Chapter 3

VLSI Architecture for MLSDA

In this chapter we propose a new VLSI architecture to implement MLSDA. The Open stack in this architecture is implemented by a modified PESPQ and all modules of the architecture are presented in details. Simulation results in decoding speed and gate counts are given for several SNRs. The architecture of MLSDA for 2/4 (3/6) codes is also decsribed.

3.1 Modified Parallel Entry Systolic Priority Queue

After slightly modifying PESPQ algorithm, We may implement **MLSDA.** There are two differences between the modifying PESPQ algorithm and the original one. The first is the difference in metrics. In Lavoie's paper, the Fano metric is used, and then PESPQ chose the largest metric in the queue. However, in MLSDA, we use the new defined metric according to Wagner rule in section 2.1, and then the modifying PESPQ needs to choose the smallest metric in the queue. The other is that we need to add an additional step for **MLSDA** to check for repeat nodes (paths). Hence, we add a new step called *compare* in the modifying PESPQ. If any newly generated node has the same depth and state as any node in the queue, we compare the metrics of these two nodes and discard the one with larger metric.

Definition 3.1.1:

Let $\mathbf{A}_i$ be the processor stores a node**,** $\mathbf{I}_i$ the insert node, $\mathbf{R}_i$ the register stores the node which will be inserted to $\mathbf{A}_i$, $\boldsymbol{m}$(N) the metric of node N, and T the extracted node. Furthermore, let P(N) indicate the position of a node in the trellis, and N(A_i)

be the node in A_i.

<Modifying PESPQ Algorithm >(Refer to Figure 3.1)

Step 1: Compare: If $P(I_j)=P(N(A_i))$, for any j=1,2 and i=1,2,…,n, then compare the metrics of I_j and $N(A_i)$. If $m(I_j) \leqq m(N(A_i))$, the register R_j keeps I_j and the metric of the node in processor Ai changes to infinite. On the other hand, the metric of the node in R_j changes to infinite and that in Ai unchanged.

Step 2: Insert and Shift right: deliver the nodes in R_1 to A_1 and R_2 to A_2 and send the node in A_{i-1} to A_{i+1} i=2, 3, …(i.e. A_1 to A_3, A_2 to A_4, …).

Step 3: Sort triple: make A_{3i-2}, A_{3i-1} and A_{3i} to be a unit, i=1,2,3…. Send the node with best metric in the unit to A3i-2.

Step 4: Shift left and extract: send A_{i+1} to A_i, i=1,2,…, and deliver T from A_1.

Step 5: Sort triple: the same as Step 3. Make A_{3i-2}, A_{3i-1} and A_{3i} to be a unit, i=1,2,3…. Send the node with best metric in the unit to A_{3i-2}.

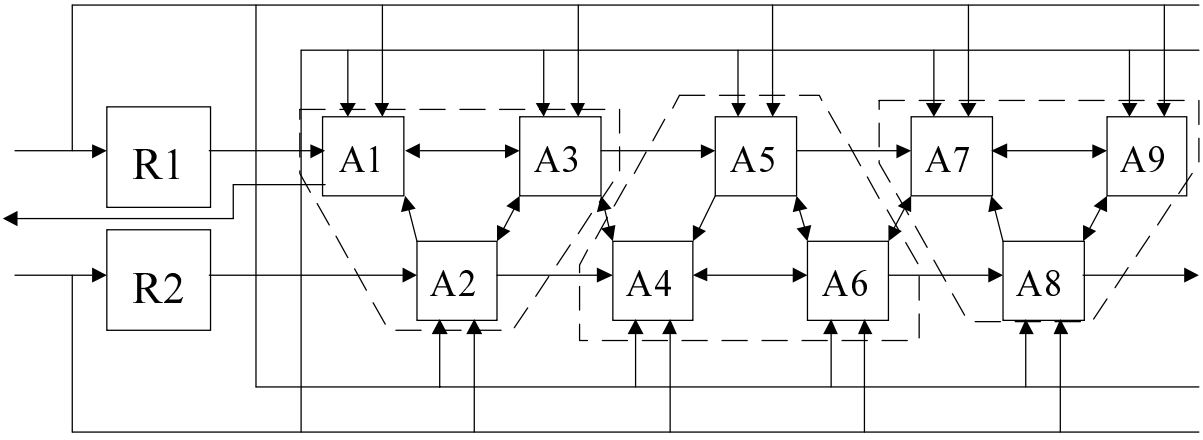


Fig. 3.1 Modified Structure of PESPQ

The fact that the modified PESPQ will always send out the node with the best metric is verified by the following theorem.

Theorem 3.1.1:

For the modified algorithm, after any cycle of the parallel entry systolic priority queue, the node with rank ***r*** , r=1,2,.., lies in a processor $\mathbf{A_k}$ such that $1 \leqq k \leqq 3r-2$.

proof:

Case 1: If the input and all nodes in processors have different states, we can apply Theorem 2.3.1 to prove it.

Case 2: If the node at processor A_k, who has the same state as the input, contains a smaller metric value than the metric of the input node, then the algorithm set the metric of the input node as infinite. Hence, Theorem 2.3.1 can be applied in this case.

Case 3: If the node at processor A_k, who has the same state as the input, contains a larger metric value than the metric of the input node, then the algorithm will set the metric of the node at processor A_k as infinite. We may assume that the previous rank of the node at the changed processor is ***l*** and the rank of the input node is ***m***, where $m < l$. After the algorithm set the metric of the node at processor A_k as infinite (the action at the step 1 of the algorithm), the new rank of any node with older rank larger than ***l*** will not be changed or one less than older rank. Now assume that the rank of the node at $A_{k'}$ is one less than that before. Then, we need to guarantee that after the cycle is end, the finial rank r of the node will satisfy the above inequality. Since one of the input with metric less than ***l***, by the similar argument given in the proof of Theorem 3.2.1 we have the inequality hold. ■

According the above theorem, the modified algorithm can totally meet the requirement of the function of the Open Stack in MLSDA and deliver the top node

with the smallest metric from the queue.

3.2 Decoder Architecture

In this section, we present VLSI architecture for realizing the MLSDA decoder based on a systolic priority queue.

In the architecture, a decoder is composed of six separate modules working in unison to decode a sequence of channel digits. A block diagram is depicted in Fig. 3.2 in which one may see a priority queue module, a data frame module, a path history module, two metric computation modules and a close stack module which is combined into the priority queue module.

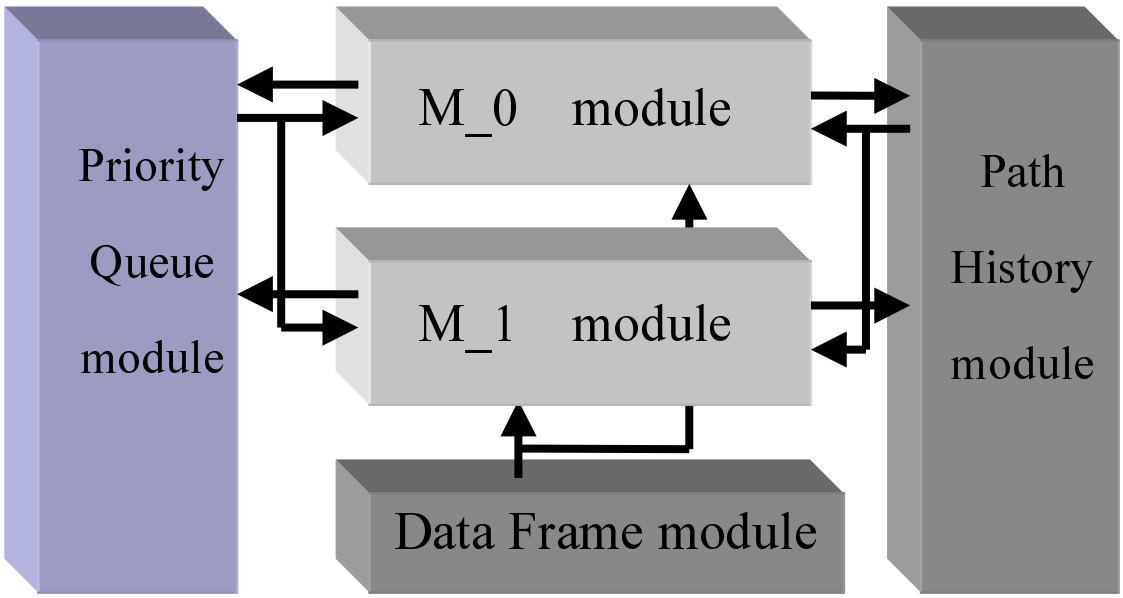


Fig. 3.2 Block diagram of the decoder architecture

In the following section 3.2.1 to 3.2.5, we will discuss the decoding flow to show how the decoder completes a decoding process. The description of functions, bits assignments, and state transient diagrams of the six modules of the decoder is given in

Section 3.2. Section 3.3 introduces the extensive version of the decoder architecture for the 2/4 (3/6) convolutional codes.

The decoding flow chart that we may decide the functions of these modules in the decoder. is shown in Fig. 3.3.

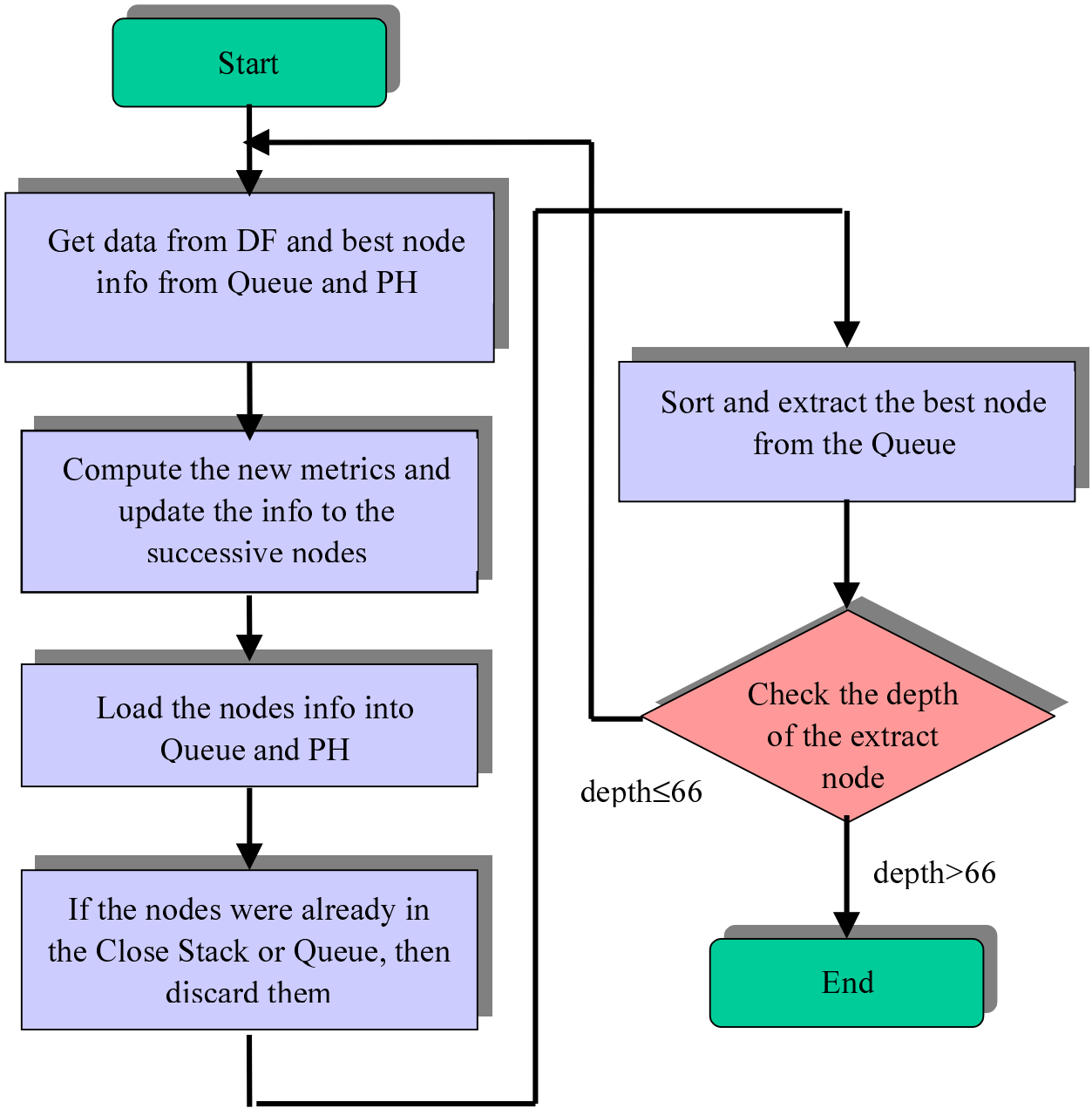


Fig. 3.3 Decoding Flow Chart

In our design, we decide the information digits to be 60, and we use a (2, 1, 6)

convolutional code. So the corresponding codeword has length 132, where the final 12 bits are generated after the last nonzero information block has entered the encoder.

We setup the information associated with a node that would be used in the decoding process as cumulative metric, Depth, State, Address, and Information bits. Address is used as an index that points to the address of information bits stored in path history module. The bit numbers of these data are assigned as those in Table I.

Field	Bits
Depth	7
State	6
Address	11
Cumulative metric	12
Information bits	66
Total	102

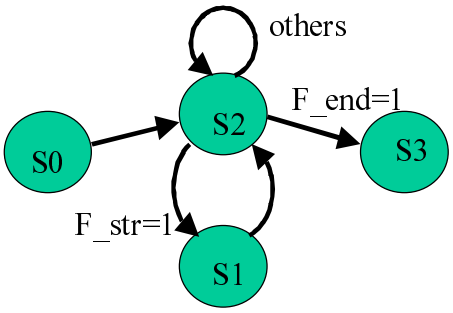
Table I Data of Node

3.2.1 Data Frame module

The data frame module holds a sequence of channel digits for the duration of its decoding. It acts like a buffer. In each decoding step, the frame module reads from the priority queue module the depth of the parent node retrieved from the stack. Then the module increases the depth by one, and extracts from its memory the corresponding channel digits.

In our design, we quantize a received codeword into 7 bits, and we load 924 bits (132*7) a time during a decoding process. When the decoder finish a decoding process, the module will refresh its memory. The I_bus shown in Fig. 3.5 is the input of received codeword sequence.

Data frame module may be implemented by a finite state machine. Fig. 3.4 shows the state diagram of this module. The input and output ports are shown in Fig. 3.5.



state	function
S0	initial state, load the 1st data to M0 & M1
S1	according to the depth, load the n-th data to M0 & M1
S2	idle state, if F_str = 1, go to S1, else if F_end=1, go to S3
S3	end state

Fig. 3.4 State diagram of data frame module

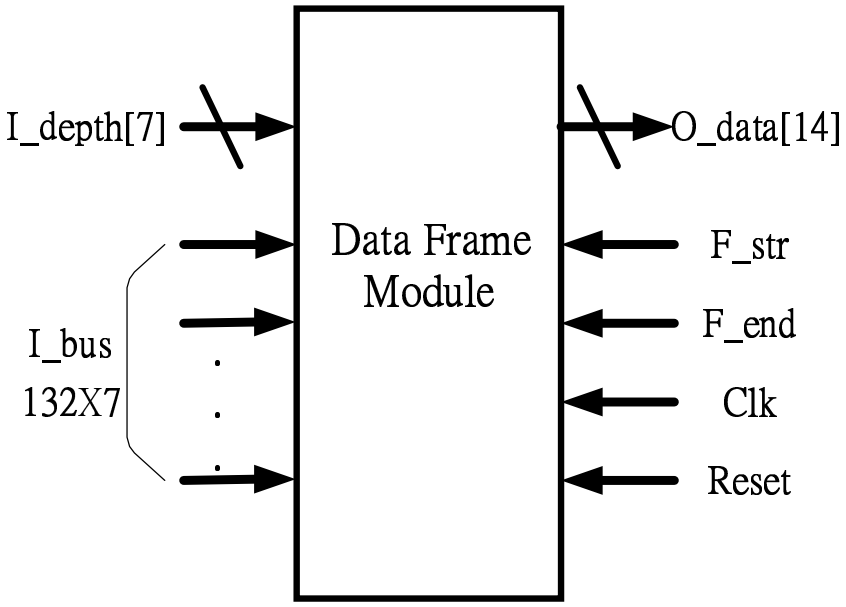
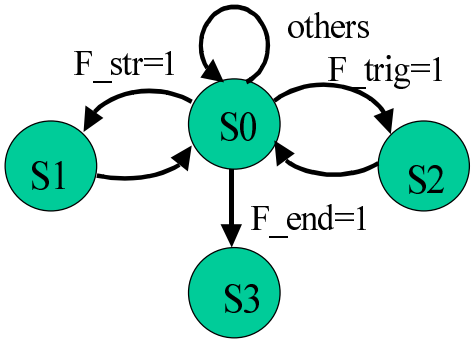


Fig. 3.5 Pins of data frame module

3.2.2 Path History module

Path History module records all the paths of nodes that were in the priority queue module. The path of a node is its information bits accumulated in each decoding step. If the queue extracts a node, the path history module sends the information bits of the node to the computation module. The action is relying on the address of each extracted node. According to the address, it can deliver the correct information sequence. When the computation module delivers the successive node, the path history module records the information bits of the node by the received addresses.

The volume of the path history module is about 32 K bytes. The finite state machine and the state diagram are illustrated in Fig. 3.6 and the ports of this module are shown in Fig. 3.6.



state	function
S0:	idle state, if F_trig = 1, go to S2, else if F_str=1, go to S1, else if F_end=1, go to S3
S1:	load the n-th information-bits sequence according to the address of the node
S2:	receive the two information-bits sequence from M0 & M1
S3:	end state

Fig. 3.6 State diagram of path history module

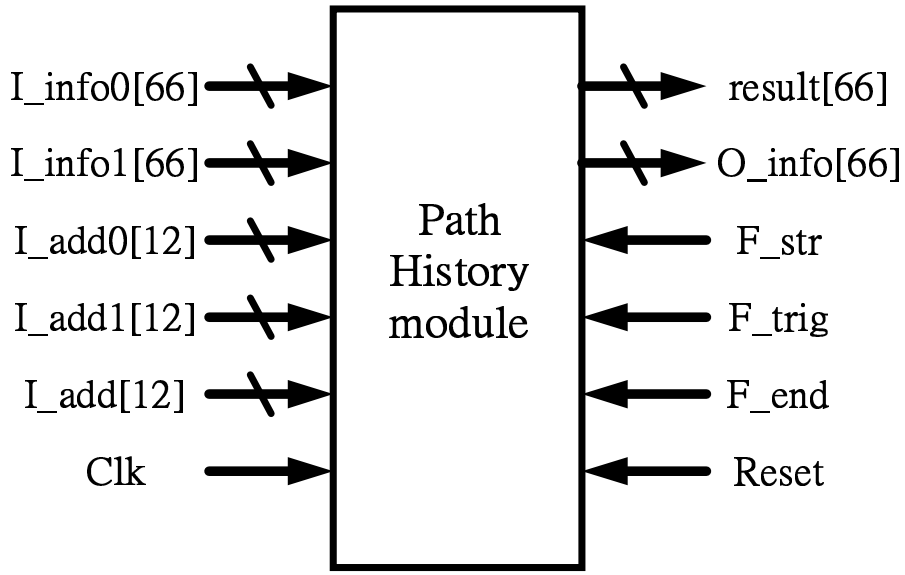


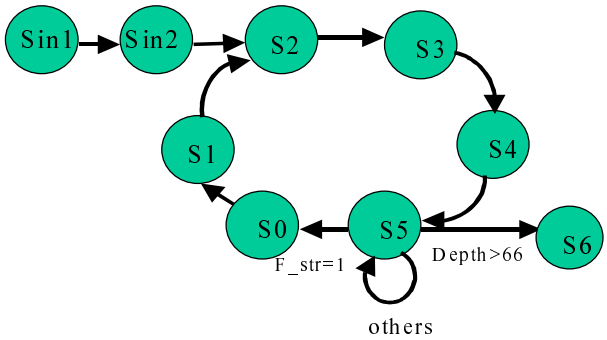
Fig. 3.7 Pins of path history module

3.2.3 Metric Computation module

The main function of metric computation module is to handle the node data received from the priority queue and to deliver the successive node data to the priority queue and path history module. Two submodules, labeled M0 and M1, compose the module. The two submodules extend the node retrieved from the priority queue along the branches corresponding to the information bits zero and one.

Each submodule contains three basic elements, a replica of the 1/2 code encoder, a table of possible branch metrics, and an adder. When the depth and state of a node enter the submodule, they will be refreshed before the computation starts. The encoder generates the successive codewords by the updated state. Then according to the table of possible branch metric, we can get the metrics by comparing the channel digits and codewords. Finally, each submodule adds the metric to the cumulative metrics of the successive nodes, and deliver the associated data of the successive nodes to other modules.

The state diagram of this module is shown in Fig. 3.7 and the ports of that are shown in Fig. 3.8.



state	function
Sin1	initial state, setup registers,
Sin2	load data from DF,
S0	update input depth and state and compute successive codewords
S1	load data from DF and information bit from PH,
S2	update information bits and decide metric by codewords
S3	add new metrics to the old metrics,
S4	send F_trig=1 and deliver the output data(address, information bits, metric, depth, state) to others,
S5	idle state, if F_str=1, load the input data (address, metric, depth, state), if depth>66, go to S6
S6	end state.

Fig. 3.8 State diagram of metric computation module

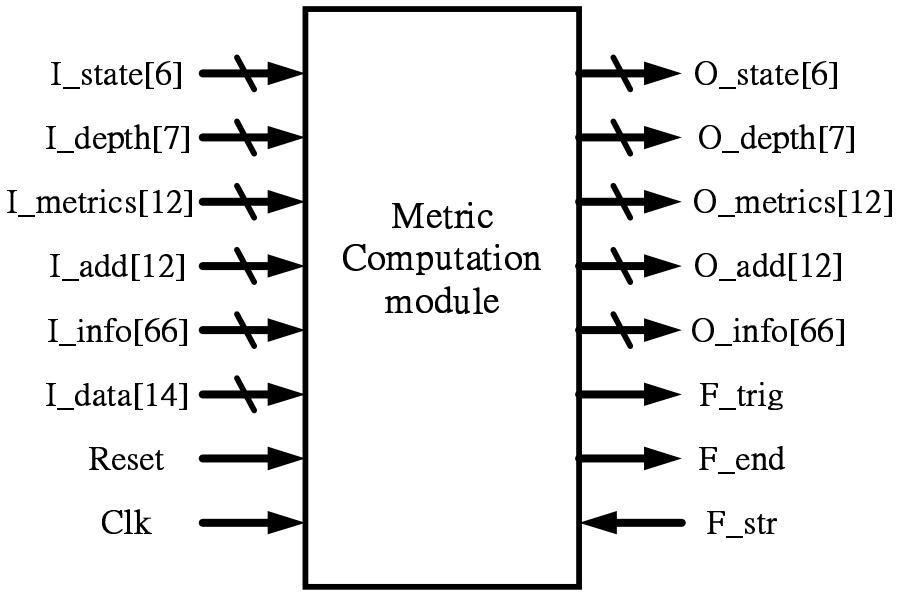
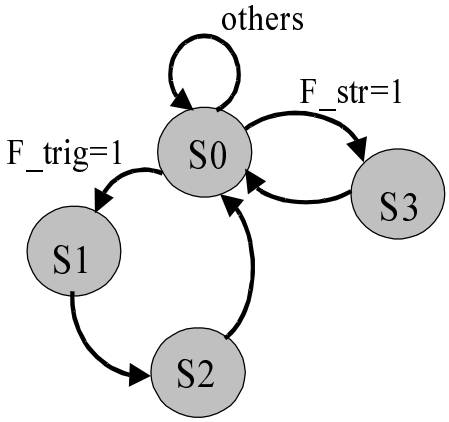


Fig. 3.9 Pins of metric computation module

3.2.4 Close Stack module

The close stack module records the state and depth of a node that has ever been extracted from the priority queue. It is a memory with 13 bits address (combining the information of depths and states) and 1 bit data. The main function of close stack is to delete the repeated path (i.e. the merging of paths) during the decoding process. Every time when the node delivered from the priority queue, the corresponding address unit is changed to 1. Then the nodes retrieved from metric computation module will be compared with the nodes in the close stack. If repetition is detected, then the node received from the priority queue will be discarded.

The state diagram and ports of this module are shown in Fig. 3.9 and Fig. 3.10. Notice that the state S1 in Fig. 3.10 is a redundant state. It is used to synchronize the output data with the priority queue module.



state	function
S0	idle state, if F_trig = 1, compare the nodes and go to S1, else if F_str=1, get the depth and state of the top node and go to S3
S1	redundancy state
S2:	reset the output signal and go back to S0
S3:	write the data into the corresponding address

Fig. 3.10 State diagram of close stack module

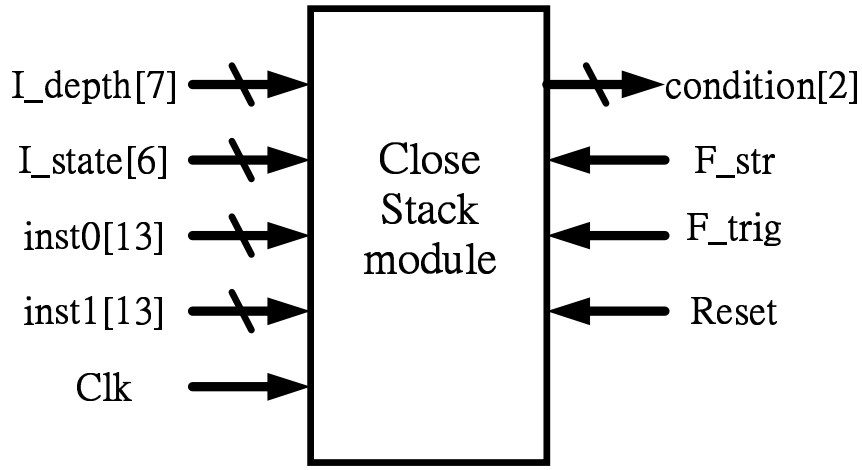


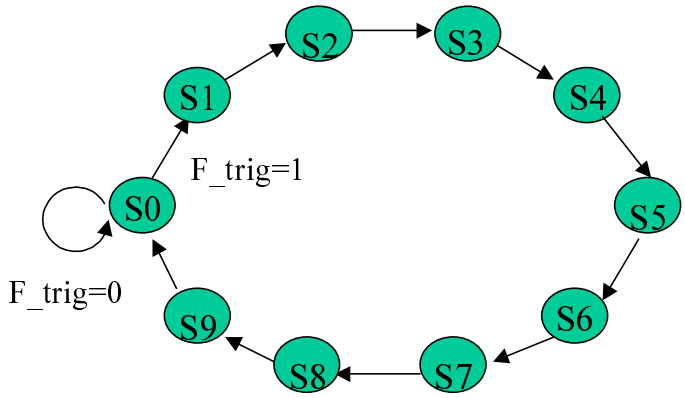
Fig. 3.11 Pins of close stack module

3.2.5 Processor submodule in Priority Queue module

At the heart of the architecture is the priority queue module. It is a smallest-in-first-out data structure that delivers the top node with the smallest metric at the beginning of every computation. The algorithm and theorem of the priority queue have been introduced in section 3.1, so we only discuss the implementation of the priority queue in this subsection. The processor submodule is the basic unit in the priority queue module. In one submodule, it has three processors as illustrated in Fig. 3.1. To connect them serially, it becomes the priority queue module.

In order to prevent the data of any node to be lossed during sorting and extracting, we add some states to backup the data of the node. In the sorting and shifting step, the backuped data are sent to the regular register.

The state diagram and ports of this module are shown in Fig. 3.12 and Fig. 3.13.



state	function
S0	idle state, if F_trig=1, compare the state and depth in processors,
S1	send data to backup registers,
S2	receive data from previous processor,
S3	send data to backup registers,
S4	sort and exchange data,
S5	send data to backup registers, extract the smallest metric out
S6	receive data from back processor,
S7	send data to backup register,
S8	sort and exchange data,
S9	send data to backup register,

Fig. 3.12 State diagram of processor submodule

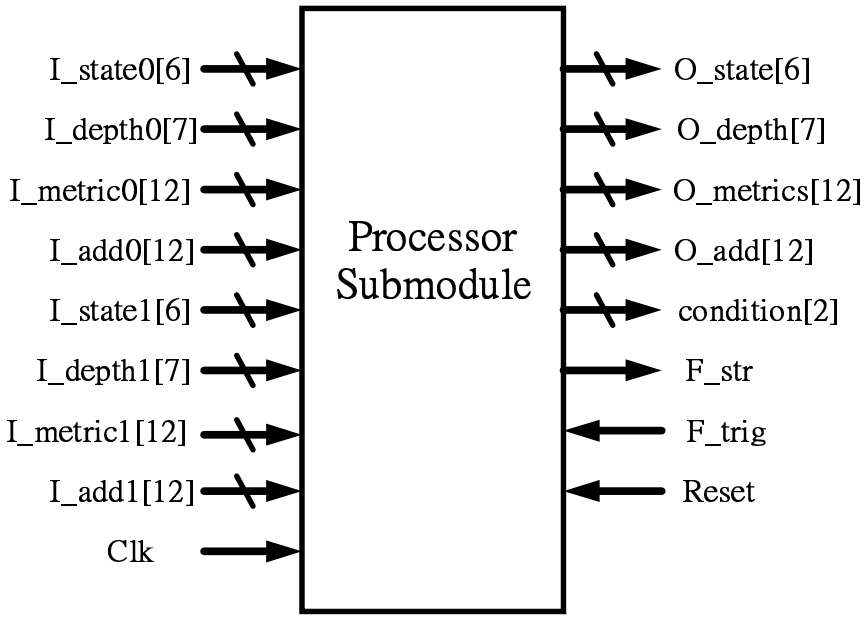


Fig. 3.13 Pins of processor submodule

3.3 The architecture of MLSDA for 2/4 and 3/6 Convolutional Codes

All 1/2 convolutional codes may be treated as an equivalent 2/4 codes during the decoding phase if we combine two input bits as one unit. Consequently, we might have some advantage in VLSI implementation such as decoding speed; however, the gate counts of decoder will be increased.

In section 3.1, we propose a priority queue for maximum-likelihood sequential decoding on 1/2 convolution code. In the section, we will propose the priority queue architecture for MLSDA where 1/2 code will be decoded as 2/4 (3/6) codes. The new architecture can also deliver the node with best metric in constraint duration. If we use the priority for 2/4 (3/6) codes decoding, the decoding speed will be faster theoretically than that of 1/2 code.

3.3.1 The 2/4 and 3/6 Convolutional Codes

Ii is convenient to explain how to convert a 1/2 code into its corresponding 2/4 code by using their trellises. The way of producing 2/4 code is a little like the punctured convolutional code, but it doesn't delete any bit in the bit stream. In other words, all codewords are the same for 1/2 and 2/4 codes. The encoders for 1/2 code and 2/4 code are the same but they have different decoder. Roughly speaking, we combine two input bits in a codeword sequence of 1/2 convolutional code during the decoding procedure. In every decoding step, decoder will process 2 information bits instead 1. The trellis diagrams for 1/2 code and its corresponding 2/4 code are shown in Fig. 3.14 and Fig. 3.15, respectively. Fig. 3.14. is the normal trellis diagram for the 1/2 convolutional code with m=2, and Fig. 3.15 is the trellis diagram for its

corresponding 2/4 convolutional code. In the 2/4 code trellis diagram, we combine two levels of states into one level so that the number of paths from one state to next states is changed from 2 to 4, and the states in the trellis become half of those of 1/2 code.

According to this method, we can extend 1/2 code to 3/6 code or more, and the trellis diagram of 3/4 code is shown in Fig. 3.16.

As shown in Fig 3.15, the number of levels for 2/4 code is half of that of 1/2 code. In VLSI implementation, we can make a tradeoff between circuit complexity and decoding speed. We can reach twice faster in speed but a little more complex in circuit architecture.

Since the MLSDA will eliminate the path merges with a path already in the Open Stack and Closed Stack. In the 2/4 code decoding, it may delete more redundant paths, and use less decoding time. Since the branches out from states are more than two we need to design a queue to handle 4 or 6 inputs.

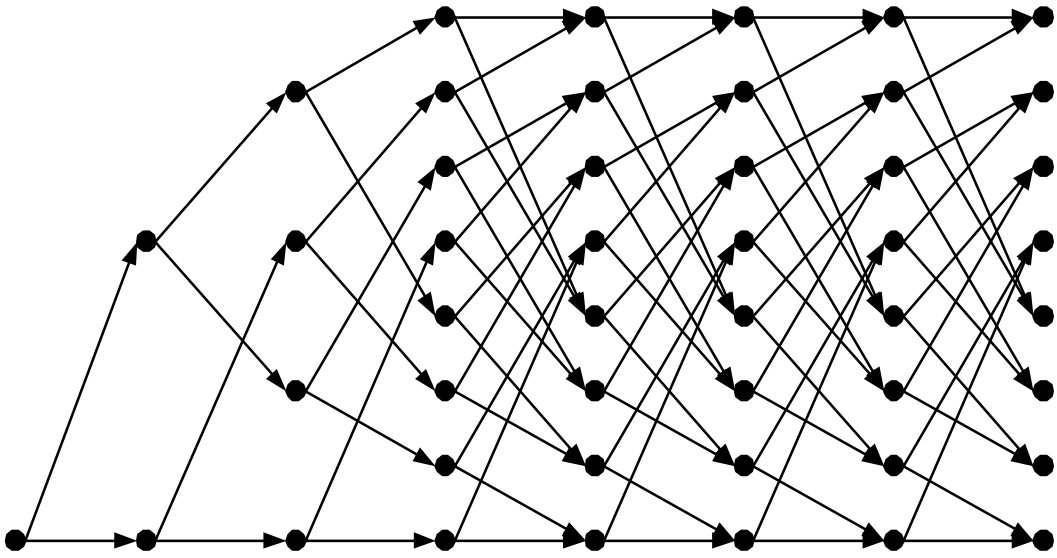


Fig. 3.14 Trellis diagram for a (2, 1, 3) code

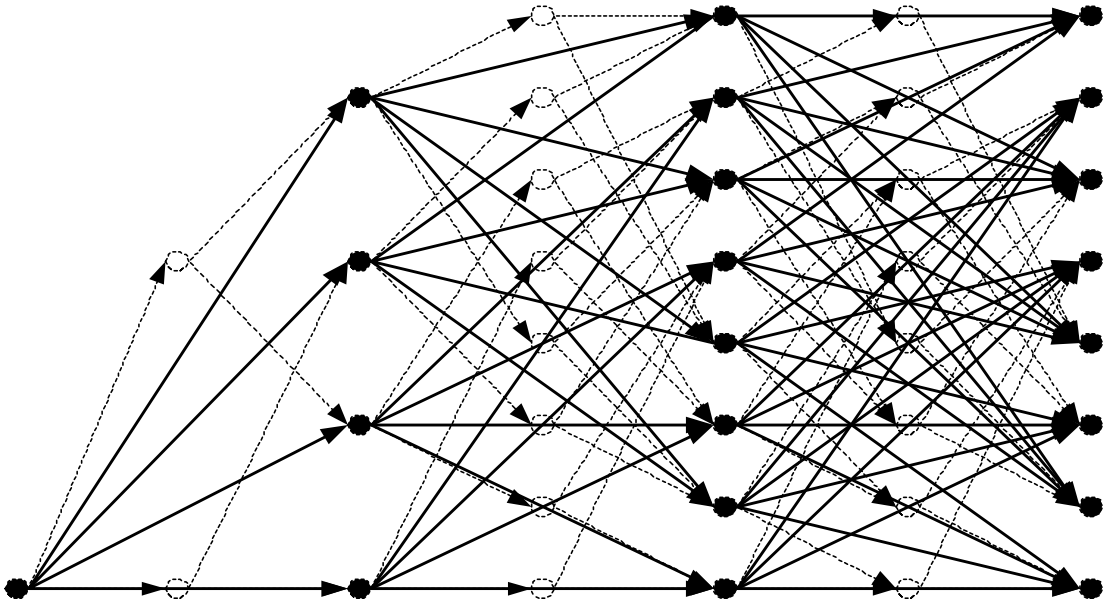


Fig. 3.15 Trellis diagram for a (4, 2, 3) code

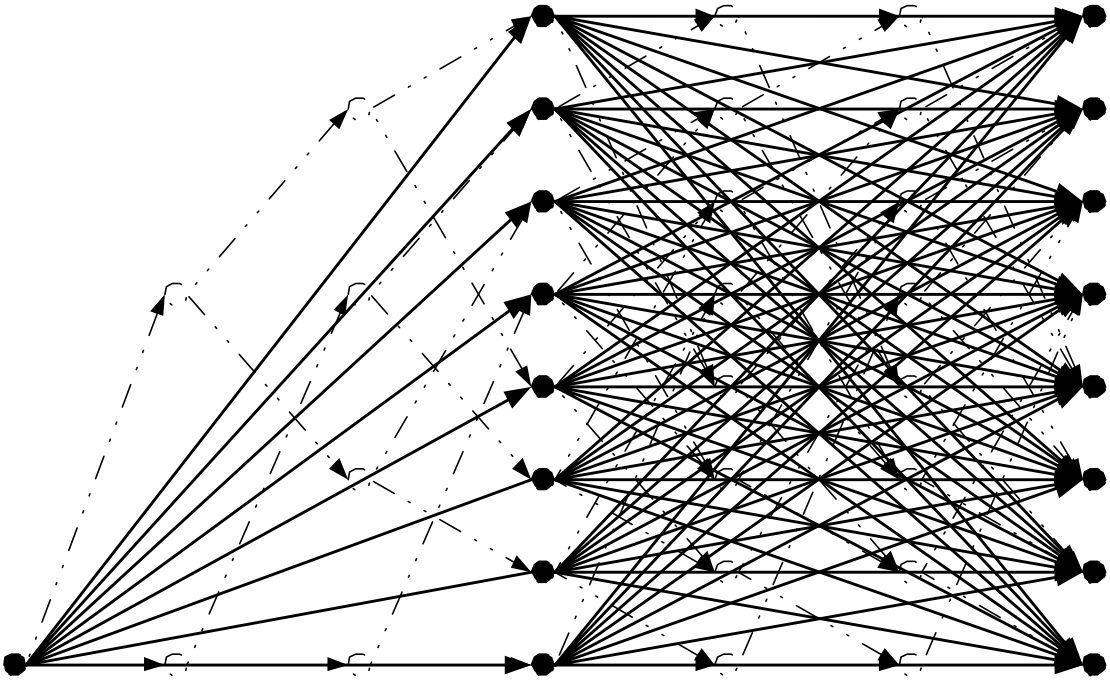


Fig. 3.16 Trellis diagram for a (6, 3, 3) code

3.3.2 The modified MISPQ

In order to implement MLSDA for 2/4 (3/6) codes, we need to make some modification on MISPQ. Like the PESPQ, the compare step is also added into the original algorithm.

Definition 3.3.1:

Let $\boldsymbol{I}_i$ be the insert node. $\boldsymbol{S}_i$ is the register stores the node which will insert to $\boldsymbol{P}_{1,i}$. ,**P**(N) the position of node N in the trellis, and **m**(N) the metric of node N. Furthermore, let $\boldsymbol{P}_{1,0}$ be the processor that stores the node with best metric.

<The modified MISPQ algorithm>

Step 1: Compare: If $\mathbf{P}(\boldsymbol{I}_j)=\mathbf{P}(\mathrm{N}(P_{\mathrm{i,j}}))$, j=1,2,…N, compare the metrics of $\boldsymbol{I}_j$ and $\boldsymbol{P}_{i,j}$. If $\mathbf{m}(\boldsymbol{I}_j)\leqq\mathbf{m}(\mathrm{N}(\boldsymbol{P}_{i,j}))$, the register $\boldsymbol{S}_j$ keeps $\boldsymbol{I}_j$ and processor $\boldsymbol{P}_{i,j}$ changes to infinite. Otherwise, $\boldsymbol{S}_j$ changes to infinite and $\boldsymbol{P}_{i,j}$ unchanged.

Step 2. Insert N metrics $\boldsymbol{S}_1$, $\boldsymbol{S}_2$, …, $\boldsymbol{S}_N$ simultaneously into the queue and shift each metric at position $\boldsymbol{P}_{i,j}$ to its right.

Step 3. Rearrange metric in each slice of processors. Move the best metric in each slice to its local top position, and the positions of other metrics are trivial.

Step 4. Extract the $\boldsymbol{P}_{1,0}$ from the queue, which is always the best metric among all. At the same time shift the metric on the top of each slice to its left.

Step 5. Rearrange metric in each slice of processors. Move the best metric in each slice to its local top position, and the positions of other metrics are trivial.

It is easy to show that In the modified MISPQ, after any number of insertion or deletion (extraction) operations, the $\boldsymbol{k}$-th good metric is stored in some processor $\boldsymbol{P}_{i,j}$, where $\boldsymbol{P}_{1,0}\leqq\boldsymbol{P}_{i,j}\leqq\boldsymbol{P}_{k,0}$. Hence, the modified MISPQ will always deliver the node with the best metric.

The modified type II MISPQ algorithm is similar to the modified MISPQ. We only add the compare step and the other steps are the same as the original type II MISPQ.

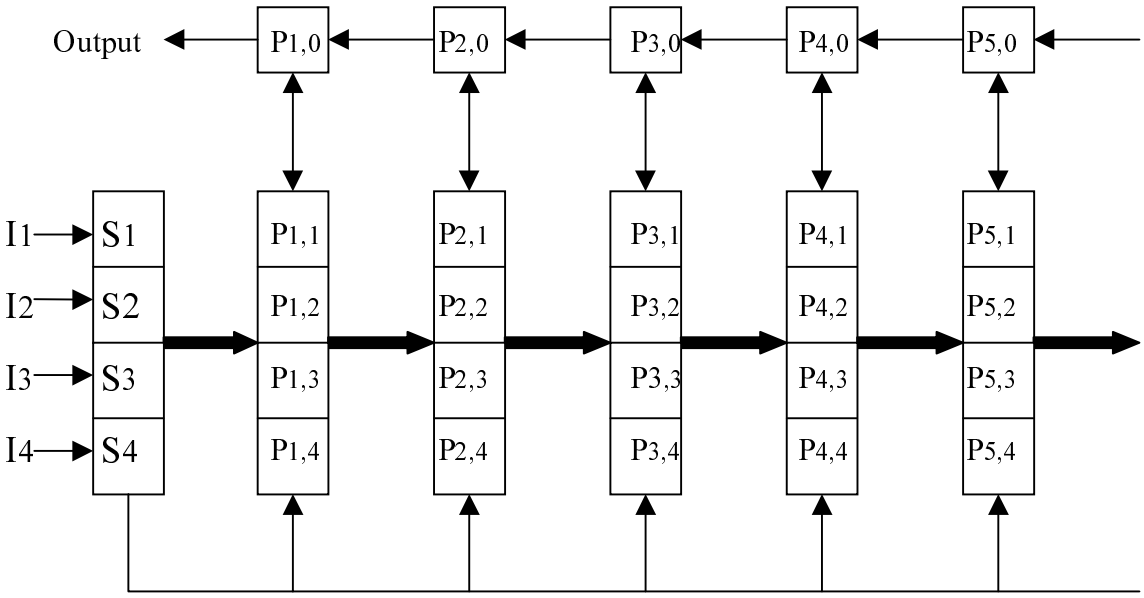


Fig. 3.17 Modified MISPQ Architecture

3.3.3 Architecture of MLSDA for 2/4 and 3/6 codes

In section 3.2, we have presented the sequential decoder structure of MLSDA for 1/2 convolutional code. The architecture for 2/4 (3/6) codes are similar to that of the 1/2 code decoder. Since the 2/4 (3/6) codes have multiple-inputs for the priority queue, we need to incorporate the modified MISPQ (modified type II MISPQ) into the priority queue module.

The number of metric computation submodules is increased when the inputs fed into queue are increased. The decoder for 2/4 code needs to have 4 metric computation submodules, named M00, M01, M10 and M11 and 8 for decoder for 3/6

code which named M000, M001, M010, M011, M100, M101, M110 and M111. The number of metric computation submodules is 2^k where k is the number of information bits for encoder to generate n output bits. When k increases linearly, the modules and decoder complexity are increased exponentially. The more the decoder complexity, the more the transmission gates. Thus, we should have a tradeoff between decoding speed and gates.

For 2/4 and 3/6 codes, the modules, except mentioned above, are almost the same as those for 1/2 code besides the pins are increased. The general architecture is shown in Fig. 3.18.

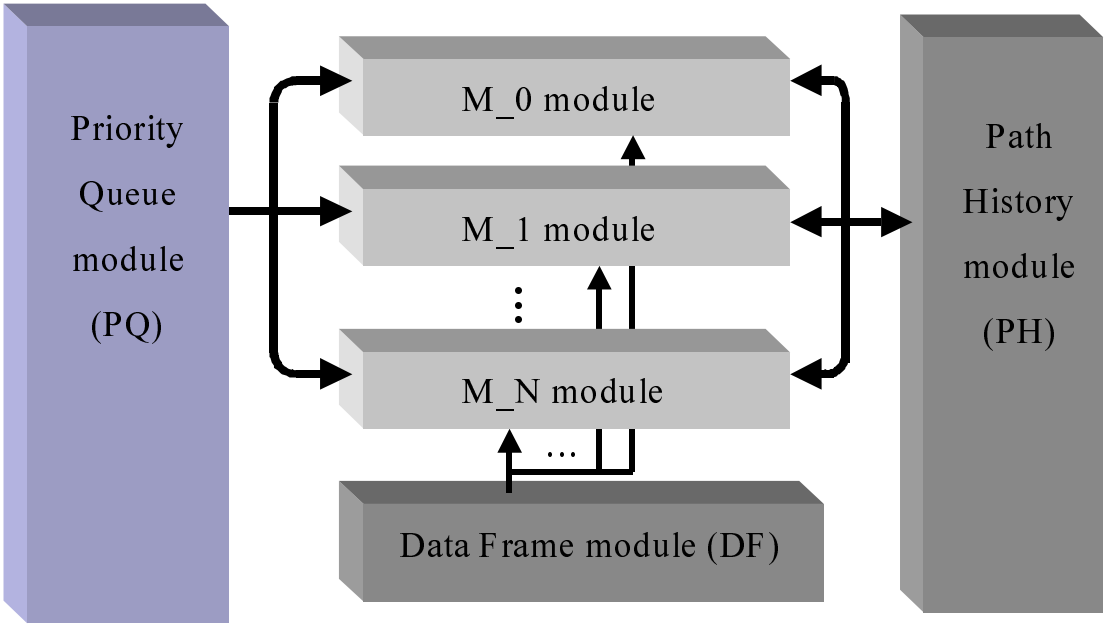


Fig. 3.18 General structure for multiple input decoder

3.4 Simulation Results

In this section we present the simulation results on decoding speed when the code is

transmitted over AWGN channel. The synthesis results, which indicates the cost and performance in IC design.

3.4.1 Simulation results on decoding speed

Since the codes will be simulated over AWGN channel, we first model the transmission system including encoder, decoder and additive white Gaussian noise (AWGN). The block diagram the system is shown in Fig. 3.19.

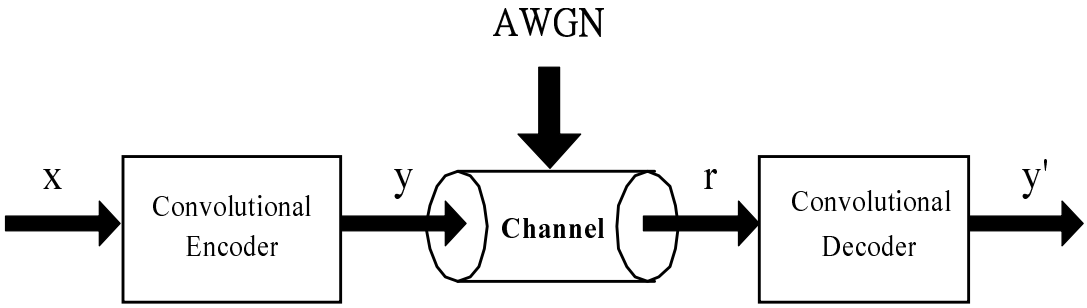


Fig. 3.19 Simulation model.

In Fig. 3.19, x is the information bits, y is the codeword sequence, r is the received sequence which is the combination of AWGN and codeword sequence, and y` is the estimated information bits. The simulation results on decoding speed for SNR from 6 dB to 10 dB are shown in Fig. 3.20. The unit for estimating the decoding speed is clock cycle that we can determine the speed more preciously for different cell library where it might have different clock rate.

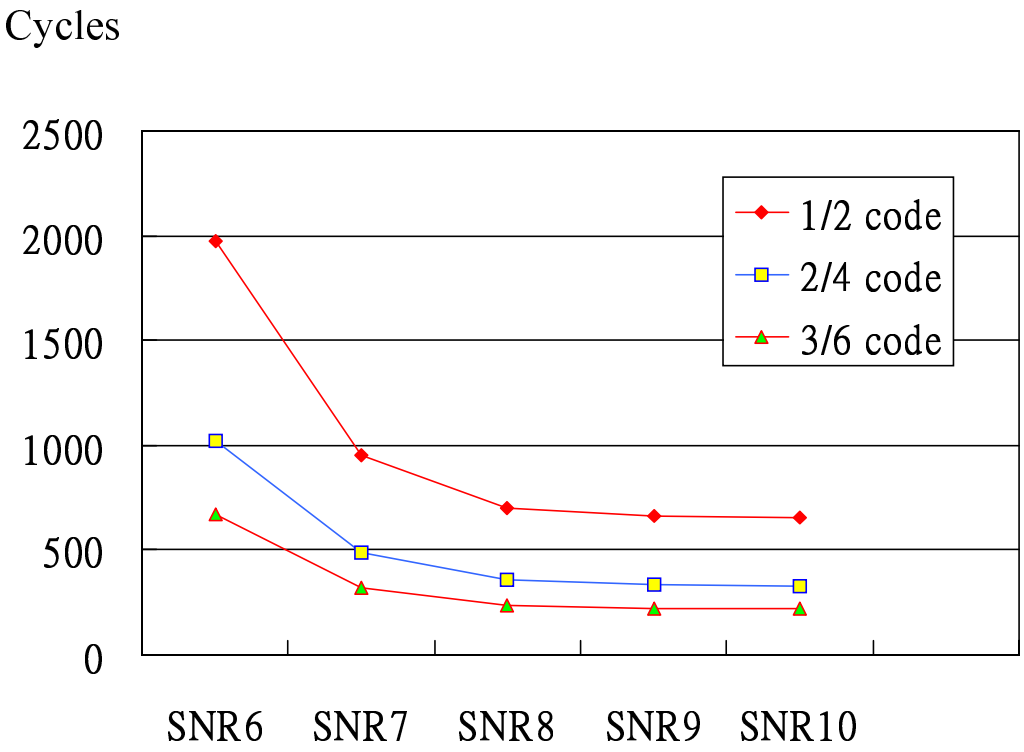


Fig. 3.20 Decoding speed for different decoders.

As shown in Fig. 3.20, the time for decoding is increased exponentially when the SNR is decreasing. Furthermore, the decoding speed ratio of the 1/2 code, 2/4 code and 3/6 code is almost 6:3:2.

3.4.2 Simulation results under Synthesis

In synthesis process, we use the 0.6 um cell library “Campass” designed by CIC to layout our decoders. The number of transmission gates and the maximum timing delay during synthesis process are concerned since they are the most critical for IC design. Furthermore, since the priority queue module and the metric computation module are the main contribution of the number of gates and maximum timing delay in the decoders we only present the number of gates and maximum timing delay in the

decoders for these two modules. The following figures are the synthesis results for the priority module and metric computation module.

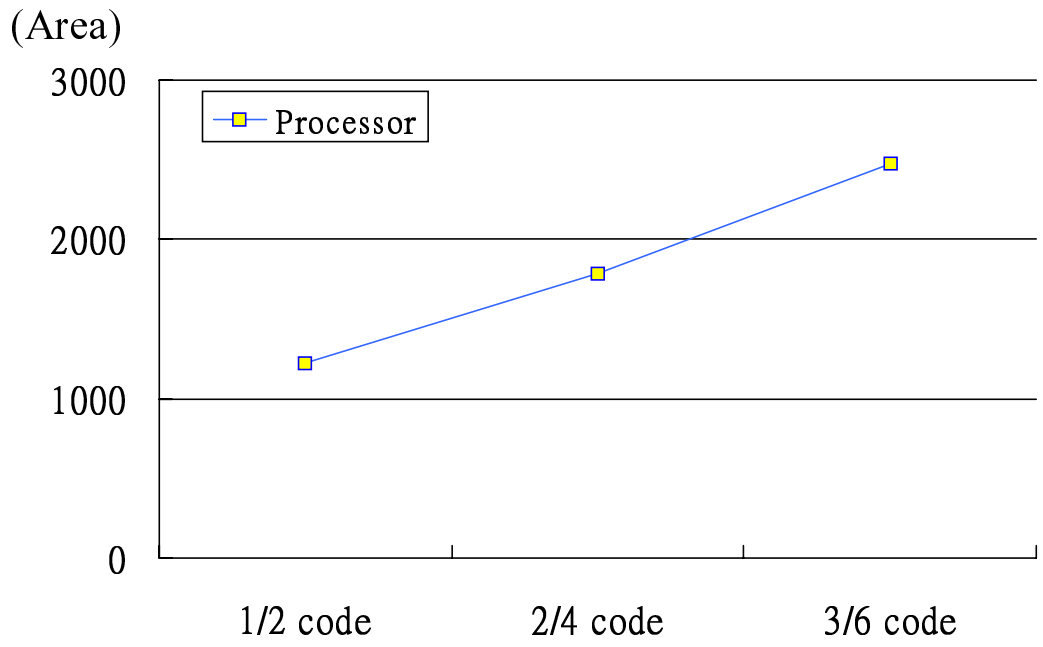


Fig. 3.21 Area of priority queue processors

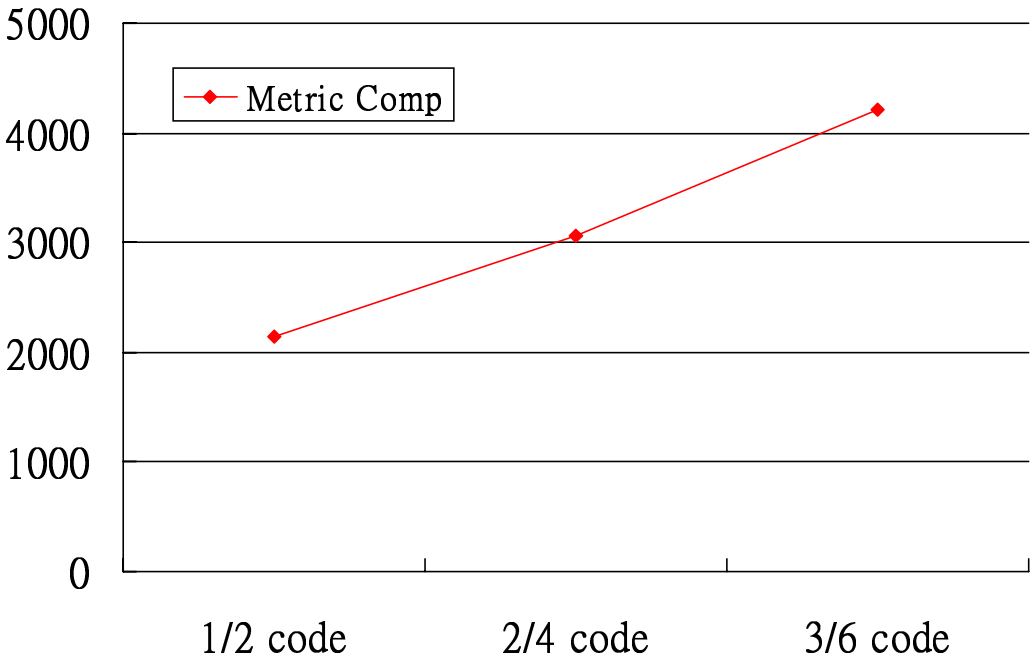


Fig 3.22 Area of computation module

As shown in Fig. 3.21, the gates of the processors for 1/2 code, 2/4 code and 3/6 code are increased linearly, so the number of gates will not raise to extremely large when the k increases.

As shown in Fig. 3.22, the areas of signal metric computation module are increased linearly for the codes simulated. However, for in the architecture of our decoder, the numbers of metric computation modules is inceased with 2^k. The layout complexity may be too large to be implemented for a larger k.

Fig. 3.23 and 3.24 show the maximum timing delay for these processors and metric computation modules. The timings for different decoder modules are almost the same. According this result, one may conclude that the 2/4 code and 3/6 code decoders have better performance in decoding time.

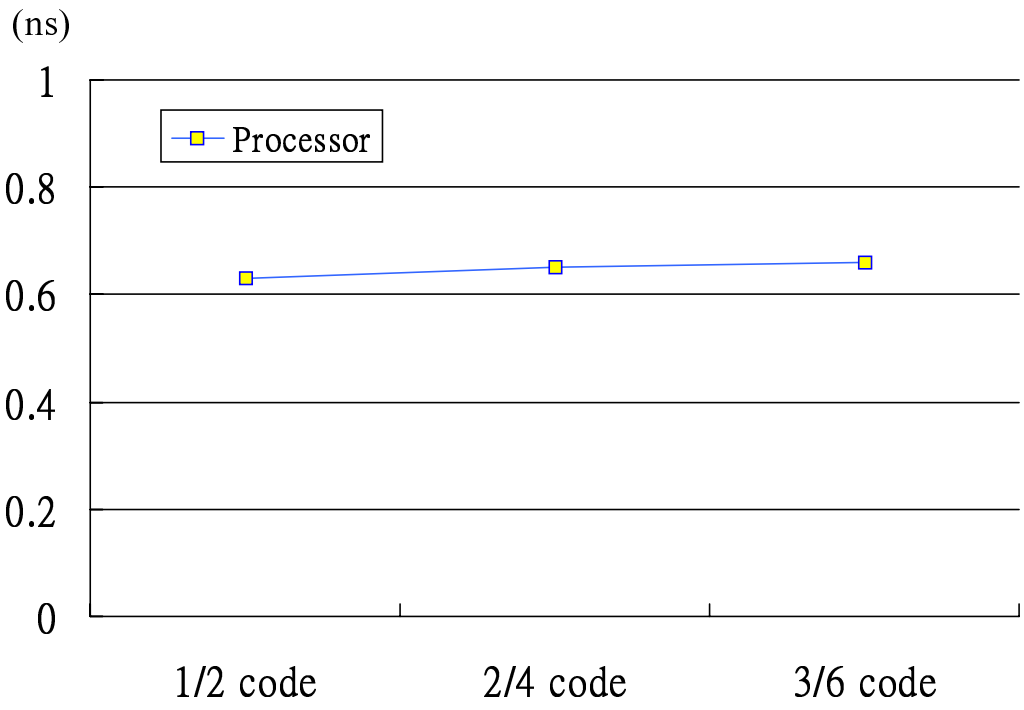


Fig. 3.23 Maximum timing delay for processors

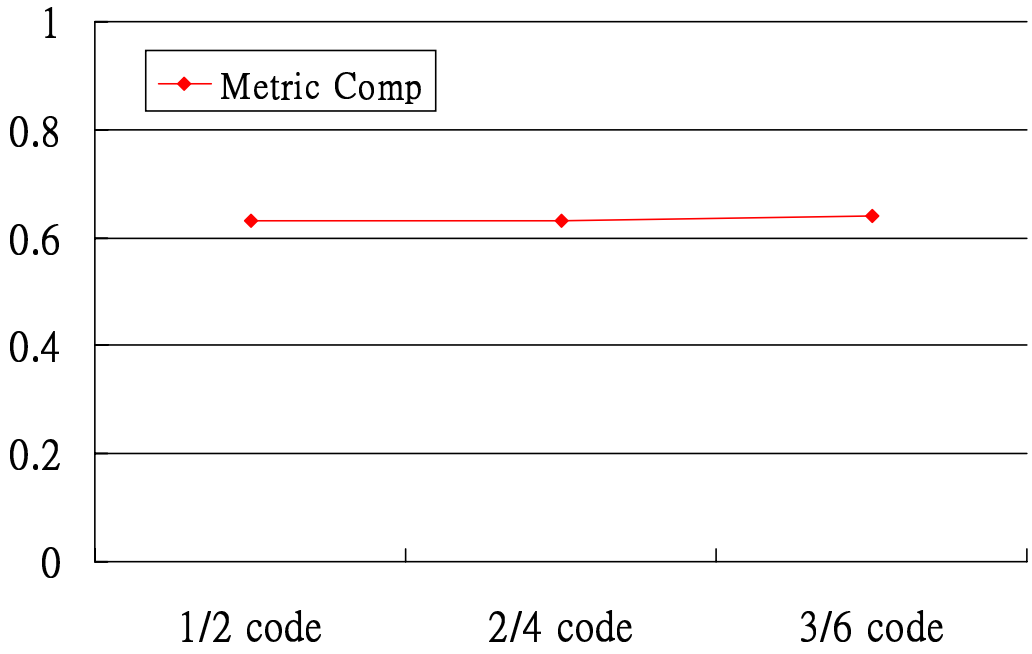


Fig. 3.24 Maximum timing delay for metric computation modules.