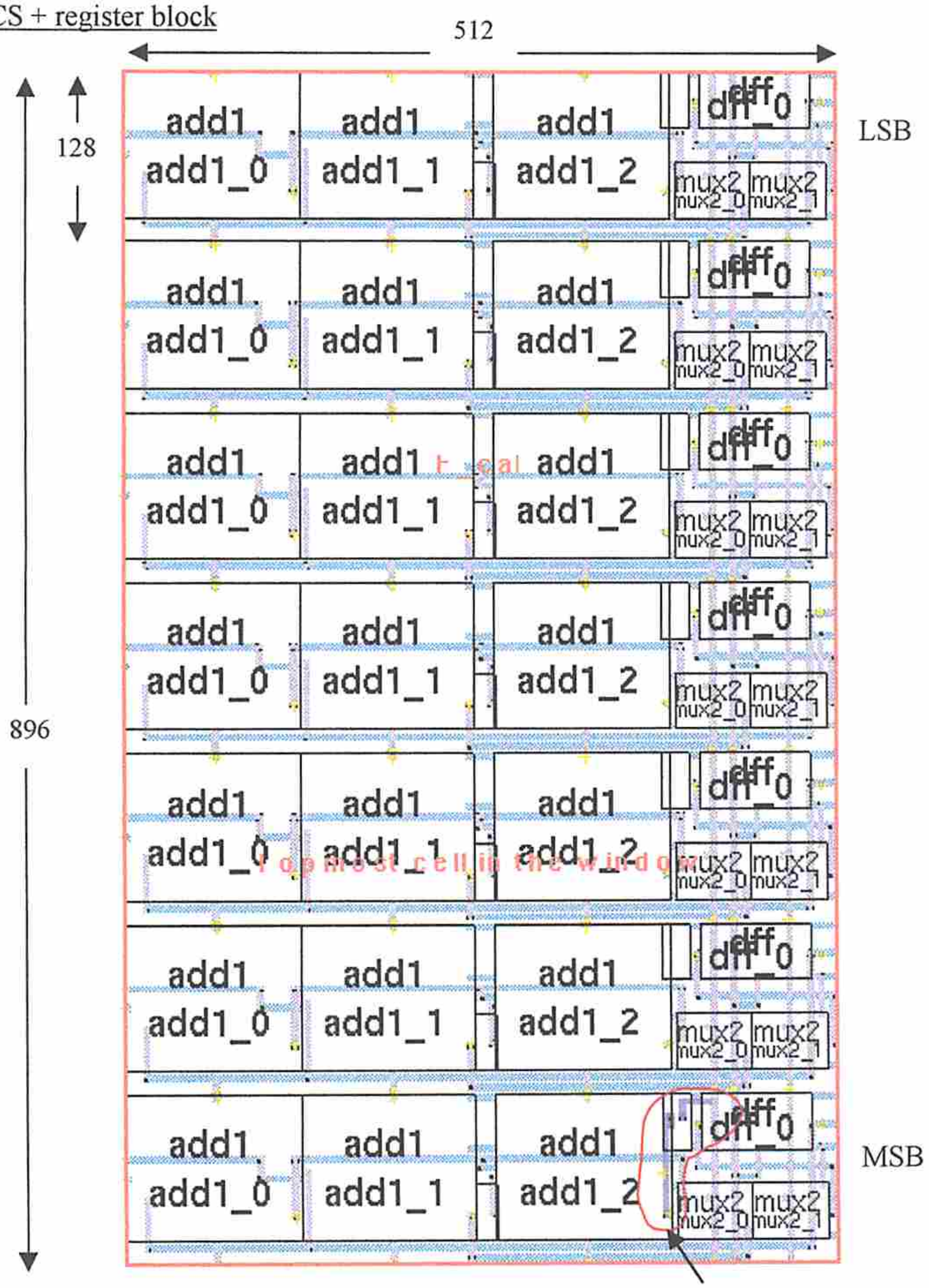
Inventer here to do complement for subtraction

Dummy feedback path. for timing testing. In real implementation the feedback would come from some other F calcalation block.



Clk-to-co2 delay = 4.5 ns (approx.)

7-bit ACS + register block



Connect the sign bit to the inverted buffer and the mux select line.

The worst case delay is composed of

a) $T_{dff,q} + T_{dff,s}$
b) wire delay from output feedback to input + delay due to load seen by this node
c) prop delay from input of add1_0 to carry-out of add1_2 for the LSB unit
d) prop delay from carry-out of add1_2 of the LSB unit to output of add1_2 of the MSB unit.
e) wire delay of the mux2_0 select signal from the MSB to LSB unit + delay due to load seen by this node

f) prop delay through mux2_0 and mux2_1

It turns out the wire delays are very small compared to other parts so they are nelegible. a), c), d) and f) can be measured from simulations of the one-bit circuit and the results are as follows:

- clk-to-cout2 delay = 4.5 ns. This value already includes $T_{\mathrm{dff,q}}$ + c) + b)
- cin-to-cout delay = 1.3ns, so d) = 1.2ns x 6 = 7.8 ns
- f) is very small about, 0.1 ns
- $T_{\mathrm{dff,s}}$ = 0.18 ns from previous DFF simulation

We need one more delay in e) due to loading on the Mux2_0 select line
The driving inverter has pmos sized 16x2 and nmos sized 8x2. Each driven mux is seen with 4 input transistors, 2 pmos sized 8x2 and 2 nmos sized 8x2. Since R(pmos) = 2 R(nmos) and width(pmos) = 2width(nmos), the rise time and fall time is balanced and can be calculated as shown below.

$R(\mathrm{pmos}) = R_{\mathrm{pmos,square}} \times L/W = 30 \times 2/16 = 3.75\ \mathrm{k\Omega}$
Given that the unit length capacitance of p-diff, n-diff, and poly are equal = 2pF/µm,
$C(\mathrm{load}) = C\_diff + C\_gate$ (of next stage)
$= (16+8)\mathrm{x}0.15\mathrm{x}2 + ((8+4)\mathrm{x}2\mathrm{x}7)\mathrm{x}0.15\ \mathrm{x}2$
$= 57.6\ \mathrm{fF}$

So, delay time = RC $\approx$ 0.22 ns

So, the total delay is 4.5 + 7.8 + 0.1 + 0.18 + 0.22 = 12.8 ns
and the maximum clock frequency = 1/delay = 78.1 MHz

Throughput

Recalling from section 2 regarding the clock frequencies, we have

$$(21(N + 2 + pl) + 2)\ T_{\mathrm{clk}} \quad \leq \quad 4(N + 2)\ T_{\mathrm{clk\_in}}$$

where N is the block length (excluding tail bits)
pl is the pipeline overhead cycles. For structural RTL here, pl = 5 cycles.
T_{clk} and $T_{\mathrm{clk\_in}}$ are processing clock period and input clock period.

Suppose we can select the clock frequencies such that they approximately equalize this equation, we can say that the time to process 2 blocks of data is $(21(N + 2 + pl) + 2)\ T_{\mathrm{clk}}$. This is, in other words, the processing time to produce 2N bits of final ouput.

So, the through put is

$$\text{Throughput} = \frac{2N}{(21(N+2+pl)+2)T_{\mathrm{clk}}}\ \text{bits/sec}$$

$$\text{Throughput} = \frac{2N \cdot f_{clk}}{(21(N+2+pl)+2)} \quad \text{bits/sec}$$

Substitute, N=1022, pl=5, we have **Throughput = 0.0946f_{clk} bit/sec** (at output)

Substitue f_{clk} = 78.1 MHz, we get a throught of 7.39 Mbits/sec

Area needed for F & B Calculation

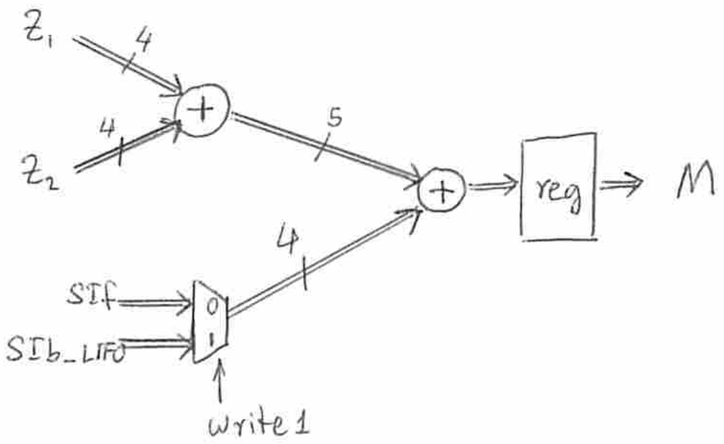
We need four blocks of 7-bit ACS for F calculation and another four for B calculation. Tile the blocks vertically with 4-λ gaps. Each block will take 512x900 and F-cal takes 512x3600

Need 1 block for F-cal and another for B_cal, so the area needed is

➔ $\boxed{512 \text{ x } 3600}$ x 2

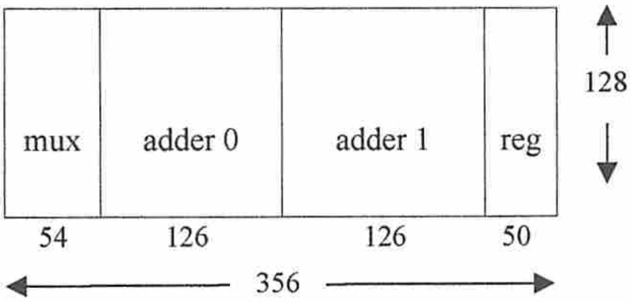
Note: For all the SISO components, I am estimating the area used by only one SISO. The area for two SISOs, which will be used in the final area estimation, will just be doubled.

SISO stage 1: M Calculation (Mf and Mb)



Reg was designed with size 80x40.
Assume there is an aquivalent
version of size 50x128.

1 bit unit



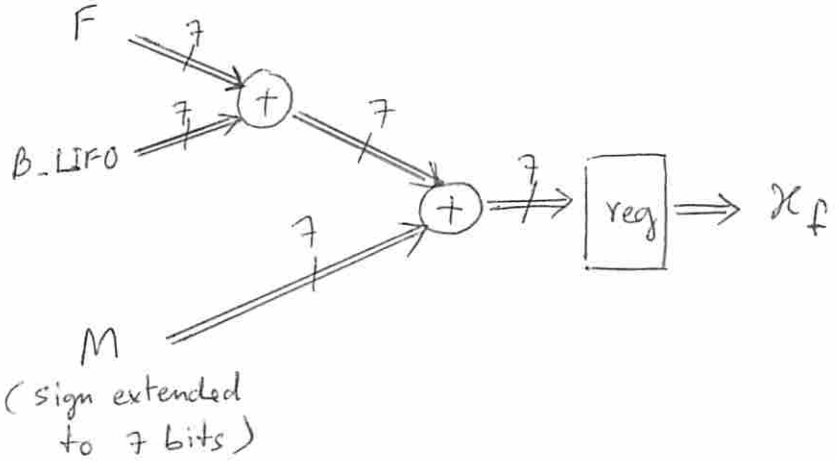
2 rows of cells
+ routing channels

Need 5 units to form a 5-bit block => 356 x 640

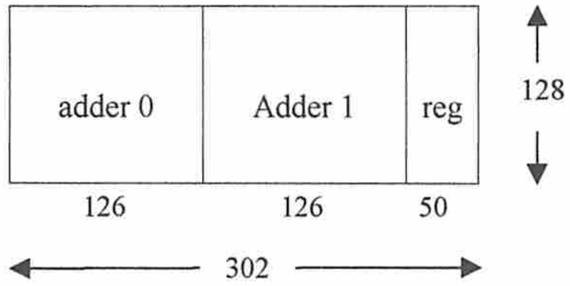
Need 3 blocks for Mf calculation and another 3 blocks for Mb. So, the area is

→ $\boxed{356 \times 1920}$ X 2

SISO stage 3: Completion Add Calculation (Comp_A)



1 bit unit

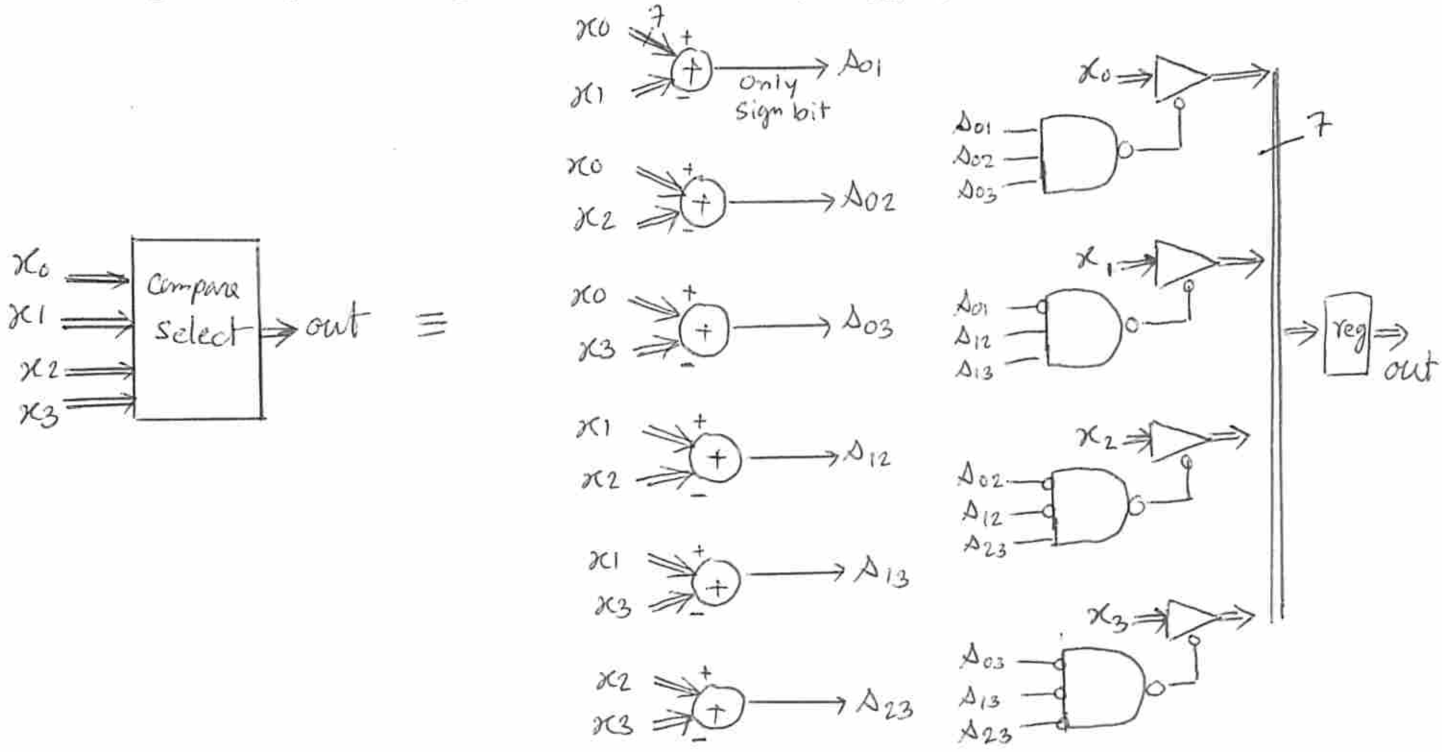


Need 7 units to form a 7-bit block → 302 x 896

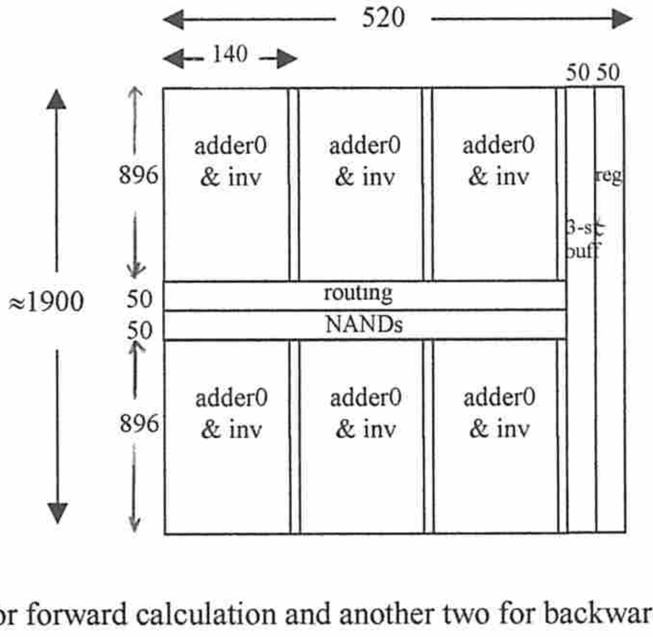
Need 8 blocks in the forward calculation → $\boxed{\text{302 x 3600}}$ X 2

And another 8 blocks in for backward → $\boxed{\text{302 x 3600}}$ X 2

SISO stage 4: Completion Compare-Select Calculation (Comp_CS)



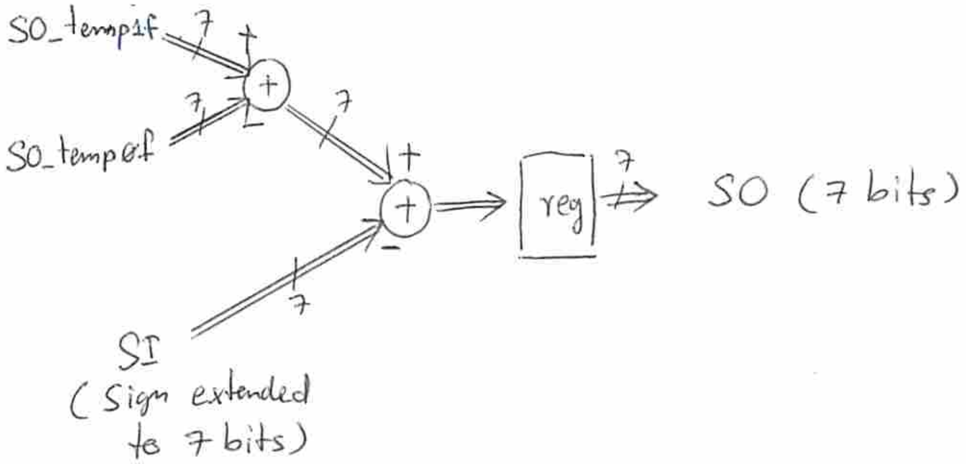
1 bit adder + inv +gap = 140 x128 ➔ form a 7 bit block of size 140 x 896



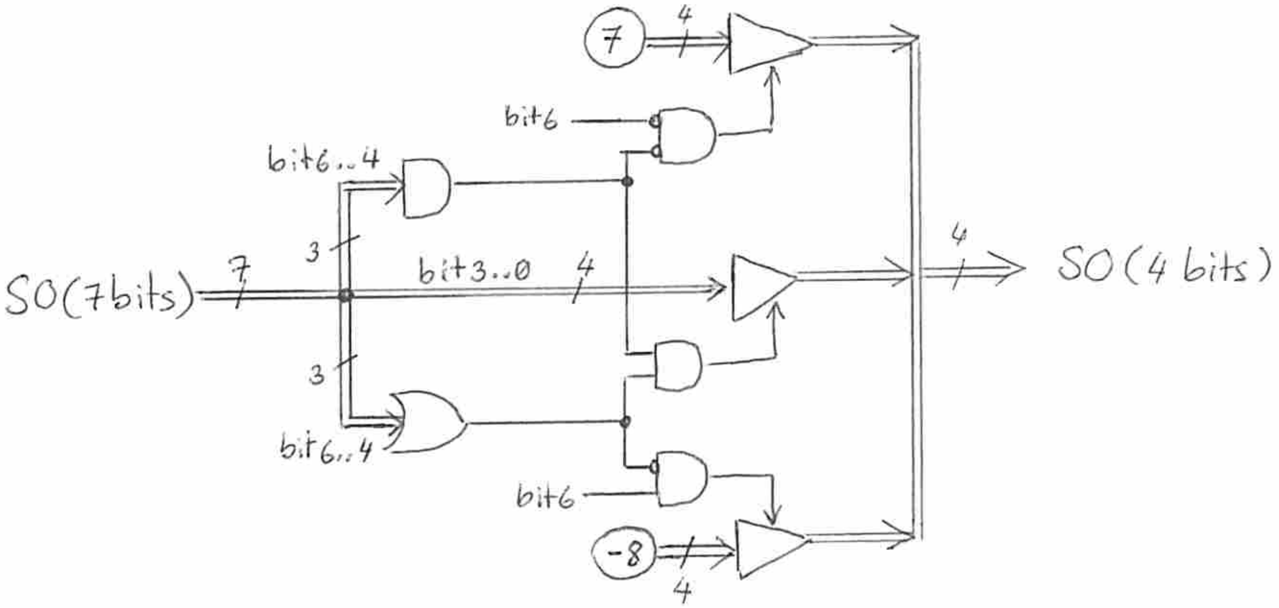
Need 2 blocks for forward calculation and another two for backward one.

➔ $\boxed{520 \text{ x } 1900}$ X 4

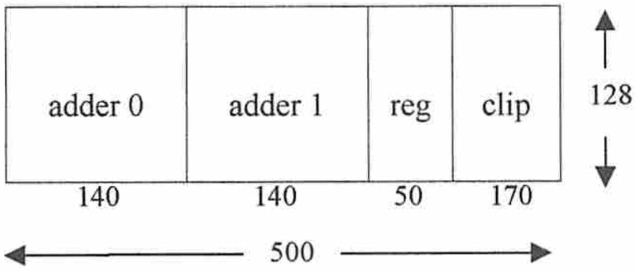
SISO stage 5: Completion Final Summation (Sum_comp)



SISO stage 6: Clipping (Clip)



1-bit unit for combined stage 5&6

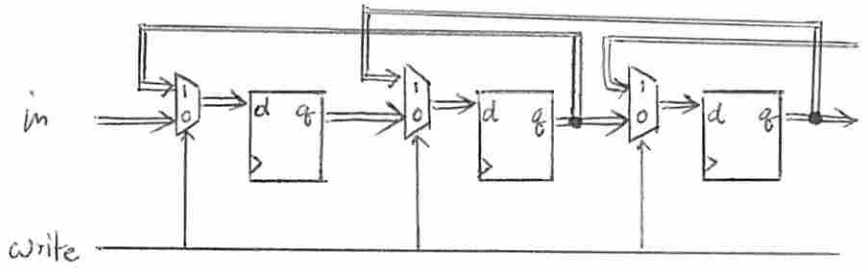


Need 7-bit block ➔ 500 x 896

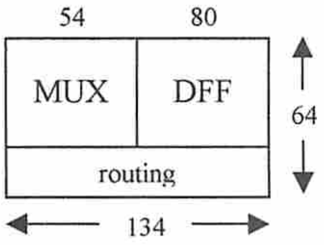
Need 1 block for forward calculation and another for backward

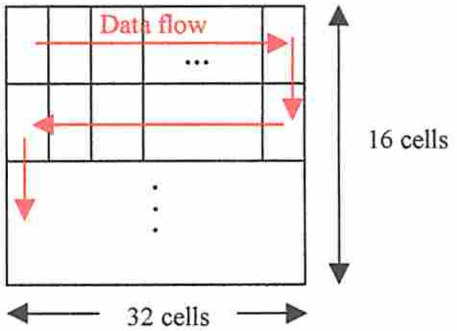
➔ 500 x 896 X 2

LIFO Buffers



1-bit unit





For LIFO_SIf and LIFO_SIb, we need 4-bit LIFO cells (1 cell ≡ 134 x 256)

➔ need 512 cells x 2, and arrange them in blocks of 32 cells x 16 cells

➔ $\boxed{4288 \text{ x } 4096}$ X 2

For LIFO_F and LIFO_B, we need 7-bit LIFO cells (1 cell ≡ 134 x 448)

➔ need 512 cells x 4 x 2, and arrange them in blocks of 32 cells x 16 cells

➔ $\boxed{4288 \text{ x } 7168}$ X 4 x 2

SISO Controller

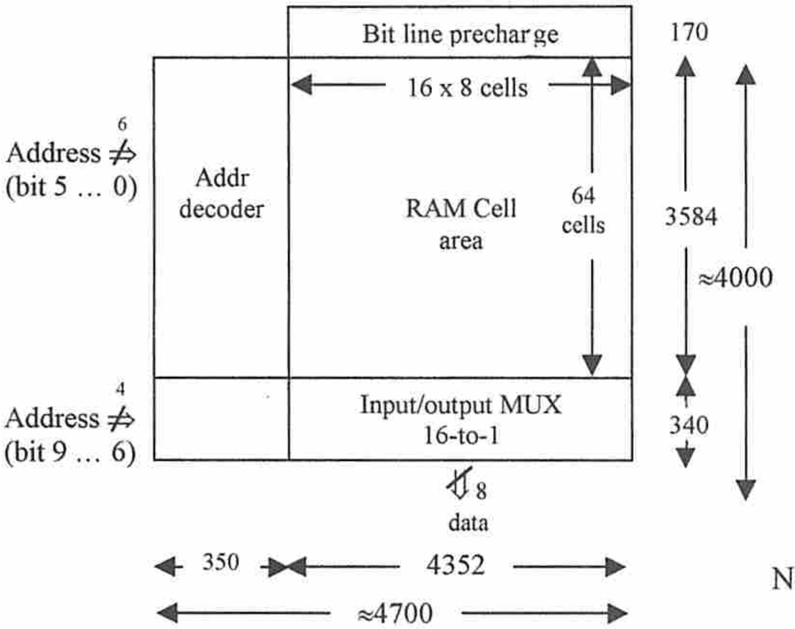
The SISO controller is composed of

1) a 10-bit counter
2) 3-state FSM with 3 output signals

Therefore, it is very small and negligible in the estimation.
(Assume it can be put in some empty hole in SISO.)

Input Buffers

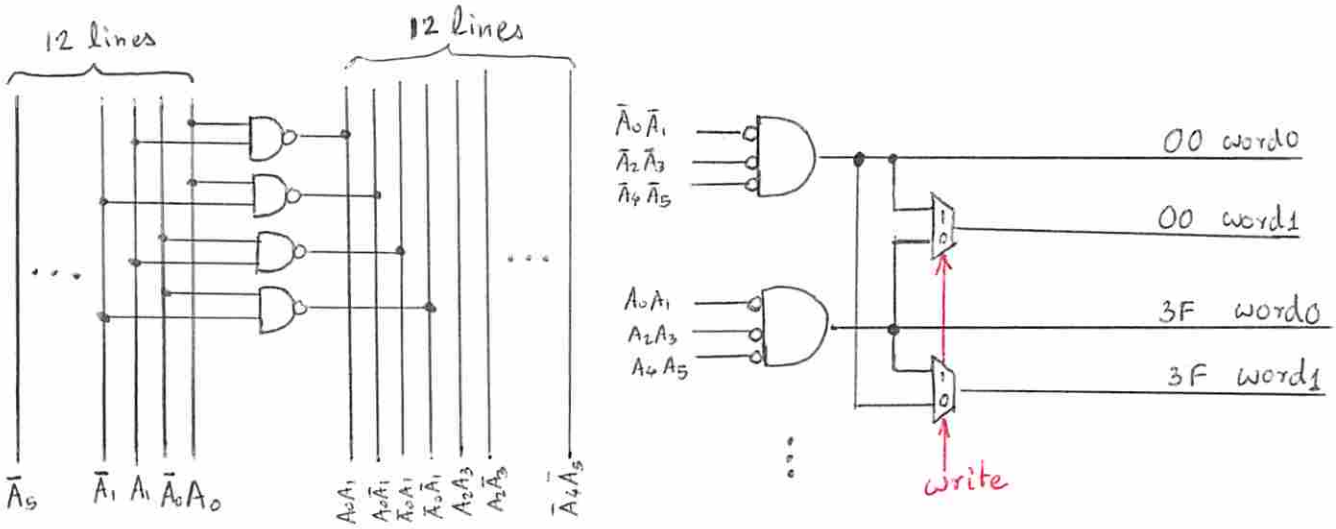
The control components are very small and assumed to reside in some small empty hole. The main components are four 2-port-read (1-port-write) RAMs of size 1024 x 8 bits. Recall from earlier in this section that a 2-port RAM cell takes 34 x 56. The RAM is arranged as shown below.



Need $\boxed{4700 \text{ x } 4000}$ X 4

Address decoder

Although this is 2-port read, the address decoders can be combined together to generate both word 0 & word1 line. This is because the data read will always be in pairs of address (1, N-1), (2, N-2), (3, N-3), … , and N-i is just a complement of i.

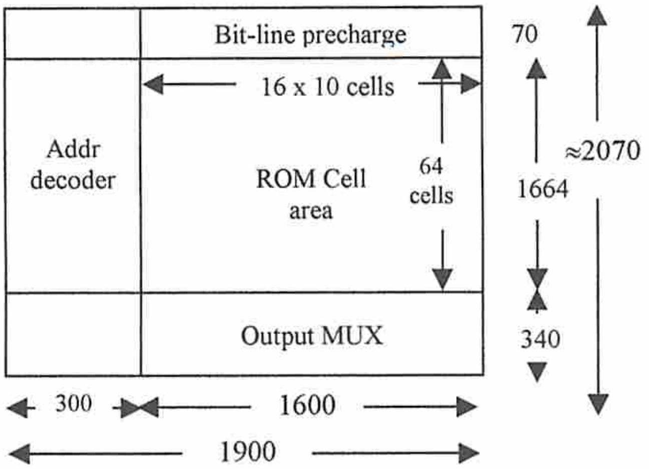


Interleaver/Deinterleaver

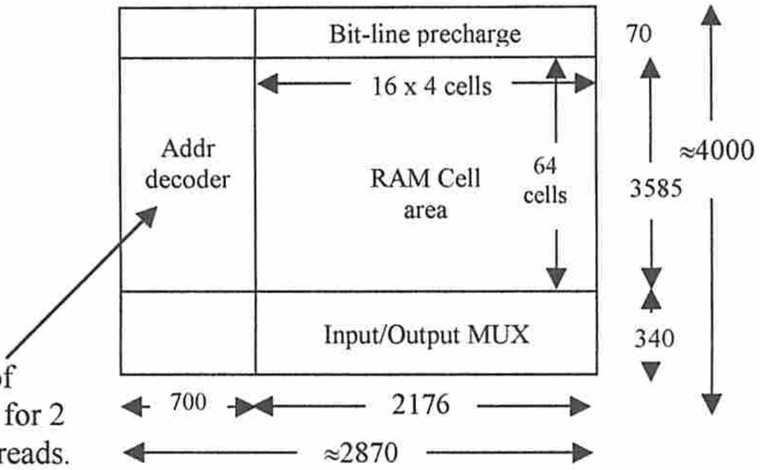
Again, neglecting small control components, we treat the area estimation of the interleaver and deiterleaver the same. Each is composed of

1) 10 bit x 1024 address ROM, 2 cells for 20 x 26
2) 4 bit x 1024 2-port-read RAM, 1 cell for 34 x 56

ROM



RAM



Need 2 sets of decoder here for 2 independent reads.

Therefore, need $\boxed{1900 \text{ x } 2070}$ X 2

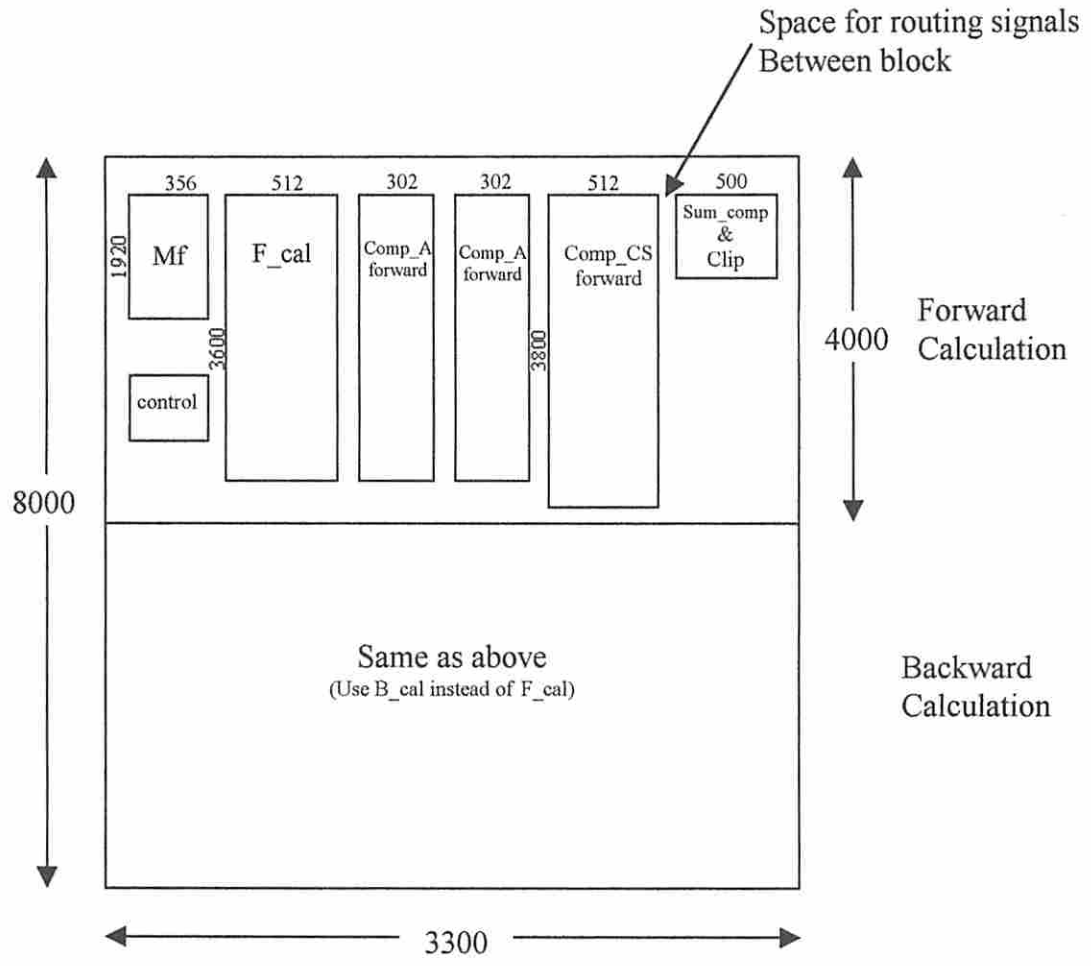
and $\boxed{2870 \text{ x } 4000}$ X 2

Main Controller

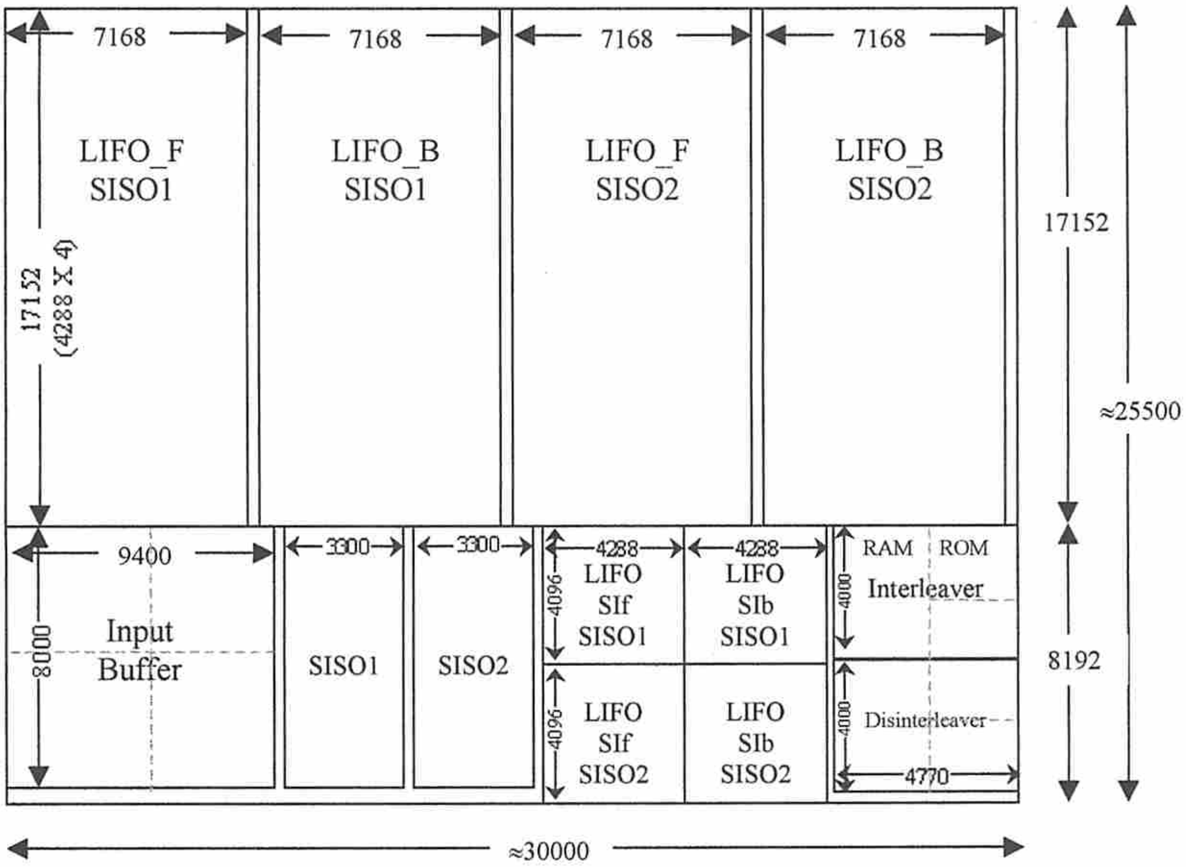
The main controller is a 3-state FSM with 5 outputs and 3 inputs. Similar to the SISO controller, this is very small and negligible.

Putting Together: Floor Plan and Total Chip Area

SISO Area Estimation



Approximately, a SISO takes 3300 x 8000 λ^2



The total chip area (excluding pads) is

$$= 30{,}000 \times 25{,}500\ \lambda^2$$

$$= 30{,}000 \times 25{,}500\ (0.15\ \mu\text{m})^2$$

$$= 4.5 \times 3.825\ (\text{mm})^2$$

Section 5 Verilog Codes for Behavioral RTL

1) Testturbo.v

```
module Testturbo;

parameter           width=4;

//parameter         N=6, input_file1="vec61.dat", input_file2="vec62.dat",
//                  result_file="hard_dec6.dat";

parameter           N=1024, input_file1="ck0.dat", input_file2="ck1.dat",
                    result_file="hard_dec.dat";

reg                 clk, clk_in, reset, start;
reg [width-1:0]     in;
wire                out, writeout;

reg [width-1:0]     z1[0:2*N+3], z2[0:2*N+3];
reg                 result[0:2*N-1];

integer             k, err, b;

Turbo DUT (in, reset, clk, clk_in, start, out, writeout);

initial
begin
    $shm_open("behav.shm");
        $shm_probe("AS");
        #6500 $shm_close();
        #100  $finish;
end

initial
begin
   $readmemb (input_file1, z1);
   $readmemb (input_file2, z2);
   $readmemb (result_file, result);

   clk    = 0;
   clk_in= 0;
   reset = 0;
   start = 0;
   b = 0;
   k=0;

   #170  reset = 1;

   #100  reset = 0;
         start = 1;

   #100  start = 0;

   for (k=0; k<2*N+4; k=k+1)
   begin
      in = z1[k];
      #100;
      in = z2[k];
      #100;
   end

   in = 0;
end

// Compare the final decision output against the output vector file

always @ (posedge writeout)
begin
   #3;
   $display (" \n");
   $display ("Block %d : Start checking final decisions", b);
   err = 0;
   for (k=0; k<N; k=k+1)
   begin
      if ( out != result[b*N+k] )
         begin
            $display ("Error at time k = %d", k);
            err = err + 1;
         end
      #6;
   end
   $display ("Error at time k = %d,  Simulation output = %d", k, out);
   b = b + 1;                    // b is block counter, err is error counter
end

always
begin
   #6   clk = ~clk;
end

always
begin
    #50    clk_in = ~clk_in;
end

endmodule
```

2) turbo.v

```
module Turbo(in, reset, clk, clk_in, start, out, writeout);

parameter           width=4;

//parameter           N=6, addr_bit=3, int_file="int6.dat", deint_file="int6.dat";

parameter           N=1024, addr_bit=11, int_file="int1024.dat", deint_file="int1024.dat";

input               reset, clk, clk_in, start;
input [width-1:0]   in;
output              out, writeout;

wire                ready, last_iter, select, start_siso1, start_siso2, done_siso2,
                    hard_dici, write1, write2;
wire [2*width-1:0]  data1F, data1B, data2F, data2B;
wire [addr_bit-1:0] addr1, addr2, addr_w1, addr_w2;
wire [width-1:0]    SI1f, SI1b, SI2f, SI2b, SO1f, SO1b, SO2f, SO2b;

Input_buff  aInput_buff (clk_in, reset, start, in, select, addr1, addr2,
                         data1F, data1B, data2F, data2B, ready);
defparam aInput_buff.N=N, aInput_buff.addr_bit=addr_bit;

SISO1 aSISO1 (clk, reset, start_siso1, data1F, data1B, SI1f, SI1b, addr1, addr_w1, SO1f,
              SO1b, done_siso1, write1);
defparam aSISO1.N=N, aSISO1.addr_bit=addr_bit;

SISO2 aSISO2 (clk, reset, start_siso2, data2F[width-1:0], data2B[width-1:0], SI2f, SI2b,
              addr2, addr_w2, SO2f, SO2b, done_siso2, write2, last_iter);
defparam aSISO2.N=N, aSISO2.addr_bit=addr_bit;

Interleave aInterleave (clk, reset, write1, addr2, addr_w1, SO1f, SO1b, SI2f, SI2b);
defparam aInterleave.N=N, aInterleave.addr_bit=addr_bit, aInterleave.coef_file=int_file;

Deinterleave aDeinterleave (clk, reset, write2, addr1, addr_w2, SO2f, SO2b, SI1f, SI1b,
                             hard_dici, out, writeout);
defparam aDeinterleave.N=N, aDeinterleave.addr_bit=addr_bit,
         aDeinterleave.coef_file=deint_file;

Control aControl (clk, reset, ready, done_siso1, done_siso2, start_siso1, start_siso2,
                  hard_dici, last_iter, select);

endmodule
```

3) Control.v

```
module Control (clk, reset, ready, done_siso1, done_siso2, start_siso1, start_siso2,
                hard_dici, last_iter, select);

input                clk, reset, ready, done_siso1, done_siso2;
output               start_siso1, start_siso2, hard_dici, last_iter, select;

reg                  start_siso1, start_siso2, hard_dici, last_iter, select;

integer              iter;

parameter   IDLE   = 0,
            START  = 1,
            RUN    = 2;

reg [1:0]  STATE, NEXT_STATE;

always @ (STATE or ready or done_siso1 or done_siso2)
begin
      case (STATE)
         IDLE:
            if (ready)  NEXT_STATE <= START;
                else    NEXT_STATE <= IDLE;

         START:
            NEXT_STATE <= RUN;

         RUN:
            if (iter==22)
               NEXT_STATE <= IDLE;
            else if (done_siso1 || done_siso2)
               NEXT_STATE <= START;
      endcase
end


always @ (STATE)
begin
   start_siso1 = 0;      //default output values
   start_siso2 = 0;
   hard_dici = 0;
   last_iter = 0;

   case (STATE)
      IDLE:
         begin
            iter <= 0;
            select = 1;
         end

      START:
         begin
            case (iter)
               0:  start_siso1 = 1;

               19: begin
                     start_siso1 = 1;
                     start_siso2 = 1;
                     last_iter = 1;
                   end

               20: begin
                     start_siso1 = 0;
                     start_siso2 = 1;
                     hard_dici = 1;
                     last_iter = 1;
                   end

               21: hard_dici = 1;

               default
                   begin
                     start_siso1 = 1;
                     start_siso2 = 1;
                   end
            endcase
            select = ~select;
            iter <= iter + 1;
         end

   endcase

end

always @ (posedge clk or posedge reset)
    if (reset)
        STATE <= IDLE;
    else
        STATE <= NEXT_STATE;

endmodule
```

4) Input_buff.v

```
module Input_buff (clk, reset, start, in, select, addr1, addr2,
                   data1F, data1B,data2F, data2B, ready);

parameter            width=4, N=6, addr_bit=3;

input                clk, reset, start, select;
input [width-1:0]    in;
input [addr_bit-1:0] addr1, addr2;
output [2*width-1:0] data1F, data1B, data2F, data2B;
output               ready;

reg [addr_bit-1:0]   back_addr1, back_addr2;
reg [2*width-1:0]    data1F, data1B, data2F, data2B;
reg                  ready, buffer_ready;
reg [width-1:0]      temp;
reg [2*width-1:0]    M1[0:2*(N+2)-1], M2[0:2*(N+2)-1];
integer              count;

parameter   IDLE  = 0,
            S1    = 1,
            S2    = 2;

reg [1:0]  STATE;

always @ (posedge clk or posedge reset)
begin
   if (reset)
      begin
         STATE <= IDLE;
         count <= 0;
      end
   else
      case (STATE)
         IDLE:
            begin
               count <= 0;
               if (start)   STATE <= S1;
                   else     STATE <= IDLE;
            end
         S1:
               if (count==4*N+7)
```

```
                  begin
                     STATE <= S2;
                     count <= 0;
                  end
               else
                  begin
                     STATE <= S1;
                     count <= count + 1;
                  end
         S2:
               if (count==4*N+7)
                  begin
                     STATE <= S1;
                     count <= 0;
                  end
               else
                  begin
                     STATE <= S2;
                     count <= count + 1;
                  end
      endcase
end

// Write process synchronous to clk

always @ (posedge clk or STATE)
begin

   case (STATE)
      IDLE:
         begin
            ready = 0;
            buffer_ready = 0;
         end

      S1:                // save input into M1
         begin
            if (buffer_ready == 1)
               if (count==0) ready = 1;
               else ready = 0;
            else
               ready = 0;

            if (count[0] == 0)
               temp = in;
            else
               M1[(count-1)/2] = (temp, in);
         end

      S2:                // save input into M2
         begin
            if (count==0) ready = 1;
            else ready = 0;
            buffer_ready = 1;

            if (count[0] == 0)
               temp = in;
            else
               M2[(count-1)/2] = (temp, in);
         end
   endcase
end

// Read process asynchronous to clk

always @ (addr1 or addr2 or STATE or select)
begin
   case (STATE)
      IDLE:
         begin
            datalF = 8'hz;
            data1B = 8'hz;
            data2F = 8'hz;
            data2B = 8'hz;
         end

      S1:
         begin
            back_addr1 = (N+1)-addr1;
            back_addr2 = (N+1)-addr2;
            data1F = M2[(N+2)*select + addr1];
            data1B = M2[(N+2)*select + back_addr1];
            data2F = M2[(N+2)*(!select) + addr2];
            data2B = M2[(N+2)*(!select) + back_addr2];
         end

      S2:
         begin
            back_addr1 = (N+1)-addr1;
            back_addr2 = (N+1)-addr2;
            data1F = M1[(N+2)*select + addr1];
            data1B = M1[(N+2)*select + back_addr1];
            data2F = M1[(N+2)*(!select) + addr2];
            data2B = M1[(N+2)*(!select) + back_addr2];
         end
   endcase
end

endmodule
```

5) SISO1.v

```
module SISO1 (clk, reset, start, data1F, data1B, SIf, SIb, addr, addr_w, SOf, SOb, done,
              write);

parameter            widthz=4, widths=4, widthf=7;
parameter            N=6, addr_bit=3;
parameter            Init = 0, Inf = 31;
parameter            sb = widthf - widthz;
parameter            maxs=7, mins=-8;

input                clk, start, reset;
input [2*widthz-1:0] data1F, data1B;
input [widths-1:0]   SIf, SIb;
output [widths-1:0]  SOf, SOb;
output               done, write;
output [addr_bit-1:0] addr, addr_w;

reg [widths-1:0]     SOf, SOb;
reg [widthf-1:0]     SOf_temp, SOb_temp;
reg                  done, write;
reg [addr_bit-1:0]   addr, addr_w;

reg [widths-1:0]     SI[0:N+1];
reg [widthz-1:0]     zf1, zf2, zb1, zb2;
reg [widthf-1:0]     M[0:3*(N+2)-1];
reg [widthf-1:0]     F[0:4*(N+3)-1], B[0:4*(N+3)-1];
integer              j, k;

parameter   IDLE   = 0,
            F_B    = 1,
            COMP   = 2;

reg [1:0]  STATE;

always @ (addr)                   //addr and addr_w are the same for behav RTL
   addr_w = addr;

always @ (data1F or data1B)
begin
      zf1 = data1F[2*widthz-1:widthz];
```

```
         zf2 = data1F[widthz-1:0];
         zb1 = data1B[2*widthz-1:widthz];
         zb2 = data1B[widthz-1:0];
   end

   always @ (posedge clk or posedge reset)
   begin
      if (reset)
         STATE <= IDLE;
      else
         case (STATE)
            IDLE:
               if (start)  STATE <= F_B;
                   else    STATE <= IDLE;

            F_B:
               if (k==N/2) STATE <= COMP;
                   else     STATE <= F_B;

            COMP:
               if (k==N+1) STATE <= IDLE;
                   else    STATE <= COMP;

         endcase
   end


   always @ (posedge clk)
   begin

      case (STATE)
         IDLE:
            begin
               k = 0; j = N+1;
               F[0] = Init;
               F[1] = Inf;  F[2] = Inf;  F[3] = Inf;
               B[4*(j+1)] = Init;
               B[4*(j+1)+1] = Inf;
               B[4*(j+1)+2] = Inf;
               B[4*(j+1)+3] = Inf;
               addr = 0;
               done = 0;
               write = 0;
            end

         F_B:
            begin
               done = 0;
               write = 0;
               if (k[0] == 0)    // k even and j odd
               begin
                  M[3*k]   = signext(SIf) + signext(zf1) + signext(zf2);
                  M[3*k+1] = signext(zf2);
                  M[3*k+2] = signext(SIf) + signext(zf1);

                  M[3*j]   = signext(SIb) + signext(zb1);
                  M[3*j+1] = 0;
                  M[3*j+2] = signext(SIb) + signext(zb1);
               end
               else
               begin
                  M[3*k]   = signext(SIf) + signext(zf1);
                  M[3*k+1] = 0;
                  M[3*k+2] = signext(SIf) + signext(zf1);

                  M[3*j]   = signext(SIb) + signext(zb1) + signext(zb2);
                  M[3*j+1] = signext(zb2);
                  M[3*j+2] = signext(SIb) + signext(zb1);
               end

               F[4*(k+1)]   = min2(F[4*k],              F[4*k+2] + M[3*k]);
      F[4*(k+1)+1] = min2(F[4*k] + M[3*k],     F[4*k+2]);
      F[4*(k+1)+2] = min2(F[4*k+1] + M[3*k+2], F[4*k+3] + M[3*k+1]);
      F[4*(k+1)+3] = min2(F[4*k+1] + M[3*k+1], F[4*k+3] + M[3*k+2]);

      B[4*j]   = min2(B[4*(j+1)],              B[4*(j+1)+1] + M[3*j]);
      B[4*j+1] = min2(B[4*(j+1)+3] + M[3*j+1], B[4*(j+1)+2] + M[3*j+2]);
      B[4*j+2] = min2(B[4*(j+1)+1],            B[4*(j+1)] + M[3*j]);
      B[4*j+3] = min2(B[4*(j+1)+2] + M[3*j+1], B[4*(j+1)+3] + M[3*j+2]);

      SI[k] = SIf;
      SI[j] = SIb;
      k <= k + 1;
      j <= j - 1;
      addr <= k + 1;
   end

COMP:
   begin
      if (k==N)
         done = 1;
      else
         done = 0;
      write = 1;
      F[4*(k+1)]   = min2(F[4*k],              F[4*k+2] + M[3*k]);
      F[4*(k+1)+1] = min2(F[4*k] + M[3*k],     F[4*k+2]);
      F[4*(k+1)+2] = min2(F[4*k+1] + M[3*k+2], F[4*k+3] + M[3*k+1]);
      F[4*(k+1)+3] = min2(F[4*k+1] + M[3*k+1], F[4*k+3] + M[3*k+2]);

      B[4*j]   = min2(B[4*(j+1)],              B[4*(j+1)+1] + M[3*j]);
      B[4*j+1] = min2(B[4*(j+1)+3] + M[3*j+1], B[4*(j+1)+2] + M[3*j+2]);
      B[4*j+2] = min2(B[4*(j+1)+1],            B[4*(j+1)] + M[3*j]);
      B[4*j+3] = min2(B[4*(j+1)+2] + M[3*j+1], B[4*(j+1)+3] + M[3*j+2]);

      SOf_temp = min4(F[4*k] + M[3*k] + B[4*(k+1)+1],
                      F[4*k+1] + M[3*k+2] + B[4*(k+1)+2],
                      F[4*k+2] + M[3*k] + B[4*(k+1)],
                      F[4*k+3] + M[3*k+2] + B[4*(k+1)+3])
               - min4(F[4*k] + B[4*(k+1)],
                      F[4*k+1] + M[3*k+1] + B[4*(k+1)+3],
                      F[4*k+2] + B[4*(k+1)+1],
                      F[4*k+3] + M[3*k+1] + B[4*(k+1)+2]) - signext(SI[k]);

      SOb_temp = min4(F[4*j] + M[3*j] + B[4*(j+1)+1],
                      F[4*j+1] + M[3*j+2] + B[4*(j+1)+2],
                      F[4*j+2] + M[3*j] + B[4*(j+1)],
                      F[4*j+3] + M[3*j+2] + B[4*(j+1)+3])
               - min4(F[4*j] + B[4*(j+1)],
                      F[4*j+1] + M[3*j+1] + B[4*(j+1)+3],
                      F[4*j+2] + B[4*(j+1)+1],
                      F[4*j+3] + M[3*j+1] + B[4*(j+1)+2]) - signext(SI[j]);

      // clipping output to 4 bits

      if (&SOf_temp[widthf-1:widthf-widths] || ~|SOf_temp[widthf-1:widthf-widths])
         SOf = SOf_temp[widths-1:0];
      else if (SOf_temp[widthf-1])
         SOf = mins;
      else
         SOf = maxs;

      if (&SOb_temp[widthf-1:widthf-widths] || ~|SOb_temp[widthf-1:widthf-widths])
         SOb = SOb_temp[widths-1:0];
      else if (SOb_temp[widthf-1])
         SOb = mins;
      else
         SOb = maxs;

      k <= k + 1;
      j <= j - 1;
      addr <= k + 1;
   end
   endcase

end


function [widthf-1:0] min2;
input [widthf-1:0]  a, b;
reg   [widthf-1:0]  diff;

begin
   diff = a-b;
   min2 = diff[widthf-1] ?  a : b;
end

endfunction


function [widthf-1:0] min4;
input [widthf-1:0]  a, b, c, d;

   min4 = min2(min2(a,b), min2(c,d));

endfunction

function [widthf-1:0] signext;
input [widthz-1:0] a;

   signext = {{sb{a[widthz-1]}}, a};

endfunction

endmodule
```

6) SISO2.v

```
module SISO2 (clk, reset, start, data1F, data1B, SIf, SIb, addr, addr_w, SOf, SOb, done,
write, last_iter);

parameter           widthz=4, widths=4, widthf=7;
parameter           N=6, addr_bit=3;
parameter           Init = 0, Inf = 31;
parameter           sb = widthf - widthz;
parameter           maxs=7, mins=-8;

input               clk, start, reset, last_iter;
input [widthz-1:0]  data1F, data1B;   // take only parity bits from channel input (4 bits)
input [widths-1:0]  SIf, SIb;
output [widths-1:0] SOf, SOb;
output              done, write;
output [addr_bit-1:0] addr, addr_w;

reg [widths-1:0]    SOf, SOb;
reg [widthf-1:0]    SOf_temp, SOb_temp;
reg                 done, write, last_iter_ff;
reg [addr_bit-1:0]  addr, addr_w;

reg [widths-1:0]    SI[0:N+1];
reg [widthz-1:0]    zf1, zf2, zb1, zb2;
reg [widthf-1:0]    M[0:3*(N+2)-1];
reg [widthf-1:0]    F[0:4*(N+3)-1], B[0:4*(N+3)-1];
integer             j, k;

parameter  IDLE   = 0,
           F_B    = 1,
           COMP   = 2;

reg [1:0]  STATE;
```

```
... same as SISO1.v ...

always @ (posedge clk)
begin

   case (STATE)
      IDLE:
         begin
            k = 0; j = N+1;
            F[0] = Init;
            F[1] = Inf;  F[2] = Inf;  F[3] = Inf;
            B[4*(j+1)] = Init;
            B[4*(j+1)+1] = Inf;
            B[4*(j+1)+2] = Inf;
            B[4*(j+1)+3] = Inf;
            addr = 0;
            done = 0;
            write = 0;
            if (last_iter)
               last_iter_ff <= 1;
            else
               last_iter_ff <= 0;
         end

      F_B:
         begin
            done = 0;
            write = 0;
            if (k[0] == 0)    // k even and j odd
            begin
               M[3*k]   = signext(SIf);
               M[3*k+1] = 0;
               M[3*k+2] = signext(SIf);

               M[3*j]   = signext(SIb) + signext(zb2);
               M[3*j+1] = signext(zb2);
               M[3*j+2] = signext(SIb);
                    end
            else
            begin
               M[3*k]   = signext(SIf) + signext(zf2);
               M[3*k+1] = signext(zf2);
               M[3*k+2] = signext(SIf);

               M[3*j]   = signext(SIb);
               M[3*j+1] = 0;
               M[3*j+2] = signext(SIb);
            end

            F[4*(k+1)]   = min2(F[4*k],              F[4*k+2] + M[3*k]);
            F[4*(k+1)+1] = min2(F[4*k] + M[3*k],     F[4*k+2]);
            F[4*(k+1)+2] = min2(F[4*k+1] + M[3*k+2], F[4*k+3] + M[3*k+1]);
            F[4*(k+1)+3] = min2(F[4*k+1] + M[3*k+1], F[4*k+3] + M[3*k+2]);

            B[4*j]   = min2(B[4*(j+1)],              B[4*(j+1)+1] + M[3*j]);
            B[4*j+1] = min2(B[4*(j+1)+3] + M[3*j+1], B[4*(j+1)+2] + M[3*j+2]);
            B[4*j+2] = min2(B[4*(j+1)+1],            B[4*(j+1)] + M[3*j]);
            B[4*j+3] = min2(B[4*(j+1)+2] + M[3*j+1], B[4*(j+1)+3] + M[3*j+2]);

            if (last_iter_ff)
               begin
                  SI[k] = 0;
                  SI[j] = 0;
               end
            else
               begin
                  SI[k] = SIf;
                  SI[j] = SIb;
               end
            k <= k + 1;
            j <= j - 1;
            addr <= k + 1;
         end

      COMP:


    ... same as SISO1.v ...


endmodule
```

7) Interleave.v

```
module Interleave (clk, reset, write, addr, addr_w, SOf, SOb, SIf, SIb);

parameter            width=4, addr_bit=3, N=6, coef_file="interleave.dat";

input                clk, reset, write;
input [addr_bit-1:0] addr, addr_w;
input [width-1:0]    SOf, SOb;
output [width-1:0]   SIf, SIb;

reg [width-1:0]      SIf, SIb;
reg [addr_bit-1:0]   P[0:N-1];
reg [width-1:0]      M[0:N-1];
reg [addr_bit-1:0]   dum_addr;

integer              i;

initial
begin
   $readmemb (coef_file, P);
end

// read process asynchronous to clk
always @ (addr or write)
begin
   if (~write)
   begin
      if (addr >= N)
         SIf = 0;
      else
         SIf = M[P[addr]];
      if (N+1-addr >= N)
         SIb = 0;
      else
         SIb = M[P[N+1-addr]];
   end
   else
   begin
      SIf = 0;
      SIb = 0;
   end
end

// write process synchronous to clk
always @ (posedge clk or posedge reset)
begin
   if (reset)
      for (i=0; i<=N+1; i=i+1)
         M[i] = 0;
   else if (write)
   begin
      if (addr_w == 0)            // change addr_w to dum_addr to account for
         dum_addr = N+1;          // data delay in behavioural RTL.
      else
         dum_addr = addr_w - 1;
      M[dum_addr] = SOf;
      M[N+1-dum_addr] = SOb;
   end
end

endmodule
```

8) Deinterleave.v

```
module Deinterleave (clk, reset, write, addr, addr_w, SOf, SOb, SIf, SIb, hard_dici, out,
                     writeout);

parameter            width=4, addr_bit=3, N=6, coef_file="interleave.dat";

input                clk, write, reset, hard_dici;
input [addr_bit-1:0] addr, addr_w;
input [width-1:0]   SOf, SOb;
output [width-1:0]  SIf, SIb;
output               out, writeout;

reg [width-1:0]     SIf, SIb;
reg [addr_bit-1:0]   P[0:N-1];
reg [width-1:0]     M[0:N-1], out_temp;
reg [addr_bit-1:0]   dum_addr;
reg                  out, writeout;

integer              count_out, i;

initial
begin
   $readmemb (coef_file, P);
end

always @ (posedge clk or negedge clk)
begin
   if (hard_dici || writeout)
      begin
         out_temp = M[count_out];
         out = out_temp[width-1];
         writeout = 1;
         count_out = count_out + 1;
      end

   if (count_out==N+1)
      begin
         count_out =0;
         writeout = 0;
         out = 0;
      end

end

// read process asynchronous to clk
always @ (addr or write)
begin
   if (~write && ~hard_dici)
   begin
      if (addr >= N)
         SIf = 0;
      else
         SIf = M[addr];

      if (N+1-addr >= N)
         SIb = 0;
      else
         SIb = M[N+1-addr];
   end
   else
   begin
      SIf = 0;
      SIb = 0;
   end
```

```
end

// write process synchronous to clk
always @ (posedge clk or posedge reset)
begin
   if (reset)
   begin
      for (i=0; i<=N+1; i=i+1)
         M[i] = 0;
      out = 0;
      writeout =0;
      count_out = 0;
   end
   else if (write)
   begin
      if (addr_w == 0)            // change addr to dum_addr to account for
         dum_addr = N+1;        // data delay in behavioural RTL.
      else
         dum_addr = addr_w - 1;
      M[P[dum_addr]] = SOf;
      M[P[N+1-dum_addr]] = SOb;
   end
end

endmodule
```

9) vec61.dat

```
// Test vector SI(Ck1=1) for block length = 6
// block 1 => 5, 2, 2, 0, -3, 3, 0, 1
// block 2 => 4, 1, 0, -2, -6, 4, -1, -5
       0101
       0010
       0010
       0000
       1101
       0011
       0000
       0001
       0100
       0001
       0000
       1110
       1010
       0100
       1111
       1011
```

vec62.dat

```
// Test vector SI(Ck1=2) for block length = 6
// block 1 => 0, 1, -5, 6, -1, -6, 0, -2
// block 2 => 5, 3, -3, -5, 6, -1, 2, 3
       0000
       0001
       1011
       0110
       1111
       1010
       0000
       1110
       0101
       0011
       1101
       1011
       0110
       1111
       0010
       0011
```

10) int6.dat

```
011
010
101
000
100
001
```

Section 6 Verilog Codes for Structural RTL

1) Testturbo.v, Turbo.v, and Control.v

These modules are the same as the behavioral RTL design.

2) Input_buff.v

```
module Input_buff (clk, reset, start, in, select, addr1, addr2,
                   data1F, data1B,data2F, data2B, ready);

parameter            width=4, N=6, addr_bit=3;

input                clk, reset, start, select;
input [width-1:0]    in;
input [addr_bit-1:0] addr1, addr2;
output [2*width-1:0] data1F, data1B, data2F, data2B;
output               ready;

reg                  clk1;
wire                 run;
wire [1:0]           bank;
wire [addr_bit-1:0]  addr_w, addrf0, addrf1, addrf2, addrf3;
wire [addr_bit-1:0]  addr_demux0, addr_demux1, addr_demux2, addr_demux3;
wire [2*width-1:0]   outf0, outf1, outf2, outf3, outb0, outb1, outb2, outb3;
wire [3:0]           write;
wire [width-1:0]     buff_in;

always @ (posedge clk)        // devide clk freq by half
begin
   if (start)
      clk1 <= 1;
   else
      clk1 <= ~clk1;
end

DFF in_FF (~clk1, in, buff_in);
defparam  in_FF.width=width;

Input_ctrl aInput_ctrl (clk1, reset, start, {bank[0],addr_w}, run, ready);

Addr_cnt aAddr_cnt (clk1, ~run, {bank, addr_w});
defparam  aAddr_cnt.addr_bit=addr_bit+2;

decoder4 w_decoder (bank, write);

mux42 data_mux ({bank[1],select}, {outf0,outb0}, {outf1,outb1}, {outf2,outb2}, {outf3,outb3},
                {data1F,data1B}, {data2F,data2B});
defparam  data_mux.width=4*width;

demux42 addr_demux ({bank[1],select} , addr1, addr2, addr_demux0, addr_demux1, addr_demux2,
                    addr_demux3);
defparam addr_demux.width=addr_bit;

RAM_in  buffer0 (clk1, reset, write[2], addrf0, {buff_in,in}, outf0, outb0);
defparam  buffer0.width=8, buffer0.addr_bit=addr_bit;

RAM_in  buffer1 (clk1, reset, write[3], addrf1, {buff_in,in}, outf1, outb1);
defparam  buffer1.width=8, buffer1.addr_bit=addr_bit;

RAM_in  buffer2 (clk1, reset, write[0], addrf2, {buff_in,in}, outf2, outb2);
defparam  buffer2.width=8, buffer2.addr_bit=addr_bit;

RAM_in  buffer3 (clk1, reset, write[1], addrf3, {buff_in,in}, outf3, outb3);
defparam  buffer3.width=8, buffer3.addr_bit=addr_bit;

mux2 amux0 (write[2], addr_demux0, addr_w, addrf0);
defparam  amux0.width=addr_bit;
```

```
mux2 amux1 (write[3], addr_demux1, addr_w, addrf1);
defparam  amux1.width=addr_bit;

mux2 amux2 (write[0], addr_demux2, addr_w, addrf2);
defparam  amux2.width=addr_bit;

mux2 amux3 (write[1], addr_demux3, addr_w, addrf3);
defparam  amux3.width=addr_bit;

endmodule
```

3) SISO1.v

```
module SISO1 (clk, reset, start, data1F, data1B, SIf, SIb, addr, addr_w, SOf, SOb, done5,
              write5);

parameter            width=4, widthf=7;
parameter            N=6, addr_bit=3;

input                clk, start, reset;
input [2*width-1:0]  data1F, data1B;
input [width-1:0]    SIf, SIb;
output [width-1:0]   SOf, SOb;
output               done5, write5;
output [addr_bit-1:0] addr, addr_w;

wire [2*width-1:0]   data1F1, data1B1;
wire                 done, done1, done2, done3, done4, write, write1, write2, write3, write4,
                      start1, start2;
wire [width-1:0]     SIf1, SIb1, SIf_mux, SIb_mux, SIf_LIFO, SIb_LIFO, SIf_LIFO2, SIb_LIFO2,
                      SIf_LIFO3, SIb_LIFO3, SIf_LIFO4, SIb_LIFO4;
wire [width+1:0]     M0f, M1f, M2f, M0b, M1b, M2b;
wire [width+1:0]     M0fg, M1fg, M2fg, M0bg, M1bg, M2bg;
wire [widthf-1:0]    F0, F1, F2, F3, B0, B1, B2, B3;
wire [widthf-1:0]    F0g, F1g, F2g, F3g, B0g, B1g, B2g, B3g;
wire [widthf-1:0]    F0_LIFO, F1_LIFO, F2_LIFO, F3_LIFO, B0_LIFO, B1_LIFO, B2_LIFO, B3_LIFO;
wire [widthf-1:0]    F0_LIFOg, F1_LIFOg, F2_LIFOg, F3_LIFOg, B0_LIFOg, B1_LIFOg, B2_LIFOg,
                      B3_LIFOg;
wire [widthf-1:0]    x0f, x1f, x2f, x3f, y0f, y1f, y2f, y3f, x0b, x1b, x2b, x3b, y0b, y1b,
                      y2b, y3b;
wire [widthf-1:0]    SO_temp1f, SO_temp0f, SO_temp1b, SO_temp0b, SOf_in, SOb_in;
wire [addr_bit-1:0]  addr1, addr2, addr3, addr4;

SISO_ctrl SISO_ctrl1 (clk, reset, start, done, write, addr);
defparam SISO_ctrl1.addr_bit=addr_bit, SISO_ctrl1.N=N;

M_SISO1 M_SISOf (clk, data1F1, SIf_mux, addr1[0], M0f, M1f, M2f);

M_SISO1 M_SISOb (clk, data1B1, SIb_mux, ~addr1[0], M0b, M1b, M2b);

F_cal F_calf (clk, start2, M0f, M1f, M2f, F0, F1, F2, F3);

B_cal B_calb (clk, start2, M0b, M1b, M2b, B0, B1, B2, B3);

Hiz_gate Hiz_gatef (write2, {M0f, M1f, M2f, F0, F1, F2, F3, B0_LIFO, B1_LIFO, B2_LIFO, B3_LIFO},
                    {M0fg, M1fg, M2fg, F0g, F1g, F2g, F3g, B0_LIFOg, B1_LIFOg, B2_LIFOg, B3_LIFOg});
defparam Hiz_gatef.width = 74;

Hiz_gate Hiz_gateb (write2, {M0b, M1b, M2b, F0_LIFO, F1_LIFO, F2_LIFO, F3_LIFO, B0, B1, B2, B3},
                    {M0bg, M1bg, M2bg, F0_LIFOg, F1_LIFOg, F2_LIFOg, F3_LIFOg, B0g, B1g, B2g, B3g});
defparam Hiz_gateb.width = 74;

Comp_A Comp_Af (clk, M0fg, M1fg, M2fg, F0g, F1g, F2g, F3g, B0_LIFOg, B1_LIFOg, B2_LIFOg, B3_LIFOg,
                x0f, x1f, x2f, x3f, y0f, y1f, y2f, y3f);

Comp_A Comp_Ab (clk, M0bg, M1bg, M2bg, F0_LIFOg, F1_LIFOg, F2_LIFOg, F3_LIFOg, B0g, B1g, B2g, B3g,
                x0b, x1b, x2b, x3b, y0b, y1b, y2b, y3b);

Comp_CS Comp_CSf (clk, x0f, x1f, x2f, x3f, y0f, y1f, y2f, y3f, SO_temp1f, SO_temp0f);
Comp_CS Comp_CSb (clk, x0b, x1b, x2b, x3b, y0b, y1b, y2b, y3b, SO_temp1b, SO_temp0b);

Sum_comp Sum_compf (clk, SO_temp1f, SO_temp0f, SIb_LIFO4, 1'b0, SOf_in);

Sum_comp Sum_compb (clk, SO_temp1b, SO_temp0b, SIf_LIFO4, 1'b0, SOb_in);

Clip Clipf (SOf_in, SOf);

Clip Clipb (SOb_in, SOb);

mux2 mux_SIf (write1, SIf1, SIb_LIFO, SIf_mux);

mux2 mux_SIb (write1, SIb1, SIf_LIFO, SIb_mux);

LIFO LIFO_F (clk, {F0, F1, F2, F3}, {F0_LIFO, F1_LIFO, F2_LIFO, F3_LIFO}, write2);
defparam LIFO_F.width=widthf*4, LIFO_F.size=N/2+1;

LIFO LIFO_B (clk, {B0, B1, B2, B3}, {B0_LIFO, B1_LIFO, B2_LIFO, B3_LIFO}, write2);
defparam LIFO_B.width=widthf*4, LIFO_B.size=N/2+1;

LIFO LIFO_SIf (clk, SIf1, SIf_LIFO, write1);
defparam LIFO_SIf.width=width, LIFO_SIf.size=N/2+1;

LIFO LIFO_SIb (clk, SIb1, SIb_LIFO, write1);
defparam LIFO_SIb.width=width, LIFO_SIb.size=N/2+1;


DFF DFF_SIf_LIFO (clk, SIf_LIFO, SIf_LIFO2);
defparam DFF_SIf_LIFO.width=width;

DFF DFF_SIb_LIFO (clk, SIb_LIFO, SIb_LIFO2);
defparam DFF_SIb_LIFO.width=width;

DFF DFF_SIf_LIFO2 (clk, SIf_LIFO2, SIf_LIFO3);
defparam DFF_SIf_LIFO2.width=width;

DFF DFF_SIb_LIFO2 (clk, SIb_LIFO2, SIb_LIFO3);
defparam DFF_SIb_LIFO2.width=width;

DFF DFF_SIf_LIFO3 (clk, SIf_LIFO3, SIf_LIFO4);
defparam DFF_SIf_LIFO3.width=width;

DFF DFF_SIb_LIFO3 (clk, SIb_LIFO3, SIb_LIFO4);
defparam DFF_SIb_LIFO3.width=width;


DFF DFF_data1F (clk, data1F, data1F1);
defparam DFF_data1F.width=2*width;

DFF DFF_data1B (clk, data1B, data1B1);
defparam DFF_data1B.width=2*width;

DFF DFF_SIf (clk, SIf, SIf1);
defparam DFF_SIf.width=width;

DFF DFF_SIb (clk, SIb, SIb1);
defparam DFF_SIb.width=width;

DFF DFF_write (clk, write, write1);
defparam DFF_write.width=1;

DFF DFF_write1 (clk, write1, write2);
defparam DFF_write1.width=1;

DFF DFF_write2 (clk, write2, write3);
defparam DFF_write2.width=1;

DFF DFF_write3 (clk, write3, write4);
defparam DFF_write3.width=1;

DFF DFF_write4 (clk, write4, write5);
```

```
defparam DFF_write4.width=1;

DFF DFF_start (clk, start, start1);
defparam DFF_start.width=1;

DFF DFF_start1 (clk, start1, start2);
defparam DFF_start1.width=1;

DFF DFF_done (clk, done, done1);
defparam DFF_done.width=1;

DFF DFF_done1 (clk, done1, done2);
defparam DFF_done1.width=1;

DFF DFF_done2 (clk, done2, done3);
defparam DFF_done2.width=1;

DFF DFF_done3 (clk, done3, done4);
defparam DFF_done3.width=1;

DFF DFF_done4 (clk, done4, done5);
defparam DFF_done4.width=1;

DFF DFF_addr (clk, addr, addr1);
defparam DFF_addr.width=addr_bit;

DFF DFF_addr1 (clk, addr1, addr2);
defparam DFF_addr1.width=addr_bit;

DFF DFF_addr2 (clk, addr2, addr3);
defparam DFF_addr2.width=addr_bit;

DFF DFF_addr3 (clk, addr3, addr4);
defparam DFF_addr3.width=addr_bit;

DFF DFF_addr4 (clk, addr4, addr_w);
defparam DFF_addr4.width=addr_bit;

endmodule
```

4) SISO2.v

```
module SISO2 (clk, reset, start, data1F, data1B, SIf, SIb, addr, addr_w, SOf, SOb, done5, write5,
              last_iter);

parameter           width=4, widthf=7;
parameter           N=6, addr_bit=3;

input               clk, start, reset, last_iter;
input [width-1:0]   data1F, data1B;   //Input for SISO2 in only the parity portion (4 bits)
input [width-1:0]   SIf, SIb;
output [width-1:0]  SOf, SOb;
output              done5, write5;
output [addr_bit-1:0] addr, addr_w;

wire [width-1:0]    data1F1, data1B1;
wire                done, done1, done2, done3, done4, write, write1, write2, write3, write4, start1,
                     start2, last_iter_ff;
wire [width-1:0]    SIf1, SIb1, SIf_mux, SIb_mux, SIf_LIFO, SIb_LIFO, SIf_LIFO2, SIb_LIFO2,
                     SIf_LIFO3, SIb_LIFO3, SIf_LIFO4, SIb_LIFO4;
wire [width+1:0]    M0f, M1f, M2f, M0b, M1b, M2b;
wire [width+1:0]    M0fg, M1fg, M2fg, M0bg, M1bg, M2bg;
wire [widthf-1:0]   F0, F1, F2, F3, B0, B1, B2, B3;
wire [widthf-1:0]   F0g, F1g, F2g, F3g, B0g, B1g, B2g, B3g;
wire [widthf-1:0]   F0_LIFO, F1_LIFO, F2_LIFO, F3_LIFO, B0_LIFO, B1_LIFO, B2_LIFO, B3_LIFO;
wire [widthf-1:0]   F0_LIFOg, F1_LIFOg, F2_LIFOg, F3_LIFOg, B0_LIFOg, B1_LIFOg, B2_LIFOg, B3_LIFOg;
wire [widthf-1:0]   x0f, x1f, x2f, x3f, y0f, y1f, y2f, y3f, x0b, x1b, x2b, x3b, y0b, y1b, y2b, y3b;
wire [widthf-1:0]   SO_temp1f, SO_temp0f, SO_temp1b, SO_temp0b, SOf_in, SOb_in;
wire [addr_bit-1:0] addr1, addr2, addr3, addr4;
```

```
Last_iter_ff Last_iter_ff1 (clk, start, last_iter, last_iter_ff);

SISO_ctrl SISO_ctrl1 (clk, reset, start, done, write, addr);
defparam SISO_ctrl1.addr_bit=addr_bit, SISO_ctrl1.N=N;

M_SISO2 M_SISOf (clk, data1F1, SIf_mux, addr1[0], M0f, M1f, M2f);

M_SISO2 M_SISOb (clk, data1B1, SIb_mux, ~addr1[0], M0b, M1b, M2b);


... same as SISO1.v ...



Sum_comp Sum_compf (clk, SO_temp1f, SO_temp0f, SIb_LIFO4, last_iter_ff, SOf_in);

Sum_comp Sum_compb (clk, SO_temp1b, SO_temp0b, SIf_LIFO4, last_iter_ff, SOb_in);


... same as SISO1.v ...
(The differences between SISO1.v and SISO2.v are shown in bold letters.)


endmodule;
```

5) Interleave.v

```
module Interleave (clk, reset, write, addr, addr_w, SOf, SOb, SIf, SIb);

parameter            width=4, addr_bit=3, N=6, coef_file="int_rom6.dat";

input                clk, reset, write;
input [addr_bit-1:0] addr, addr_w;
input [width-1:0]    SOf, SOb;
output [width-1:0]   SIf, SIb;

wire [addr_bit-1:0]  addr_f, addr_f_rom, addr_b_rom;


RAM2r data_RAM (clk, reset, write, addr_f, addr_b_rom, {SOf, SOb}, SIf, SIb);
defparam data_RAM.width=width, data_RAM.addr_bit=addr_bit;

ROM addr_ROM (addr[addr_bit-2:0], {addr_f_rom, addr_b_rom});
defparam addr_ROM.width=2*addr_bit, addr_ROM.addr_bit=addr_bit-1, addr_ROM.datafile=coef_file;

mux2 addr_muxb (write, addr_f_rom, ~addr_w, addr_f);
defparam addr_muxb.width=addr_bit;

endmodule
```

6) Deinterleave.v

```
module Deinterleave (clk, reset, write, addr, addr_w, SOf, SOb, SIf, SIb, hard_dici, out, writeout);

parameter            width=4, addr_bit=3, N=6, coef_file="deint_rom6.dat";

input                clk, reset, write, hard_dici;
input [addr_bit-1:0] addr, addr_w;
input [width-1:0]    SOf, SOb;
output [width-1:0]   SIf, SIb;
output               out, writeout;

reg                  out, writeout;
wire                 half;
reg [width-1:0]      SIb, SIf;
wire [addr_bit-1:0]  addr_f, addr_f_rom, addr_b_rom;
wire [width-1:0]     SIb_ram, SIf_ram;
wire [addr_bit-2:0]  count_out, addr_rom;

RAM2r data_RAM (clk, reset, write, addr_f, addr_b_rom, {SOf, SOb}, SIf_ram, SIb_ram);
defparam data_RAM.width=width, data_RAM.addr_bit=addr_bit;

ROM addr_ROM (addr_rom, {addr_f_rom, addr_b_rom});
defparam addr_ROM.width=2*addr_bit, addr_ROM.addr_bit=addr_bit-1, addr_ROM.datafile=coef_file;

mux2 addr_muxf (write, addr_f_rom, ~addr_w, addr_f);
defparam addr_muxf.width=addr_bit;

mux2 addr_rom_mux (writeout, addr[addr_bit-2:0], count_out, addr_rom);
defparam addr_rom_mux.width=addr_bit-1;

Out_cnt out_counter (clk, ~writeout, half, count_out);
defparam out_counter.addr_bit = addr_bit;

// Generate writeout (active during final decision output)
always @ (posedge clk)
begin
   if (hard_dici || writeout)
      writeout <= 1;
   else
      writeout <= 0;

   if (half && count_out==2)      // reach the last output position
      writeout <= 0;
end


// Output mux
always @ (writeout or SIb_ram or SIf_ram or half)
   if (writeout)
      begin
         if (half)
            out = SIb_ram[width-1];
         else
            out = SIf_ram[width-1];

         SIb = 0;
         SIf = 0;
      end
   else
      begin
         out = 0;
         SIf = SIf_ram;
         SIb = SIb_ram;
      end

endmodule
```

7) Input_ctrl.~~dat~~✓ (used in Input_buff.v)

```
module Input_ctrl (clk, reset, start, count, run, ready);
parameter addr_bit=3, N=6;

input                 clk, start, reset;
input [addr_bit:0]    count;
output                run, ready;
reg                   run, ready, buffer_ready;

parameter   IDLE  = 0,
            S1    = 1,
            S2    = 2;

reg [1:0]  STATE, NEXT_STATE;


always @ (posedge clk or posedge reset)
```

```
   if (reset)
      STATE <= IDLE;
   else
      STATE <= NEXT_STATE;


always @ (STATE or start or count)
begin
   case (STATE)
      IDLE:
         if (start)  NEXT_STATE <= S1;
             else    NEXT_STATE <= IDLE;

      S1:
         if (count==2*N+3)
            NEXT_STATE <= S2;
         else
            NEXT_STATE <= S1;

      S2:
         if (count==2*N+3)
            NEXT_STATE <= S1;
         else
            NEXT_STATE <= S2;

   endcase
end


always @ (STATE or count)
begin
   case (STATE)
      IDLE:
         begin
            run = 0;
            ready = 0;
            buffer_ready = 0;
         end

      S1:
         begin
            run = 1;
            if (buffer_ready == 1)          //identify whether it is from IDLE or S2
               if (count==0) ready = 1;
               else ready = 0;
            else
               ready = 0;
         end

      S2:
         begin
            run = 1;
            if (count==0) ready = 1;
            else ready = 0;
            buffer_ready = 1;
         end
   endcase
end

endmodule
```

8) Addr_cnt.v (used in Input_buff.v and Deinterleaver.v)

```
module Addr_cnt (clk, clear, addr);

parameter            addr_bit=3;

input                clk, clear;
output [addr_bit-1:0] addr;
reg [addr_bit-1:0]   addr;
```

```
always @ (posedge clk)
begin
   if (clear)          //sync clear
      addr <= 0;
   else
      addr <= addr + 1;
end

endmodule
```

9) RAM_in.v   (used in Input_buff.v)

```
module RAM_in (clk, reset, write, addra, in, outa, outb);
parameter  width=4, addr_bit=3;
parameter  size=1<<addr_bit;
parameter  N=6;

input                clk, reset, write;
input [addr_bit-1:0] addra;
input [width-1:0]    in;
output [width-1:0]   outa, outb;
reg [width-1:0]      outa, outb;

reg [addr_bit-1:0]   addrb;
reg [width-1:0]      M[0:size-1];
integer              i;


// create backward addr (addrb) from received forward address (addra)
always @ (addra)
   addrb = ~addra;


// write process sync to clk and uses addra, write 2 adjacent locations at a time
// in 1 port, 8 bits
always @ (posedge clk or posedge reset)
   if (reset)
      for (i=0; i<size; i=i+1)
          M[i] = 0;
   else if (write)
         M[addra] = in;

// read process async, 2 port 4 bits each, pointed by addra and addrb
always @ (addra or addrb or write)
   if (~write)
      begin
         outa = M[addra];
         outb = M[addrb];
      end

endmodule
```

12) decode4.v   (used in Input_buff.v)

```
module decoder4 (in, out);

input [1:0]          in;
output [3:0]         out;
reg [3:0]            out;

always @ (in)
   case (in)
      0:  out = 4'b0001;
      1:  out = 4'b0010;
      2:  out = 4'b0100;
      3:  out = 4'b1000;
   endcase
endmodule
```

13) mux42.v (used in Input_buff.v)

```
module mux42 (sel, in0, in1, in2, in3, out0, out1);
parameter  width=4;

input [1:0]          sel;
input [width-1:0]    in0, in1, in2, in3;
output [width-1:0]   out0, out1;
reg [width-1:0]      out0, out1;

always @ (sel or in0 or in1 or in2 or in3)
   case (sel)
      0: begin
            out0 = in0;
            out1 = in1;
         end

      1: begin
            out0 = in1;
            out1 = in0;
         end

      2: begin
            out0 = in2;
            out1 = in3;
         end

      3: begin
            out0 = in3;
            out1 = in2;
         end

   endcase
endmodule
```

14) demux42.v (used in Input_buff.v)

```
module demux42 (sel, in0, in1, out0, out1, out2, out3);
parameter  width=4;

input [1:0]          sel;
input [width-1:0]    in0, in1;
output [width-1:0]   out0, out1, out2, out3;
reg [width-1:0]      out0, out1, out2, out3;

always @ (sel or in0 or in1)
   case (sel)
      0: begin
            out0 = in0;
            out1 = in1;
            out2 = {width{1'hz}};
            out3 = {width{1'hz}};
         end

      1: begin
            out0 = in1;
            out1 = in0;
            out2 = {width{1'hz}};
            out3 = {width{1'hz}};
         end

      2: begin
            out1 = {width{1'hz}};
            out2 = {width{1'hz}};
            out2 = in0;
            out3 = in1;
         end

      3: begin
            out1 = {width{1'hz}};
            out2 = {width{1'hz}};
            out2 = in1;
            out3 = in0;
         end
   endcase
endmodule
```

15) mux2.v (used in many files)

```
module mux2 (sel, in0, in1, out);
parameter  width=4;

input                 sel;
input [width-1:0]     in0, in1;
output [width-1:0]    out;
reg [width-1:0]       out;

always @ (sel or in0 or in1)
   if (sel)
      out = in1;
   else
      out = in0;

endmodule
```

16) DFF.v (used in many files)

```
module DFF (clk, in, out);

parameter             width=4;

input                 clk;
input [width-1:0]     in;
output [width-1:0]    out;
reg [width-1:0]       out;

always @ (posedge clk)
begin
   out <= in;
end

endmodule
```

17) SISO_ctrl.v (used in SISO1.v and SISO2.v)

```
module SISO_ctrl (clk, reset, start, done, write, addr);

parameter             N=6, addr_bit=3;

input                 clk, start, reset;
output                done, write;
wire                  reset_cnt;
output [addr_bit-1:0] addr;

SISO_ctrl_gen  ctrl_gen1 (clk, reset, start, done, write, reset_cnt, addr);

Addr_cnt  Addr_cnt1 (clk, reset_cnt, addr);

Endmodule
```

18) SISO_ctrl_gen.v (used in SISO_ctrl.v)

```
module SISO_ctrl_gen (clk, reset, start, done, write, reset_cnt, addr);

parameter             N=6, addr_bit=3;

input                 clk, start, reset;
input [addr_bit-1:0]  addr;
output                done, write, reset_cnt;
```

```
reg                 done, write, reset_cnt;

parameter   IDLE   = 0,
            F_B    = 1,
            COMP   = 2;

reg [1:0]  STATE, NEXT_STATE;


always @ (posedge clk or posedge reset)
   if (reset)
      STATE <= IDLE;
   else
      STATE <= NEXT_STATE;


always @ (STATE or start or addr)
begin

   NEXT_STATE <= IDLE;      //default for unspecified conditions

   case (STATE)
      IDLE:
         if (start)  NEXT_STATE <= F_B;
             else    NEXT_STATE <= IDLE;

      F_B:
         if (addr==N/2) NEXT_STATE <= COMP;
             else     NEXT_STATE <= F_B;

      COMP:
         if (addr==N+1) NEXT_STATE <= IDLE;
             else    NEXT_STATE <= COMP;
   endcase
end


always @ (STATE or addr)
begin

   done = 0;          // default for unspecified conditions
   write = 0;
   reset_cnt = 0;

   case (STATE)
      IDLE:
         reset_cnt = 1;

      COMP:
      begin
         write = 1;
         if (addr==N)
            done = 1;
         else
            done = 0;
      end
   endcase
end

endmodule
```

19) M_SISO1.v (used in SISO1.v)

```
module M_SISO1 (clk, data, S1, k, M0, M1, M2);

parameter             width=4;

input                 clk, k;
input [2*width-1:0]   data;
input [width-1:0]     SI;
output [width+1:0]    M0, M1, M2;

reg [width+1:0]       M0, M1, M2;
reg [width-1:0]       z1, z2;

always @ (posedge clk)
begin

   z1 = data[2*width-1:width];

   if (k == 0)     // k even
      z2 = data[width-1:0];
   else
      z2 = 0;

   M0 = signext(SI) + signext(z1) + signext(z2);
   M1 = signext(z2);
   M2 = signext(SI) + signext(z1);

end

function [width+1:0] signext;     // extend 2 bits of sign
input [width-1:0] a;

   signext = {{2{a[width-1]}}, a);

endfunction

endmodule
```

20) M_SISO2.v (used in SISO2.v)

```
module M_SISO2 (clk, data2, SI, k, M0, M1, M2);

parameter             width=4;

input                 clk, k;
input [width-1:0]     data2;    //only 4 LSBs of data, SISO2 doesn't use system bit info (z1)
input [width-1:0]     SI;
output [width+1:0]    M0, M1, M2;

reg [width+1:0]       M0, M1, M2;
reg [width-1:0]       z2;

always @ (posedge clk)
begin

   if (k == 1)     // k odd
      z2 = data2;
   else
      z2 = 0;

   M0 = signext(SI) + signext(z2);
   M1 = signext(z2);
   M2 = signext(SI);

end

function [width+1:0] signext;      // extend 2 bits of sign
input [width-1:0] a;

   signext = {{2{a[width-1]}}, a);

endfunction

endmodule
```

21) F_cal.v (used in SISO1.v and SISO2.v)

```
module F_cal (clk, clear, M0, M1, M2, F0, F1, F2, F3);

parameter            width=4, widthf=7;
parameter            Init = 0, Inf = 31;

input                clk, clear;
input [width+1:0]    M0, M1, M2;
output [widthf-1:0]  F0, F1, F2, F3;

reg [widthf-1:0]     F0, F1, F2, F3;

always @ (posedge clk)
begin
   if (clear)
      begin
         F0 <= Init;
         F1 <= Inf;
         F2 <= Inf;
         F3 <= Inf;
      end
   else
      begin
         F0 <= min2(F0,              F2 + signext(M0));
         F1 <= min2(F0 + signext(M0), F2);
         F2 <= min2(F1 + signext(M2), F3 + signext(M1));
         F3 <= min2(F1 + signext(M1), F3 + signext(M2));
      end
end

function [widthf-1:0] min2;
input [widthf-1:0]  a, b;
reg   [widthf-1:0]  diff;

begin
   diff = a-b;
   min2 = diff[widthf-1] ?  a : b;
end

endfunction

function [width+2:0] signext;    //extend 1 bit of sign
input [width+1:0] a;

   signext = {{a[width+1]}, a};

endfunction

endmodule
```

22) B_cal.v (used in SISO1.v and SISO2.v)

```
module B_cal (clk, clear, M0, M1, M2, B0, B1, B2, B3);

parameter            width=4, widthf=7;
parameter            Init = 0, Inf = 31;

input                clk, clear;
input [width+1:0]    M0, M1, M2;
output [widthf-1:0]  B0, B1, B2, B3;

reg [widthf-1:0]     B0, B1, B2, B3;

always @ (posedge clk)
begin
   if (clear)
      begin
         B0 <= Init;
         B1 <= Inf;
         B2 <= Inf;
         B3 <= Inf;
         end
      else
         begin
            B0 <= min2(B0,                B1 + signext(M0));
            B1 <= min2(B3 + signext(M1), B2 + signext(M2));
            B2 <= min2(B1               , B0 + signext(M0));
            B3 <= min2(B2 + signext(M1), B3 + signext(M2));
         end
end

function [widthf-1:0] min2;
input [widthf-1:0]  a, b;
reg   [widthf-1:0]  diff;

begin
   diff = a-b;
   min2 = diff[widthf-1] ?  a : b;
end

endfunction

function [width+2:0] signext;     //extend 1 bit of sign
input [width+1:0] a;

   signext = {{a[width+1]}, a};

endfunction

endmodule
```

23) Comp_A.v (used in SISO1.v and SISO2.v)

```
module Comp_A (clk, M0, M1, M2, F0, F1, F2, F3, B0, B1, B2, B3, x0, x1, x2, x3, y0, y1, y2, y3);

parameter             width=4, widthf=7;

input                 clk;
input [width+1:0]     M0, M1, M2;
input [widthf-1:0]    F0, F1, F2, F3, B0, B1, B2, B3;
output [widthf-1:0]   x0, x1, x2, x3, y0, y1, y2, y3;

reg [widthf-1:0]      x0, x1, x2, x3, y0, y1, y2, y3;

always @ (posedge clk)
begin

   // output to be compared to generate SO_templ (SO(uk=1))
   x0 = F0 + signext(M0) + B1;
   x1 = F1 + signext(M2) + B2;
   x2 = F2 + signext(M0) + B0;
   x3 = F3 + signext(M2) + B3;

   // output to be compared to generate SO_templ (SO(uk=0))
   y0 = F0 + B0;
   y1 = F1 + signext(M1) + B3;
   y2 = F2 + B1;
   y3 = F3 + signext(M1) + B2;

end


function [width+2:0] signext;     //extend 1 bit of sign
input [width+1:0] a;

   signext = {{a[width+1]}, a};

endfunction

endmodule
```

24) Comp_CS.v (used in SISO1.v and SISO2.v)

```
module Comp_CS (clk, x0, x1, x2, x3, y0, y1, y2, y3, SO_temp1, SO_temp0);

parameter            width=4, widthf=7;

input                clk;
input [widthf-1:0]   x0, x1, x2, x3, y0, y1, y2, y3;
output [widthf-1:0]  SO_temp1, SO_temp0;

reg [widthf-1:0]     SO_temp1, SO_temp0;

always @ (posedge clk)
begin

   SO_temp1 = min4(x0, x1, x2, x3);

   SO_temp0 = min4(y0, y1, y2, y3);

end

function [widthf-1:0] min4;
input [widthf-1:0]  a, b, c, d;
reg   [widthf-1:0]  a_b, a_c, a_d, b_c, b_d, c_d;
reg                 amin, bmin, cmin;

begin
   a_b = a - b;
   a_c = a - c;
   a_d = a - d;
   b_c = b - c;
   b_d = b - d;
   c_d = c - d;

   amin =  a_b[widthf-1] &  a_c[widthf-1] &  a_d[widthf-1];
   bmin = ~a_b[widthf-1] &  b_c[widthf-1] &  b_d[widthf-1];
   cmin = ~a_c[widthf-1] & ~b_c[widthf-1] &  c_d[widthf-1];

   if (amin)
      min4 = a;
   else if (bmin)
      min4 = b;
   else if (cmin)
      min4 = c;
   else
      min4 = d;
end

endfunction

endmodule
```

25) Sum_comp.v (used in SISO1.v and SISO2.v)

```
module Sum_comp (clk, SO_temp1, SO_temp0, SI, last_iter, SO);

parameter            width=4, widthf=7;

input [widthf-1:0]   SO_temp1, SO_temp0;
input [width-1:0]    SI;
input                clk, last_iter;
output [widthf-1:0]  SO;

reg [widthf-1:0]     SO;
reg [widthf-1:0]     SI_temp;

always @ (posedge clk)
begin
```

```
   if (last_iter)
      SI_temp = 0;
   else
      SI_temp = signext(SI);

   SO= SO_temp1 - SO_temp0 - SI_temp;

end

function [widthf-1:0] signext;    // extend 3 bits of sign
input [width-1:0] a;

   signext = {{3{a[width-1]}}, a};

endfunction

endmodule
```

26) Clip.v (used in SISO1.v and SISO2.v)

```
module Clip (SO_in, SO_out);

parameter             width=4, widthf=7;

input [widthf-1:0]    SO_in;
output [width-1:0]    SO_out;

reg [width-1:0]       SO_out;

always @ (SO_in)
begin

   if (&SO_in[widthf-1:widthf-width] || ~|SO_in[widthf-1:widthf-width])
      SO_out = SO_in[width-1:0];
   else if (SO_in[widthf-1])
      SO_out = -8;
   else
      SO_out = 7;

end

endmodule
```

27) LIFO.v (used in SISO1.v and SISO2.v)

```
module LIFO (clk, in, out, read);

parameter             width=4, size=4;

input                 clk, read;
input [width-1:0]     in;
output [width-1:0]    out;

reg [width-1:0]       out;
reg [width-1:0]       mem [size-1:0];
integer               i;

always @ (posedge clk)
begin
   if (read)
      begin
         mem[size-1] <= mem[0];
         for (i=0; i<size-1; i=i+1)
            mem[i] <= mem[i+1];
      end
   else
      begin
         mem[0] <= in;
         for (i=1; i<size; i=i+1)
            mem[i] <= mem[i-1];
      end
end

always @ (mem[0])
   out = mem[0];

endmodule
```

28) Last_iter_ff.v (used in Deinterleaver.v)

```
module Last_iter_ff (clk, start, last_iter, out);

input          clk, start, last_iter;
output         out;
reg            out;

always @ (posedge clk)
   if (start)
      out <= last_iter;

endmodule
```

29) RAM2r.v (used in Interleaver.v and Deinterleaver.v)

```
module RAM2r (clk, reset, write, addra, addrb, in, outa, outb);
parameter  width=4, addr_bit=3;
parameter  size=1<<addr_bit;
parameter  N=6;

input                clk, reset, write;
input [addr_bit-1:0] addra, addrb;
input [2*width-1:0]  in;
output [width-1:0]   outa, outb;
reg [width-1:0]      outa, outb;

reg [width-1:0]      M[0:size-1];
integer              i;

// write process sync to clk and uses addra, write 2 adjacent locations at a time
// in 1 port, 8 bits
always @ (posedge clk or posedge reset)
   if (reset)
      for (i=0; i<size; i=i+1)
          M[i] = 0;
   else if (write)
      begin
         M[2*addra] = in[width-1:0];
         if (addra <= 1)
            M[2*addra+1] = 0;              // force tail bit locations to be zero
         else
            M[2*addra+1] = in[2*width-1:width];
      end

// read process async, 2 port 4 bits each
always @ (addra or addrb or write)
   if (~write)
      begin
         outa = M[addra];
         outb = M[addrb];
      end

endmodule
```

30) ROM.v (used in Deinterleaver.v)

```
module ROM (addr, out);
parameter  width=4, addr_bit=2, datafile="file.dat";
parameter  size = 1<<addr_bit;
```

```
input [addr_bit-1:0] addr;
output [width-1:0]   out;
reg [width-1:0]      out;

reg [width-1:0]      M[0:size-1];

initial
   $readmemb (datafile, M);

always @ (addr)
   out = M[addr];

endmodule
```

31) Out_cnt.v (used in Interleaver.v and Deinterleaver.v)

```
module Out_cnt (clk, clear, half, out);

parameter               addr_bit=3;

input                   clk, clear;
output                  half;
output [addr_bit-2:0]   out;
reg [addr_bit-1:0]      addr;
reg [addr_bit-2:0]      out;
reg                     half;

always @ (posedge clk)
begin
   if (clear)          //sync clear
      addr <= 0;
   else
      addr <= addr + 1;
end

always @ (addr)
begin
   half = addr[addr_bit-1];
   if (half)
      out = ~addr[addr_bit-2:0];
   else
      out = addr[addr_bit-2:0];
end

endmodule
```

32) vec61.dat

```
// Test vector SI(Ck1=1) for block length = 6
// block 1 => 5, 2, 2, 0, -3, 3, 0, 1
// block 2 => 4, 1, 0, -2, -6, 4, -1, -5
        0101
        0010
        0010
        0000
        1101
        0011
        0000
        0001
        0100
        0001
        0000
        1110
        1010
        0100
        1111
        1011
```

vec62.dat

```
// Test vector SI(Ck1=2) for block length = 6
// block 1 => 0, 1, -5, 6, -1, -6, 0, -2
// block 2 => 5, 3, -3, -5, 6, -1, 2, 3
        0000
        0001
        1011
        0110
        1111
        1010
        0000
        1110
        0101
        0011
        1101
        1011
        0110
        1111
        0010
        0011
```

33) Int_rom.dat (block length = 6)

```
110001
100011
101010
000111
```

33) Deint_rom.dat (block length = 6)

```
110001
101011
010100
000111
```

Section 7 Matlab Codes for Test Vector Generation

```
% MATLAB Code for Turbo Decoder
%
% M(k,1) = M(0,0,0) = M(2,0,1)
% M(k,2) = M(0,1,1) = M(2,1,0)
% M(k,3) = M(1,0,3) = M(3,0,2)
% M(k,4) = M(1,1,2) = M(3,1,3)
%
% Note on array indexes:
% MATLAB supports only indexes from 1, 2, 3, ... So, I define two
% variable to account for the idexes of M, F, and B.
%       k          is the base time index
%       kk = k+1   is the time index for M and B
%       kf = k+2 is the time index for F
% Using kk and kf, we can specify M, F, and B the same way as described
% in the Turbo Decoder algorithm.

clear
Init=0; Inf=31;

% Data for block length 1024 (read from test vector files)
N=1024;
load ck0.dat; load ck1.dat;
load interleaver.dat;
z1 = bin2sign(ck0,4);  z2 = bin2sign(ck1,4);
Interleave = bin2dec(interleaver,10);

% Data for block length 6
%N=6;
%z1=[5 2 2 0 -3 3 0 1];  z2=[0 1 -5 6 -1 -6 0 -2];          %block 1
%z1 = [4 1 0 -2 -6 4 -1 -5]; z2 = [5 3 -3 -5 6 -1 2 3];     %block 2
%Interleave=[3, 2, 5, 0, 4, 1];

Interleave = Interleave + 1;          % Add +1 offset to make non-zero indexes
Deinterleave(Interleave) = 1:N;
SI1=zeros(1,N+2);
SI2=zeros(1,N+2);
No_of_iter = 10;

for i = 1:No_of_iter;       (Main Loop)

%SISO1 Transition Matrix
for k=0:N+1
   kk=k+1;
   if rem(k,2) == 0                     %k even
      M(kk,1) = 0;
      M(kk,2) = SI1(kk) + z1(kk) + z2(kk);
      M(kk,3) = z2(kk);
      M(kk,4) = SI1(kk) + z1(kk);
   else                                 %k odd
      M(kk,1) = 0;
      M(kk,2) = SI1(kk) + z1(kk);
      M(kk,3) = 0;
      M(kk,4) = SI1(kk) + z1(kk);
   end
end

%SISO1 Forward & Backward ACS
F(1,:) = [Init Inf Inf Inf];
for k=0:N+1
   kk=k+1;
   kf=k+2;
   F(kf,1) = min( F(kf-1,1)+M(kk,1), F(kf-1,3)+M(kk,2));
   F(kf,2) = min( F(kf-1,1)+M(kk,2), F(kf-1,3)+M(kk,1));
   F(kf,3) = min( F(kf-1,2)+M(kk,4), F(kf-1,4)+M(kk,3));
   F(kf,4) = min( F(kf-1,2)+M(kk,3), F(kf-1,4)+M(kk,4));
end

B(N+3,:) = [Init Inf Inf Inf];
```

```
for k=N+1:-1:0
   kk=k+1;
   B(kk,1) = min( B(kk+1,1)+M(kk,1), B(kk+1,2)+M(kk,2));
   B(kk,2) = min( B(kk+1,4)+M(kk,3), B(kk+1,3)+M(kk,4));
   B(kk,3) = min( B(kk+1,2)+M(kk,1), B(kk+1,1)+M(kk,2));
   B(kk,4) = min( B(kk+1,3)+M(kk,3), B(kk+1,4)+M(kk,4));
end

%SISO1 Completion
for k=0:N+1
   kk=k+1;
   kf=k+2;
   SO1(kk) = min([F(kf-1,1) + M(kk,2) + B(kk+1,2), ...
                  F(kf-1,2) + M(kk,4) + B(kk+1,3), ...
                  F(kf-1,3) + M(kk,2) + B(kk+1,1), ...
                  F(kf-1,4) + M(kk,4) + B(kk+1,4)]) ...
           - min([F(kf-1,1) + M(kk,1) + B(kk+1,1), ...
                  F(kf-1,2) + M(kk,3) + B(kk+1,4), ...
                  F(kf-1,3) + M(kk,1) + B(kk+1,2), ...
                  F(kf-1,4) + M(kk,3) + B(kk+1,3)]) - SI1(kk);
end

%clipping soft information output
SO1 = (SO1<=maxs).*SO1 + (SO1>maxs)*maxs;
SO1 = (SO1>=mins).*SO1 + (SO1<mins)*mins;

%Interleave SI1 for SISO2
SI2 = SO1(Interleave);
SI2 = [SI2, 0, 0];

%SISO2 Transition Matrix
for k=0:N+1
   kk=k+1;
   if rem(k,2) == 0                    %k even
      M(kk,1) = 0;
      M(kk,2) = SI2(kk);
      M(kk,3) = 0;
      M(kk,4) = SI2(kk);
   else                                %k odd
      M(kk,1) = 0;
      M(kk,2) = SI2(kk) + z2(kk);
      M(kk,3) = z2(kk);
      M(kk,4) = SI2(kk);
   end
end

%SISO2 Forward & Backward ACS
F(1,:) = [Init Inf Inf Inf];
for k=0:N+1
   kk=k+1;
   kf=k+2;
   F(kf,1) = min( F(kf-1,1)+M(kk,1), F(kf-1,3)+M(kk,2));
   F(kf,2) = min( F(kf-1,1)+M(kk,2), F(kf-1,3)+M(kk,1));
   F(kf,3) = min( F(kf-1,2)+M(kk,4), F(kf-1,4)+M(kk,3));
   F(kf,4) = min( F(kf-1,2)+M(kk,3), F(kf-1,4)+M(kk,4));
end

B(N+3,:) = [Init Inf Inf Inf];
for k=N+1:-1:0
   kk=k+1;
   B(kk,1) = min( B(kk+1,1)+M(kk,1), B(kk+1,2)+M(kk,2));
   B(kk,2) = min( B(kk+1,4)+M(kk,3), B(kk+1,3)+M(kk,4));
   B(kk,3) = min( B(kk+1,2)+M(kk,1), B(kk+1,1)+M(kk,2));
   B(kk,4) = min( B(kk+1,3)+M(kk,3), B(kk+1,4)+M(kk,4));
end

%SISO2 Completion
for k=0:N+1
   kk=k+1;
   kf=k+2;
   SO2(kk) = min([F(kf-1,1) + M(kk,2) + B(kk+1,2), ...
                    F(kf-1,2) + M(kk,4) + B(kk+1,3), ...
                    F(kf-1,3) + M(kk,2) + B(kk+1,1), ...
                    F(kf-1,4) + M(kk,4) + B(kk+1,4)]) ...
            - min([F(kf-1,1) + M(kk,1) + B(kk+1,1), ...
                    F(kf-1,2) + M(kk,3) + B(kk+1,4), ...
                    F(kf-1,3) + M(kk,1) + B(kk+1,2), ...
                    F(kf-1,4) + M(kk,3) + B(kk+1,3)]) - SI2(kk);

   if i == No_of_iter
      SO2(kk) = SO2(kk) + SI2(kk);
   end
end

%clipping soft information output
SO2 = (SO2<=maxs).*SO2 + (SO2>maxs)*maxs;
SO2 = (SO2>=mins).*SO2 + (SO2<mins)*mins;

% print out soft information input and output for each iteration
fprintf('\nInteration #%d\n',i);
    SI1
    SO1
    SI2
    SO2

%Deinterleave SI2 for SISO1
SI1 = SO2(Deinterleave);
SI1 = [SI1, 0, 0];

end  %i        ( End Main Loop )


%Final Hard Decision
SIout = SI1(1:N)
out = SIout < 0
```

Results of Block 1

```
Interation #1
SI1 =      0     0     0     0     0     0     0     0
SO1 =      4     0     0     0    -4     4     1     4
SI2 =      0     0     4     4    -4     0     0     0
SO2 =      3     4     0     0     0     3     4    -3

Interation #2
SI1 =      0     3     4     3     0     0     0     0
SO1 =      2    -4    -2     1    -3     3     2     1
SI2 =      1    -2     3     2    -3    -4     0     0
SO2 =     -3     0     0    -4     1     1    -3    -3

Interation #3
SI1 =     -4     1     0    -3     1     0     0     0
SO1 =      4     1     0    -1    -3     2     2     0
SI2 =     -1     0     2     4    -3     1     0     0
SO2 =      2     1    -1     0    -1     2     4    -3

Interation #4
SI1 =      0     2     1     2    -1    -1     0     0
SO1 =      3    -2    -1    -1    -3     4     0     2
SI2 =     -1    -1     4     3    -3    -2     0     0
SO2 =      0     0    -1    -4     1     1    -1    -1

Interation #5
SI1 =     -4     1     0     0     1    -1     0     0
SO1 =      4     0     0    -1    -2     2     1     0
SI2 =     -1     0     2     4    -2     0     0     0
SO2 =      2     1    -1    -1     0     2     2    -2
Interation #6
SI1 =     -1     2     1     2     0    -1     0     0
SO1 =      3    -2    -1    -1    -3     4     0     2
SI2 =     -1    -1     4     3    -3    -2     0     0
```

```
SO2 =      0      0     -1     -4      1      1     -1     -1

Interation #7
SI1 =     -4      1      0      0      1     -1      0      0
SO1 =      4      0      0     -1     -2      2      1      0
SI2 =     -1      0      2      4     -2      0      0      0
SO2 =      2      1     -1     -1      0      2      2     -2

Interation #8
SI1 =     -1      2      1      2      0     -1      0      0
SO1 =      3     -2     -1     -1     -3      4      0      2
SI2 =     -1     -1      4      3     -3     -2      0      0
SO2 =      0      0     -1     -4      1      1     -1     -1

Interation #9
SI1 =     -4      1      0      0      1     -1      0      0
SO1 =      4      0      0     -1     -2      2      1      0
SI2 =     -1      0      2      4     -2      0      0      0
SO2 =      2      1     -1     -1      0      2      2     -2

Interation #10
SI1 =     -1      2      1      2      0     -1      0      0
SO1 =      3     -2     -1     -1     -3      4      0      2
SI2 =     -1     -1      4      3     -3     -2      0      0
SO2 =     -1     -1      3     -1     -2     -1     -1     -1

SIout =     -1     -1     -1     -1     -2      3

out   =      1      1      1      1      1      0
```

Results of Block 2

```
Interation #1
SI1 =      0      0      0      0      0      0      0      0
SO1 =      7      3      3     -3     -6      5     -5     -5
SI2 =     -3      3      5      7     -6      3      0      0
SO2 =     -2     -8      0      3     -3      4     -6      7

Interation #2
SI1 =      3      4     -8     -2     -3      0      0      0
SO1 =      4     -2      7      1      1     -1     -1     -1
SI2 =      1      7     -1      4      1     -2      0      0
SO2 =      4      2     -5      2      2     -1      3      5

Interation #3
SI1 =      2     -1      2      4      2     -5      0      0
SO1 =      5      1     -2     -4     -4      4      1     -5
SI2 =     -4     -2      4      5     -4      1      0      0
SO2 =     -4     -8      6      5     -1      4     -5      5

Interation #4
SI1 =      5      4     -8     -4     -1      6      0      0
SO1 =      4      0      7      1      2      0      1     -2
SI2 =      1      7      0      4      2      0      0      0
SO2 =      4      1     -4      0      0     -2      2      4

Interation #5
SI1 =      0     -2      1      4      0     -4      0      0
SO1 =      6      1     -2     -3     -5      5     -1     -5
SI2 =     -3     -2      5      6     -5      1      0      0
SO2 =     -4     -8      5      6     -1      4     -5      5

Interation #6
SI1 =      6      4     -8     -4     -1      5      0      0
SO1 =      4      0      7      1      2      0      1     -2
SI2 =      1      7      0      4      2      0      0      0
SO2 =      4      1     -4      0      0     -2      2      4
```

```
Interation #7
SI1 =      0     -2      1      4      0     -4      0      0
SO1 =      6      1     -2     -3     -5      5     -1     -5
SI2 =     -3     -2      5      6     -5      1      0      0
SO2 =     -4     -8      5      6     -1      4     -5      5

Interation #8
SI1 =      6      4     -8     -4     -1      5      0      0
SO1 =      4      0      7      1      2      0      1     -2
SI2 =      1      7      0      4      2      0      0      0
SO2 =      4      1     -4      0      0     -2      2      4

Interation #9
SI1 =      0     -2      1      4      0     -4      0      0
SO1 =      6      1     -2     -3     -5      5     -1     -5
SI2 =     -3     -2      5      6     -5      1      0      0
SO2 =     -4     -8      5      6     -1      4     -5      5

Interation #10
SI1 =      6      4     -8     -4     -1      5      0      0
SO1 =      4      0      7      1      2      0      1     -2
SI2 =      1      7      0      4      2      0      0      0
SO2 =      5      7     -4      4      2     -2      2      4

SIout =      4     -2      7      5      2     -4

out   =      0      1      0      0      0      1
```

gencoeff.m to calculate the interleaver coefficient to be saved in ROM for stuctural RTL

```
N=6; addr_bit=3; interleave = [3 2 5 0 4 1];
%N=1024; addr_bit=11; load interleaver.dat; interleave=bin2dec(interleaver',10);
output_file='int_rom6.dat'

N=N+2;
interleave = interleave + 1;          %offset by 1
interleave = [interleave, N-1, N];    %add 2 coeff for tail bits

forw = zeros(1,N);
forw(1:2:N) = 1:N/2;          %forw = [1 0 2 0 3 0 ...]

back = zeros(1,N);
back(2:2:N) = N:-1:N/2+1;   %back = [0 N 0 N-1 0 N-2 0 ...]

for i = 1:N
   coeff(i) = find(interleave(i) == forw_back);
end

coeff = coeff - 1;     % take off offset 1

% combine coefficients in pairs and write to output_file
fid = fopen(output_file,'w');
for i = 1:2:N
   fprintf(fid,'%s%s\n', dec2bin(coeff(forw_back(i)),addr_bit), dec2bin(coeff(forw_back
(i+1)),addr_bit));
end
fclose(fid);


% function dec2bin.m to translate decimal to binary (B bits)
function out = dec2bin(in, B)
out = 0;
for i = 0:B-1
   bit = rem(in, 2);
   out = out*10 + bit;
   in = in - bit;
   in = in / 2;
end
```

```
% function dec2bin.m to translate binary (B bits) to decimal
function out = bin2dec(in, B)
out = 0;
for i = 0:B-1
   bit = rem(in, 10);
   out = out + bit*2^i;
   in = in - bit;
   in = in / 10;
end

% function dec2bin.m to translate 2's complement (B bits) to decimal
function out = bin2sign(in, B)
out = 0;
for i = 0:B-2
   bit = rem(in, 10);
   out = out + bit*(2^i);
   in = in - bit;
   in = in / 10;
end

bit = rem(in, 10);
out = out - bit*(2^(i+1));
```

Reducing Power Consumption of Turbo Decoder [4]

Introduction

One of the disadvantages of Turbo codes is its complexity in the decoding scheme. The SISO algorithm of a Turbo decoder is known to be twice as complex as a classical Viterbi decoder. Furthermore, a Turbo decoder requires many iterations to achieve a good performance. The huge amount of processing in turbo decoding has a major impact on power consumption especially in mobile applications. Therefore, people have tried to find ways to reduce the power consumption without sacrifying the performance. Leung et. al. [4] has proposed a method using adaptive iteration with variable supply voltage, which will be summarized in this report.

Adaptive Iteration Decoding Algorithm

The idea is that the number of iteration required to achieve a good performance should depend on SNR of the received signal. Since SNR is unknown and changing, a fixed number of iteration might be doing too much or too little for a particular situation. Therefore, we would like to adaptively determine the condition of the signal and stop the SISO iteration accordingly.

One easy way to check the signal condition is to embed a CRC (cyclic redundancy checking) in each block of data and perform a CRC check after every iteration of SISOs. If the sequence pass the CRC check, the Turbo decoding then stops at this iteration. Since the CRC check requires very little processing power, the overhead is trivial.

Power Reduction Scheme for Adaptive Iteration

If the required iteration were fewer, that would mean we could run the Turbo decoding slower and use a lower supply voltage to further reduce the power consumption. Basically, we want to make the supply voltage changing adaptively according to the signal condition. However, we need to know the required iteration for each data frame ahead of the start of processing. Using CRC would not work for this problem although we may use the number of iteration of a previous frame to help judging the condition of the current frame, given that the signal condition is not changing too rapidly.

Another scheme is called "as slow as possible" assignment, which combines with CRC check to stop processing a frame as described above. Given that there are input buffers for some frames, if we can finish processing a current frame earlier (determined by CRC), we will have more time to process the next time before the deadline of the next frame comes. Thus, the scheme is very simple, that is, process a frame at as slow a speed as possible. In this sense, we know ahead of each frame what speed to use and, thus, use a supply voltage accordingly.

SNR-Assisted Voltage Assignment Algorithm

The simple as-slow-as possible algorithm alone will not wok efficiently given a stream of input data. The reason is if we process a current frame as slow as possible, it is likely that we have to speed up on the next frame. A better algorithm would have to really estimate the SNR of the signal to predict the number of iterations better.

Estimation of SNR is very possible. In many mobile systems, estimate of SNR is already available for other purposes and we can take advantage of that available measurement to assist the as-slow-as-possible algorithm.

Post-Design Notes

The purpose of this project is to learn about VLSI design. Lots of thanks and credits should be given to the instructors of the class, Prof. Peter A. Beerel and Prof. Keith M. Chugg, who have made the project practical and exciting. The turbo decoder is certainly one of the hot chips people in the industry are designing right now.

There are many design issues that I have learned and would like to discuss some of them here.

1) Simpler is better. If we could make algorithms simpler, we would be able to use fewer resources and sometimes make the hardware faster. There are many examples of simplifications in the project. First, normalization of the algorithm tremendously make the algorithm simpler (please be referred to [5]). Second, by recognizing redundancies and zeros in the transition matrix, we can calculate and store only three values of the transition matrix instead of eight for each time step. (I did not show the derivation of this but one can easily derive it from the original algorithm description to arrive at the same equation I have in the code.) Third, in the implementation, the controllers have been made simple by dividing the controlling job into two parts: the main controller and the local SISO controller. Each controller has its own local counter.

2) The memory components occupy the majority area of the chip. In this project, about 90% of the chip area is occupied by ROMs, RAMs, and flip-flops. Any efforts that try to reduce the memory usage would tremendously reduce the chip area. For example, by scheduling the completion process to start after the F&B matrices have been calculated halfway, we can cut the storage area for F&B matrices in half. In this project, LIFOs are used to store F&B matrices to minimize delay in retrieving the data. However, considering that they take up about 65% of the chip area, one could better off implement them with registered RAMs, which are much more area-efficient.

3) In this project, the full adders are implemented using half adders and pass transistors for simplicity and area efficiency. It turns out this is not a good decision. Pass-transistor circuits have poor driving capability and are not good for cascading. So, the adder is slow and could possibly fail to work (I did not test all the possible inputs). Considering that computation components take relatively very small area in the chip but their speed, especially in the critical path, would directly improve the speed of the chip, one could have designed them better with as-fast-as-possible circuits.

Due to limitation in time and experience, I apologize for any incompleteness and mistake in this report. Should the reader need the source codes or any more information, please contact me at pornchai@ieee.org.

References

[1] Jan M. Rabaey, Digital Integrated Circuits, A Design Perspective, Prentice Hall, 1996.

[2] Neil Weste and Kamram Eshrighan, Principles of CMOS VLSI Design, Addison-Wesley, 1993.

[3] Adrian Barbulescu, Turbo Codes: A Tutorial on a New Class of Powerful Error Correcting Coding Schemes, ITS, University of South Australia, Oct 1998.

[4] Leung, O.Y.-H., Chung-Wai Yue, Chi-Ying Tsui, Cheng, R.S, *Reducing power consumption of turbo code decoder using adaptive iteration with variable supply voltage,* Low Power Electronics and Design, 1999. Proceedings. 1999 International Symposium on , 1999 , Page (s): 36 –41.

[5] Keith M. Chugg, Slides on Let's Build a Turbo Decoder, USC, Jan 2000.