ICS

TECHNICAL REPORT

Design of a JBIG Encoder using SpecC Methodology

Technical Report ICS-00-13
June 21, 2000

Junyu Peng, Lukai Cai, Anand Selka, Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{pengj, lcai, aselka, gajski}@ics.uci.edu

Justin Denison, Arkady M. Horak

Motorola Semiconductor Products Sector
System on a Chip Design Technology
Austin, TX 78731, USA

{justin, RVKA30}@email.sps.mot.com

Information and Computer Science
University of California, Irvine

Design of a JBIG Encoder using SpecC Methodology

Technical Report ICS-00-13
June 21, 2000

Junyu Peng, Lukai Cai, Anand Selka, Daniel D. Gajski

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{pengj, lcai, aselka, gajski}@ics.uci.edu

Arkady M. Horak, Justin Denison

Motorola Semiconductor Products Sector
System on a Chip Design Technology
Austin, TX 78731, USA

{justin, RVKA30}@email.sps.mot.com

Abstract

This report describes the design of a JBIG encoder, based on the ITU-T Recommendation T.82, using the SpecC system level design methodology being developed at CAD Lab, UC Irvine. We begin with an executable specification in SpecC, and explore design alternatives for the system architecture, and refine the specification into a final communication model where the communication protocols between the system components are defined. In this report, we document the different design stages undergone, and also the results in the process.

Contents

List of Figures

Design of a JBIG Encoder using SpecC Methodology

J. Peng, L. Cai, A. Selka, D. Gajski

Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA

A. Horak, J. Denison

Motorola Semiconductor Products Sector
System on a Chip Design Technology
Austin, TX 78731, USA

Abstract

This report describes the design of a JBIG encoder, based on the ITU-T Recommendation T.82, using the SpecC system level design methodology being developed at CAD Lab, UC Irvine. We begin with an executable specification in SpecC, and explore design alternatives for the system architecture, and refine the specification into a final communication model where the communication protocols between the system components are defined. In this report, we document the different design stages undergone, and also the results in the process.

1 Introduction

1.1 Project Goal

The goal of the project is to verify SpecC methodology by applying it to design a JBIG Encoder. JBIG stands for Joint Bi-level Image Compression Experts Group. It defines a codec system based on progressivity, which starts with a reduced resolution version of an image and builds it up enhancing it as needed by transmitting additional information [7]. The JBIG image compression standard is available as ITU-T Recommendation T.82.

1.2 JBIG Algorithm

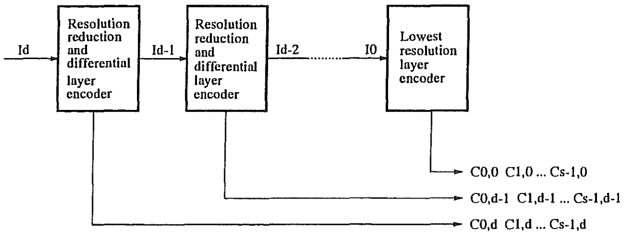

Id
Resolution reduction and differential layer encoder
Id-1
Resolution reduction and differential layer encoder
Id-2
I0
Lowest resolution layer encoder
C0,0 C1,0 ... Cs-1,0
C0,d-1 C1,d-1 ... Cs-1,d-1
C0,d C1,d ... Cs-1,d

Figure 1: JBIG Algorithm.

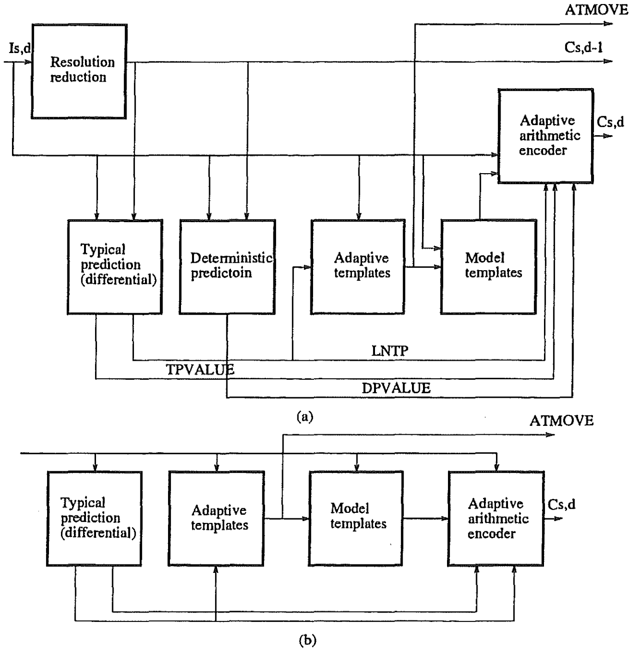


Figure 2: Functional Blocks of JBIG. (a) Differential Layer Encoding (b) Lowest Layer Encoding

The JBIG algorithm implementation used in this project is also available as Recommendation — International Standard prepared by the *Joint Bi-level Image Expert Group* (JBIG) of *ISO/IEC JTC1/SC29/WG9* and *CCITT SGVIII*. The JBIG experts group was formed to establish a standard for the progressive encoding of bi-level image (an image that, like a black-and-white image, has only two colors). A progressive encoding system transmits a compressed image by first sending the compressed data for a reduced-resolution version of the image and then enhancing it as needed by transmitting additional compressed data, which builds on the already transmitted. The JBIG standard defines a method for lossless compression encoding of a bi-level image which means the encoding is distortionless and the final decoded image is identical to the original. The method also has "progressive" capability. When decoding a progressively coded image, a low-resolution rendition of the original image is made available first with subsequent doublings of resolution as more data is decoded.

Let D denote the number of doublings in resolution (called differential layers) provided by the progressive coding. Let ID denote the highest resolution image and let its horizontal and vertical dimensions in pixels be XD and YD. Then the general encoding process can be decomposed as shown in Figure 1. Each of the resolution reduction and differential layer encoder blocks of Figure 1. is identical in function and can be decomposed further as in Figure 2(a).

Resolution Reduction (RR) This block performs resolution reduction. This block accepts a high-resolution image and creates a low-resolution image with half as many rows and half as many columns as the original.

Typical Prediction (TP) This block provides some coding gain, but its primary purpose is to speed implementations. It looks for regions of solid color and when it finds that a given current high-resolution pixel for coding is in such a region, this pixel needs not to be encoded.

Deterministic Prediction (DP) This block provides coding gain. When images are reduced in resolution by a particular resolution reduction algorithm, it sometimes happens that the value of a particular current high-resolution pixel to be coded is inferable from the pixels already known to both the encoder and the decoder, that is, all the pixels in the lower-resolution image and those in the high-resolution image that are causally related to the current pixel. When this occurs, the current pixel is said to be deterministically predictable. The DP block flags any such pixels and inhibits their encoding by the arithmetic coder. DP is a table driven algorithm. The values of particular surrounding pixels in the low-resolution image and causal high-resolution image are used to index into a table to check for determinicity and, when it is present, obtain the deterministic prediction.

Adaptive Templates (AT) This block provides substantial coding gain on images rendering greyscale with halftoning. AT looks for periodicity in the image and on finding it changes the template so that the pixel preceding the current pixel by this periodicity is incorporated into the template. Such changes are infrequent, and when one occurs, a control sequence (indicated symbolically by *ATMOVE* in the figreffig:jbig2.) is added to the output datastream.

Model Templates For each high-resolution pixel to be coded, the MT block provides the arithmetic coder with an integer called the context. This integer is determined by the values of particular surrounding pixels in both the high-resolution and low-resolution images.

Adaptive Arithmetic Encoder (AAE) This block is an entropy coder. It notes the outputs of the TP and DP blocks to determine if it is even necessary to code a given pixel. Assume it is, it then notes the context and uses its internal probability estimator to estimate the conditional probability that the current pixel will be a given value.

The lowest resolution layer encoder is conceptually simpler than the differential-layer encoder because the RR and DP blocks are not applicable and the TP, AT and MT blocks are different since there is no lower resolution layer to be used as input. This can be seen in Figure 2(b).

2 Specification Capture

2.1 JBIG Specification

Our first objective was to understand the JBIG algorithm and identify the control flow and data dependencies among the various functional blocks of JBIG. The motivation behind this was to arrive at an executable specification of JBIG. We started off with an implementation that was available as a portable, free JBIG library, in C. Most of the encoder functionality was however coded as a single function. Thus, the C code was modified by making it more modular.

We had certain objectives while rewriting the available implementation. We had to make it suitable for profiling. We would have to analyze the usage of the functions and the operations within the functions. We tried to make the implementation as close to the specification as possible. As a result, this would expose maximum possible parallelism. The T.82 specification was used to arrive at the more granular code.

2.2 JBIG Computation

In the following we discuss some aspects of the JBIG implementation.

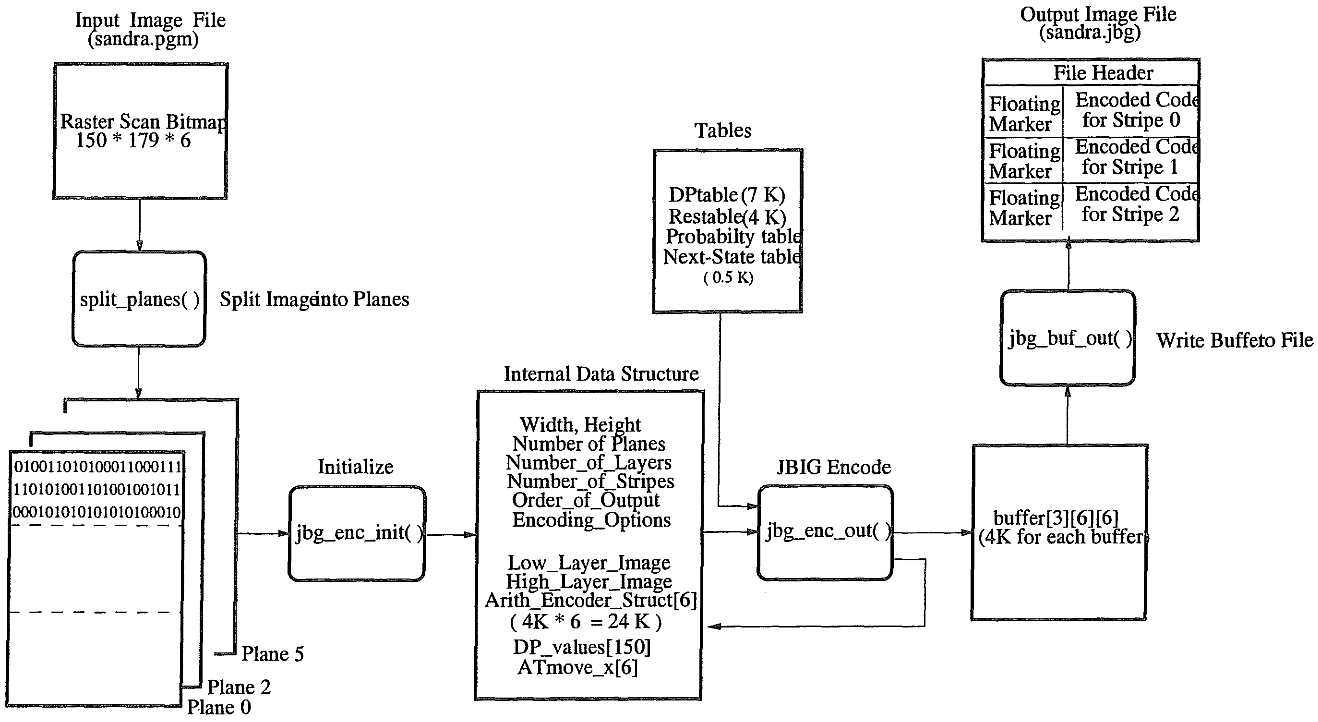


Figure 3: JBIG Input/Output.

Figure 3 shows the input/output context for the JBIG Encoder. The input image is read from a bitmap file and is formatted into the appropriate number of planes. This formatting is accomplished by the function *split_planes()*. Each plane is divided into several stripes. *jbg_enc_init()* uses the image information to initialize the data structures used by JBIG Encoder. *Width* and *Height* are the width and height of the image in pixels. In addition there are other variables initialized such as *Number_of_Planes, Number_of_Layers, Number_of_Stripes, Order_of_Output, Encoding_Options. Low_Layer_Image* and *High_Layer_Image* are pointers to the low and high resolution layer images at a time. There are other data structures as well like for instance to hold information on ATMOVE. JBIG also uses several static tables like the DP table and Resolution table. It uses an output buffer which can go up-to 4KB at a time. *jbg_enc_out()* does the actual encoding of the image and stores the code information in the buffer, while *jbg_buf_out()* writes the buffer to the output image file.

The output image file has a specific image format as can be seen in the figure. The coded image is represented as one or more Bi-level Image Entities (BIEs). Each BIE is composed of a Bi-level Image Header(BIH) and a Bi-level Image Data(BID). The header information(BIH) contains some global image information along with the static tables which were used to encode the image. A BID is a concatenation of stripe data entities(SDEs) and marker segments. An SDE encapsulates an encoded stripe of the image while the markers provide control information. They may for instance declare the following certain number of bytes to be comments.

The C functions from the implementation were encapsulated into SpecC behaviors and the new specification was verified using a test bench and some test images. The output of this phase was the SpecC behavioral model.

2.3 Behavior Model

We finally arrived at the behavior model in SpecC as can be seen in Figure 4. It shows the various behavior blocks of JBIG. The whole of JBIG behavior has been encapsulated as *jbg_enc_out*. It is composed of two sub-behaviors *write_BIH* and *output_sde*. The former, is a leaf behavior which writes the header information for a stripe of image into the output file (described in the previous subsection) and the later writes the stripe data entity into the output file. *output_sde* is further composed of two mutually exclusive sets of behaviors: set 1 - {*jbg_buf_output*} and set

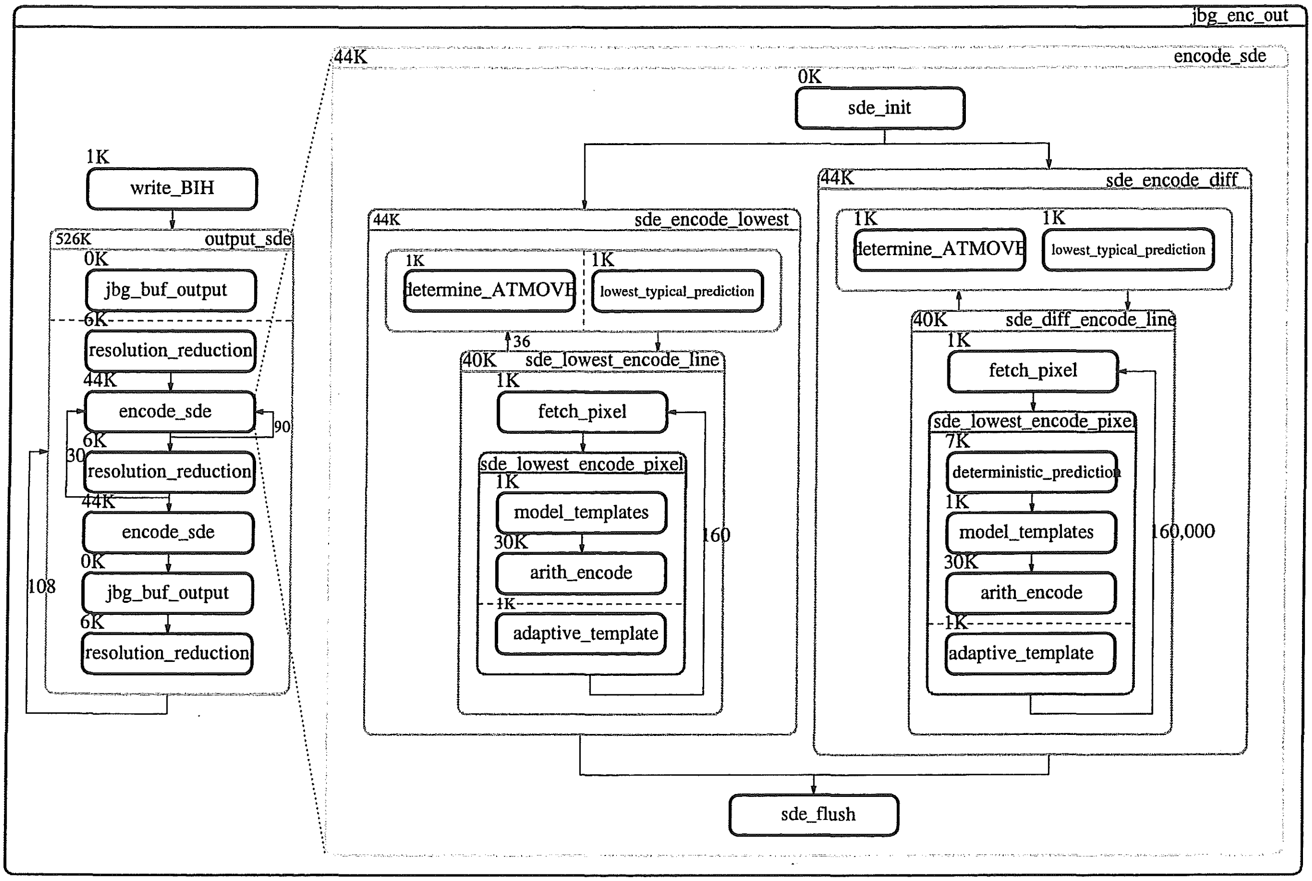


Figure 4: SpecC Behavioral Model.

2 - {*resolution_reduction*, *encode_sde*, ...}. The idea has been borrowed from the C implementation we got, where a stripe data entity(sde) is output directly if it has already undergone reduction and encoding(set 1) or else is passed through resolution reduction and encoding before being output(set 2). *encode_sde* is further exploded into a leaf behavior *sde_init* followed by either *sde_encode_lowest* or *sde_encode_diff*, followed by *sde_flush*.

sde_init and *sde_flush* perform initialization and cleanup respectively for the encoding process. The functionality of the other behaviors have already been discussed in Section 1.2. In the figure, an arrow from one behavior to another implies a sequential flow of control, while a dotted line of separation implies a possible parallel execution of the behaviors. Each label on the top left of a box in the figure gives an estimate of the memory requirements for the corresponding block.

Our next objective was to do HW/SW estimation followed by an optimal partitioning and arrive at an architectural model for JBIG.

3 Architecture Exploration

3.1 Crude Profiling

We used the modified C code in order to do preliminary profiling. We found out that the C code had to be further refined in order to reduce the input-output overhead for certain functions. Profiling was done to determine the number of operations executed in each function. This would serve as a crude Software estimate. The easiest way to profiling was using an Instruction Set Simulator (ISS, for a specific processor, DSP 56600 in our case). However the dynamic memory requirements of the JBIG implementation were so huge that the ISS approach did not work.

3.2 HW/SW Partitioning

3.2.1 FSMD Modeling

There were 6 functional blocks within JBIG. Each of them could potentially be implemented as a HW

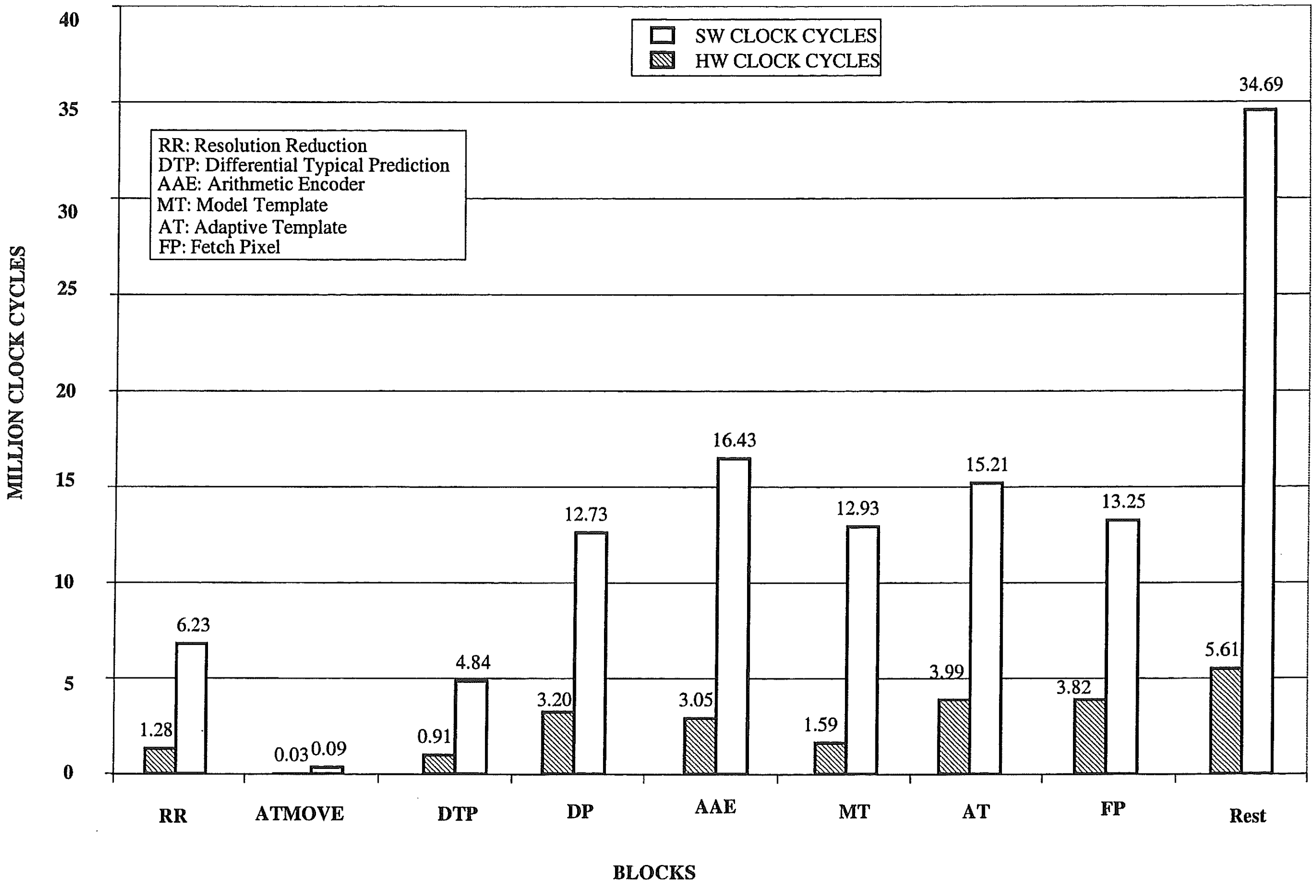


Figure 5: HW/SW Estimation.

block(ASIC). Each block was thus analyzed in order to arrive at its FSMD model. Based on the functionality of the block, an architecture was sketched. This architecture consisted of a control unit and other components such as register files, ALUs, and multiplexers. In hardware, certain functions could be more efficiently implemented than the way they were implemented in C. A control flow chart was also obtained for each block in order to assist generation of the FSMD.

An FSMD model can be derived from such an RTL architecture consisting of a control unit, controlling a datapath. The controller comprises output logic and next state logic. Thus, it exercises the datapath every clock cycle. This clock cycle can be determined by computing the longest delay path in the RTL architecture. Since we know the functionality to be performed using the architecture and also the operations which can be performed in every clock cycle, we can estimate the number of clock cycles (states) required to execute the function. This gives us an estimate of the delay of the control unit. The delay of other components can be obtained by mapping the components of the architecture to the components from an actual technology library. This way we can estimate the critical path delay and hence the clock cycle time. At the end of this phase we also get the RTL model of the specification.

3.2.2 Specification Refinement and Estimation

In order to estimate the HW/SW performance we needed to compute the number of clock cycles taken for execution of the the functional block of JBIG. To simplify the analysis, we divided each functional block into its *basic blocks*, in which control flow enters at the top and leaves only at the end. Then, using a test image, we captured the number of times each of the basic blocks was executed. Given this, we could go ahead and determine the performance as follows:

HW performance:

$$\sum_{bb} S_{bb} N_{bb}$$

where *bb* is a basic block, S is the number of states in the basic block using the FSMD model as the reference, N is the number of times the basic block is executed.

SW performance:

$$\sum_{bb}\sum_{i} N_{bb,i}$$

where *i* corresponds to an instruction in the basic block and N is the number of clock cycles required to execute the instruction on a target microprocessor.

For the HW performance, we know exactly how many states the basic block has using the FSMD model. Each state in the FSMD model corresponds to one clock cycle. Thus the first equation directly gives the required performance estimate.

The HW/SW performance results were thus tabulated. They can be seen in Figure 5. The horizontal axis indicates the different functional blocks of the whole of JBIG. The last one i.e., *Rest* includes the remaining blocks, such as processing the input bitmap image, writing the output of AAE into the output file etc. For each block we can see its performance in custom hardware as well as Software, in which case it would be running on a microprocessor. For the performance estimation, we used Motorola DSP 65500 as the underlying processor.

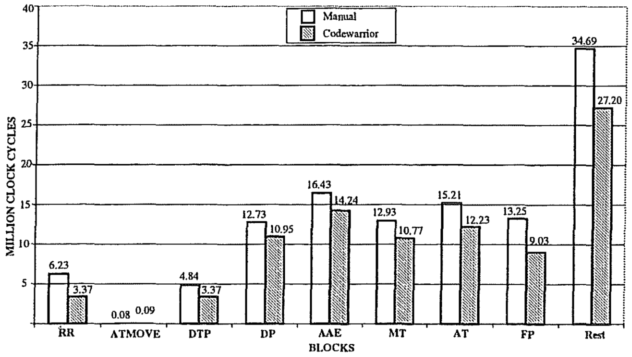


Figure 6: Comparison of Manual and Codewarrior SW Estimation.

Meanwhile, Codewarrior, a profiling tool was installed. We used Codewarrior and performed SW estimation again and found out that the manual estimation was comparable to that of Codewarrior. The comparative results can be seen in Figure 6.

The differences can be explained as follows. While we assumed the underlying processor to be Motorola DSP 56600, Codewarrior used the Intel Pentium model. Also, we were looking at code unoptimized by a compiler, at the source level (rather than an optimized assembly code). As a result, certain operations that may have been eliminated by a compiler were also accounted for during manual estimation.

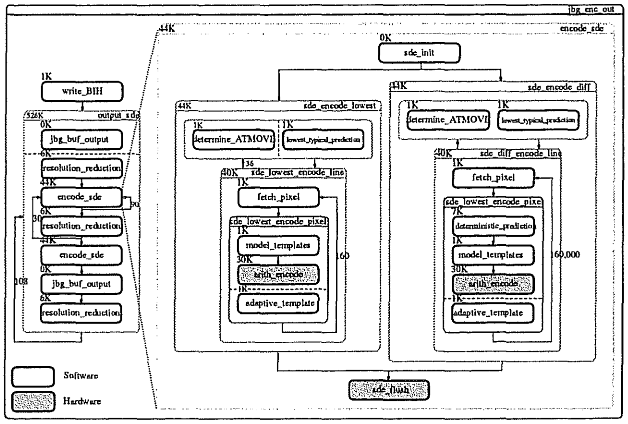


Figure 7: Solution 1.

3.3 Partitioning

Knowing the HW/SW performance of each block, we could consider different partitioning solutions. For each partition, we could compute the number of clock cycles required for the HW block(s), number of clock cycles required for the SW block(s) and hence the total number of clock cycles. Naturally, the more the functionality was put into HW, the less was the required number of clock cycles. However, each partition was also associated with a communication overhead in terms of the amount of data transferred at the interface. Based on the communication overhead, certain decisions regarding local HW memory and shared global memory were made.

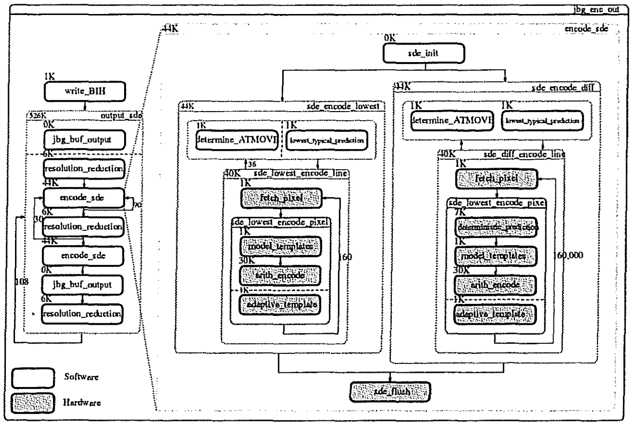


Figure 8: Solution 2.

The result of this phase was 5 partitioning solutions varying from pure software solution to most of the JBIG Encoder in hardware. In the complete software solution, all the behaviors as can be seen in Figure 4 are implemented in software. Figure 7 shows Solution 1. Here, *arith_encode, SDE_flush* are implemented in HW. We note that the corresponding blocks in both *sde_encode_lowest* and *sde_encode_diff* are implemented in HW. Other solutions are dealt similarly since the hardware required to perform the functionality are similar for the lowest resolution layer as well as the differential layers. In Solution 2, as seen in Figure 8, *sde_lowest_encode_line, sde_diff_encode_line and SDE_flush* are implemented in hardware. In other words, the behaviors for line encoding which are *sde_lowest_encode_line, sde_diff_encode_line* are mapped to hardware. Figure 9 shows Solution 3, where the entire *encode_sde* is implemented in hardware. This means that the behavior to encode a whole stripe data entity(Section 1.2) runs as an ASIC. In Solution 4, Figure 10 everything other than *jbg_buf_output, write_BIH* are implemented in hardware. JBIG_LITE implements all of sde_encode_lowest except determine_ATMOVE in hardware.

The performance of each of these partitions can be seen in Figure 11. There are up to three bars per solution. The dark solid bar gives the performance of the HW blocks, the empty bar gives the performance of the SW blocks and the light solid bar give the total performance. As expected, the SW Solution takes the maximum execution time, since software tends to be slower than an ASIC. Likewise, Solution 4 yields the best performance since the maximum number of blocks are implemented in hardware. For each of the solutions, the ASIC structure is almost the same. The other factor of consideration would be the communication overhead.

3.4 Architecture Model

For each solution, the communication overhead was computed in terms of the number of the total number of bytes of data that may need to be transferred. Figure 12 tabulates the communication occurring in each partition. This represents the communication at the HW - SW interface. We compute it by first computing the communication overhead when the HW - SW interface is exercised once. Then we compute the *communication usage* which gives the number of times such a data transfer needs to occur. Thus the total communication overhead is computed by multiplying the above two quantities. For the SW solution, it is 0. Among the other solutions, it is the lowest for Solution 4 since most of the blocks are implemented in HW.

We also tabulated the partitioning solution used by Motorola JBIG_LITE architecture and compared it with our own partitioning solutions. Interestingly, the performance of JBIG_LITE was close to the performance of the pure software solution. This observation was a chief motivation to experiment with one of our own and better partitioning solutions.

Each solution defined an architecture model. In general, architecture model consisted of three major components namely, ColdFire core to run the SW blocks of JBIG, system memory and the JBIG_HW, implementing the HW blocks of JBIG as can be seen in Figure 13. We thus have defined the architecture model.

Among all the solutions we had to choose one of them and proceed in our methodology. To simplify things, we chose Solution 1. We therefore discuss the

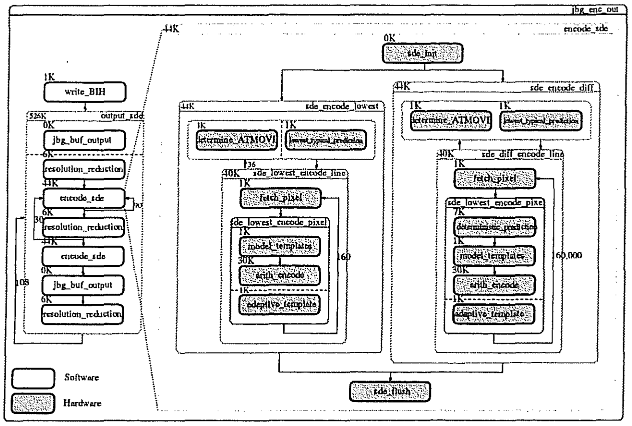


Figure 9: Solution 3.

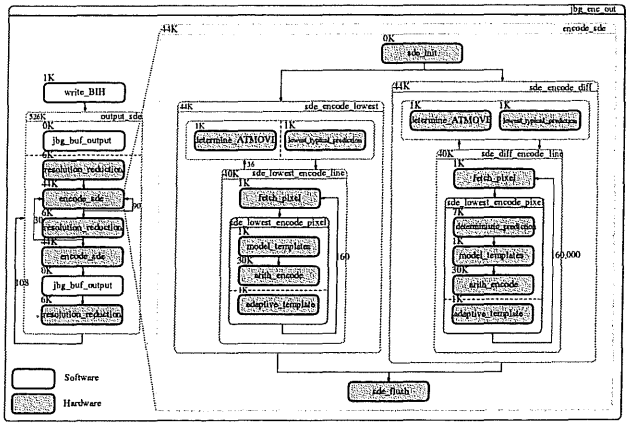


Figure 10: Solution 4.

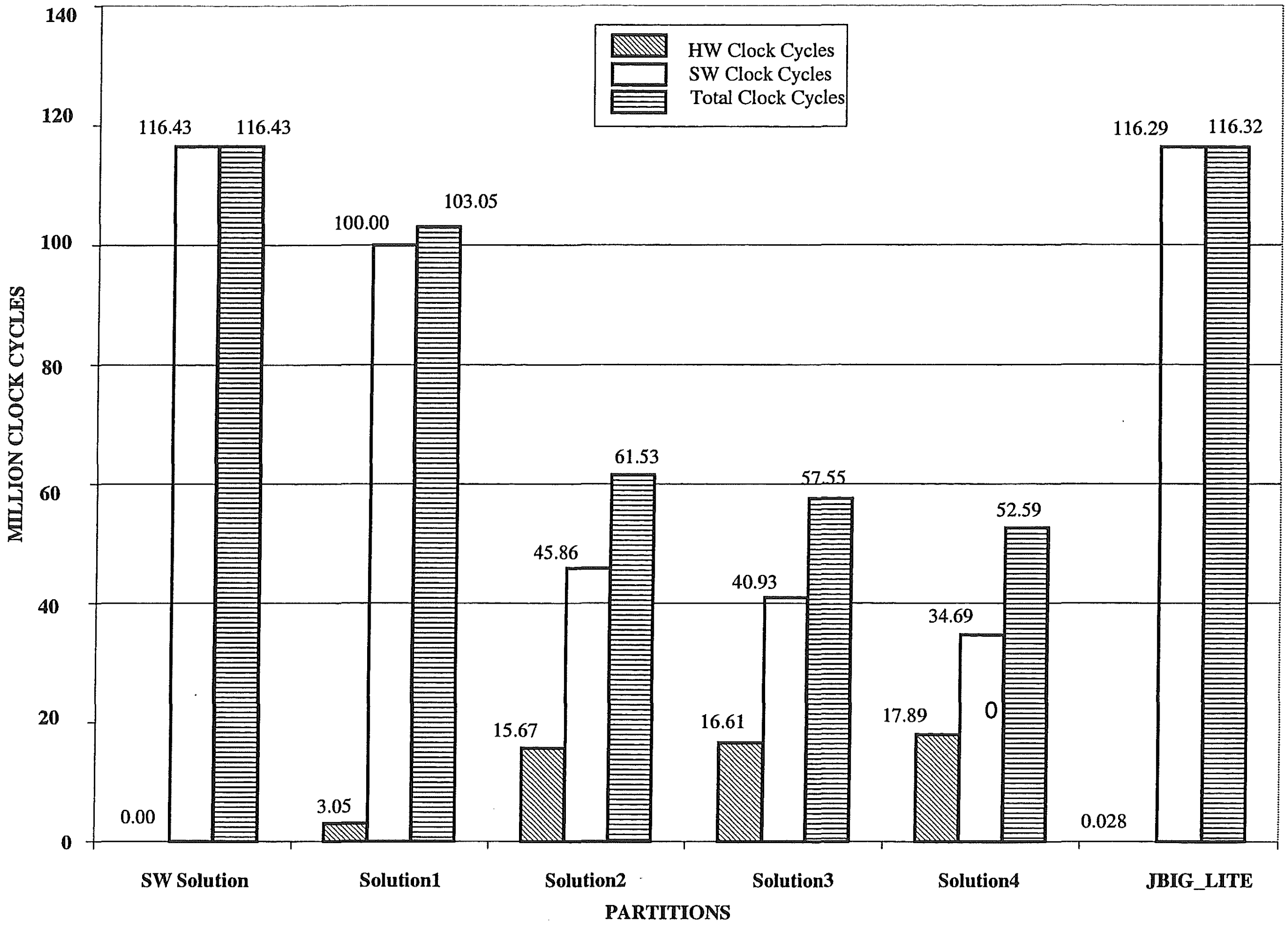
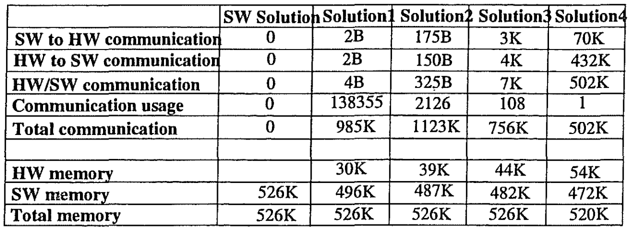


Figure 11: Performance of Partitions.



	SW Solution	Solution1	Solution2	Solution3	Solution4
SW to HW communication	0	2B	175B	3K	70K
HW to SW communication	0	2B	150B	4K	432K
HW/SW communication	0	4B	325B	7K	502K
Communication usage	0	138355	2126	108	1
Total communication	0	985K	1123K	756K	502K
HW memory		30K	39K	44K	54K
SW memory	526K	496K	487K	482K	472K
Total memory	526K	526K	526K	526K	520K

Figure 12: Communication Estimation.

other aspects of Solution 1.

Figure 14 shows the HW block implementation for Solution 1, where *arith_encode* and *SDE_flush* are implemented in HW. The control unit decides the execution state and generated appropriate signals for computation or communication via the interface/M Bus. The datapath is composed of a register file, ROM, registers and some computational units. ROM contains static tables used by the encoder.

The behavior of *arith_encode* can be summarized as follows. Before actual computation, certain values which reflect the current state of arith_encode are read from Memory. After the computation, the updated values are written back to memory. This is essential since there is a set of such registers for each plane to be encoded, and the encoding may occur in any order, where more than one plane may be active at the same time. *arith_encode* uses an index (address) into a *probability estimate table.* This table is stored in Memory.

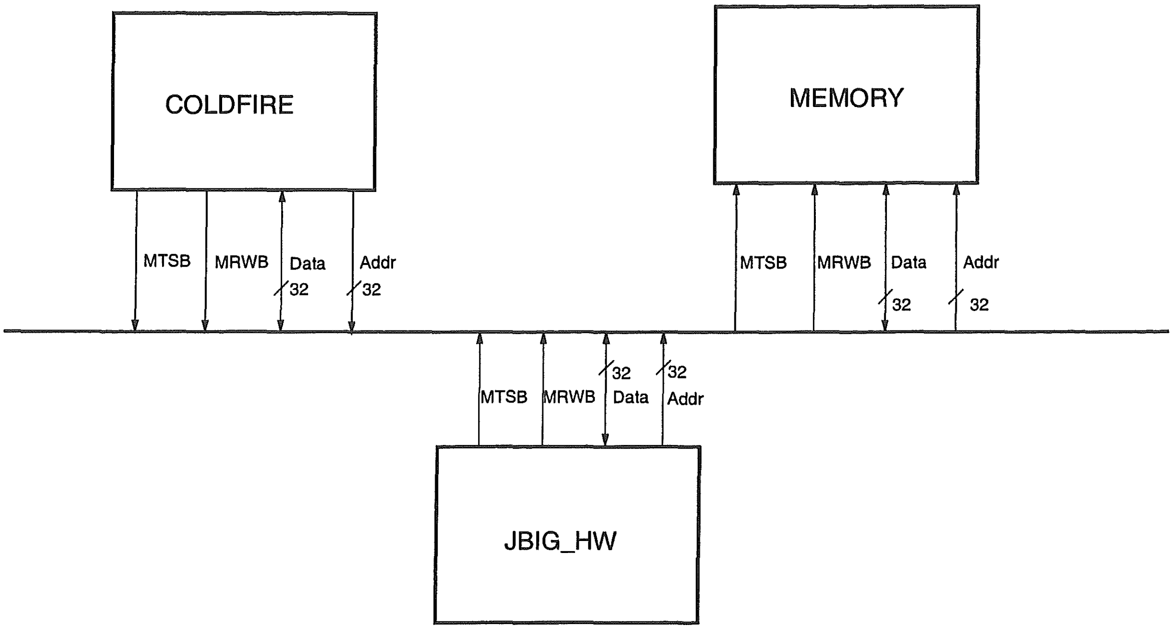


Figure 13: Architecture Model.

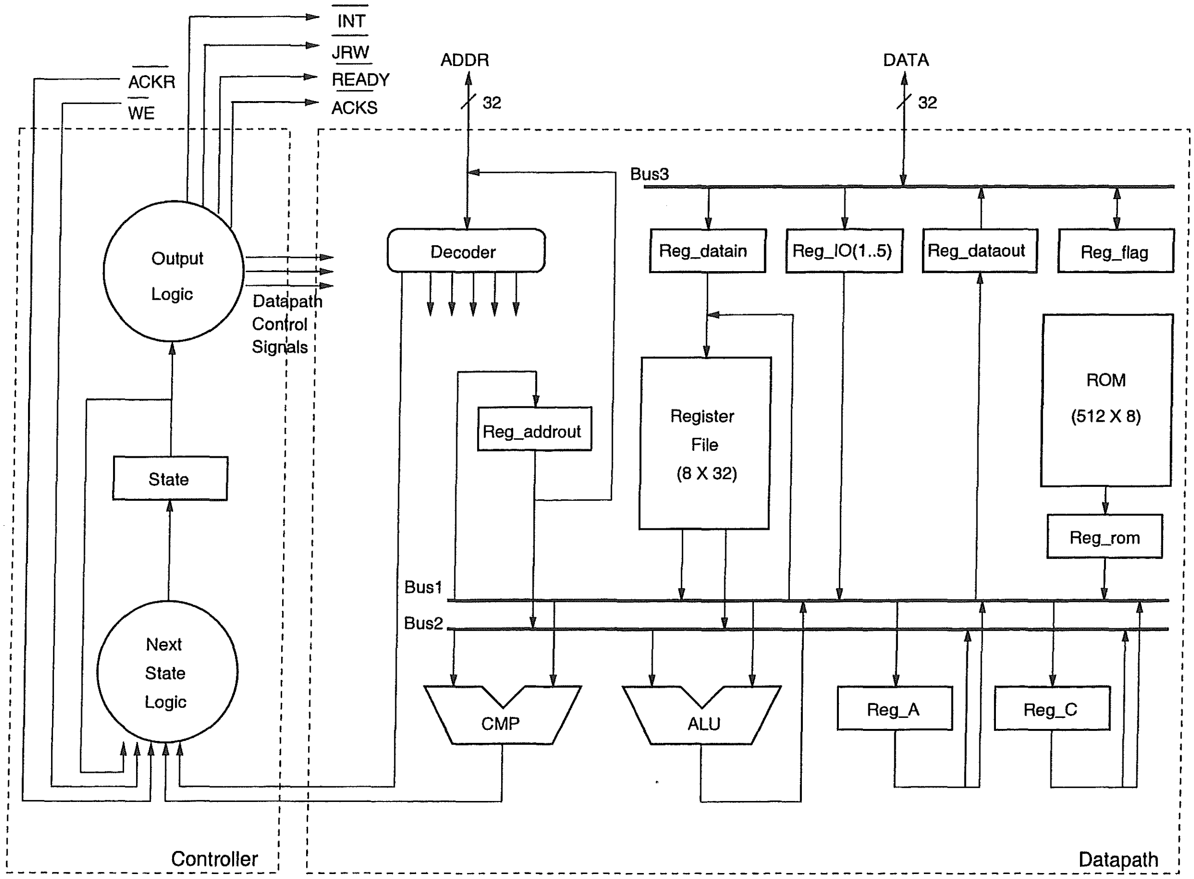


Figure 14: HW Implementation of Arithmetic Encoder.

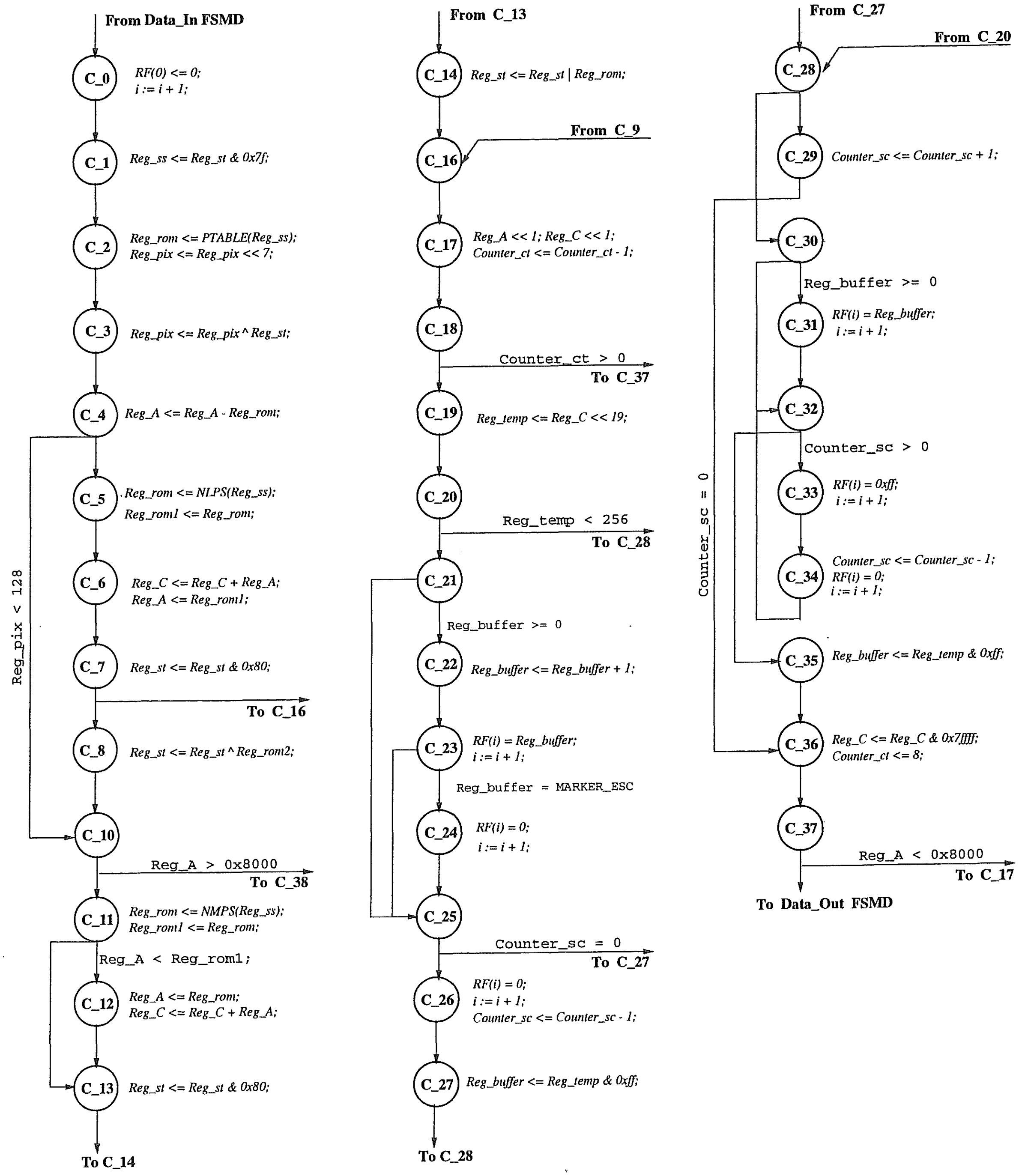


Figure 15: FSMD for JBIG_HW Computation.

Each table entry is 8 bits with the first bit giving the value of the most probable symbol(MPS which can be 0 or 1) followed by a 7 bit index into a probability estimation table stored in ROM(ST). This table, aside certain other arrays, includes the next state arrays. There are two such arrays - one to be used when the current symbol(currently processed pixel) is the same as MPS, and the other to be used when the current symbol differs from MPS. Thus, the current symbol is compared with MPS and the next state is determined. In the process, both MPS and ST may get modified. Based on the probabilities, the sequence of symbols(0s and 1s) are continuously mapped to a sequence of real numbers in their binary format, which form the arithmetic *code*. The real number is actually obtained from the size of *current coding interval*, a number between 0 and 1, which is proportional to the aforementioned probability. This interval is always divided into two - one portion for MPS and another for LPS(1-MPS).The latter gives the size of the interval for the least probability symbol and is denoted by LSZ.

The FSMD for the JBIG_HW architecture of Figure 14 implementing the above behavior can be seen in Figure 15. The index of the probability estimation table is available in Reg_st. Since only 7 bits are essential, state **C_1** masks out the first bit. State **C_2** extracts LSZ and state **C_4** computes the new coding interval(Reg_A). Then, depending upon whether the current symbol was a MPS or LPS, the appropriate coding procedure is followed. For a LPS, the next state is **C_5**; for a MPS the next state is **C_10**. This coding may modify Reg_C (the *code* register) and may end in performing *re-normalization*. *Re-normalization* occurs whenever Reg_A falls below 0x8000 and in Figure 15 this starts from state **C_17** and ends in state **C_43**. During the process of *re-normalization*, both Reg_A and Reg_C are shifted a certain number of times (indicated by Counter_ct) and then a procedure *byteout* is followed.

Byteout is implemented in states **C_19** - **C_36**. The code register contains a *completed byte of data*(extracted into Reg_temp) which is examined for an overflow in states **C_19** and **C_20**. If an overflow has occurred, states **C_21** - **C_27** output the necessary number(Counter_sc) of 0x00 bytes by virtue of the carry. This is known as *carry resolution*. Otherwise, states **C_28** - **C_35** output the required number(Counter_sc) of 0xff bytes (if *carry resolution* has completed). Finally, state **C_36** clears the portion corresponding to Reg_temp in Reg_c and initializes Counter_ct.

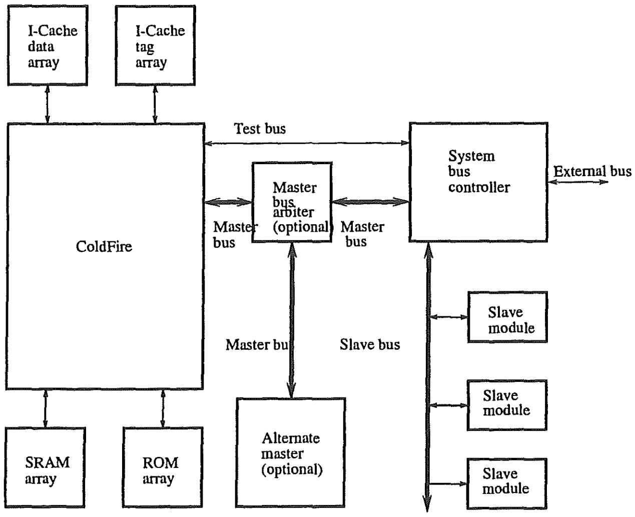


Figure 16: ColdFire System Diagram.

The ColdFire system diagram can be seen in Figure 16. We can relate it to our architecture model of Figure 13 as follows. 'ColdFire' in both the figures corresponds to the core microprocessor and other components. JBIG_HW in the architecture model is the alternate master. Memory is one of the slave modules, connected to the master bus via an interface bus controller. Only a master component can initiate bus transactions. The slave bus provides the interface between the internal master bus and the on chip peripherals and is controlled by the system bus controller. The System Bus Controller supplied along with ColdFire has programmable registers which can be used to configure the memory map and interrupt control. The master bus is the primary data interface for the ColdFire. It is a two cycle bus and devices on it are capable of initiating transactions. It is made of 32-bit address, 32-bit read data, 32-bit write data buses and control signals. We will analyze it in greater detail in the next section on communication synthesis. We will also analyze the memory component from communication point of view in the next section.

Now that we had the basic building blocks for Solution 1, the next objective was to define the communication interfaces. These are the interfaces between ColdFire [10], JBIG_HW and Memory. They can be specified as FSM models implementing appropriate communication protocols thereby defining the communication model.

4 Communication Synthesis

In Figure 13 we find that there are three components communicating via system bus. We decide to use the *Master Bus (M bus)* coming along with ColdFire System as the system bus. ColdFire interacts with JBIG_HW to give initialization information. Thus, there must be some SW running on ColdFire to handle the initialization of JBIG_HW. JBIG_HW also gets data from the memory, updates them and writes them back into memory. We thus need to design a *bus interface for JBIG_HW* that responds to the bus while talking to ColdFire and also takes care of memory read/write. On the other hand, memory must be able to respond to the read and write requests from ColdFire or JBIG_HW. This again requires design of a *bus interface for memory* to respond to bus read/write requests. We note that we need not design a similar *bus interface for ColdFire* since we are using M bus and it already has its own M bus controller.

The behavior and complexity of such interfaces depends on the time constrained behavior of the components at their ports. We hence discus the timing behavior of the components in the next subsection. We then analyze the software needed for initialization, the JBIG_HW bus interface and the memory bus interface.

4.1 Analysis of Communicating Components

4.1.1 ColdFire

ColdFire communicates via M bus. Figure 17 shows the M bus signals. A subset of those signals are described below.

Master Address Bus These signals provide the address of the first item of a bus transfer.

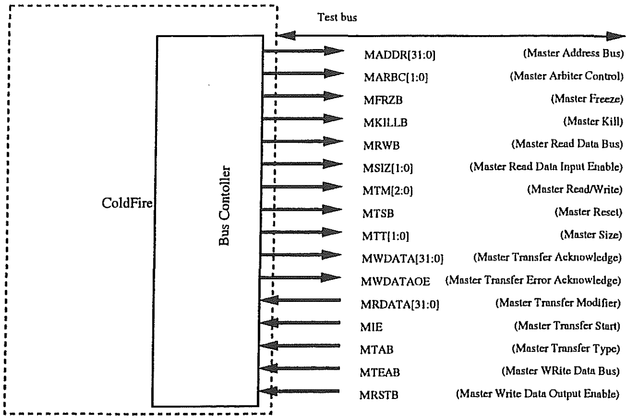


Figure 17: M Bus Signals.

Master Read Data Bus These signals provide the read path for for ColdFire and can transfer 8/16/32 bits of data per bus transfer.

Master Read Data Input Enable This signal, when active high enables capturing of MRDATA.

Master Read/Write This signal indicates the direction of data transfer for the current bus cycle.

Master Transfer Acknowledge This signal is active low and is asserted by a peripheral to indicate successful completion of the bus transfer.

Master Transfer Start This signal is active low and it indicates the start of each bus transfer.

Master Write Data Bus These signals provide write data path for ColdFire and can transfer 8/16/32 bits of data per bus transfer.

Master Write Data Output Enable This signal is active high and it indicates that ColdFire is driving the master. It may be used to control the optional bidirectional data bus three state drivers.

Figure 18 shows the timing diagram for the M bus read and write transactions. The M bus has two basic cycles which are the transfer start and transfer acknowledge cycles. In the first cycle(C1), the address(*MADDR*) and control information(*MTSB, MRWB, MIE*) are driven onto the bus. In the second cycle(C3), the transfer acknowledge signal(*MTAB*) is asserted by the slave and data becomes valid. For a read transaction, data(*MRDATA*) are latched by the M bus controller. For a write transaction, the slave can latch the data the data(*MWDATA*). If MTAB is not yet asserted, then the bus inserts wait cycles(C2); such a 'handshake' may be used to accommodate slow slaves.

4.1.2 Memory

We experimented with Samsung memory KM68257C [9], a CMOS static RAM. It has 8 common input and output lines. The different pins and the memory specification for read and write cycles are shown in Figure 19. Figure 19(a) gives the pin functions. We find that the $\overline{CS}$ signal is active low and needs to be asserted during memory read or write transactions.

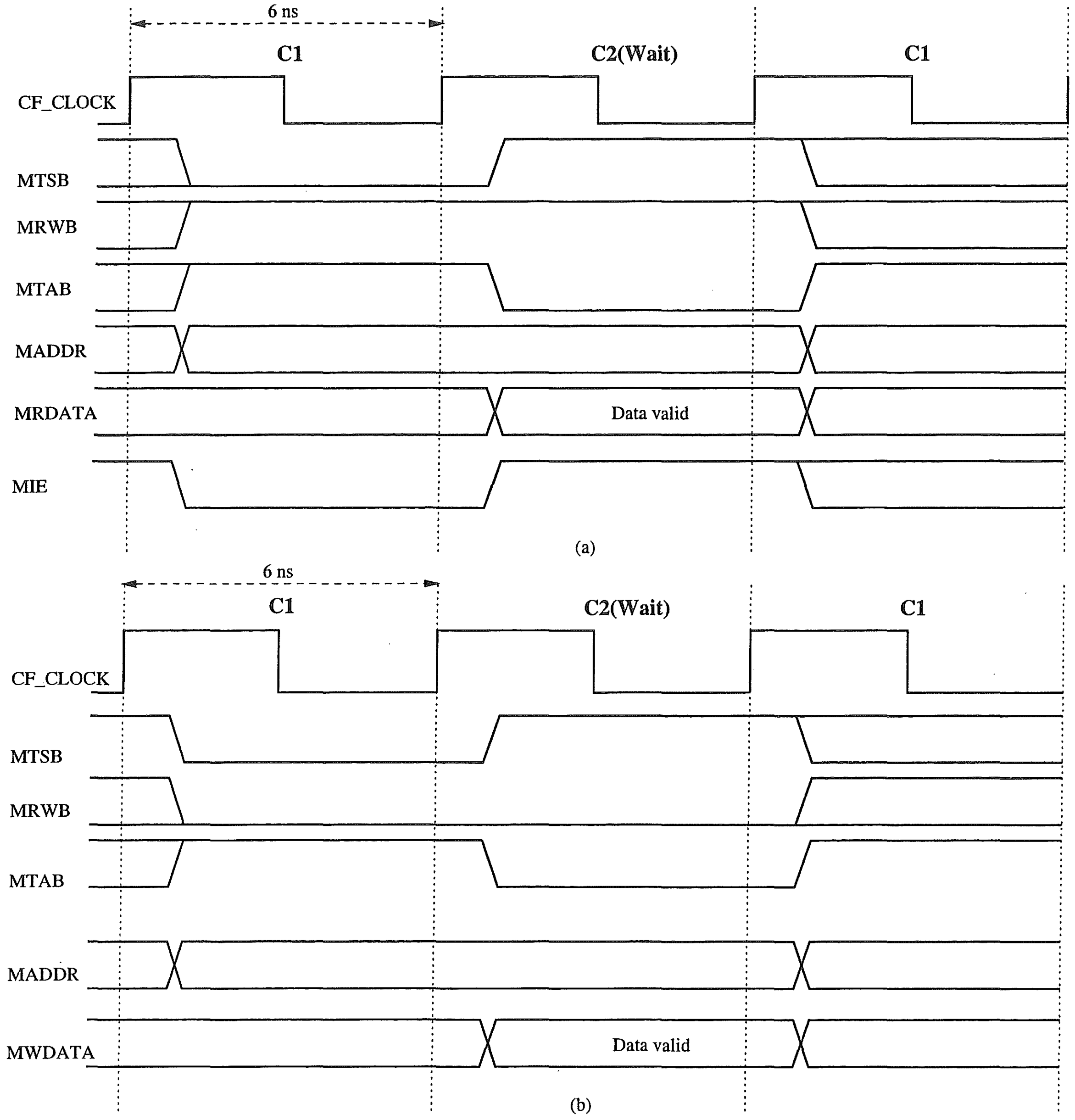
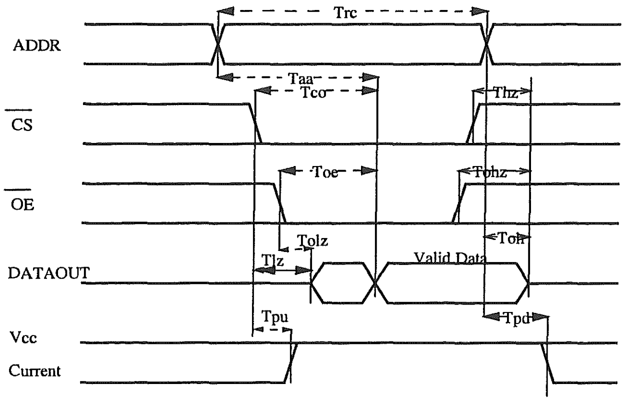


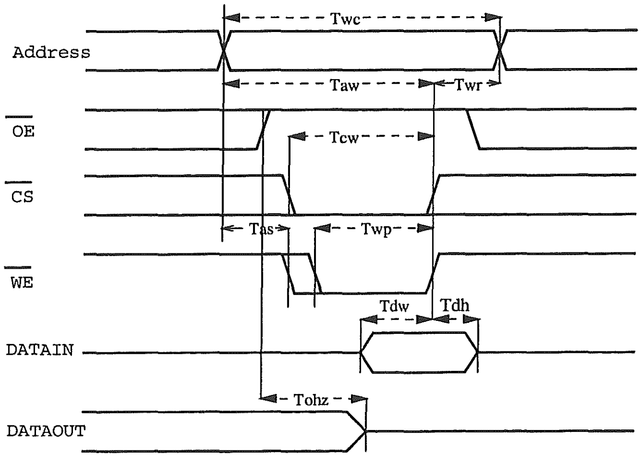
Figure 18: ColdFire Timing Diagram. (a) Read (b) Write

Pin Name	Pin Function
ADDR	Address Inputs
$\overline{WE}$	Write Enable
$\overline{CS}$	Chip Select
$\overline{OE}$	Output Enable
DATAIN/OUT	Data Inputs/Outputs
Vcc	Power(+5.0V)
Vss	Ground

(a)



(b)



(d)

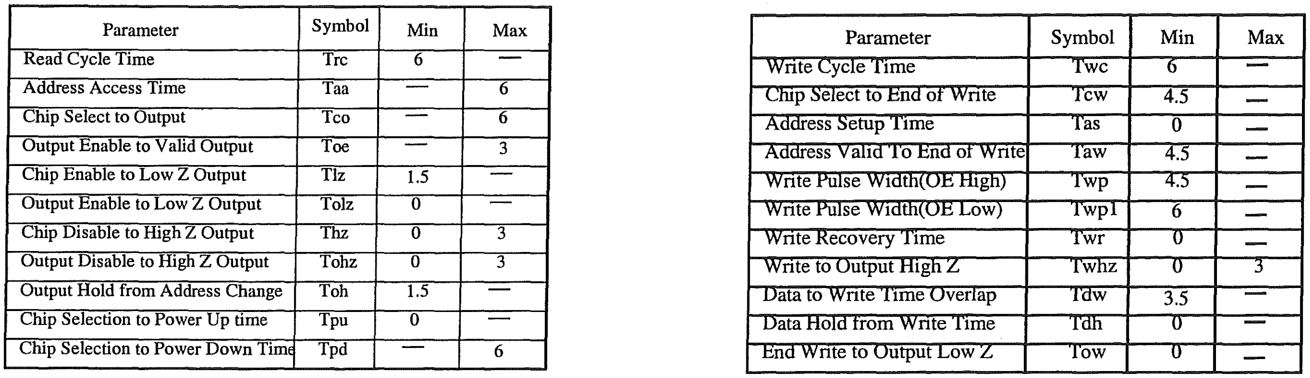
Parameter	Symbol	Min	Max
Read Cycle Time	Trc	12	—
Address Access Time	Taa	—	12
Chip Select to Output	Tco	—	12
Output Enable to Valid Output	Toe	—	6
Chip Enable to Low Z Output	Tlz	3	—
Output Enable to Low Z Output	Tolz	0	—
Chip Disable to High Z Output	Thz	0	6
Output Disable to High Z Output	Tohz	0	6
Output Hold from Address Change	Toh	3	—
Chip Selection to Power Up time	Tpu	0	—
Chip Selection to Power Down Time	Tpd	—	12

(c)

Parameter	Symbol	Min	Max
Write Cycle Time	Twc	12	—
Chip Select to End of Write	Tcw	9	—
Address Setup Time	Tas	0	—
Address Valid To End of Write	Taw	9	—
Write Pulse Width(OE High)	Twp	9	—
Write Pulse Width(OE Low)	Twp1	12	—
Write Recovery Time	Twr	0	—
Write to Output High Z	Twhz	0	6
Data to Write Time Overlap	Tdw	7	—
Data Hold from Write Time	Tdh	0	—
End Write to Output Low Z	Tow	0	—

(e)

Figure 19: Memory Specification(KM68257c). (a) Pin Functions (b) Read Timing (c) Read Constraints (d) Write Timing (e) Write Constraints



Parameter	Symbol	Min	Max
Read Cycle Time	Trc	6	—
Address Access Time	Taa	—	6
Chip Select to Output	Tco	—	6
Output Enable to Valid Output	Toe	—	3
Chip Enable to Low Z Output	Tlz	1.5	—
Output Enable to Low Z Output	Tolz	0	—
Chip Disable to High Z Output	Thz	0	3
Output Disable to High Z Output	Tohz	0	3
Output Hold from Address Change	Toh	1.5	—
Chip Selection to Power Up time	Tpu	0	—
Chip Selection to Power Down Time	Tpd	—	6

(a)

Parameter	Symbol	Min	Max
Write Cycle Time	Twc	6	—
Chip Select to End of Write	Tcw	4.5	—
Address Setup Time	Tas	0	—
Address Valid To End of Write	Taw	4.5	—
Write Pulse Width(OE High)	Twp	4.5	—
Write Pulse Width(OE Low)	Twp1	6	—
Write Recovery Time	Twr	0	—
Write to Output High Z	Twhz	0	3
Data to Write Time Overlap	Tdw	3.5	—
Data Hold from Write Time	Tdh	0	—
End Write to Output Low Z	Tow	0	—

(b)

Figure 20: Fast Memory Specifications. (a) Read Constraints (b) Write Constraints

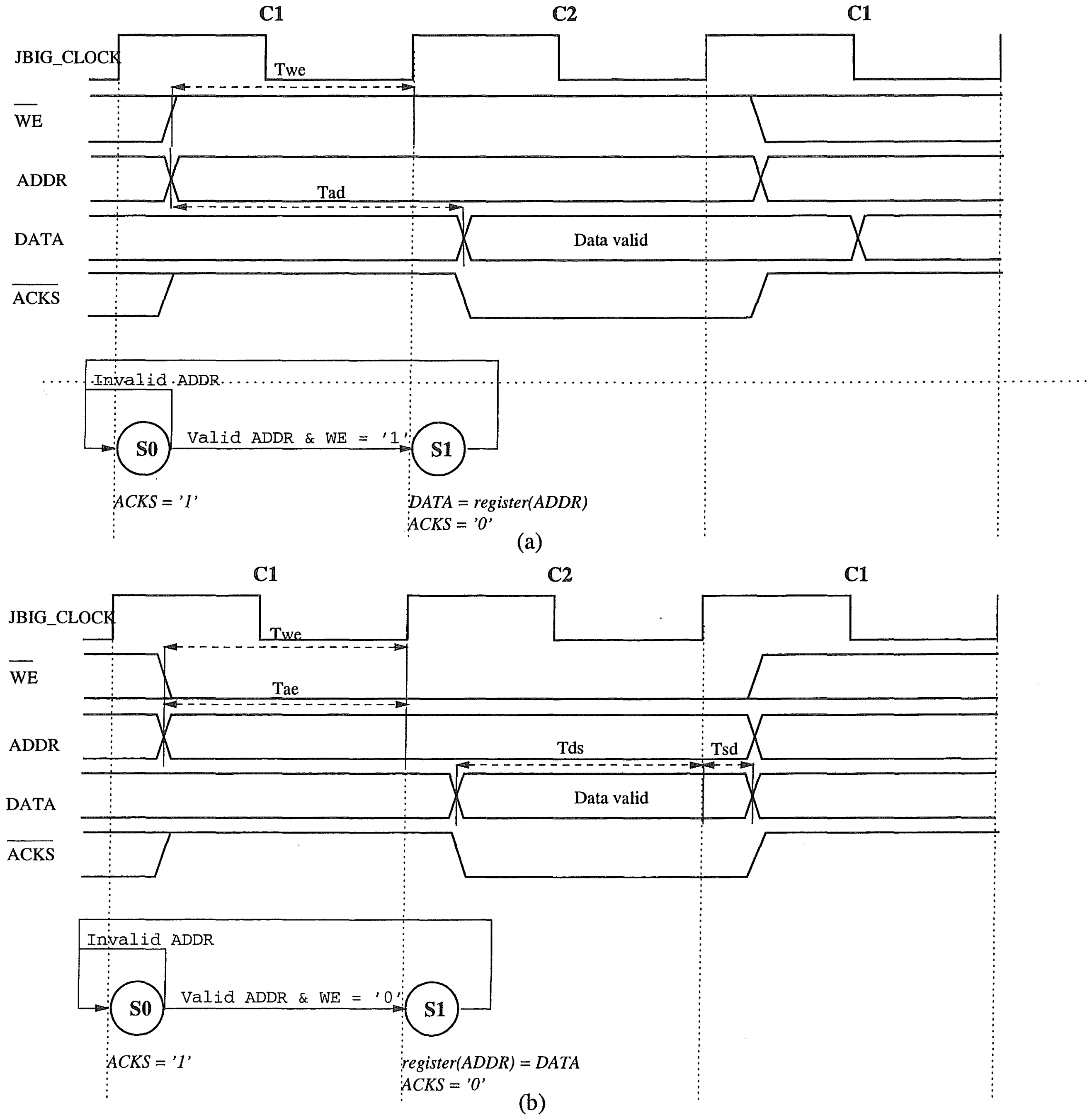
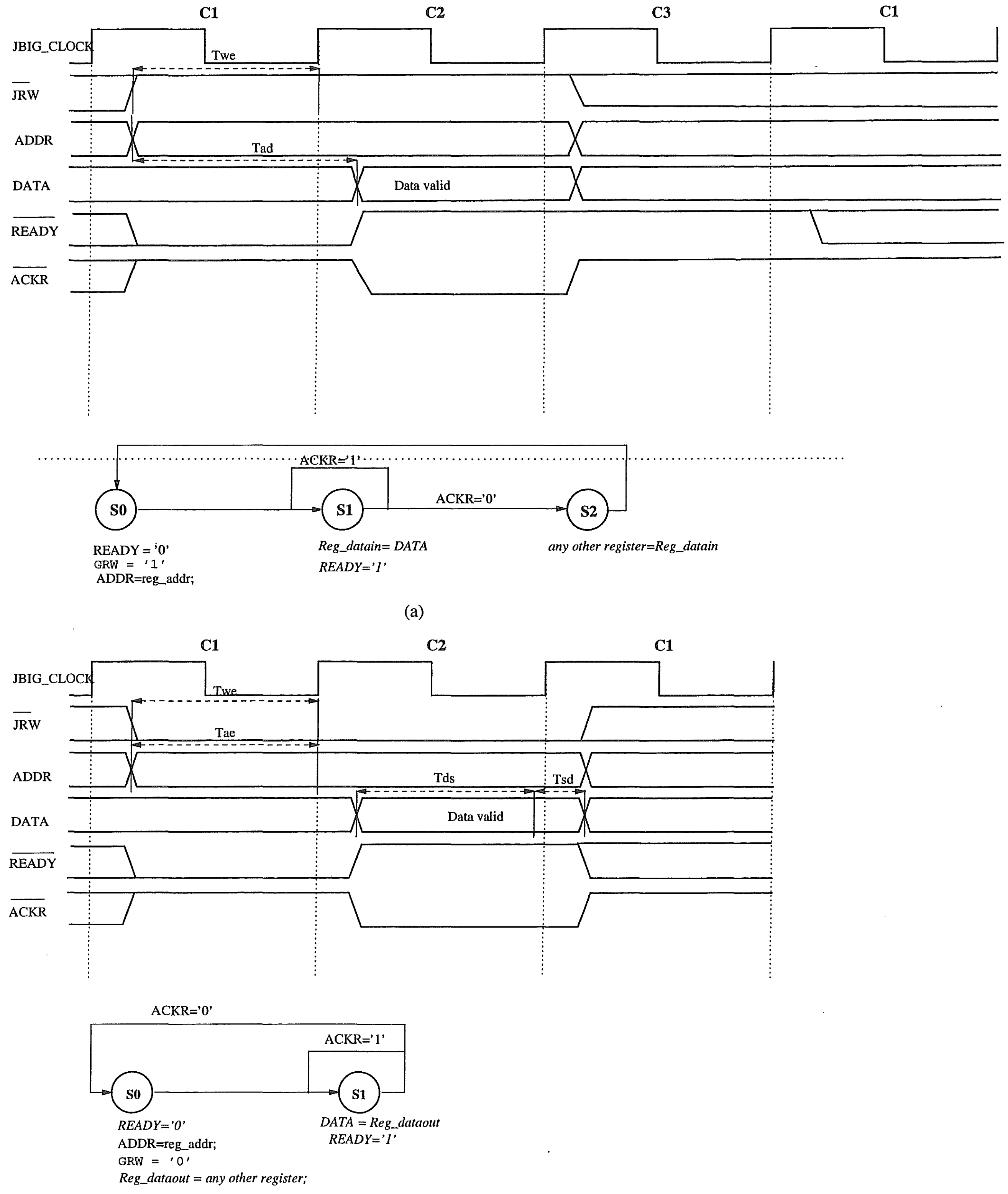


Figure 21: M Bus - JBIG_HW Timing in Slave Mode. (a) Reading JBIG_HW Registers (b) Writing JBIG_HW Registers



(a)

(b)

Figure 22: M Bus - JBIG_HW Timing in Master Mode. (a) JBIG_HW M Bus Read (b) JBIG_HW M Bus Write

Read Cycle Figure 19(b) shows the timing waveform of the read cycle. Figure 19(c) shows the description of the different constraints in the timing waveform and their values. $\overline{WE}$ is high for a read cycle. The extreme reference points for any of the given timing constraint are the end points of *Trc*. Also, the address has to be valid prior to chip selection. We note that the address access time, which gives the maximum possible delay between transition to valid address and transition to valid data is 12ns, which implies that after 12ns we can be sure of finding the required data on the data out pins.

Write Cycle Figure 19(d) shows the timing waveform of the write cycle. Figure 19(e) shows the description of the different constraints in the timing waveform and their values. $\overline{WE}$ is low for a write cycle. The extreme reference points for any given timing constraint are similar to those for a read cycle. Write occurs during the overlap of low $\overline{CS}$ and low $\overline{WE}$. *Tdw* tells us that the data must remain valid for at least 7ns after it is driven onto the data input bus of the memory.

Figure 20 shows hypothetical memory timing constraints, which have been derived from Figure 19(c) and (e) by halving their values, i.e., this memory is twice as fast as KM68257c. We will be using this to demonstrate a simple bus interface for memory in the next section.

4.1.3 JBIG_HW

Figure 14 shows that JBIG_HW has 8 ports that may be used for communication. $\overline{INT}$ can be used to interrupt the ColdFire processor. *DATA*, *ADDR* are used for receiving or sending data and address respectively. JBIG_HW may receive address when it functions as a *slave*. $\overline{WE}$ and $\overline{JRW}$ are used for indicating read/write while JBIG_HW is a slave and *master* respectively. $\overline{ACKS}$ *is used for sending acknowledgment while JBIG_HW is a master and* $\overline{ACKR}$ is used for receiving acknowledgment while JBIG_HW is a slave. $\overline{READY}$ is used to initiate a transaction such as read or write in the master mode.

4.1.3.1 JBIG_HW as Slave

Figure 21 shows the timing diagram of JBIG_HW when it functions as a slave. Below each timing diagram we can also see a corresponding FSMD which responds to the ports.

JBIG_HW Read Figure 21(a) shows the read timing. It takes two clock cycles. At the end of C1, $\overline{WE}$ is high indicating that a read transaction is to be performed. The valid address is available at the ADDR port. This address is decoded and the data are sent out of *DATA* port. This may be latched by the reader at the end of C2. $\overline{ACKS}$ goes low in C2 indicating that JBIG_HW has received the read request. Timing constraints are defined as:

$$\begin{aligned} T_{we} &>= CF\_ClOCK - (\delta(decoder) + \delta(output\_logic) \\ T_{ad} &<= \delta(decoder) + \delta(output\_logic) \end{aligned}$$

JBIG_HW Write Figure 21(b) shows the write timing. It also takes two clock cycles. *ADDR* must be available in C1. $\overline{WE}$ must be low during a write transaction. Valid data must be available in C2, and they are latched on at the and of C2. $\overline{ACKS}$ goes low in C2 indicating that JBIG_HW has received the write request. Timing constraints are defined as:

$$\begin{aligned} T_{we} &>= CF\_ClOCK - (\delta(decoder) + \delta(output\_logic) \\ T_{ae} &>= setuptime(register(ADDR)) \\ T_{sd} &>= holdtime(register(ADDR)) \end{aligned}$$

4.1.3.2 JBIG_HW as Master

Figure 22 shows the timing diagram of JBIG_HW when it functions as a master. It is similar to the ColdFire timing of Figure 18, except that there is no signal equivalent to *MIE*. $\overline{ACKR}$ is equivalent to *MTAB*. $\overline{READY}$ and $\overline{ACKR}$ are equivalent to *MTSB* and *MTAB* respectively. $\overline{WE}$ is equivalent to *MRWB*.

JBIG_HW Read Figure 22(a) shows the read timing when JBIG_HW is a master. In C1, the memory read address are driven into *ADDR*. $\overline{JRW}$ goes high to indicate a read transaction. In C2 $\overline{ACKR}$ goes low. This means valid data are available which can be latched into JBIG_HW register(Reg_datain). Data are latched on only when $\overline{ACKR}$ is low. Such a scheme can be used to accommodate slower components during communication. In C3, the data are moved from Reg_datain to any other register.

JBIG_HW Write Figure 22(b) shows the write timing. In C1 the memory write address are driven into ADDR and the data to be written are moved into the appropriate register(Reg_dataout). In C2, the data are driven into *DATA*, and $\overline{READY}$ is asserted. The data are kept valid as long as $\overline{ACKR}$ remains high. This, like in the case of read, is used to accommodate slower components.

4.2 Interface Design

Now that we have understood the components and their means of communication, the next step is to design the necessary interfaces for communication.

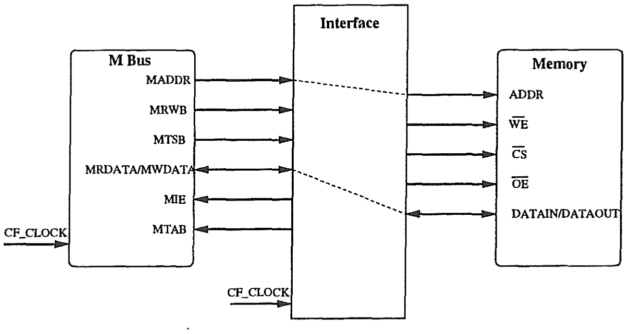


Figure 23: M Bus - Memory Interface.

Figure 23 shows the signals to be dealt with for an interface between the memory and the bus. Some of the signals are read by the interface while some are generated by the interface. Some may be either read or generated depending upon the nature of the transaction(read/write). *MADDR, MTSB, MRWB* are read by the interface and they are the address, transaction start and read/write signals respectively. *MTAB, MIE,* $\overline{CS}$, $\overline{OE}$, $\overline{WE}$ are generated and they are transaction acknowledgment, bus input enable, memory chip select, memory output enable, and memory write enable signals respectively. *MDATA* are read or generated depending on whether the transaction is a read or write transaction. We use the bus clock as the clock for the interface.

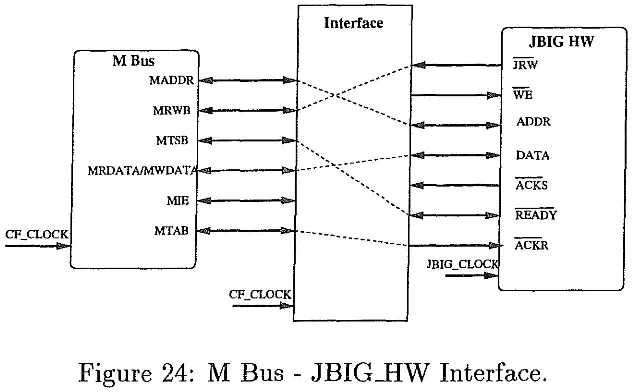


Figure 24: M Bus - JBIG_HW Interface.

Similarly, Figure 24 shows the signals to be dealt with for an interface between the JBIG_HW and the bus. We see that both *DATA* and *ADDR* are bidirectional since JBIG_HW can behave like a slave as well as a master. *MTAB* is the read by the interface when JBIG_HW is a master, and generated by the interface while, JBIG_HW is a slave.

Now we are in a position to discuss what is inside the *interfaces.* As mentioned earlier, it depends upon the characteristics of the components that are communicating. We will discuss two cases. In the first case, the interfaces are made simple. This would so because the components are already perfectly synchronized, i.e., their speeds match with each other and that of the communication bus. In the second case, the speeds do not match. For instance:

1. Consider that the memory puts the required data into the bus. But if the bus master, say for instance JBIG_HW, is not fast enough, invalid data will be latched. The same problem occurs if Memory is slower compared to JBIG_HW.

2. Consider the communication between ColdFire and JBIG_HW. ColdFire puts some data into the bus at its clock speed. This data must be valid until JBIG_HW is able to latch it at its own clock speed.

For this case we would need to design some kind of a *handshake* to ensure proper communication. Thus the interface we design must be robust to ensure proper communication depending upon the characteristics of the components. These include input/output ports and timing that we studied before. In the following two sub-sections, we discuss the two cases.

4.2.1 Case 1: Simple Interfaces

4.2.1.1 **M Bus - Memory**

In this case, we assume that memory is fast enough to respond to any bus read/write request. We thus use the specification of Figure 20.

Figure 26 shows the FSMD of the bus interface for Memory. We have direct connections as the following:

$$ADDR \leftrightarrow MADDR$$
$$DATA \leftrightarrow MWDATA$$

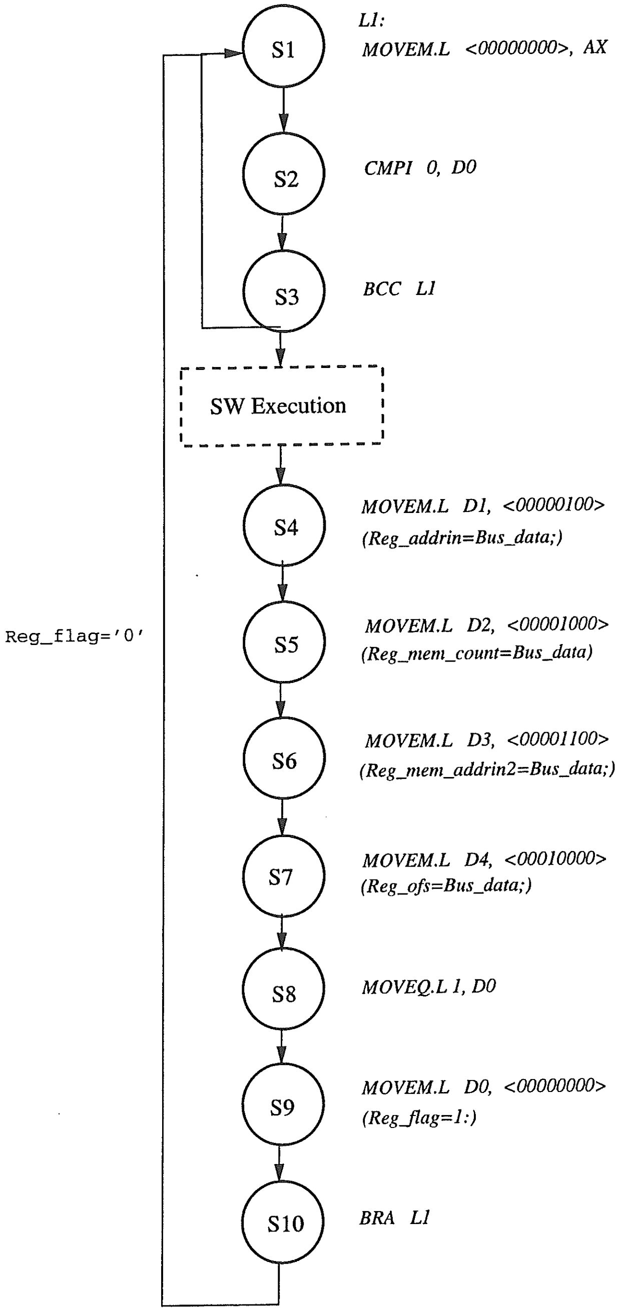


Figure 25: ColdFire Assembly Code for JBIG_HW DMA Setup(case1).

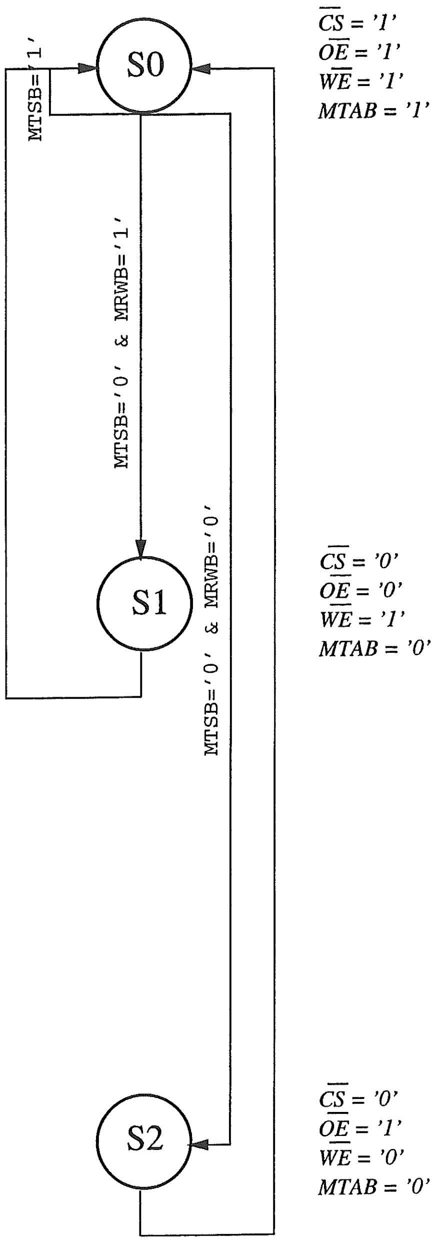


Figure 26: FSMD of M Bus - Memory Interface.

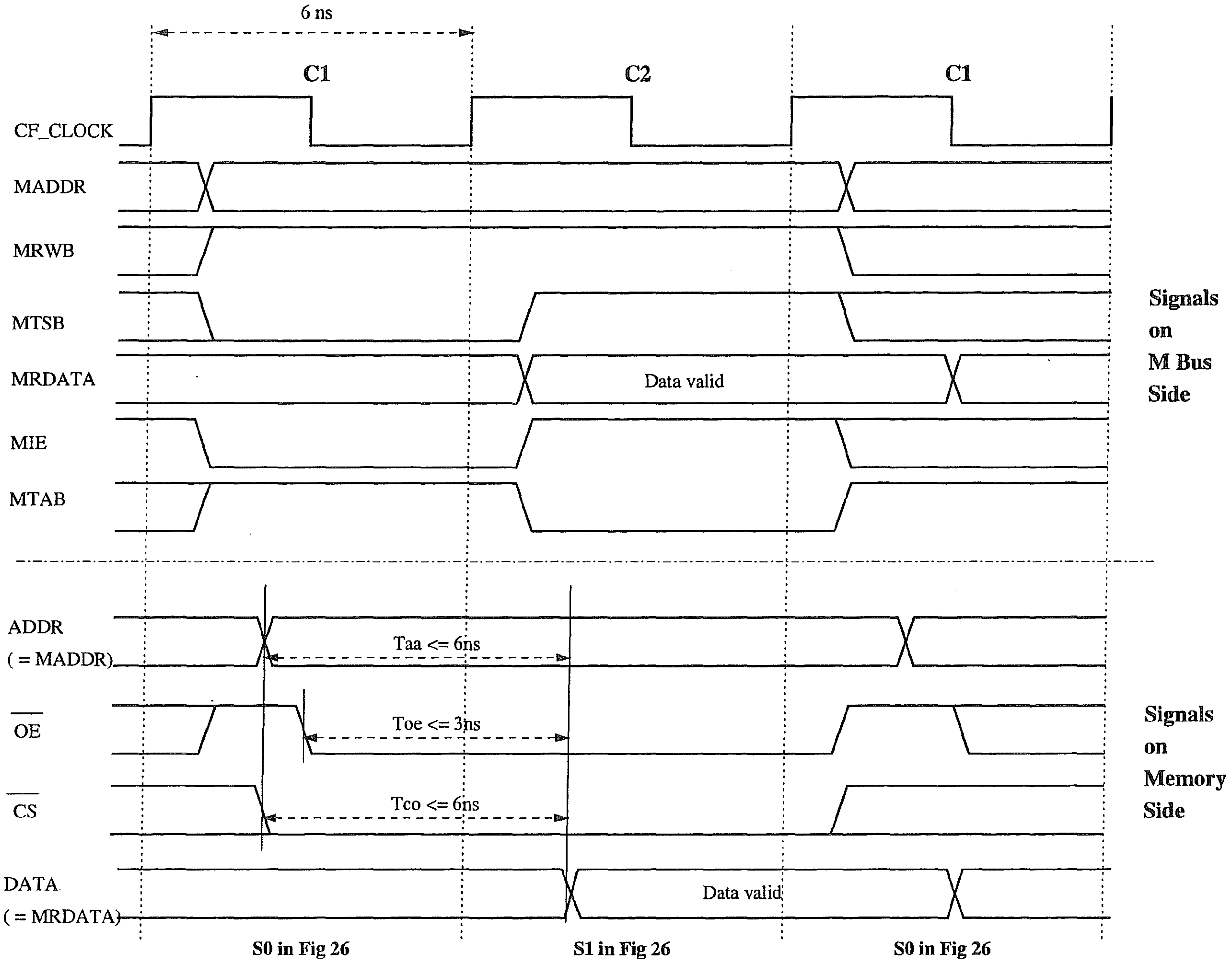


Figure 27: M Bus - Memory Read Timing(case1).

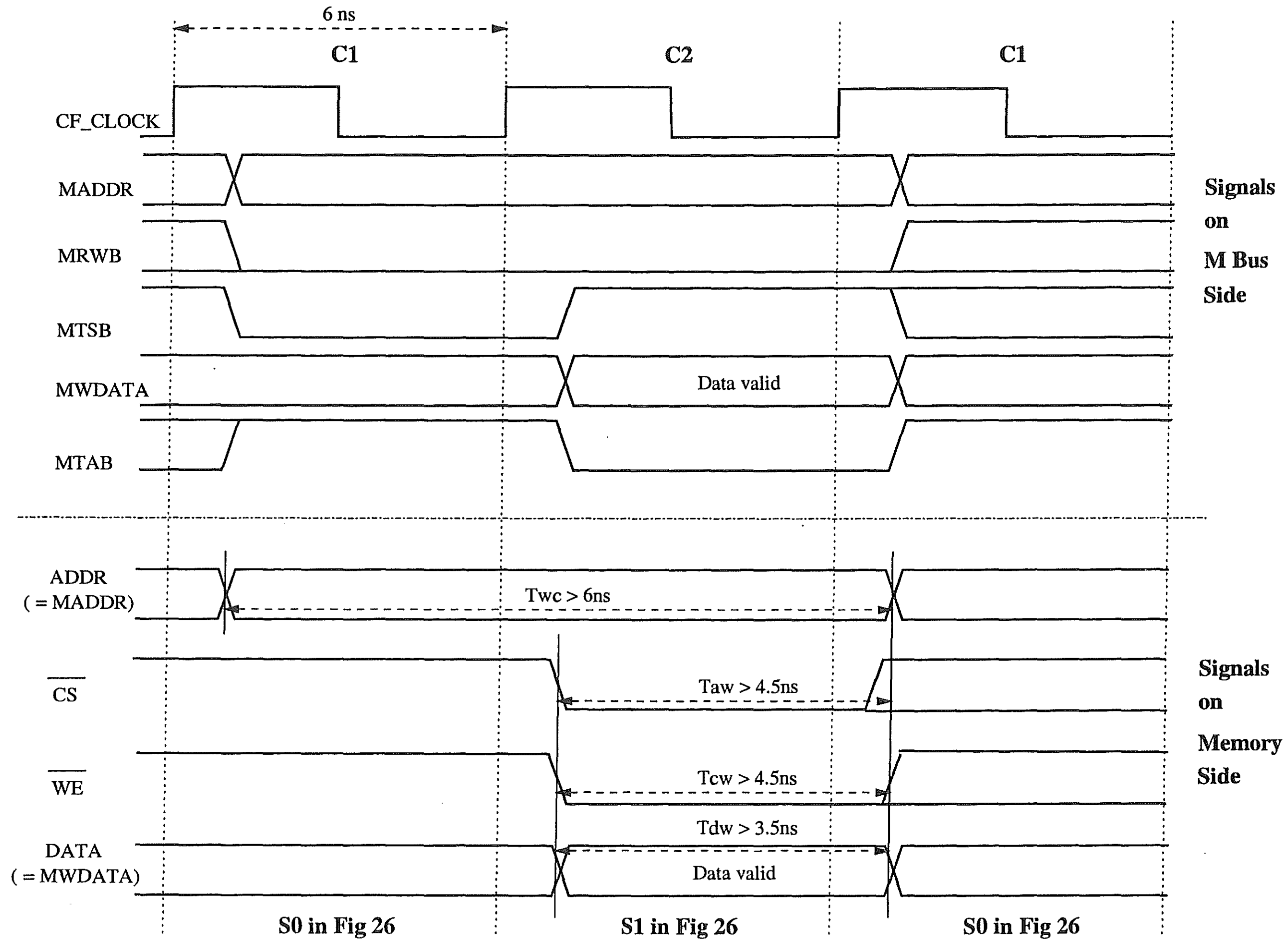


Figure 28: M Bus - Memory Write Timing(case1).

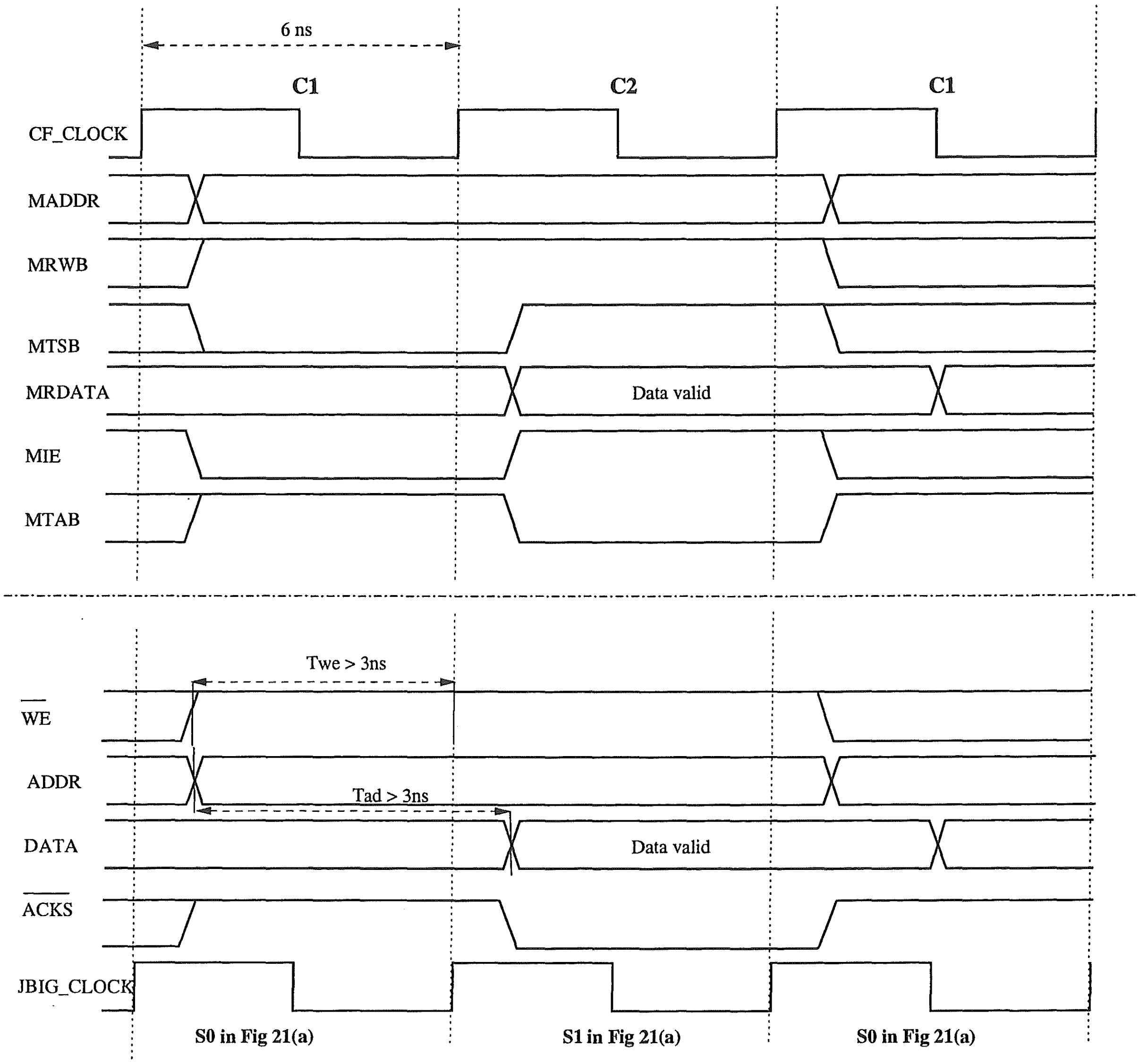


Figure 29: M Bus - JBIG_HW(Slave) Read Timing(case1).

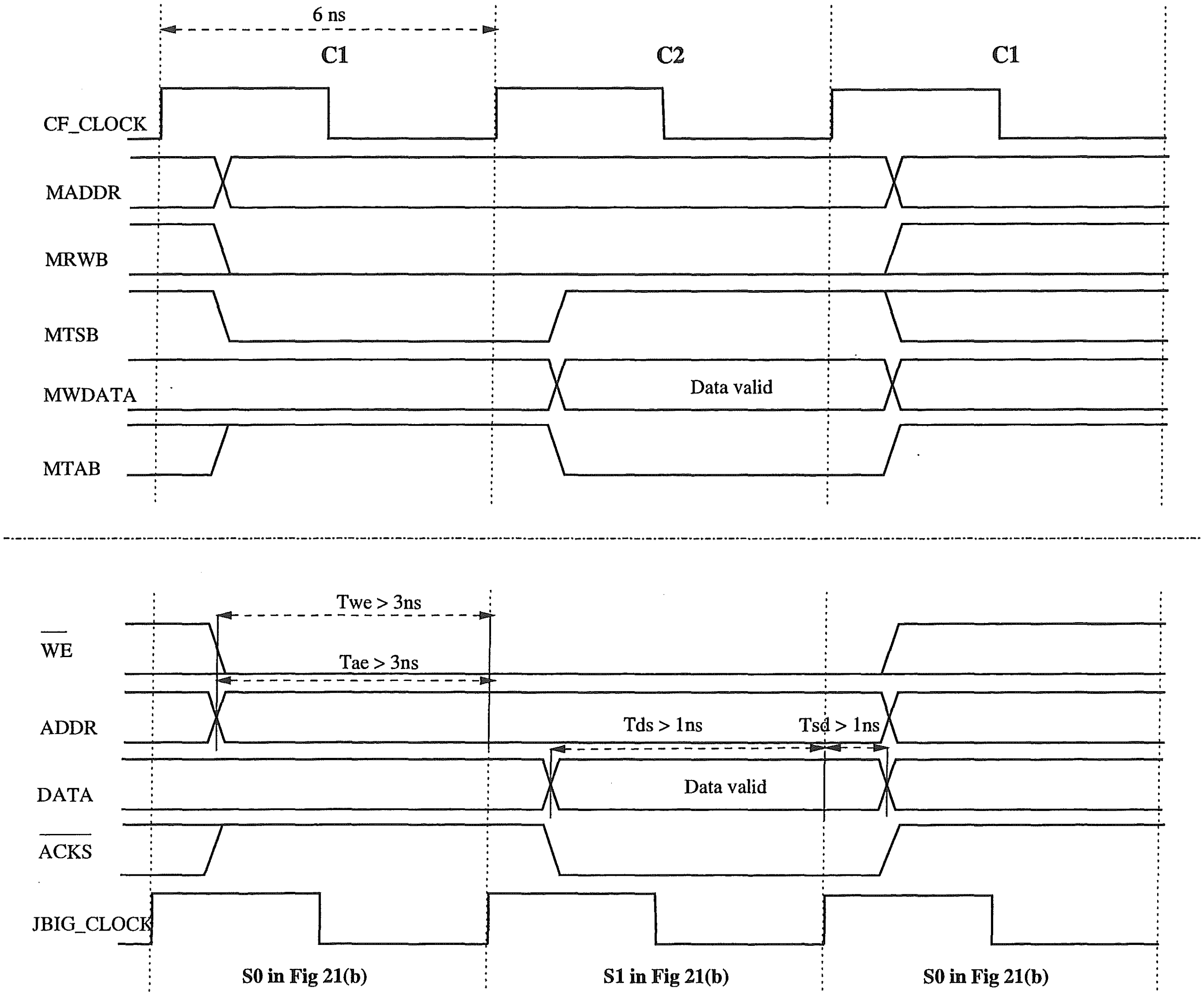


Figure 30: M Bus - JBIG_HW(Slave) Write Timing(case1).

It converts the memory response to the basic 2 cycle protocol of the M bus. In **S0** all signals are inactive and *MTSB* is 1 indicating no bus transaction. When *MTSB* becomes 1, the next state is **S1** or **S2** depending upon *MRWB* for a read or write transaction respectively. In **S1**, $\overline{CS}$ goes low, $\overline{OE}$ goes low for memory read, and *MTAB* goes low to indicate that no wait cycles are needed. $\overline{WE}$ is made high. In **S2** same things happen except that $\overline{OE}$ and $\overline{WE}$ are reversed.

Memory Read Figure 27 shows the timing diagram for a memory read transaction. In the first cycle, MTSB is made low and MRWB goes high indicating a read transaction. In the second clock cycle, $\overline{CS}$ and $\overline{OE}$ go low selecting the memory chip for read. Then, memory puts valid data into the data bus. At this time, MTAB is deasserted. To enable capturing of data, MIE goes high indicating that data are being written into the bus.

Memory Write Figure 28 shows the timing diagram for a memory write transaction. In the first cycle, MTSB is made low and MRWB is set low indicating a write transaction. In the second clock cycle, $\overline{CS}$, $\overline{WE}$ go low selecting the memory chip for write. Valid data are driven onto the bus. MTAB is deasserted.

4.2.1.2 JBIG_HW initialization SW

Figure 25 shows the assembly code FSMD running on ColdFire, to initialize JBIG_HW. After initialization, JBIG_HW computation takes place and then eventually the control again transfers back to ColdFire. This 'transfer of control' is achieved via an interrupt (*INT*) from JBIG_HW to ColdFire. Figure 25 is actually a superstate FSMD, in the sense that each state might take more than one clock cycle. For instance, states **S4** - **S10** take two clock cycles each in accordance with the M bus protocol. All thee registers occurring here are mapped to one of the JBIG_HW registers in Figure 14 such as Reg_IO(1..5) or a register in the Register File.

S1 is the wait state for the interrupt from JBIG_HW. After the interrupt, the SW portion of JBIG executes on ColdFire and then initialization of JBIG_HW takes place in states **S4** - **S8**. This initialization will be used for JBIG_HW memory access. **S4** - **S7** involve a series of writes into address mapped JBIG_HW registers(0x00000100, 0x00001000,...). In **S4**, the start address is written into JBIG_HW register 0x00000100(Reg_addrin). **S5** writes the count value i.e., number of data words to be transferred from the start address(Reg_cont). **S6** writes the address where the results of JBIG_HW computation must be put in memory(Reg_addrin2). **S7** gives the number of data words that need to be skipped after each data word access(Reg_ofs). States **S8** and **S9** set the JBIG_HW flag register, to indicate that JBIG_HW can start its computation. **S10** executes the branch instruction, to the wait state.

4.2.1.3 M Bus - JBIG_HW Interface

In this case, we assume that JBIG_HW is fast enough to read the data from the bus when it is available. Besides we also assume that the data are available on the bus at the end of the first clock cycle after the request is made.

Such an interface can be achieved by simply directly connecting as follows:

$$\begin{aligned} \overline{JRW} &\leftrightarrow MRWB \\ ADDR &\leftrightarrow MADDR \\ DATA &\leftrightarrow MWDATA \\ \overline{READY} &\leftrightarrow MTSB \\ \overline{ACKR} &\leftrightarrow MTAB \end{aligned}$$

The required interface FSMD is thus a one state FSMD with the above connections.

JBIG_HW Slave Read Figure 29 captures the timing diagrams for a JBIG_HW read transaction i.e., using the bus to read the registers of JBIG_HW. This could be used for ColdFire to read the results of JBIG_HW computation directly. The interface has the job of matching the two timing diagrams. For this case, a direct connection of the signals as mentioned above will suffice. These timing diagrams have been directly taken from the specifications discussed in the previous sections(Figures Figure 18 and Figure 21(a).

JBIG_HW Slave Write Figure 30 captures the timing diagrams for a JBIG_HW slave write transaction, i.e., using the bus to write into JBIG_HW registers. This is used for initialization of JBIG_HW. The timing diagrams have been taken from Figures Figure 18 and Figure 21(b).

JBIG_HW Master Read Figure 31 captures the timing diagrams for a JBIG_HW master read transaction. This is used when JBIG_HW wants to read from Memory. Keeping in mind that peripheral components such as memory could be slow, the protocol has provision for an $\overline{ACKR}$ signal which can be used by them to indicate any wait cycles necessary. This is

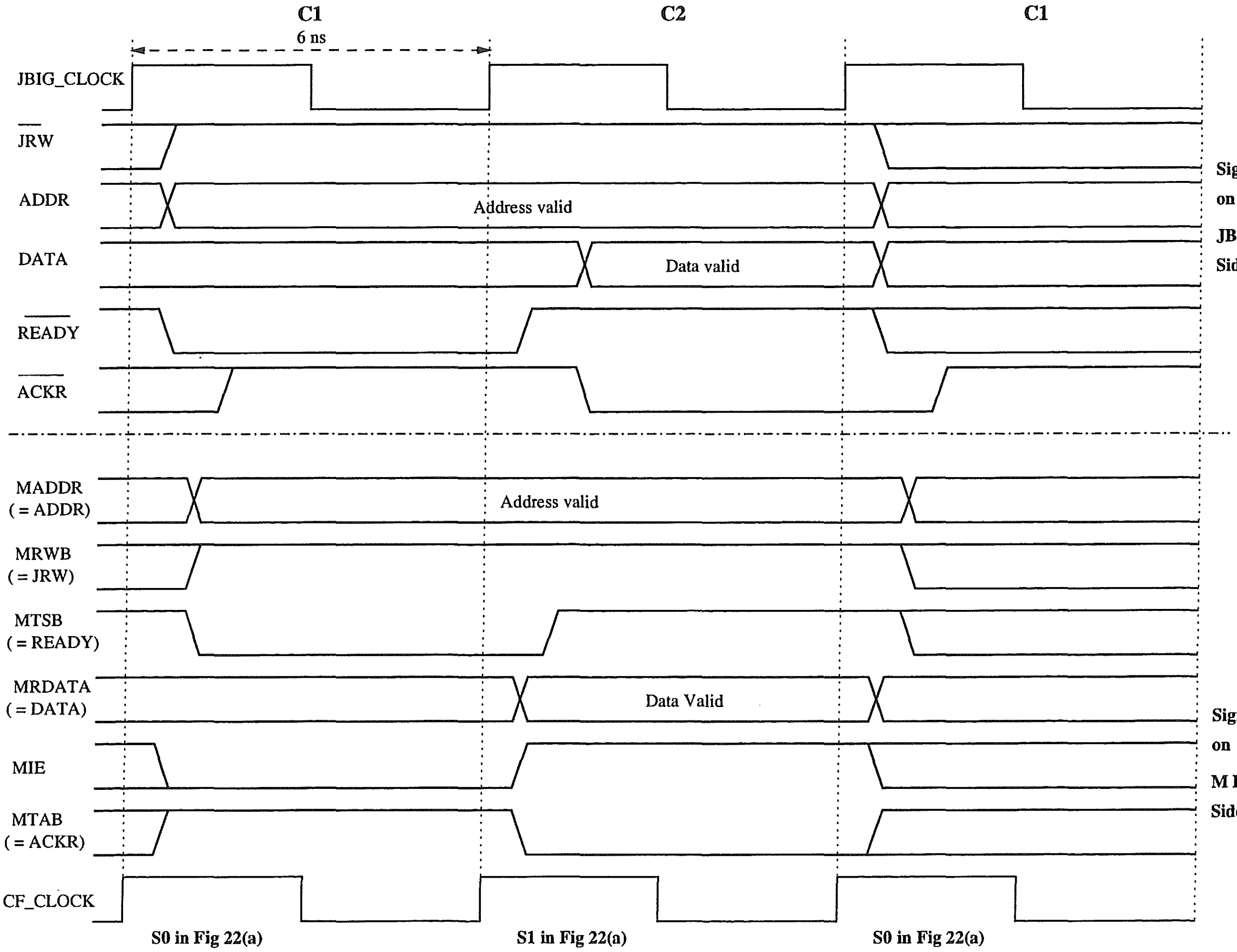


Figure 31: M Bus - JBIG_HW(Master) Read Timing(case1).

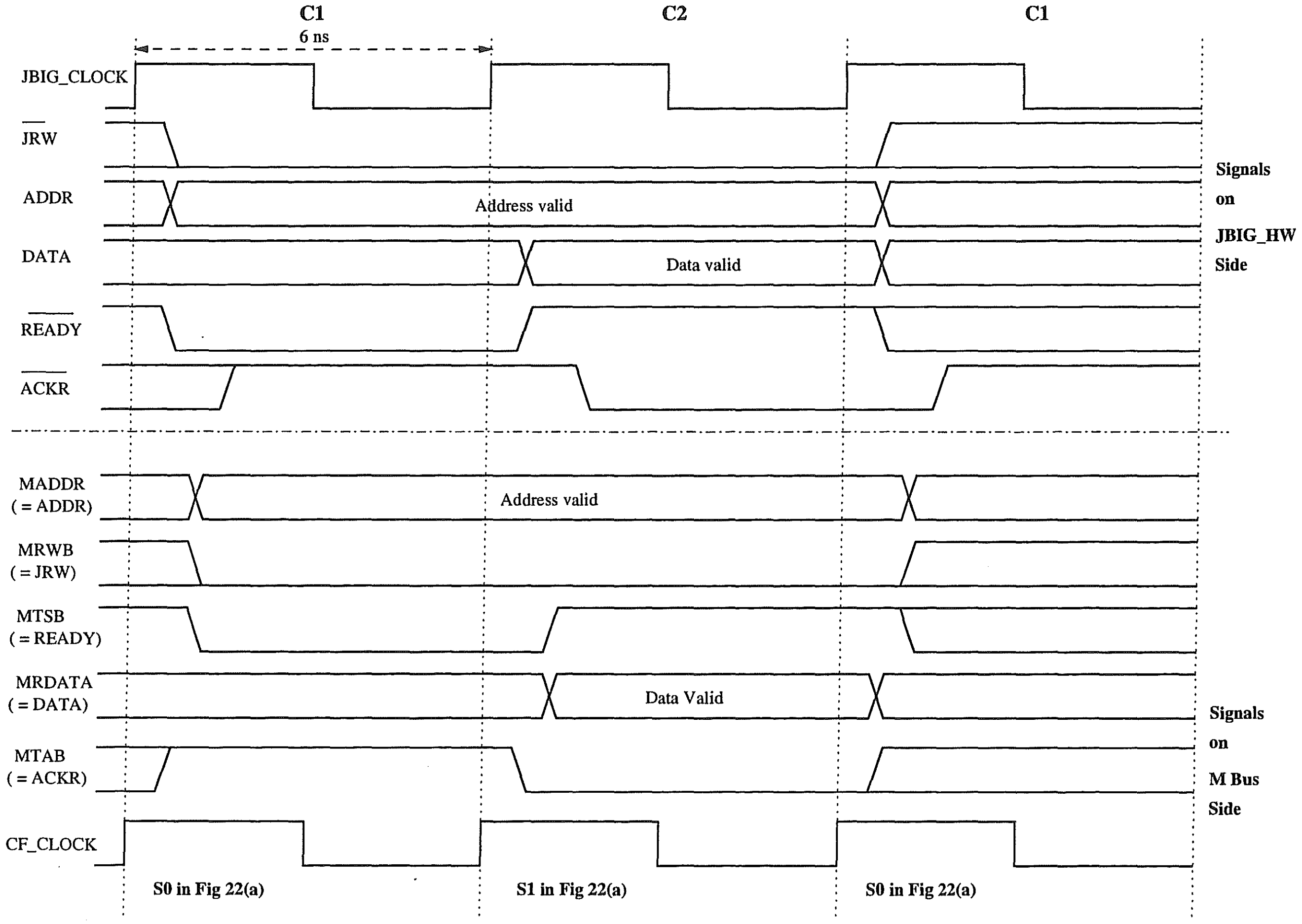


Figure 32: M Bus - JBIG_HW(Master) Write Timing(case1).

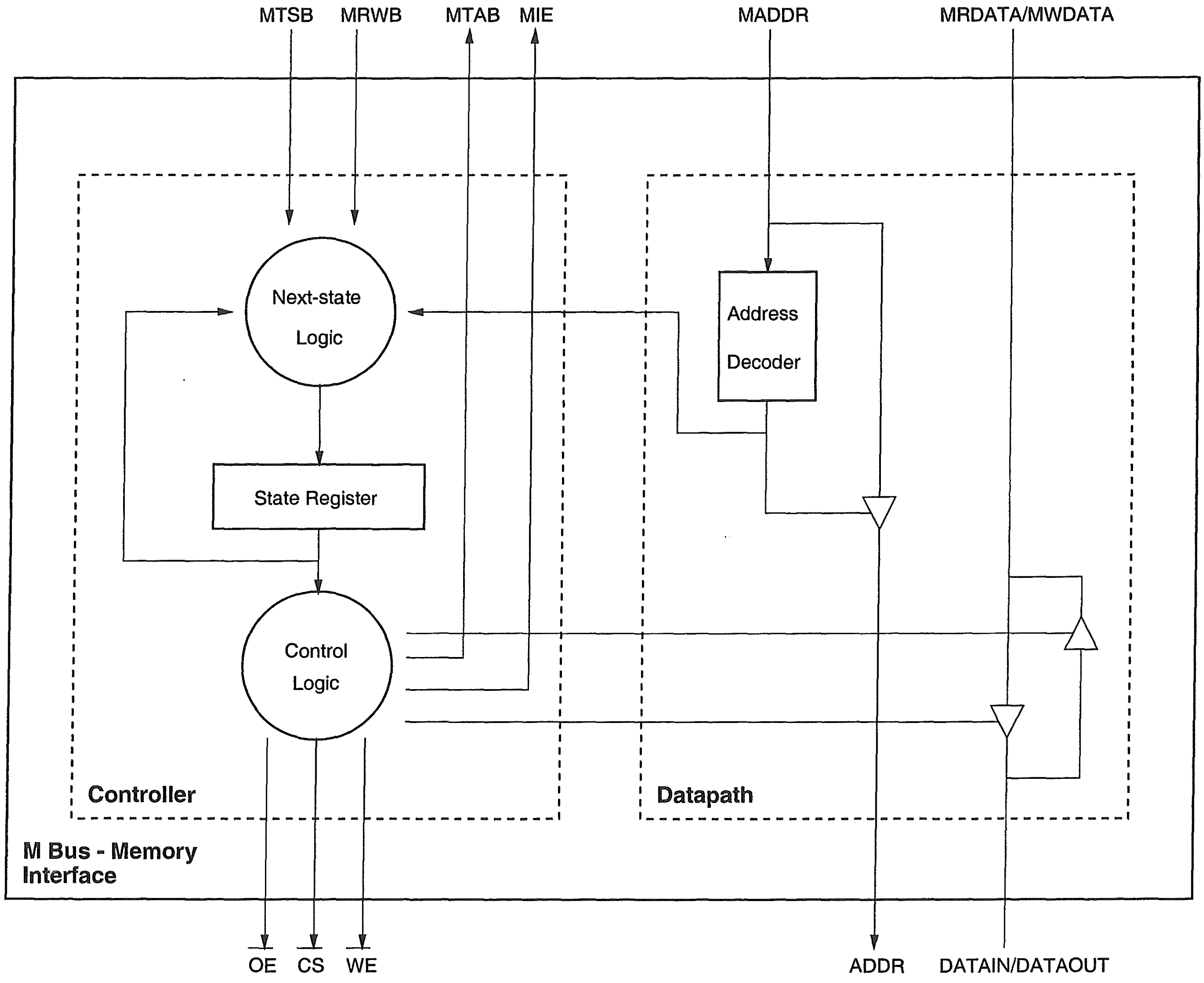


Figure 33: M Bus - Memory Interface(case 2).

similar to the M bus protocol. However for this case, since we assume that the timings match, no such wait cycles are necessary. These diagrams have been taken from Figures Figure 19 and Figure 22(a).

JBIG_HW Master Write Figure 32 captures the timing diagrams for a JBIG_HW master write transaction. This is used when JBIG_HW wants to write into Memory. Again, no wait cycles are needed for a memory write since we assume that memory (or in general any other component) is fast enough. These timing diagrams have been taken from Figures Figure 19 and Figure 22(b). We see that they satisfy memory timing requirements.

4.2.2 Case 2: Interfaces with handshake

4.2.2.1 M Bus - Memory Interface (BMI)

In this case we use the memory specification in Figure 19. We find that the memory is slower than the M bus and BMI needs to take care of this fact. Figure 33 shows how BMI looks like and Figure 34 shows the FSMD of the BMI. We have direct connections as the following:

$$ADDR \leftrightarrow MADDR$$
$$DATA \leftrightarrow MWDATA$$

Initially it is in **S0**, where $\overline{CS}, \overline{OE}, \overline{WE}$, *MTAB* are asserted. It remains in this state as long as no bus transaction is initiated, i.e., *MTSB* is 1. When *MTSB* becomes 0, it goes to **S1** for a read transaction, and **S3** for a write transaction. These are wait states, and hence *MTAB* remains asserted, indicating that read/write is not yet over. During these states, the corresponding actions take place. For both $\overline{CS}$ is deasserted, selecting the memory chip. For a read, $\overline{OE}$ is low and $\overline{WE}$ is high while the reverse is true for a write. Finally, **S2** and **S4** mark the end of read and write respectively, where *MTAB* is deasserted, thereby indicating end of transaction.

Memory Read Figure 35 shows the timing diagram for a memory read transaction. In the first cycle, MTSB is made low and MRWB goes high indicating a read transaction. In the second clock cycle, $\overline{CS}$ and $\overline{OE}$ go low selecting the memory chip for read. Then, memory puts valid data into the data bus. However *MTAB* is not deasserted at this time, since it takes 12 ns, i.e., two clock cycles to ensure valid data. Thus C2 is a wait cycle. In C3, to enable capturing of data, MIE goes high indicating that data are being written into the bus and *MTAB* is made low to indicate end of read transaction. Thus data can be latched at the end of C3.

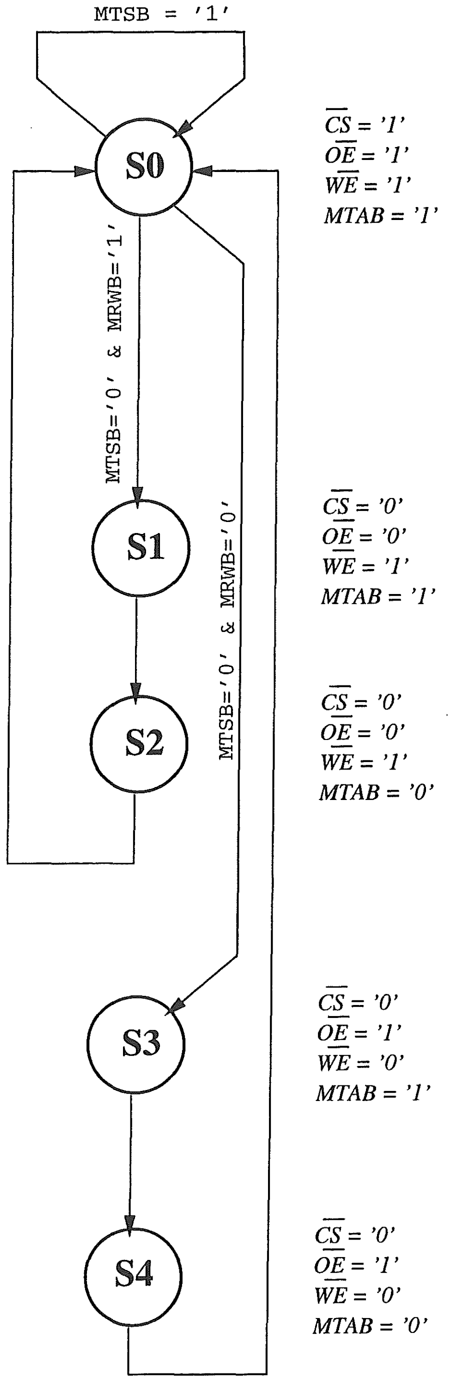


Figure 34: FSMD of M Bus - Memory Interface(case2).

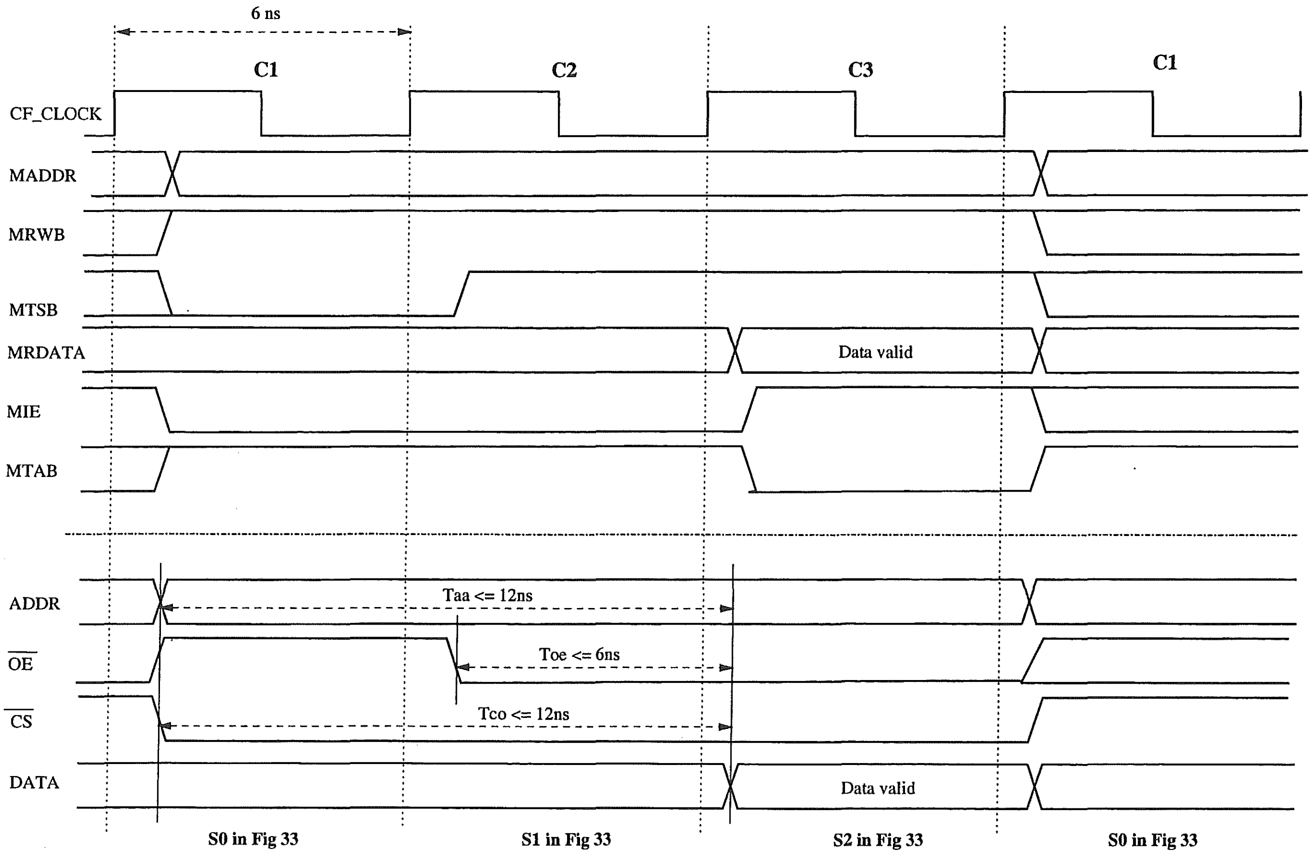


Figure 35: M Bus - Memory Read Timing(case2).

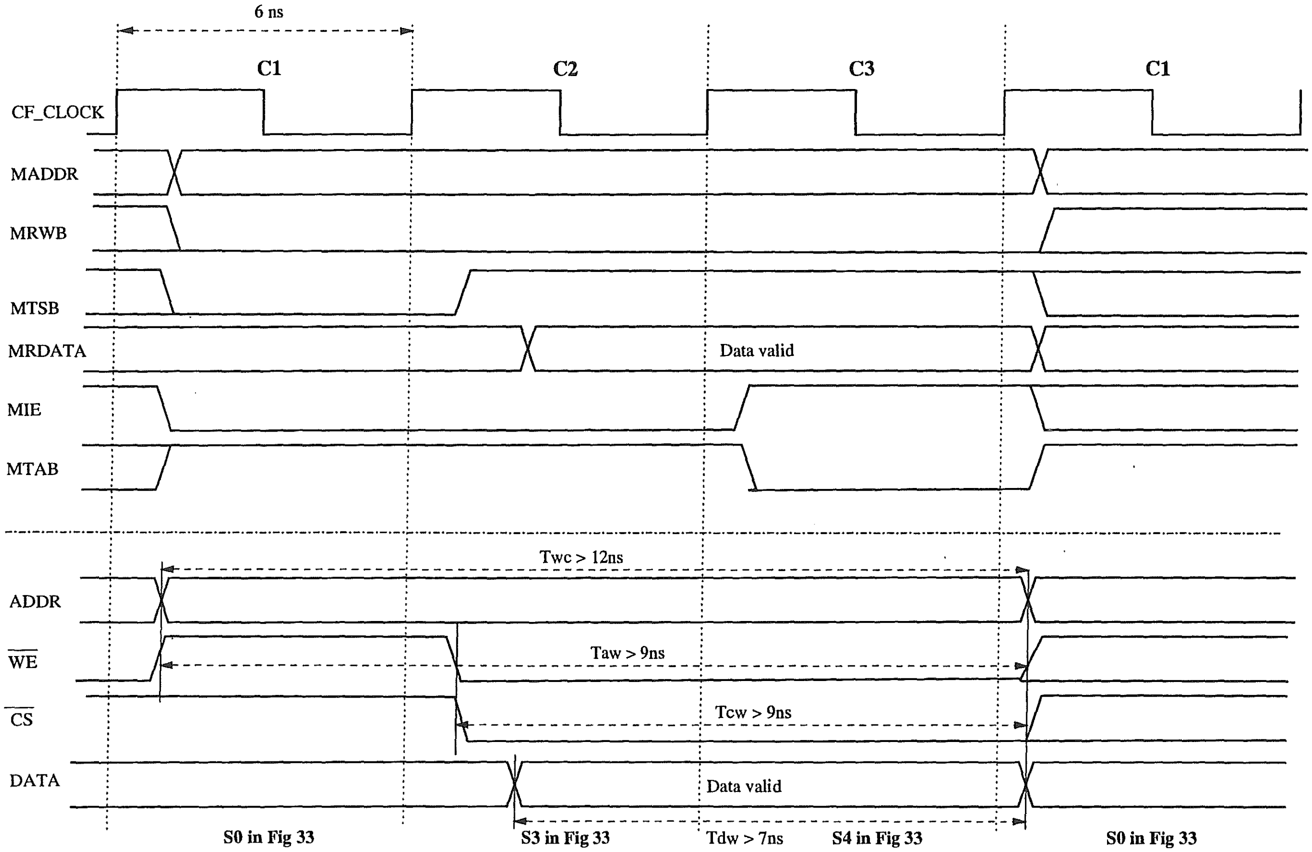


Figure 36: M Bus - Memory Write Timing(case2).

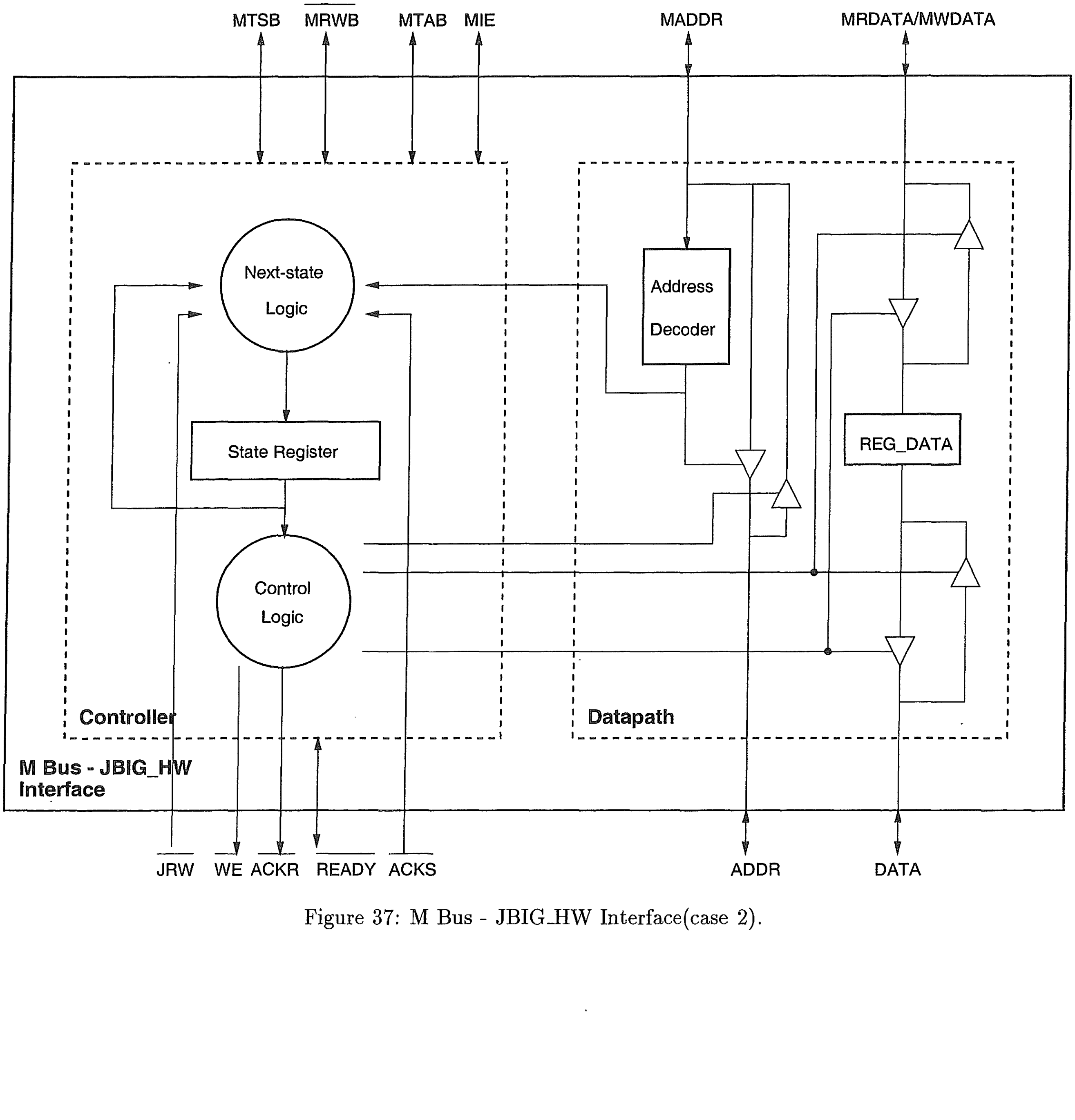


Figure 37: M Bus - JBIG_HW Interface(case 2).

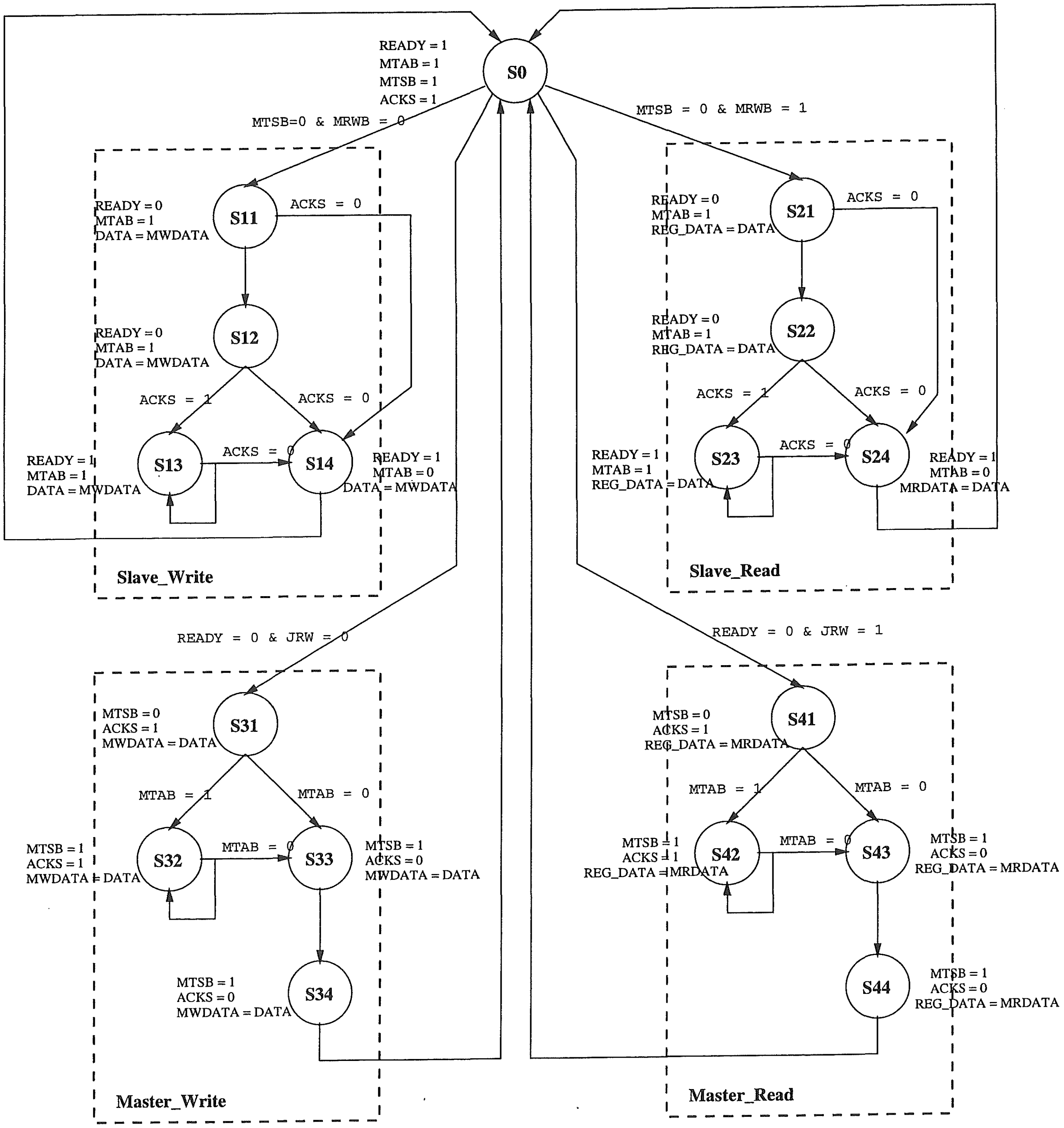


Figure 38: M Bus - JBIG_HW Interface FSMD(case 2).

Memory Write Figure 36 shows the timing diagram for a memory write transaction. In the first cycle, MTSB is made low and MRWB is set high indicating a write transaction. In the second clock cycle, $\overline{CS}$, $\overline{WE}$ go low selecting the memory chip for write. Valid data are driven onto the bus. C2 is also a wait cycle since memory takes at least 7ns for writing. Thus *MTAB* is kept asserted during C2. In C3, *MTAB* is deasserted, indicating that the write transaction is complete.

4.2.2.2 JBIG_HW initialization SW

This is identical to that of case1. Since there is no dependency of SW for its correctness, on the timing of any of the components, the SW need not be changed. When the read/write instructions are executed, the M Bus - JBIG_HW interface takes care of the required handshake. We discuss this interface in the next subsection.

4.2.2.3 M Bus - JBIG_HW Interface

In this case, we assume that JBIG_HW has a 8 ns clock. It is thus slower than the bus. We thus need an interface to convert the JBIG_HW protocol which runs at 8 ns into bus protocol that runs at 6 ns and vice-versa. Figure 37 shows how this interface looks like and Figure 38 shows the FSMDs required to take care of JBIG_HW transactions in the slave and master modes. In **S0**, $\overline{READY}$, *MTAB* are asserted and it waits for initiation of a bus transaction.

Slave_Read part in Figure 38 shows the FSMD that takes care of reading of JBIG_HW registers. The following are direct connections:

$$\begin{gathered} ADDR \leftrightarrow MADDR \\ \overline{WE} \leftrightarrow MRWB \end{gathered}$$

The Slave_Read FSMD starts when *MTSB* goes low and MRWB is '1'. In **S21**, $\overline{READY}$ is deasserted and *DATA* is latched into REG_DATA. Depending upon $\overline{ACKS}$, it goes to **S22** or **S24**. **S22** is a wait state, which is needed in case JBIG_HW misses the deassertion of $\overline{READY}$. It can miss this by at most one clock cycle of the bus. In the next state (**S23** or **S24**), $\overline{READY}$ is asserted. **S23** is the wait state for $\overline{ACKS}$ to go low. *DATA* is latched in both **S22** and **S23**. **S24** is the final state where the latched data are driven into the bus *MRDATA* and *MTAB* is deasserted. The transition from **S1** to **S24** is in case JBIG_HW senses $\overline{READY}$ the the same clock cycle it is deasserted.

Slave_Write part in Figure 38 shows the FSMD that takes care of writing JBIG_HW registers. The following are direct connections:

$$\begin{gathered} ADDR \leftrightarrow MADDR \\ DATA \leftrightarrow MWDATA \\ \overline{WE} \leftrightarrow MRWB \end{gathered}$$

The Slave_Write FSMD starts when *MTSB* goes low and MRWB is '0'. In **S11**, $\overline{READY}$ is deasserted. Depending upon $\overline{ACKS}$, it goes to **S12** or **S14**. **S12** is a wait state, which is needed in case JBIG_HW misses the deassertion of $\overline{READY}$. It can miss this by at most one clock cycle of the bus. In the next state (**S13** or **S14**), $\overline{READY}$ is asserted. **S13** is the wait state for $\overline{ACKS}$ to go low. **S14** is the final state where *MTAB* is deasserted, indicating the end of write transaction. A low $\overline{ACKS}$ is an indication that JBIG_HW has latched the data into its register. The transition from **S11** to **S41** is in case JBIG_HW senses $\overline{READY}$ the the same clock cycle it is deasserted.

Master_Read part of Figure 38 shows the FSMD that takes care of bus read for JBIG_HW. The following are direct connections:

$$\begin{gathered} ADDR \leftrightarrow MADDR \\ \overline{JRW} \leftrightarrow MRWB \end{gathered}$$

The Master_Read FSMD starts when *READY* goes low and JRW is '1'. In **S41**, *MTSB* is deasserted. Depending upon *MTAB*, it then goes to **S42** or **S43**. **S42** is a wait state, for *MTAB* to go low. In this state the bus data *MRDATA* are latched into REG_DATA. Then it goes to **S43** where *MTSB* is asserted, $\overline{ACKR}$ is deasserted and the latched data are driven into *DATA*. Deassertion of $\overline{ACKR}$ indicates that valid data will be available on *DATA* and that JBIG_HW can latch the data into its input register. **S44** is the final state which is a wait state, so that JBIG_HW can move the data from the input register to any other register.

Master_Write of Figure 38 shows the FSMD that takes care of bus write for JBIG_HW. The following are direct connections:

$$\begin{gathered} ADDR \leftrightarrow MADDR \\ DATA \leftrightarrow MWDATA \\ \overline{JRW} \leftrightarrow MRWB \end{gathered}$$

The Master_Write FSMD starts when *READY* goes low and JRW is '0'. In **S0**, $\overline{ACKR}$, *MTSB* are

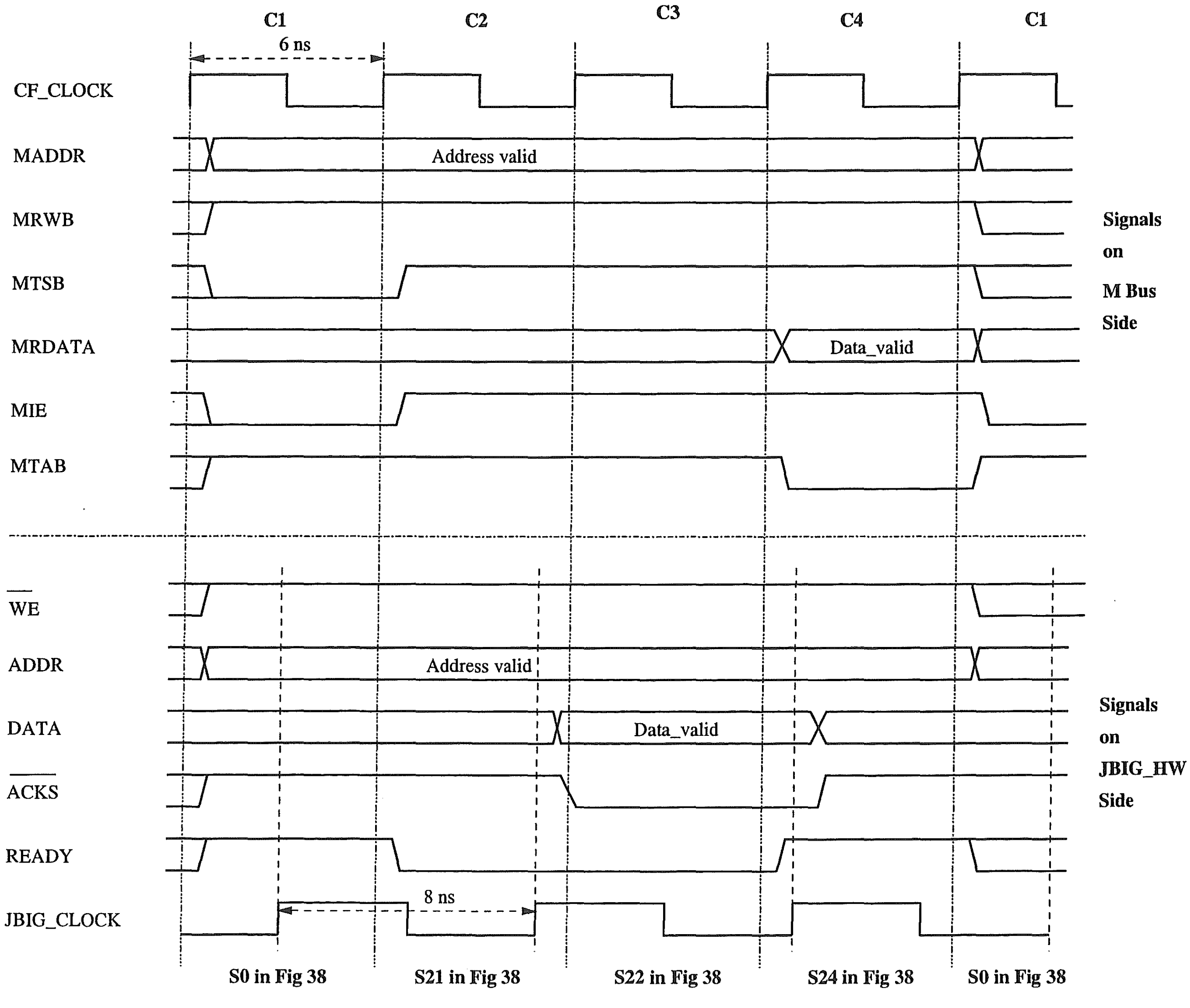


Figure 39: M Bus - JBIG_HW(Slave) Read Timing(case2).

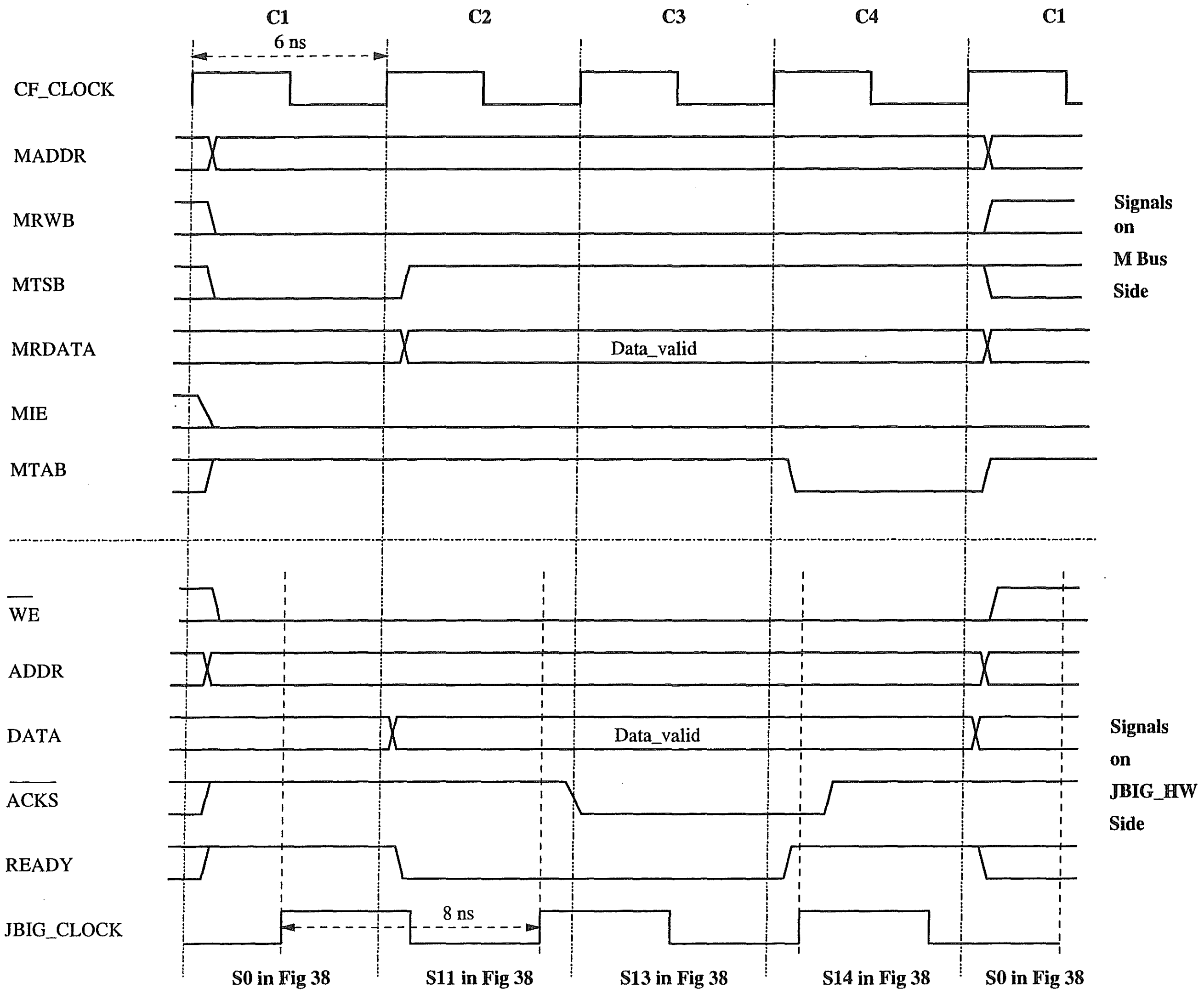


Figure 40: M Bus - JBIG_HW(Slave) Write Timing(case2).

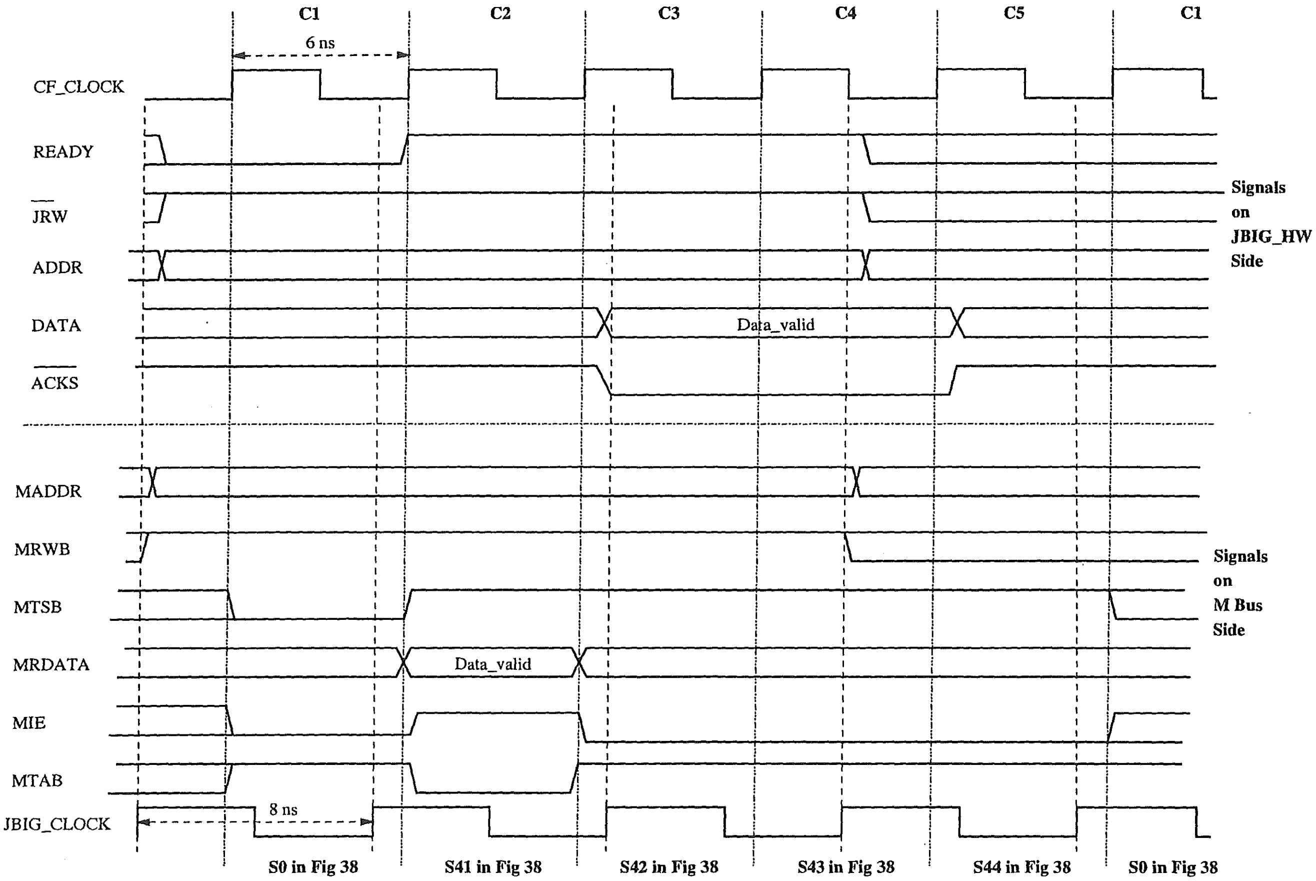


Figure 41: M Bus - JBIG_HW(Master) Read Timing(case2).

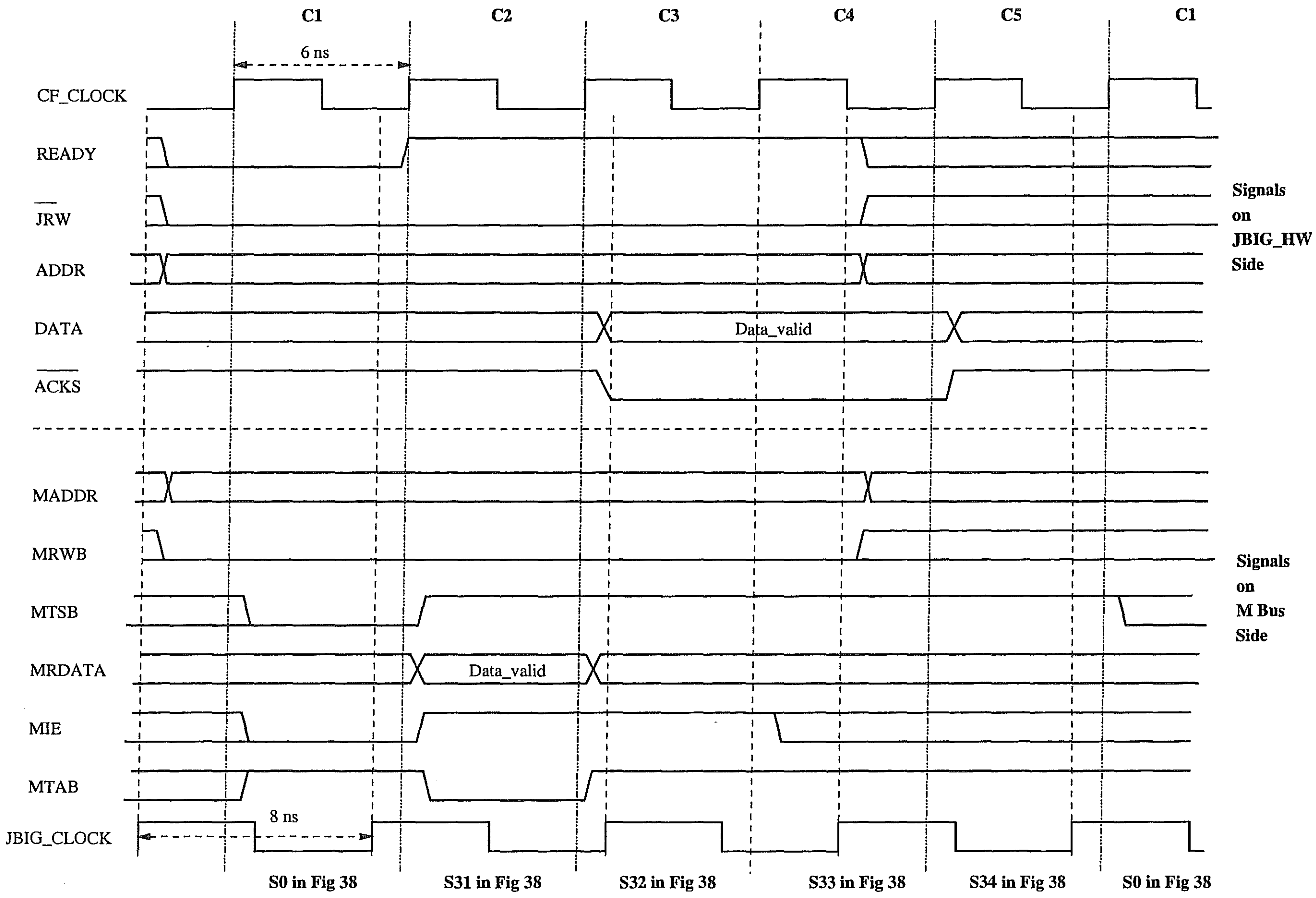


Figure 42: M Bus - JBIG_HW(Master) Write Timing(case2).

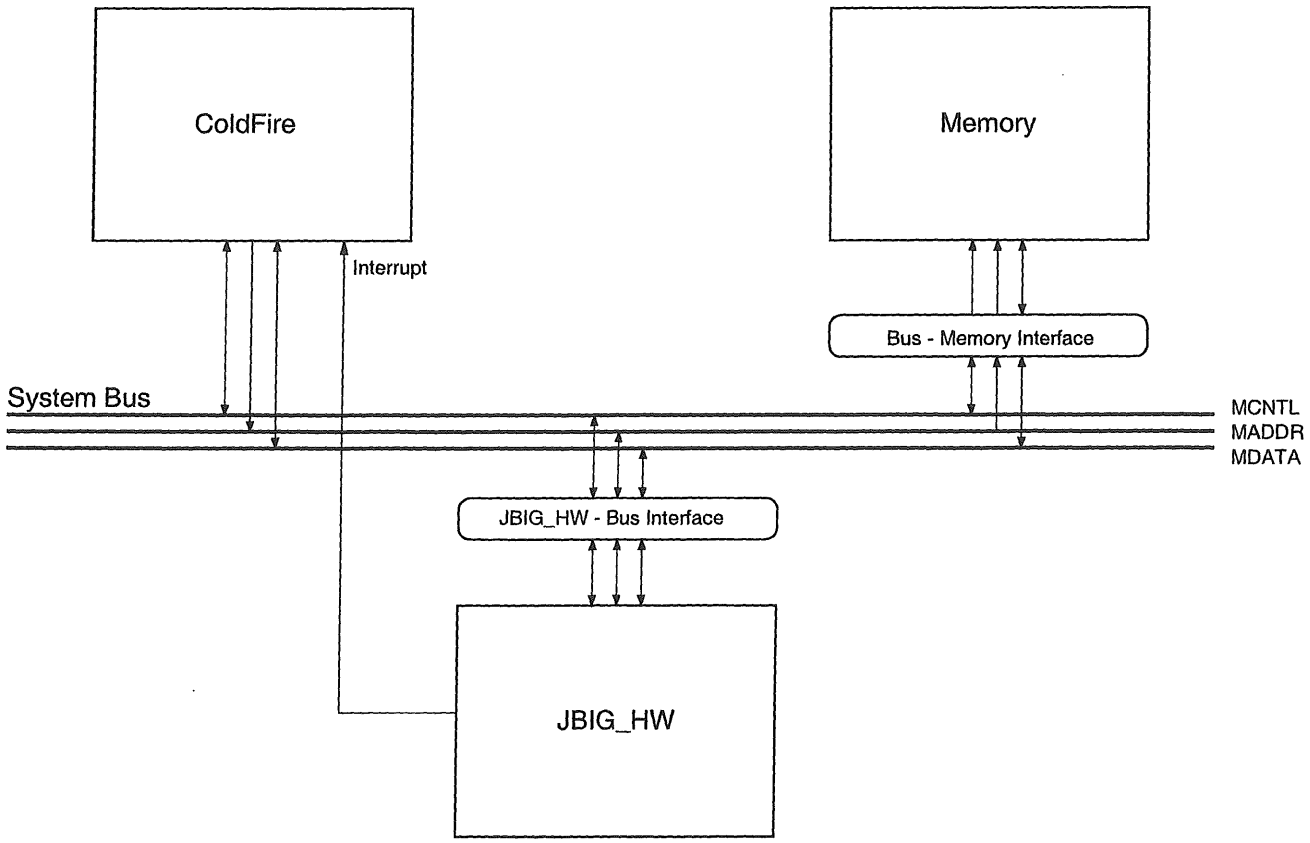


Figure 43: Communication Model.

asserted and it waits for $\overline{READY}$ to go low. This indicates that JBIG_HW is initiating a transaction. In **S31**, *MTSB* is deasserted. Depending upon *MTAB*, it then goes to **S32** or **S33**. **S32** is a wait state, for *MTAB* to go low. *MTAB* going low indicates that data have been read from the bus. Then it goes to **S33** where *MTSB* is asserted, $\overline{ACKR}$ is deasserted. Deassertion of $\overline{ACKR}$ indicates the end of write transaction. This indicated to JBIG_HW that the data it had put into the bus have been latched.

JBIG_HW Slave Read Figure 39 captures the timing diagrams for a JBIG_HW slave read transaction i.e., using the bus to read the registers of JBIG_HW. The top portion shows the signals of the bus and the remaining are the signals of JBIG_HW. The timing for these signals are derived from the Figures Figure 18, Figure 21(a) and Figure 38. We can see how the valid data are available on *MRDATA* with respect to *DATA*.

JBIG_HW Slave Write Figure 40 captures the timing diagrams for a JBIG_HW slave write transaction. The timing diagrams have been derived from Figures Figure 18, Figure 21(b) and Figure 38.

JBIG_HW Master Read Figure 41 captures the timing diagrams for a JBIG_HW master read transaction. These diagrams have been derived from Figures Figure 18, Figure 22(a) and Figure 38.

JBIG_HW Master Write Figure 42 captures the timing diagrams for a JBIG_HW master write transaction. These timing diagrams have been derived from Figures Figure 19, Figure 22(b) and Figure 38.

4.3 Communication Model

Figure 13 of the architecture model can be refined to indicate the interfaces as shown in Figure 43. We find that ColdFire, memory and JBIG_HW are interconnected via M Bus, which is composed of Data lines that include signals in *MDATA*, Address lines that

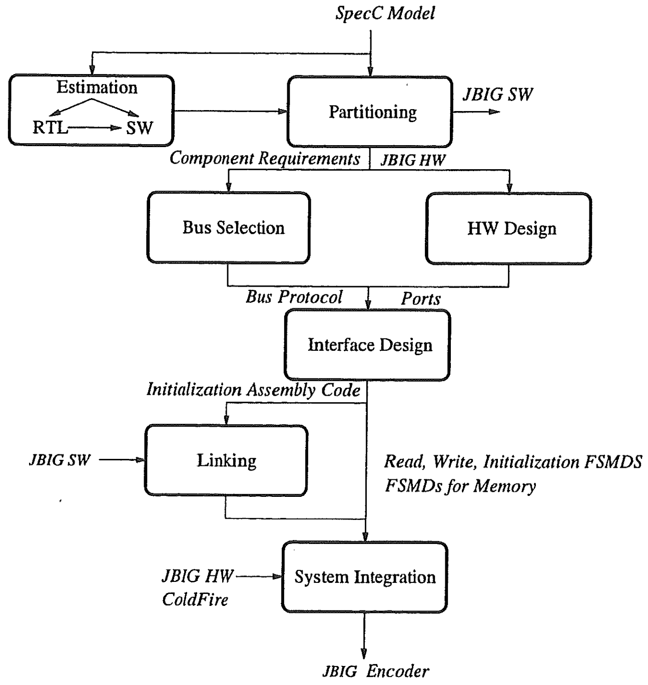


Figure 44: Activities.

include signals in *MADDR* and Control lines, that include all other signals. These signals are the same as those that appeared in the M Bus in Figures Figure 23, Figure 24. *INT* is used to interrupt ColdFire to indicate that JBIG_HW computation is complete and that ColdFire can reinitialize the JBIG_HW.

5 Conclusions

In this report, we presented the SpecC system-level design methodology by applying it to design a JBIG encoder. Figure 44 shows the activities that were performed. We produced the behavioral SpecC model, which was used for estimation and partitioning. For estimation, we found out the total number of clock cycles required for execution, for both HW and SW. For HW, we estimated using clock accurate FSMD (RTL). For SW, we estimated in two ways. One was using C code and the other was using assembly instructions corresponding to the RTL. These estimation results were used for partitioning and the results of partitioning were a set of HW modules (in our case, the JBIG_HW) and a set of SW modules (JBIG_SW). Then we analyzed the components for a suitable bus protocol. This was followed by bus selection and the design of the HW modules. Then, interface design was taken up, which produced SW interface in terms of assembly code, and HW interface in terms of FSMDs for memory and JBIG_HW. This assembly code will have to be linked with the rest of the SW, and finally integrated with JBIG_HW, HW interfaces and ColdFire. This system integration produces the desired JBIG encoder.

References

[1] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, March, 2000.

[2] J. Zhu, R. Dömer, D. Gajski, *Syntax and Semantics of the SpecC+ Language*, University of California, Irvine, Technical Report ICS-TR-97-16, April 1997.

[3] R. Dömer, J. Zhu, D. Gajski, *The SpecC Language Reference Manual*, University of California, Irvine, Technical Report ICS-TR-98-13, March 1998.

[4] A. Gerstlauer, S. Zhao, D. Gajski, A. Horak, *Design of a GSM Vocoder using SpecC Methodology*, University of California, Irvine, Technical Report ICS-TR-99-11, February 1999.

[5] H. Lehr, D. Gajski, *Modeling Custon Hardware in VHDL*, University of California, Irvine, Technical Report ICS-TR-99-29, July 1999.

[6] N. Fan, V. Chaiyakul, D. Gajski, "Usage-Based Characterization of Complex Functional Blocks for Reuse in Behavioral Synthesis," *Proceedings of Asia-Pacific DAC, 2000,*

[7] International Telecommunication Union(ITU-T Recommendation T.82), *Information Technology - Coded Representation of Picture and Audio Information - Prograssive Bi-Level Image Compression*, March 1993.

[8] Motorola, Inc., Semiconductor Products Sector, DSP Division, *DSP56600 16-bit Digital Signal Processor Family Manual*, DSP56600FM/AD, 1996.

[9] Samsung Semiconductor Inc., North America, http://dev.usa.samsungsemi.com/products/summary/speedsram/KM68257C.htm. *SRAM Products - KM68257C*, February 1998.

[10] Motorola, Inc., Semiconductors Products Sector, DSP Division, *ColdFire 2/2M Integrated Microprocessor User's Manual*, 1998.

A SpecC Specification Model for JBIG Encoder

A.1 constant.sh

```
#define NULL 0

#define stderr ((FILE *) 1)
#define stdin ((FILE *) 0)
#define stdout ((FILE *) 2)
#define EOF 0

#define JBG_HITOLO      0x08
#define JBG_SEQ         0x04
#define JBG_ILEAVE      0x02
#define JBG_SMID        0x01

#define JBG_LRLTWO      0x40
#define JBG_VLENGTH     0x20
#define JBG_TPDON       0x10
#define JBG_TPBON       0x08
#define JBG_DPON        0x04
#define JBG_DPPRIV      0x02
#define JBG_DPLAST      0x01

#define JBG_DELAY_AT    0x100   /* delay ATMOVE until the first line of the next
                                 * stripe. Option available for compatibility
                                 * with conformance test example in clause 7.2.*/



/*
 * Possible error code return values
 */

#define JBG_EOK         0
#define JBG_EOK_INTR    1
#define JBG_EAGAIN      2
#define JBG_ENOMEM      3
#define JBG_EABORT      4
#define JBG_EMARKER     5
#define JBG_ENOCONT     6
#define JBG_EINVAL      7
#define JBG_EIMPL       8

/* cut from jbig.sh*/
#define JBG_VERSION     "1.0"
#define MX_MAX  23      /* maximal supported mx offset for
                         * adaptive template in the encoder */
#define JBG_BUFSIZE 4000
#define JBG_ATMOVES_MAX  64
#define JBG_EN          0       /* English */
#define JBG_DE_8859_1   1       /* German in ISO Latin 1 character set */
#define JBG_DE_UTF_8    2       /* German in Unicode UTF-8 encoding */
#define jbg_dec_getplanes(s)     ((s)->planes)

/* cut from jbig.sc */
#define TPB2CX  0x195   /* contexts for TP special pixels */
#define TPB3CX  0x0e5
#define TPDCX   0xc3f

/* marker codes */
#define MARKER_STUFF     0x00
#define MARKER_RESERVE   0x01
#define MARKER_SDNORM    0x02
#define MARKER_SDRST     0x03
```

```
#define MARKER_ABORT     0x04
#define MARKER_NEWLEN    0x05
#define MARKER_ATMOVE    0x06
#define MARKER_COMMENT   0x07
#define MARKER_ESC       0xff

/* loop array indices */
#define STRIPE  0
#define LAYER   1
#define PLANE   2

/* special jbg_buf pointers (instead of NULL) */
#define SDE_DONE ((struct jbg_buf *) -1)
#define SDE_TODO ((struct jbg_buf *) 0)
```

A.2 jbig_tab.sh

```
/*
 *  Probability estimation tables for the arithmetic encoder/decoder
 *  given by ITU T.82 Table 24.
 *
 *  £Id: jbig_tab.c,v 1.6 1998-04-05 18:36:19+01 mgk25 Rel £
 */

short jbg_lsz[113] = {
  0x5a1d, 0x2586, 0x1114, 0x080b, 0x03d8, 0x01da, 0x00e5, 0x006f,
  0x0036, 0x001a, 0x000d, 0x0006, 0x0003, 0x0001, 0x5a7f, 0x3f25,
  0x2cf2, 0x207c, 0x17b9, 0x1182, 0x0cef, 0x09a1, 0x072f, 0x055c,
  0x0406, 0x0303, 0x0240, 0x01b1, 0x0144, 0x00f5, 0x00b7, 0x008a,
  0x0068, 0x004e, 0x003b, 0x002c, 0x5ae1, 0x484c, 0x3a0d, 0x2ef1,
  0x261f, 0x1f33, 0x19a8, 0x1518, 0x1177, 0x0e74, 0x0bfb, 0x09f8,
  0x0861, 0x0706, 0x05cd, 0x04de, 0x040f, 0x0363, 0x02d4, 0x025c,
  0x01f8, 0x01a4, 0x0160, 0x0125, 0x00f6, 0x00cb, 0x00ab, 0x008f,
  0x5b12, 0x4d04, 0x412c, 0x37d8, 0x2fe8, 0x293c, 0x2379, 0x1edf,
  0x1aa9, 0x174e, 0x1424, 0x119c, 0x0f6b, 0x0d51, 0x0bb6, 0x0a40,
  0x5832, 0x4d1c, 0x438e, 0x3bdd, 0x34ee, 0x2eae, 0x299a, 0x2516,
  0x5570, 0x4ca9, 0x44d9, 0x3e22, 0x3824, 0x32b4, 0x2e17, 0x56a8,
  0x4f46, 0x47e5, 0x41cf, 0x3c3d, 0x375e, 0x5231, 0x4c0f, 0x4639,
  0x415e, 0x5627, 0x50e7, 0x4b85, 0x5597, 0x504f, 0x5a10, 0x5522,
  0x59eb
};

unsigned char jbg_nmps[113] = {
    1,   2,   3,   4,   5,   6,   7,   8,
    9,  10,  11,  12,  13,  13,  15,  16,
   17,  18,  19,  20,  21,  22,  23,  24,
   25,  26,  27,  28,  29,  30,  31,  32,
   33,  34,  35,   9,  37,  38,  39,  40,
   41,  42,  43,  44,  45,  46,  47,  48,
   49,  50,  51,  52,  53,  54,  55,  56,
   57,  58,  59,  60,  61,  62,  63,  32,
   65,  66,  67,  68,  69,  70,  71,  72,
   73,  74,  75,  76,  77,  78,  79,  48,
   81,  82,  83,  84,  85,  86,  87,  71,
   89,  90,  91,  92,  93,  94,  86,  96,
   97,  98,  99, 100,  93, 102, 103, 104,
   99, 106, 107, 103, 109, 107, 111, 109,
  111
};

/*
 * least significant 7 bits (mask 0x7f) of jbg_nlps[] contain NLPS value,
 * most significant bit (mask 0x80) contains SWTCH bit
 */
```

```
unsigned char jbg_nlps[113] = {
  129,  14,  16,  18,  20,  23,  25,  28,
   30,  33,  35,   9,  10,  12, 143,  36,
   38,  39,  40,  42,  43,  45,  46,  48,
   49,  51,  52,  54,  56,  57,  59,  60,
   62,  63,  32,  33, 165,  64,  65,  67,
   68,  69,  70,  72,  73,  74,  75,  77,
   78,  79,  48,  50,  50,  51,  52,  53,
   54,  55,  56,  57,  58,  59,  61,  61,
  193,  80,  81,  82,  83,  84,  86,  87,
   87,  72,  72,  74,  74,  75,  77,  77,
  208,  88,  89,  90,  91,  92,  93,  86,
  216,  95,  96,  97,  99,  99,  93, 223,
  101, 102, 103, 104,  99, 105, 106, 107,
  103, 233, 108, 109, 110, 111, 238, 112,
  240
};

/*
 * Resolution reduction table given by ITU-T T.82 Table 17
 */

char jbg_resred[4096] = {
  0,0,0,1,0,0,0,1,0,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,
  0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,
  0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
  1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,0,0,0,0,1,0,1,0,0,1,1,1,0,1,1,
  0,0,0,1,0,0,0,1,0,0,1,0,0,0,1,1,0,1,1,1,0,0,0,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,0,1,1,1,0,1,1,0,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,1,0,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,
  1,0,0,1,0,0,1,1,0,1,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,1,1,1,1,1,1,1,1,
  0,0,1,1,0,0,0,1,0,0,0,1,0,0,1,1,0,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,1,1,0,1,1,1,1,1,1,0,1,1,1,0,
  0,0,0,0,0,0,0,1,0,0,1,0,0,0,0,1,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,1,0,0,1,0,0,1,1,0,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,
  0,0,0,1,0,0,0,1,0,1,1,0,1,0,1,1,1,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,
  1,1,1,0,1,0,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,
  1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,1,1,1,1,1,1,
  0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,1,1,0,1,0,1,1,0,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,1,1,0,0,0,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,0,1,0,0,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,0,0,1,1,1,1,1,1,0,1,1,1,1,0,1,1,
  1,0,0,1,0,0,1,0,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,
  0,0,1,0,1,1,1,1,0,0,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,0,
0,0,0,0,1,0,0,1,0,0,1,1,0,1,1,1,0,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,1,1,0,1,0,0,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,1,1,0,1,0,1,0,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,
0,0,1,0,0,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,0,1,1,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,1,
0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,1,0,0,1,0,0,1,1,
0,0,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,0,1,0,0,0,0,1,0,1,0,1,0,1,0,1,
0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,1,0,0,0,0,0,0,0,1,1,1,1,0,1,1,1,
0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,1,0,1,0,1,1,0,0,0,1,0,0,1,1,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,1,1,0,0,1,1,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,1,
0,0,1,0,0,1,1,1,0,0,0,0,1,0,0,1,0,0,0,1,1,1,1,0,1,0,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,0,1,0,1,1,0,
0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,0,0,0,1,0,1,1,1,0,1,1,1,
0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,1,1,0,1,0,0,0,1,1,0,1,0,0,0,0,1,1,1,1,0,0,1,1,1,0,1,1,0,0,1,1,
0,0,0,0,0,0,0,0,1,1,0,1,0,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
0,0,1,1,0,0,1,1,0,0,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,1,0,0,0,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,1,1,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,
0,0,0,1,0,0,1,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,0,1,0,0,1,0,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,
0,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,1,0,0,0,1,0,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,
0,0,1,1,1,1,1,1,0,0,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,1,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,1,1,0,1,1,
0,0,0,1,0,0,0,1,0,0,1,0,0,0,1,1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,0,1,0,1,0,1,1,0,1,1,1,0,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,1,1,0,0,0,0,0,1,0,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,
0,0,0,0,1,0,0,1,0,0,1,1,0,0,1,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,1,1,0,1,1,
1,0,1,0,1,0,0,1,1,0,1,1,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,1,0,1,1,0,1,1,1,
0,0,1,0,0,0,0,1,0,0,0,0,0,0,1,1,0,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,0,1,1,1,1,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,1,1,0,0,0,0,0,1,0,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,1,0,1,0,1,0,1,1,0,1,0,1,0,0,0,1,1,1,1,1,1,1,1,1,
1,1,1,0,1,0,0,0,1,1,0,1,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,
1,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,1,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,1,1,0,1,1,
0,0,1,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0,0,1,1,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,0,1,0,1,0,0,1,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,1,0,1,1,0,1,1,0,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,1,1,1,0,0,0,0,0,0,1,1,1,1,0,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,1,0,1,0,0,0,1,1,1,0,1,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1,0,1,1,
1,0,0,0,1,0,0,0,0,1,1,1,0,1,0,1,1,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,1,1,1,0,1,1,0,
0,0,1,1,1,1,1,1,0,0,0,0,1,0,0,1,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,1,1,0,1,1,1,1,1,1,1,
0,0,0,0,1,0,0,0,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,0,0,1,1,1,1,1,1,1,
0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,1,1,1,0,1,1,1,1,1,1,1,1,1,1,1,
  0,0,1,0,1,0,1,1,0,0,1,0,1,1,1,1,0,1,1,1,1,1,1,1,0,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,1,1,0,0,0,1,0,1,1,1,1,1,1,1,0,1,1,1,0,1,1,1,
  0,0,1,0,1,0,1,1,0,1,1,1,1,1,1,1,0,0,1,1,1,0,1,1,0,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,0,0,0,0,1,1,0,0,0,0,0,0,0,1,0,0,0,0,1,0,0,1,
  0,0,0,1,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,0,0,0,0,0,0,1,
  0,0,0,0,0,0,0,1,0,1,1,1,0,1,0,1,0,0,1,0,0,0,0,1,0,1,0,1,0,1,0,1,
  0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,1,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,
  0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,1,0,1,0,0,1,0,0,1,0,0,0,0,0,0,0,1,
  0,0,0,0,1,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,1,0,0,1,1,
  0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,1,0,0,0,0,0,0,0,0,0,1,0,1,0,0,1,1,
  1,0,0,0,0,0,0,0,0,1,0,0,0,0,0,1,0,0,0,1,0,0,1,1,0,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,1,1,0,0,0,0,1,1,0,0,0,0,0,0,0,
  0,0,1,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,0,0,1,0,0,1,1,
  0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,
  0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,
  0,0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,0,0,1,1,
  0,0,0,0,0,0,0,1,0,0,0,0,0,0,0,1,0,1,0,1,0,0,0,1,0,1,1,1,1,1,1,1,
  0,0,0,0,0,0,0,0,0,1,0,1,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,0,0,1,1,
  0,0,0,0,0,0,0,1,0,1,0,1,0,1,0,0,0,0,1,1,0,0,0,1,0,1,1,1,0,1,1,1
};

/*
 * Deterministic prediction tables given by ITU–T T.82 tables
 * 19 to 22. The table below is organized differently, the
 * index bits are permutated for higher efficiency.
 */

char jbg_dptable[256 + 512 + 2048 + 4096] = {
  /* phase 0: offset=0 */
  0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
  0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,
  0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,2,2,2,2,2,0,2,2,2,2,2,2,2,2,
  0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,2,2,2,2,2,0,2,0,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
  /* phase 1: offset=256 */
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,
  0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,2,2,2,2,2,0,2,0,2,2,2,2,2,
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,
  0,2,2,2,2,1,2,1,2,2,2,2,1,1,1,1,2,0,2,0,2,2,2,2,0,2,0,2,2,2,2,2,
  0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,2,2,2,2,0,2,2,2,2,2,2,2,
  0,2,0,2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,0,0,2,2,2,2,2,0,0,2,2,2,2,2,
  0,2,2,2,2,1,2,1,2,2,2,2,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,
  1,2,1,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,1,1,2,2,2,2,2,0,2,2,2,2,2,2,
  2,2,2,2,2,0,2,0,2,2,2,2,0,0,0,0,0,2,0,2,2,2,2,2,0,2,2,2,2,2,2,2,
  0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,2,0,2,0,2,2,2,2,2,
  2,2,2,2,2,1,1,1,2,2,2,2,1,1,1,1,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,1,
  2,2,2,2,2,2,2,2,2,2,2,2,2,0,1,2,0,2,0,2,2,2,2,2,0,2,0,2,2,2,2,1,
  0,2,0,2,2,1,2,1,2,2,2,2,1,1,1,1,0,0,0,0,2,2,2,2,0,2,0,2,2,2,2,1,
  2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,0,0,0,2,2,2,2,2,
  2,2,2,2,2,1,2,1,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,1,2,1,2,2,2,2,1,
  2,2,2,2,2,2,2,2,0,2,0,2,2,1,2,2,2,2,2,2,2,2,2,2,0,0,0,2,2,2,2,2,
  /* phase 2: offset=768 */
  2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,2,2,2,2,1,1,1,1,
  0,2,2,2,2,1,2,1,2,2,2,2,1,2,1,2,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,
  2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,1,2,1,2,2,2,2,2,1,1,1,
  2,0,2,2,2,1,2,1,0,2,2,2,1,2,1,2,2,2,2,0,2,2,2,2,0,2,0,2,2,2,2,2,
  0,2,0,0,1,1,1,1,2,2,2,2,1,1,1,1,0,2,0,2,1,1,1,1,2,2,2,2,1,1,1,1,
  2,2,0,2,2,2,1,2,2,2,2,2,1,2,1,2,2,2,0,2,2,1,2,1,0,2,0,2,1,1,1,1,
  2,0,0,2,2,2,2,2,0,2,0,2,2,0,2,0,2,0,2,0,2,2,2,1,2,2,0,2,1,1,2,1,
  2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,1,
  2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,2,2,2,2,1,1,1,1,
  0,0,0,0,2,2,2,2,0,0,0,0,2,2,2,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,
  2,2,0,2,2,2,2,1,0,2,2,2,1,1,1,1,2,0,2,2,2,2,2,2,0,2,0,2,2,1,2,1,
2,0,2,0,2,2,2,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,
0,2,2,2,1,2,1,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,
2,2,0,2,2,2,2,2,2,2,2,2,2,2,0,2,2,0,0,2,2,1,2,1,0,2,2,2,1,1,1,1,
2,2,2,0,2,2,2,2,2,2,0,2,2,0,2,0,2,1,2,2,2,2,2,2,1,2,1,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,1,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,1,1,1,2,2,2,2,1,1,1,1,
2,2,2,1,2,2,2,2,2,2,1,2,0,0,0,0,2,2,0,2,2,1,2,2,2,2,2,2,1,1,1,1,
2,0,0,0,2,2,2,2,0,2,2,2,2,2,2,0,2,2,2,0,2,2,2,2,2,0,0,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,0,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,1,
0,2,0,2,2,1,1,2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,
2,0,2,0,2,1,2,1,0,2,0,2,2,2,1,2,2,0,2,0,2,2,2,2,0,2,0,2,2,2,1,2,
2,2,2,0,2,2,2,2,2,2,0,2,2,2,2,2,2,2,1,2,2,2,2,2,2,0,1,2,2,2,2,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
0,2,2,2,1,2,1,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,
2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,2,2,1,2,1,0,2,2,2,1,1,1,1,
2,0,2,0,2,1,2,2,0,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,1,2,2,
2,0,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,0,2,0,2,2,2,2,0,0,0,0,2,1,2,1,
2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,0,0,2,2,2,1,2,2,2,
0,0,2,0,2,2,2,2,0,2,0,2,2,0,2,0,1,1,1,2,2,2,2,2,2,2,2,2,2,1,1,1,
2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,1,
2,2,0,0,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,1,1,2,2,2,2,1,1,1,1,
0,2,2,2,1,2,1,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,
2,0,0,2,2,2,2,2,0,2,0,2,2,2,2,2,1,0,1,2,2,2,2,1,0,2,2,2,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,0,2,2,0,2,0,2,1,2,2,2,2,2,2,2,2,0,2,2,1,2,2,
0,2,0,0,1,1,1,1,0,2,2,2,1,1,1,1,2,2,2,2,2,2,2,2,2,0,2,2,1,2,1,1,
2,2,0,2,2,1,2,2,2,2,2,2,1,2,2,2,2,0,2,2,2,2,2,2,0,2,0,2,1,2,1,1,
2,0,2,0,2,2,2,2,0,2,0,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,2,1,
2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
0,2,2,2,2,0,2,0,2,2,2,2,0,0,0,0,2,2,2,2,2,1,1,2,2,2,2,2,1,2,2,2,
2,0,2,2,2,1,2,1,0,2,2,2,2,2,1,2,2,0,2,0,2,2,2,2,0,2,0,2,2,1,2,2,
0,2,0,0,2,2,2,2,1,2,2,2,2,2,2,0,2,1,2,2,2,2,2,2,1,2,2,2,2,2,2,2,
0,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,1,0,2,2,
0,0,0,2,2,1,1,1,2,2,2,2,1,2,2,2,2,0,2,0,2,2,2,1,2,2,2,2,1,2,1,2,
0,0,0,0,2,2,2,2,2,2,0,2,2,1,2,2,2,1,2,1,2,2,2,2,1,2,1,2,0,2,2,2,
2,0,2,0,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
0,2,2,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,1,2,2,2,2,2,0,2,2,1,2,2,0,0,0,2,2,2,2,2,1,2,2,0,2,2,2,1,2,1,2,
2,0,2,0,2,2,2,2,0,2,0,2,2,1,2,2,0,2,0,0,2,2,2,2,2,2,2,2,2,1,2,2,
2,2,2,2,2,2,2,2,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,0,2,0,2,2,2,1,
1,2,0,2,2,1,2,1,2,2,2,2,1,2,2,2,2,0,2,0,2,2,2,2,2,0,2,2,1,1,1,1,
0,2,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,2,1,2,1,
2,2,0,0,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,1,
2,2,2,0,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,2,
2,0,2,0,2,2,2,2,2,1,1,2,2,2,2,2,2,2,2,2,2,2,2,1,0,2,0,2,2,2,1,2,
2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
2,0,2,0,2,2,2,2,2,0,2,0,2,2,2,2,2,0,2,0,2,2,2,2,0,0,0,0,2,1,2,1,
2,2,2,2,2,1,2,1,0,2,0,2,2,2,2,2,2,0,2,0,2,2,2,2,0,2,0,2,2,2,2,1,
2,0,2,0,2,2,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,0,
2,0,2,0,2,2,2,1,2,2,2,0,2,2,2,1,2,0,2,0,2,2,2,2,0,0,0,2,2,2,2,1,
2,0,2,0,2,2,2,2,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,2,
/* phase 3: offset=2816 */
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,1,2,1,2,0,2,0,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,0,2,2,2,1,2,0,2,2,2,1,2,2,2,2,0,2,0,2,1,2,1,0,0,0,0,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,0,2,2,2,1,2,
2,2,2,1,2,2,2,0,1,1,1,1,0,0,0,0,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,0,0,0,0,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,0,0,0,0,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,2,
2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,2,0,2,0,2,1,2,1,
2,0,0,0,2,1,1,1,0,0,0,0,1,1,1,1,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,
2,0,2,2,2,1,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,0,0,2,0,1,1,2,1,
2,2,2,0,2,2,2,1,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,
0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,2,0,2,0,2,1,2,1,0,0,0,0,1,1,1,1,
2,0,0,2,2,1,1,2,2,2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,2,1,1,1,2,0,0,0,
2,1,2,1,2,0,2,0,1,2,1,2,0,2,0,2,2,2,2,0,2,2,2,1,2,0,2,0,2,1,2,1,
2,0,2,0,2,1,2,1,0,0,0,0,1,1,1,1,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,
2,2,2,2,2,2,2,2,2,0,0,0,2,1,1,1,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,
2,0,0,0,2,1,1,1,0,0,0,0,1,1,1,1,2,0,2,0,2,1,2,1,0,0,2,0,1,1,2,1,
2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,0,0,0,2,1,1,1,
2,2,2,1,2,2,2,0,2,1,1,1,2,0,0,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,1,2,1,2,0,2,0,2,
2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,2,1,1,1,0,0,0,0,1,1,1,1,
2,0,2,2,2,1,2,2,0,0,2,0,1,1,2,1,2,1,2,1,2,0,2,0,2,2,2,2,2,2,2,2,
2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,0,0,0,0,1,1,1,1,
2,0,0,0,2,1,1,1,0,0,0,0,1,1,1,1,2,2,2,2,2,2,2,2,2,1,0,2,2,0,1,2,
2,2,2,1,2,2,2,0,2,1,1,1,2,0,0,0,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,
2,1,2,1,2,0,2,0,1,2,1,1,0,2,0,0,0,0,2,1,1,1,2,0,0,0,0,0,1,1,1,1,
2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,0,2,1,2,1,2,0,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,0,2,2,2,1,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,0,2,2,2,1,2,2,2,0,0,2,2,1,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,
2,0,2,0,2,1,2,1,0,0,0,0,1,1,1,1,2,2,2,2,2,2,2,2,0,0,0,0,1,1,1,1,
2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,0,0,0,2,1,1,1,
2,2,2,0,2,2,2,1,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
2,0,2,2,2,1,2,2,2,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
2,1,2,1,2,0,2,0,1,2,1,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,1,2,1,2,0,2,0,1,2,1,1,0,2,0,0,2,0,2,2,2,1,2,2,0,2,1,2,1,2,0,2,
2,2,2,1,2,2,2,0,2,2,1,2,2,2,0,2,2,1,2,2,2,0,2,2,2,2,0,2,2,2,1,2,
0,0,2,0,1,1,2,1,0,0,1,0,1,1,0,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,0,2,2,2,1,1,2,2,2,0,2,2,2,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,
2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,0,0,2,2,1,1,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,
0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,
2,0,0,0,2,1,1,1,0,0,0,0,1,1,1,1,2,2,2,1,2,2,2,0,2,1,2,1,2,0,2,0,
2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,0,2,0,0,1,2,1,1,2,0,0,0,2,1,1,1,
2,2,2,2,2,2,2,2,2,1,1,1,2,0,0,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
2,2,2,0,2,2,2,1,2,2,0,2,2,2,1,2,2,1,2,1,2,0,2,0,2,0,2,2,2,1,2,2,
2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,1,1,1,2,0,0,0,
2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,0,0,1,2,1,1,
2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,0,2,2,2,1,2,2,2,
2,1,2,1,2,0,2,0,2,1,2,2,2,0,2,2,2,2,2,0,2,2,2,1,2,0,2,0,2,1,2,1,
2,0,2,0,2,1,2,1,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,0,1,0,0,1,0,1,1,
    2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,2,2,1,2,2,2,0,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,1,2,2,1,0,2,0,2,2,2,1,2,2,2,
    2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,2,2,0,2,2,2,1,2,2,0,2,2,2,1,2,
    2,0,2,0,2,1,2,1,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
    0,2,0,0,1,2,1,1,2,0,0,0,2,1,1,1,2,2,2,2,2,2,2,2,1,0,1,2,0,1,0,2,
    2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,1,2,2,2,0,2,2,1,1,2,2,0,0,2,2,
    0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,1,2,1,2,0,2,0,2,1,2,2,2,0,2,2,2,0,2,2,2,1,2,2,0,2,2,2,1,2,2,2,
    0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,1,2,2,2,0,2,2,2,
    2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,2,
    0,0,0,0,1,1,1,1,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,2,2,0,2,2,2,1,2,
    2,0,2,0,2,1,2,1,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,
    0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,
    2,2,2,1,2,2,2,0,1,1,2,1,0,0,2,0,2,0,2,2,2,1,2,2,0,2,2,2,1,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,2,2,0,2,2,2,1,2,
    2,0,2,0,2,1,2,1,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,2,1,2,2,2,0,2,2,
    0,2,0,0,1,2,1,1,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,0,0,0,2,1,1,1,2,
    2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
    2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,0,0,2,1,1,1,2,0,0,2,2,2,1,2,2,2,
    2,1,2,1,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,0,2,0,0,1,2,1,1,
    0,0,2,2,1,1,2,2,0,2,1,2,1,2,0,2,2,1,2,1,2,0,2,0,1,2,1,2,0,2,0,2,
    2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,2,2,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,
    2,2,0,0,2,2,1,1,2,2,0,0,2,2,1,1,2,2,2,2,2,2,2,2,2,2,0,0,2,2,1,1,
    2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,0,2,0,0,1,2,1,1,
    2,2,2,0,2,2,2,1,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,1,1,1,2,0,0,0,2,
    2,2,2,2,2,2,2,2,1,1,1,2,0,0,0,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,
    2,0,2,0,2,1,2,1,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,0,0,0,2,1,1,1,
    2,0,2,2,2,1,2,2,0,2,2,2,1,2,2,2,2,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,
    2,0,2,0,2,1,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,0,2,0,2,1,2,1,2,1,2,0,2,0,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,0,2,0,2,1,2,1,1,2,1,2,0,2,0,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,1,2,1,2,0,2,0,2,2,1,2,1,2,0,2,0,2,2,2,2,2,2,2,2,
    2,0,2,1,2,1,2,0,0,2,1,2,1,2,0,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
    2,0,2,0,2,1,2,1,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,0,2,0,2,1,2,1,
    2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,1,2,1,2,0,2,0,1,1,1,2,0,0,0,2,2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,
    2,0,2,0,2,1,2,1,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
    2,2,2,2,2,2,2,2,2,2,2,0,2,2,2,1,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,2,
};
```

A.3 jbig_head.sh

```
#! / bin/csh -f
```

A.4 jbig.sh

```
/*
 *   Header file for the portable free JBIG compression library
 *
 *   Markus Kuhn -- mkuhn@acm.org
 *
 *  £Id: jbig.h,v 1.8 1998-04-11 02:26:33+01 mgk25 Rel £
 */

#ifndef JBG_H
#define JBG_H

#include <stddef.h>
#include "constant.sh"

/*
 * JBIG-KIT version number
 */


/*
 * Buffer block for SDEs which are temporarily stored by encoder
 */






struct jbg_buf {
  unsigned char d[JBG_BUFSIZE];              /* one block of a buffer list */
  int len;                             /* length of the data in this block */
  struct jbg_buf *next;                           /* pointer to next block */
  struct jbg_buf *previous;                   /* pointer to previous block *
                                               * (unused in freelist)      */
  struct jbg_buf *last;     /* only used in list head: final block of list */
  struct jbg_buf **free_list;   /* pointer to pointer to head of free list */
};

/*
 * Maximum number of allowed ATMOVEs per stripe
 */


/*
 * Option and order flags
 */

/*
 * Language code for error message strings (based on ISO 639 2-letter
 * standard language name abbreviations).
 */


/*
 * Status description of an arithmetic encoder
 */

typedef struct jbg_arenc_state {
  unsigned char st[4096];    /* probability status for contexts, MSB = MPS */
  unsigned long c;                /* C register, base of coding intervall, *
                                   * layout as in Table 23                 */
  unsigned long a;       /* A register, normalized size of coding intervall */
  long sc;           /* counter for buffered 0xff values which might overflow */
  int ct;  /* bit shift counter, determines when next byte will be written */
  int buffer;                /* buffer for most recent output byte != 0xff */
```

```
    void (*byte_out)(int, void *); /* function which receives all PSCD bytes */
    void *file;                                  /* parameter passed to byte_out */
  }jbg_arenc_state;


  /*
   * Status description of an arithmetic decoder
   */

  /* copy the enum from the inside of struct to outside of struct */
  typedef enum{
      JBG_OK,                           /* symbol has been successfully decoded */
      JBG_READY,                /* no more bytes of this PSCD required, marker *
                                 * encountered, probably more symbols available */
      JBG_MORE,             /* more PSCD data bytes required to decode a symbol */
      JBG_MARKER    /* more PSCD data bytes required, ignored final 0xff byte */
  } JBG_RESULT;                                  /* result of previous decode call */

  typedef struct jbg_ardec_state {
    unsigned char st[4096];     /* probability status for contexts, MSB = MPS */
    unsigned long c;                      /* C register, base of coding intervall, *
                                          * layout as in Table 25               */
    unsigned long a;       /* A register, normalized size of coding intervall */
    int ct;        /* bit shift counter, determines when next byte will be read */
    unsigned char *pscd_ptr;                  /* pointer to next PSCD data byte */
    unsigned char *pscd_end;                     /* pointer to byte after PSCD */
    JBG_RESULT result;

    int startup;                                /* controls initial fill of s->c */
  }jbg_ardec_state;



  /* self-defined struct */
  typedef struct local_data {
    unsigned char *hp, *lp1, *lp2, *p0, *p1, *q1, *q2;
    unsigned long hl, ll, hx, hy, lx, ly, hbpl, lbpl;
    unsigned long line_h0, line_h1;
    unsigned long line_h2, line_h3, line_l1, line_l2, line_l3;
    struct jbg_arenc_state *se;
    unsigned long i, j, y;
    unsigned t;
    int ltp, ltp_old, cx;
    unsigned long c_all, c[MX_MAX + 1], cmin, cmax, clmin, clmax;
    int tmax, at_determined;
    int new_tx;
    long new_tx_line;
    struct jbg_buf *new_jbg_buf;
  }local_data;




  /*
   * Status of a JBIG encoder
   */

  typedef struct jbg_enc_state {
    int d;                               /* resolution layer of the input image */
    unsigned long xd, yd;     /* size of the input image (resolution layer d) */
    int planes;                             /* number of different bitmap planes */
    int dl;                          /* lowest resolution layer in the next BIE */
    int dh;                         /* highest resolution layer in the next BIE */
    unsigned long l0;                   /* number of lines per stripe at lowest *
                                        * resolution layer 0                    */
  unsigned long stripes;      /* number of stripes required  (determ. by l0) */
  unsigned char **lhp[2];      /* pointers to lower/higher resolution images */
  int *highres;                   /* index [plane] of highres image in lhp[] */
  int order;                                      /* SDE ordering parameters */
  int options;                                        /* encoding parameters */
  unsigned mx, my;                             /* maximum ATMOVE window size */
  int *tx;       /* array [plane] with x-offset of adaptive template pixel */
  char *dppriv;         /* optional private deterministic prediction table */
  char *res_tab;           /* table for the resolution reduction algorithm */
  struct jbg_buf ****sde;       /* array [stripe][layer][plane] pointers to *
                                 * buffers for stored SDEs                 */
  struct jbg_arenc_state *s;  /* array [planes] for arithm. encoder status */
  struct jbg_buf *free_list; /* list of currently unused SDE block buffers */
  void (*data_out)(unsigned char *start, size_t len, void *file);
                                                      /* data write callback */
  void *file;                             /* parameter passed to data_out() */
  char *tp;     /* buffer for temp. values used by diff. typical prediction */
} jbg_enc_state;


/*
 * Status of a JBIG decoder
 */

typedef struct jbg_dec_state {
  /* data from BIH */
  int d;                              /* resolution layer of the full image */
  int dl;                            /* first resolution layer in this BIE */
  unsigned long xd, yd;     /* size of the full image (resolution layer d) */
  int planes;                          /* number of different bitmap planes */
  unsigned long l0;                  /* number of lines per stripe at lowest *
                                      * resolution layer 0                   */
  unsigned long stripes;    /* number of stripes required  (determ. by l0) */
  int order;                                      /* SDE ordering parameters */
  int options;                                        /* encoding parameters */
  int mx, my;                                  /* maximum ATMOVE window size */
  char *dppriv;         /* optional private deterministic prediction table */

  /* loop variables */
  unsigned long ii[3];  /* current stripe, layer, plane (outer loop first) */

  /*
   * Pointers to array [planes] of lower/higher resolution images.
   * lhp[d & 1] contains image of layer d.
   */
  unsigned char **lhp[2];

  /* status information */
  int **tx, **ty;   /* array [plane][layer-dl] with x,y-offset of AT pixel */
  struct jbg_ardec_state **s;    /* array [plane][layer-dl] for arithmetic *
                                  * decoder status */
  int **reset;       /* array [plane][layer-dl] remembers if previous stripe *
                      * in that plane/resolution ended with SDRST.           */
  unsigned long bie_len;                        /* number of bytes read so far */
  unsigned char buffer[20]; /* used to store BIH or marker segments fragm. */
  int buf_len;                                   /* number of bytes in buffer */
  unsigned long comment_skip;        /* remaining bytes of a COMMENT segment */
  unsigned long x;              /* x position of next pixel in current SDE */
  unsigned long i; /* line in current SDE (first line of each stripe is 0) */
  int at_moves;                 /* number of AT moves in the current stripe */
  unsigned long at_line[JBG_ATMOVES_MAX];           /* lines at which an    *
                                                     * AT move will happen */
  int at_tx[JBG_ATMOVES_MAX], at_ty[JBG_ATMOVES_MAX]; /* ATMOVE offsets in *
                                                       * current stripe    */
  unsigned long line_h1, line_h2, line_h3;     /* variables of decode_pscd */
  unsigned long line_l1, line_l2, line_l3;
  int pseudo;          /* flag for TPBON/TPDON:  next pixel is pseudo pixel */
  int **lntp;         /* flag [plane][layer-dl] for TP: line is not typical */

  unsigned long xmax, ymax;          /* if possible abort before image gets *
                                      * larger than this size */
  int dmax;                                      /* abort after this layer */
}jbg_dec_state;


/* some macros (too trivial for a function) */





#endif /* JBG_H */
```

A.5 jbig_head.sh

```
/*
 *  Header file for the portable free JBIG compression library
 *
 *  Markus Kuhn -- mkuhn@acm.org
 *
 *  £Id: jbig.h,v 1.8 1998-04-11 02:26:33+01 mgk25 Rel £
 */

#ifndef JBG_H
#define JBG_H

#include <stddef.h>
#include "constant.sh"

/*
 * JBIG-KIT version number
 */



/*
 * Buffer block for SDEs which are temporarily stored by encoder
 */






struct jbg_buf {
  unsigned char d[JBG_BUFSIZE];                 /* one block of a buffer list */
  int len;                                /* length of the data in this block */
  struct jbg_buf *next;                               /* pointer to next block */
  struct jbg_buf *previous;                       /* pointer to previous block *
                                                   * (unused in freelist)     */
  struct jbg_buf *last;      /* only used in list head: final block of list */
  struct jbg_buf **free_list;    /* pointer to pointer to head of free list */
};

/*
 * Maximum number of allowed ATMOVEs per stripe
 */


```

```
/*
 * Option and order flags
 */


/*
 * Language code for error message strings (based on ISO 639 2-letter
 * standard language name abbreviations).
 */



/*
 * Status description of an arithmetic encoder
 */


typedef struct jbg_arenc_state {
  unsigned char st[4096];     /* probability status for contexts, MSB = MPS */
  unsigned long c;                  /* C register, base of coding intervall, *
                                     * layout as in Table 23               */
  unsigned long a;      /* A register, normalized size of coding intervall */
  long sc;      /* counter for buffered 0xff values which might overflow */
  int ct;  /* bit shift counter, determines when next byte will be written */
  int buffer;                 /* buffer for most recent output byte != 0xff */
  void (*byte_out)(int, void *); /* function which receives all PSCD bytes */
  void *file;                               /* parameter passed to byte_out */
}jbg_arenc_state;



/*
 * Status description of an arithmetic decoder
 */

/* copy the enum from the inside of struct to outside of struct */
typedef enum{
    JBG_OK,                          /* symbol has been successfully decoded */
    JBG_READY,              /* no more bytes of this PSCD required, marker *
                             * encountered, probably more symbols available */
    JBG_MORE,            /* more PSCD data bytes required to decode a symbol */
    JBG_MARKER    /* more PSCD data bytes required, ignored final 0xff byte */
} JBG_RESULT;                              /* result of previous decode call */

typedef struct jbg_ardec_state {
  unsigned char st[4096];     /* probability status for contexts, MSB = MPS */
  unsigned long c;                  /* C register, base of coding intervall, *
                                     * layout as in Table 25               */
  unsigned long a;      /* A register, normalized size of coding intervall */
  int ct;     /* bit shift counter, determines when next byte will be read */
  unsigned char *pscd_ptr;               /* pointer to next PSCD data byte */
  unsigned char *pscd_end;                  /* pointer to byte after PSCD */
  JBG_RESULT result;

  int startup;                              /* controls initial fill of s->c */
}jbg_ardec_state;



/* self-defined struct */
typedef struct local_data {
  unsigned char *hp, *lp1, *lp2, *p0, *p1, *q1, *q2;
  unsigned long hl, ll, hx, hy, lx, ly, hbpl, lbpl;
  unsigned long line_h0, line_h1;
  unsigned long line_h2, line_h3, line_l1, line_l2, line_l3;
  struct jbg_arenc_state *se;
  unsigned long i, j, y;
  unsigned t;
  int ltp, ltp_old, cx;
  unsigned long c_all, c[MXMAX + 1], cmin, cmax, clmin, clmax;
  int tmax, at_determined;
  int new_tx;
  long new_tx_line;
  struct jbg_buf *new_jbg_buf;
}local_data;



/*
 * Status of a JBIG encoder
 */

typedef struct jbg_enc_state {
  int d;                            /* resolution layer of the input image */
  unsigned long xd, yd;    /* size of the input image (resolution layer d) */
  int planes;                         /* number of different bitmap planes */
  int dl;                       /* lowest resolution layer in the next BIE */
  int dh;                      /* highest resolution layer in the next BIE */
  unsigned long l0;                /* number of lines per stripe at lowest *
                                    * resolution layer 0                   */
  unsigned long stripes;    /* number of stripes required  (determ. by l0) */
  unsigned char **lhp[2];    /* pointers to lower/higher resolution images */
  int *highres;                 /* index [plane] of highres image in lhp[] */
  int order;                                    /* SDE ordering parameters */
  int options;                                      /* encoding parameters */
  unsigned mx, my;                           /* maximum ATMOVE window size */
  int *tx;       /* array [plane] with x-offset of adaptive template pixel */
  char *dppriv;         /* optional private deterministic prediction table */
  char *res_tab;           /* table for the resolution reduction algorithm */
  struct jbg_buf ****sde;      /* array [stripe][layer][plane] pointers to *
                                * buffers for stored SDEs                  */
  struct jbg_arenc_state *s;  /* array [planes] for arithm. encoder status */
  struct jbg_buf *free_list; /* list of currently unused SDE block buffers */
  void (*data_out)(unsigned char *start, size_t len, void *file);
                                                    /* data write callback */
  void *file;                            /* parameter passed to data_out() */
  char *tp;    /* buffer for temp. values used by diff. typical prediction */
}jbg_enc_state;


/*
 * Status of a JBIG decoder
 */

typedef struct jbg_dec_state {
  /* data from BIH */
  int d;                             /* resolution layer of the full image */
  int dl;                            /* first resolution layer in this BIE */
  unsigned long xd, yd;     /* size of the full image (resolution layer d) */
  int planes;                         /* number of different bitmap planes */
  unsigned long l0;                /* number of lines per stripe at lowest *
                                    * resolution layer 0                   */
  unsigned long stripes;    /* number of stripes required  (determ. by l0) */
  int order;                                    /* SDE ordering parameters */
  int options;                                      /* encoding parameters */
  int mx, my;                                /* maximum ATMOVE window size */
  char *dppriv;         /* optional private deterministic prediction table */

  /* loop variables */
  unsigned long ii[3];  /* current stripe, layer, plane (outer loop first) */
  /*
   * Pointers to array [planes] of lower/higher resolution images.
   * lhp[d & 1] contains image of layer d.
   */
  unsigned char **lhp[2];

  /* status information */
  int **tx, **ty;   /* array [plane][layer-dl] with x,y-offset of AT pixel */
  struct jbg_ardec_state **s;    /* array [plane][layer-dl] for arithmetic *
                                  * decoder status */
  int **reset;       /* array [plane][layer-dl] remembers if previous stripe *
                      * in that plane/resolution ended with SDRST.          */
  unsigned long bie_len;                    /* number of bytes read so far */
  unsigned char buffer[20]; /* used to store BIH or marker segments fragm. */
  int buf_len;                                /* number of bytes in buffer */
  unsigned long comment_skip;        /* remaining bytes of a COMMENT segment */
  unsigned long x;              /* x position of next pixel in current SDE */
  unsigned long i; /* line in current SDE (first line of each stripe is 0) */
  int at_moves;                /* number of AT moves in the current stripe */
  unsigned long at_line[JBG_ATMOVES_MAX];              /* lines at which an *
                                                        * AT move will happen */
  int at_tx[JBG_ATMOVES_MAX], at_ty[JBG_ATMOVES_MAX]; /* ATMOVE offsets in *
                                                       * current stripe    */
  unsigned long line_h1, line_h2, line_h3;       /* variables of decode_pscd */
  unsigned long line_l1, line_l2, line_l3;
  int pseudo;          /* flag for TPBON/TPDON: next pixel is pseudo pixel */
  int **lntp;        /* flag [plane][layer-dl] for TP: line is not typical */

  unsigned long xmax, ymax;          /* if possible abort before image gets *
                                      * larger than this size */
  int dmax;                                       /* abort after this layer */
} jbg_dec_state;


/* some macros (too trivial for a function) */



#endif /* JBG_H */
```

A.6 std_include.sc

```
#include <stdio.h>
#include <stdlib.h>
```

A.7 jbig.sc

```
/*
 *  Portable Free JBIG image compression library
 *
 *  Markus Kuhn -- mkuhn@acm.org
 *
 *  £Id: jbig.c,v 1.10 1998-04-11 02:26:33+01 mgk25 Rel £
 *
 *  This module implements a portable standard C encoder and decoder
 *  using the JBIG lossless bi-level image compression algorithm as
 *  specified in International Standard ISO 11544 or equivalently as
 *  specified in ITU-T Recommendation T.82. See the file jbig.doc
 *  for usage instructions and application examples.
 *
 *  This program is free software; you can redistribute it and/or modify
 *  it under the terms of the GNU General Public License as published by
 *  the Free Software Foundation; either version 2 of the License, or
 *  (at your option) any later version.
 *
 *  This program is distributed in the hope that it will be useful,
 *  but WITHOUT ANY WARRANTY; without even the implied warranty of
 *  MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 *  GNU General Public License for more details.
 *
 *  You should have received a copy of the GNU General Public License
 *  along with this program; if not, write to the Free Software
 *  Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
 *
 *  If you want to use this program under different license conditions,
 *  then contact the author for an arrangement.
 *
 *  It is possible that certain products which can be built using this
 *  software module might form inventions protected by patent rights in
 *  some countries (e.g., by patents about arithmetic coding algorithms
 *  owned by IBM and AT&T in the USA). Provision of this software by the
 *  author does NOT include any licences for any patents. In those
 *  countries where a patent licence is required for certain applications
 *  of this software module, you will have to obtain such a licence
 *  yourself.
 */

#ifdef DEBUG


#else
#define NDEBUG
#endif

#include <string.h>
#include <assert.h>
#include "constant.sh"


 #include "jbig_tab.sh"

import "std_include";
import "jbig_head";


/* optional export of arithmetic coder functions for test purposes */
#ifdef TEST_CODEC
#define ARITH
#define ARITH_INL
#else
#define ARITH      static
#ifdef __GNUC__
#define ARITH_INL  static
#else
#define ARITH_INL  static
#endif
#endif



/* object code version id */

```

```
static void jbg_buf_write(int b, void *head);
static struct jbg_buf *jbg_buf_init(struct jbg_buf **free_list);
static void jbg_buf_prefix(struct jbg_buf *new_prefix, struct jbg_buf **start);
ARITH void arith_encode_init(struct jbg_arenc_state *s, int reuse_st);


const char jbg_version[] =
"␣JBIG-KIT␣" JBG_VERSION "␣--␣Markus␣Kuhn␣--␣"
"$Id:␣jbig.c,v␣1.10␣1998-04-11␣02:26:33+01␣mgk25␣Rel␣$␣";

/*
 * the following array specifies for each combination of the 3
 * ordering bits, which ii[] variable represents which dimension
 * of s->sde.
 */
static const int index[8][3] = {
  { 2, 1, 0 },    /* no ordering bit set */
  { -1, -1, -1},  /* SMID -> illegal combination */
  { 2, 0, 1 },    /* ILEAVE */
  { 1, 0, 2 },    /* SMID + ILEAVE */
  { 0, 2, 1 },    /* SEQ */
  { 1, 2, 0 },    /* SEQ + SMID */
  { 0, 1, 2 },    /* SEQ + ILEAVE */
  { -1, -1, -1 }  /* SEQ + SMID + ILEAVE -> illegal combination */
};


/*
 * Array [language][message] with text string error messages that correspond
 * to return values from public functions in this library.
 */
#define NEMSG         9  /* number of error codes */
#define NEMSG_LANG    3  /* number of supported languages */
static const char *errmsg[NEMSG_LANG][NEMSG] = {
  /* English (JBG_EN) */
  {
    "Everything␣is␣ok",                                      /* JBG_EOK */
    "Reached␣specified␣maximum␣size",                        /* JBG_EOK_INTR */
    "Unexpected␣end␣of␣data",                                /* JBG_EAGAIN */
    "Not␣enough␣memory␣available",                           /* JBG_ENOMEM */
    "ABORT␣marker␣found",                                    /* JBG_EABORT */
    "Unknown␣marker␣segment␣encountered",                    /* JBG_EMARKER */
    "Incremental␣BIE␣does␣not␣fit␣to␣previous␣one",          /* JBG_ENOCONT */
    "Invalid␣data␣encountered",                              /* JBG_EINVAL */
    "Unimplemented␣features␣used"                            /* JBG_EIMPL */
  },
  /* German (JBG_DE_8859_1) */
  {
    "Kein␣Problem␣aufgetreten",                              /* JBG_EOK */
    "Angegebene␣maximale␣Bildgr\366\337e␣erreicht",          /* JBG_EOK_INTR */
    "Unerwartetes␣Ende␣der␣Daten",                           /* JBG_EAGAIN */
    "Nicht␣gen\374gend␣Speicher␣vorhanden",                  /* JBG_ENOMEM */
    "Es␣wurde␣eine␣Abbruch-Sequenz␣gefunden",                /* JBG_EABORT */
    "Eine␣unbekannte␣Markierungssequenz␣wurde␣gefunden",     /* JBG_EMARKER */
    "Neue␣Daten␣passen␣nicht␣zu␣vorangegangenen␣Daten",      /* JBG_ENOCONT */
    "Es␣wurden␣ung\374ltige␣Daten␣gefunden",                 /* JBG_EINVAL */
    "Noch␣nicht␣implementierte␣Optionen␣wurden␣benutzt"      /* JBG_EIMPL */
  },
  /* German (JBG_DE_UTF_8) */
  {
    "Kein␣Problem␣aufgetreten",                              /* JBG_EOK */
    "Angegebene␣maximale␣Bildgr\303\266\303\237e␣erreicht",  /* JBG_EOK_INTR */
    "Unerwartetes␣Ende␣der␣Daten",                           /* JBG_EAGAIN */
    "Nicht␣gen\303\274gend␣Speicher␣vorhanden",              /* JBG_ENOMEM */
```

```
      "Es␣wurde␣eine␣Abbruch−Sequenz␣gefunden",                /* JBG_EABORT */
      "Eine␣unbekannte␣Markierungssequenz␣wurde␣gefunden",     /* JBG_EMARKER */
      "Neue␣Daten␣passen␣nicht␣zu␣vorangegangenen␣Daten",      /* JBG_ENOCONT */
      "Es␣wurden␣ung\303\274ltige␣Daten␣gefunden",             /* JBG_EINVAL */
      "Noch␣nicht␣implementierte␣Optionen␣wurden␣benutzt"      /* JBG_EIMPL */
    }
  };


  /*
   * Calculate y = ceil(x/2) applied n times. This function is used to
   * determine the number of pixels per row or column after n resolution
   * reductions. E.g. X[d−1] = jbg_ceil_half(X[d], 1) and X[0] =
   * jbg_ceil_half(X[d], d) as defined in clause 6.2.3 of T.82.
   */
  unsigned long jbg_ceil_half(unsigned long x, int n)
  {
    unsigned long mask;

    mask = (1UL << n) − 1;     /* the lowest n bits are 1 here */
    return (x >> n) + ((mask & x) != 0);
  }


  /*
   * The following three functions are the only places in this code, were
   * C library memory management functions are called. The whole JBIG
   * library has been designed in order to allow multi−threaded
   * execution. no static or global variables are used, so all fuctions
   * are fully reentrant. However if you want to use this multi−thread
   * capability and your malloc, realloc and free are not reentrant,
   * then simply add the necessary semaphores or mutex primitives below.
   */

  static void *checked_malloc(size_t size)
  {
    void *p;

    p = malloc(size);
    /* Full manual exception handling is ugly here for performance
     * reasons. If an adequate handling of lack of memory is required,
     * then use C++ and throw a C++ exception here. */
    if (!p)
      abort();

  #if 0
    fprintf(stderr, "%p␣=␣malloc(%ld)\n", p, (long) size);
  #endif

    return p;
  }

  /*
  static void *checked_realloc(void *ptr, size_t size)
  {
    void *p;

    p = realloc(ptr, size);

    if (!p)
      abort();

  #if 0
    fprintf(stderr, "%p = realloc(%p, %ld)\n", p, ptr, (long) size);
  #endif
```

```
    return p;
  }
*/

static void checked_free(void *ptr)
{
  free(ptr);

#if 0
  fprintf(stderr, "free(%p)\n", ptr);
#endif

}


/*
 * Memory management for buffers which are used for temporarily
 * storing SDEs by the encoder.
 *
 * The following functions manage a set of struct jbg_buf storage
 * containers were each can keep JBG_BUFSIZE bytes. The jbg_buf
 * containers can be linked to form linear double-chained lists for
 * which a number of operations are provided. Blocks which are
 * tempoarily not used any more are returned to a freelist which each
 * encoder keeps. Only the destructor of the encoder actually returns
 * the block via checked_free() to the stdlib memory management.
 */


/*
 * Allocate a new buffer block and initialize it. Try to get it from
 * the free_list, and if it is empty, call checked_malloc().
 */
static struct jbg_buf *jbg_buf_init(struct jbg_buf **free_list)
{
  struct jbg_buf *new_block;

  /* Test whether a block from the free list is available */
  if (*free_list) {
    new_block = *free_list;
    *free_list = new_block->next;
  } else {
    /* request a new memory block */
    new_block = (struct jbg_buf *) checked_malloc(sizeof(struct jbg_buf));
  }
  new_block->len = 0;
  new_block->next = NULL;
  new_block->previous = NULL;
  new_block->last = new_block;
  new_block->free_list = free_list;

  return new_block;
}


/*
 * Return an entire free_list to the memory management of stdlib.
 * This is only done by jbg_enc_free().
 */
static void jbg_buf_free(struct jbg_buf **free_list)
{
  struct jbg_buf *tmp;

  while (*free_list) {
    tmp = (*free_list)->next;
    checked_free(*free_list);
    *free_list = tmp;
  }

  return;
}


/*
 * Append a single byte to a single list that starts with the block
 * *(struct jbg_buf *) head. The type of *head is void here in order to
 * keep the interface of the arithmetic encoder gereric, which uses this
 * function as a call-back function in order to deliver single bytes
 * for a PSCD.
 */
static void jbg_buf_write(int b, void *head)
{
  struct jbg_buf *now;

  now = ((struct jbg_buf *) head)->last;
  if (now->len < JBG_BUFSIZE - 1) {
    now->d[now->len++] = b;
    return;
  }
  now->next = jbg_buf_init(((struct jbg_buf *) head)->free_list);
  now->next->previous = now;
  now->next->d[now->next->len++] = b;
  ((struct jbg_buf *) head)->last = now->next;

  return;
}


/*
 * Remove any trailing zero bytes from the end of a linked jbg_buf list,
 * however make sure that no zero byte is removed which directly
 * follows a 0xff byte (i.e., keep MARKER_ESC MARKER_STUFF sequences
 * intact). This function is used to remove any redundant final zero
 * bytes from a PSCD.
 */
static void jbg_buf_remove_zeros(struct jbg_buf *head)
{
  struct jbg_buf *last;

  while (1) {
    /* remove trailing 0x00 in last block of list until this block is empty */
    last = head->last;
    while (last->len && last->d[last->len - 1] == 0)
      last->len--;
    /* if block became really empty, remove it in case it is not the
     * only remaining block and then loop to next block */
    if (last->previous && !last->len) {
      head->last->next = *head->free_list;
      *head->free_list = head->last;
      head->last = last->previous;
    } else
      break;
  }

  /*
   * If the final non-zero byte is 0xff (MARKER_ESC), then we just have
   * removed a MARKER_STUFF and we will append it again now in order
   * to preserve PSCD status of byte stream.
   */
  if (head->last->len && head->last->d[head->last->len - 1] == MARKER_ESC)
    jbg_buf_write(MARKER_STUFF, head);
  return;
}


/*
 * The jbg_buf list which starts with block *new_prefix is concatenated
 * with the list which starts with block **start and *start will then point
 * to the first block of the new list.
 */
static void jbg_buf_prefix(struct jbg_buf *new_prefix, struct jbg_buf **start)
{
  new_prefix->last->next = *start;
  new_prefix->last->next->previous = new_prefix->last;
  new_prefix->last = new_prefix->last->next->last;
  *start = new_prefix;

  return;
}


/*
 * Send the contents of a jbg_buf list that starts with block **head to
 * the call back function data_out and return the blocks of the jbg_buf
 * list to the freelist from which these jbg_buf blocks have been taken.
 * After the call, *head == NULL.
 */
static void jbg_buf_output(struct jbg_buf **head,
                           void (*data_out)(unsigned char *start,
                                            size_t len, void *file),
                           void *file)
{
  struct jbg_buf *tmp;

  while (*head) {
    (*data_out)((*head)->d, (*head)->len, file);
    tmp = (*head)->next;
    (*head)->next = *(*head)->free_list;
    *(*head)->free_list = *head;
    *head = tmp;
  }

  return;
}


/*
 * Convert the table which controls the deterministic prediction
 * process from the 1728 byte long DPTABLE format into the 6912 byte long
 * internal format.
 */
void jbg_dppriv2int(char *internal, const unsigned char *dptable)
{
  int i, j, k;
  int trans0[ 8] = { 1, 0, 3, 2, 7, 6, 5, 4 };
  int trans1[ 9] = { 1, 0, 3, 2, 8, 7, 6, 5, 4 };
  int trans2[11] = { 1, 0, 3, 2, 10, 9, 8, 7, 6, 5, 4 };
  int trans3[12] = { 1, 0, 3, 2, 11, 10, 9, 8, 7, 6, 5, 4 };

#define FILL_TABLE2(offset, len, trans) \
  for (i = 0; i < len; i++) { \
    k = 0; \
    for (j = 0; j < 8; j++) \
      k |= ((i >> j) & 1) << trans[j]; \
    internal[k + offset] = \
      (dptable[(i + offset) >> 2] >> ((3 - (i & 3)) << 1)) & 3; \
    }

    FILL_TABLE2(   0,  256, trans0);
    FILL_TABLE2( 256,  512, trans1);
    FILL_TABLE2( 768, 2048, trans2);
    FILL_TABLE2(2816, 4096, trans3);

    return;
  }




/*
 * Convert the error codes used by jbg_dec_in() into a string
 * written in the selected language and character set.
 */
const char *jbg_strerror(int errnum, int language)
{
  if (errnum < 0 || errnum >= NEMSG)
    return "Unknown␣error␣code␣passed␣to␣jbg_strerror()";
  if (language < 0 || language >= NEMSG_LANG)
    return "Unknown␣language␣code␣passed␣to␣jbg_strerror()";

  return errmsg[language][errnum];
}
```

A.8 jbg_enc_init.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";

/*
 * Initialize the status struct for the encoder.
 */
behavior jbg_enc_init(in struct jbg_enc_state *s,
                      in unsigned long x,
                      in unsigned long y,
                      in int planes,
                      in unsigned char **p)

{
void main(void){
  unsigned long l, lx;
  int i;
  size_t bufsize;
/*
    char jbg_resred[NUM], jbg_dptable[NUM];
*/
  s->xd = x;
  s->yd = y;
  s->planes = planes;


  s->d = 0;
  s->dl = 0;
```

```
    s->dh = s->d;
    s->l0 = jbg_ceil_half(s->yd, s->d) / 35;    /* 35 stripes/image */
    while ((s->l0 << s->d) > 128)               /* but <= 128 lines/stripe */
      --s->l0;
    if (s->l0 < 2) s->l0 = 2;
    s->mx = 8;
    s->my = 0;
    s->order = JBG_ILEAVE | JBG_SMID;
    s->options = JBG_TPBON | JBG_TPDON | JBG_DPON;
    s->dppriv = jbg_dptable;
    s->res_tab = jbg_resred;

    s->highres = (int *) checked_malloc(planes * sizeof(int));
    s->lhp[0] = p;
    s->lhp[1] = (unsigned char **) checked_malloc(planes * sizeof(unsigned char *));
    bufsize = ((jbg_ceil_half(x, 1) + 7) / 8) * jbg_ceil_half(y, 1);
    for (i = 0; i < planes; i++) {
      s->highres[i] = 0;
      s->lhp[1][i] = (unsigned char *) checked_malloc(sizeof(unsigned char) * bufsize);
    }

    s->free_list = NULL;
    s->s = (struct jbg_arenc_state *)
      checked_malloc(s->planes * sizeof(struct jbg_arenc_state));
    s->tx = (int *) checked_malloc(s->planes * sizeof(int));
    lx = jbg_ceil_half(x, 1);
    s->tp = (char *) checked_malloc(lx * sizeof(char));
    for (l = 0; l < lx; s->tp[l++] = 2);
    s->sde = NULL;

  }
};
```

A.9 jbg_enc_layers.sc

```
#include "constant.sh"


import "jbig_head";
import "jbig";


/*
 * As an alternative to jbg_enc_lrlmax(), the following function allows
 * to specify the number of layers directly. The stripe height and layer
 * range is also adjusted automatically here.
 */
behavior jbg_enc_layers(in struct jbg_enc_state *s,
                        in int d)
{ void main(void)
{
  if (d < 0 || d > 255)
    return;
  s->d  = d;
  s->dl = 0;
  s->dh = s->d;

  s->l0 = jbg_ceil_half(s->yd, s->d) / 35;    /* 35 stripes/image */
  while ((s->l0 << s->d) > 128)               /* but <= 128 lines/stripe */
    --s->l0;
  if (s->l0 < 2) s->l0 = 2;

  return;
  }
};
```

A.10 jbg_enc_lrange.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";

/*
 * Specify the highest and lowest resolution layers which will be
 * written to the output file. Call this function not before
 * jbg_enc_layers() or jbg_enc_lrlmax(), because these two functions
 * reset the lowest and highest resolution layer to default values.
 * Negative values are ignored. The total number of layers is returned.
 */


behavior jbg_enc_lrange(in struct jbg_enc_state *s,
                        in  int dl,
                        in int dh)
{
void main(void){
  if (dl >= 0      && dl <= s->d) s->dl = dl;
  if (dh >= s->dl && dh <= s->d) s->dh = dh;


}
};
```

A.11 jbg_enc_lrlmax.sc

```
#include "constant.sh"


import "jbig_head";
import "jbig";

/*
 * This function selects the number of differential layers based on
 * the maximum size requested for the lowest resolution layer. If
 * possible, a number of differential layers is selected, which will
 * keep the size of the lowest resolution layer below or equal to the
 * given width x and height y. However not more than 6 differential
 * resolution layers will be used. In addition, a reasonable value for
 * l0 (height of one stripe in the lowest resolution layer) is
 * selected, which obeys the recommended limitations for l0 in annex A
 * and C of the JBIG standard. The selected number of resolution layers
 * is returned.
 */


behavior jbg_enc_lrlmax(in struct jbg_enc_state *s,
                        in  long x,
                        in  long y)
{
void main(void){
  for (s->d = 0; s->d < 6; s->d++)
    if ((long) jbg_ceil_half(s->xd, s->d) <= x && (long) jbg_ceil_half(s->yd, s->d) <= y)
      break;
  s->dl = 0;
```

```
    s->dh = s->d;

    s->l0 = jbg_ceil_half(s->yd, s->d) / 35;  /* 35 stripes/image */
    while ((s->l0 << s->d) > 128)             /* but <= 128 lines/stripe */
      --s->l0;
    if (s->l0 < 2) s->l0 = 2;

    /* return s->d;*/
  }
  };
```

A.12 jbg_enc_options.sc

```
#include "constant.sh"


import "jbig_head";
import "jbig";

/*
 * The following function allows to specify the bits describing the
 * options of the format as well as the maximum AT movement window and
 * the number of layer 0 lines per stripes.
 */
behavior jbg_enc_options(in struct jbg_enc_state *s,
                         in int order,
                         in int options,
                         in long l0,
                         in int mx,
                         in int my)
{
void main(void){
   if (order >= 0 && order <= 0x0f) s->order = order;
   if (options >= 0) s->options = options;
   if (l0 >= 0) s->l0 = l0;
   if (mx >= 0 && my < 128) s->mx = mx;
   if (my >= 0 && my < 256) s->my = my;


}
};
```

A.13 jbg_split_planes.sc

```
#include "constant.sh"


import "jbig_head";
import "jbig";




behavior jbg_split_planes(in unsigned long x,
                          in unsigned long y,
                          in int has_planes,
                          in int encode_planes,
                          in unsigned char *src,
                          in unsigned char **dest,
                          inout int use_graycode)
{
  void main(void){
    unsigned char *localsrc;
    unsigned bpl;
    unsigned i, k = 8;
    int p;
    unsigned long line;
  /*
     void *memset(void *s, int c, size_t n);
  */
    unsigned prev;      /* previous *src byte shifted by 8 bit to the left */
    register int bits, msb;
    int bitno;

    localsrc=src;


    bpl = (x + 7) / 8;              /* bytes per line in dest plane */
    msb = has_planes - 1;

    /* sanity checks */
    if (encode_planes > has_planes)
      encode_planes = has_planes;
    use_graycode = use_graycode != 0 && encode_planes > 1;

    for (p = 0; p < encode_planes; p++)
      memset(dest[p], 0, bpl * y);

    for (line = 0; line < y; line++) {                      /* lines loop */
      for (i = 0; i * 8 < x; i++) {                          /* dest bytes loop */
        for (k = 0; k < 8 && i * 8 + k < x; k++) {           /* pixel loop */
          prev = 0;
          for (p = 0; p < encode_planes; p++) {              /* bit planes loop */
            /* calculate which bit in *src do we want */
            bitno = (msb - p) & 7;
            /* put this bit with its left neighbor right adjusted into bits */
            bits = (prev | *localsrc) >> bitno;
            /* go to next *src byte, but keep old */
            if (bitno == 0)
              prev = *localsrc++;
            /* make space for inserting new bit */
            dest[p][bpl * line + i] <<= 1;
            /* insert bit, if requested apply Gray encoding */
            dest[p][bpl * line + i] |= (bits ^ (use_graycode & (bits>>1))) & 1;
            /*
             * Theorem: Let b(n),..., b(1),b(0) be the digits of a
             * binary word and let g(n),..., g(1),g(0) be the digits of the
             * corresponding Gray code word, then g(i) = b(i) xor b(i+1).
             */
          }
          /* skip unused *src bytes */
          for (;p < has_planes; p++)
            if (((has_planes - 1 - p) & 7) == 0)
              localsrc++;
        }
      }
      for (p = 0; p < encode_planes; p++)                    /* right padding loop */
        dest[p][bpl * (line + 1) - 1] <<= 8 - k;
    }

  return;
}
  /* for main */

}; /*for behavior */
```

A.14 jbg_enc_out.sc

```
#include "constant.sh"


import "std_include";
import "jbig_head";
import "jbig";
import "output_sde";
import "encode_sde";
import "resolution_reduction";
import "jbg_int2dppriv";



/*
 * Encode one full BIE and pass the generated data to the specified
 * call-back function
 */


behavior jbg_enc_out(inout struct jbg_enc_state *s)
{
  long bpl;
  unsigned char bih[20];
  unsigned long xd, yd, y;
  long ii[3], is[3], ie[3];     /* generic variables for the 3 nested loops */
  unsigned long stripe;
  int layer, plane;
  int order;
  unsigned char *dpbuf;
  char *local_dppriv;

output_sde output_sde_exec(s, stripe, layer, plane);
encode_sde encode_sde_exec(s, stripe, layer, plane);
resolution_reduction resolution_reduction_exec(s, plane, layer);
jbg_int2dppriv jbg_int2dppriv_exec(dpbuf, local_dppriv);

void main(void)
{

/*
    char jbg_dptable[NUM];
*/

  dpbuf = (unsigned char *) malloc(sizeof(char) * 1728);
  /* some sanity checks */
  s->order &= JBG_HITOLO | JBG_SEQ | JBG_ILEAVE | JBG_SMID;
  order = s->order & (JBG_SEQ | JBG_ILEAVE | JBG_SMID);
  if (index[order][0] < 0)
    s->order = order = JBG_SMID | JBG_ILEAVE;
  if (s->options & JBG_DPON && s->dppriv != jbg_dptable)
    s->options |= JBG_DPPRIV;
  if (s->mx > MX_MAX)
    s->mx = MX_MAX;
  s->my = 0;
  if (s->mx && s->mx < ((s->options & JBG_LRLTWO) ? 5U : 3U))
    s->mx = 0;
  if (s->d > 255 || s->d < 0 || s->dh > s->d || s->dh < 0 ||
      s->dl < 0 || s->dl > s->dh || s->planes < 0 || s->planes > 255)
    return;

  /* ensure correct zero padding of bitmap at the final byte of each line */
  if (s->xd & 7) {
    bpl = (s->xd + 7) / 8;      /* bytes per line */
    for (plane = 0; plane < s->planes; plane++)
      for (y = 0; y < s->yd; y++)
        s->lhp[0][plane][y * bpl + bpl - 1] &= ~((1 << (8 - (s->xd & 7))) - 1);
  }

  /* calculate number of stripes that will be required */
  s->stripes = ((s->yd >> s->d) +
                ((((1UL << s->d) - 1) & s->xd) != 0) + s->l0 - 1) / s->l0;

  /* allocate buffers for SDE pointers */
  if (s->sde == NULL) {
    s->sde = (struct jbg_buf ****)
      checked_malloc(s->stripes * sizeof(struct jbg_buf ***));
    for (stripe = 0; stripe < s->stripes; stripe++) {
      s->sde[stripe] = (struct jbg_buf ***)
        checked_malloc((s->d + 1) * sizeof(struct jbg_buf **));
      for (layer = 0; layer < s->d + 1; layer++) {
        s->sde[stripe][layer] = (struct jbg_buf **)
          checked_malloc(s->planes * sizeof(struct jbg_buf *));
        for (plane = 0; plane < s->planes; plane++)
          s->sde[stripe][layer][plane] = SDE_TODO;
      }
    }
  }

  /* output BIH */
  bih[0] = s->dl;
  bih[1] = s->dh;
  bih[2] = s->planes;
  bih[3] = 0;
  xd = jbg_ceil_half(s->xd, s->d - s->dh);
  yd = jbg_ceil_half(s->yd, s->d - s->dh);
  bih[4] = xd >> 24;
  bih[5] = (xd >> 16) & 0xff;
  bih[6] = (xd >> 8) & 0xff;
  bih[7] = xd & 0xff;
  bih[8] = yd >> 24;
  bih[9] = (yd >> 16) & 0xff;
  bih[10] = (yd >> 8) & 0xff;
  bih[11] = yd & 0xff;
  bih[12] = s->l0 >> 24;
  bih[13] = (s->l0 >> 16) & 0xff;
  bih[14] = (s->l0 >> 8) & 0xff;
  bih[15] = s->l0 & 0xff;
  bih[16] = s->mx;
  bih[17] = s->my;
  bih[18] = s->order;
  bih[19] = s->options & 0x7f;
  (*s->data_out)(bih, 20, s->file);
  if ((s->options & (JBG_DPON | JBG_DPPRIV | JBG_DPLAST)) ==
      (JBG_DPON | JBG_DPPRIV)) {
    /* write private table */
    local_dppriv = s->dppriv;
    jbg_int2dppriv_exec.main();
    (*s->data_out)(dpbuf, 1728, s->file);
  }


  /*
   * Encode everything first. This is a simple-minded alternative to
   * all the tricky on-demand encoding logic in output_sde() for
   * debugging purposes.
   */
/*
  for (layer = s->dh; layer >= s->dl; layer--) {
    for (plane = 0; plane < s->planes; plane++) {
        if (layer > 0)
          resolution_reduction_exec.main();
        for (stripe = 0; stripe < s->stripes; stripe++)
          encode_sde_exec.main();
        s->highres[plane] ^= 1;
      }
    }
  */

  /*
   * Generic loops over all SDEs. Which loop represents layer, plane and
   * stripe depends on the option flags.
   */

  /* start and end value vor each loop */
  is[index[order][STRIPE]] = 0;
  ie[index[order][STRIPE]] = s->stripes - 1;
  is[index[order][LAYER]] = s->dl;
  ie[index[order][LAYER]] = s->dh;
  is[index[order][PLANE]] = 0;
  ie[index[order][PLANE]] = s->planes - 1;

  for (ii[0] = is[0]; ii[0] <= ie[0]; ii[0]++)
    for (ii[1] = is[1]; ii[1] <= ie[1]; ii[1]++)
      for (ii[2] = is[2]; ii[2] <= ie[2]; ii[2]++) {

        stripe = ii[index[order][STRIPE]];
        if (s->order & JBG_HITOLO)
          layer = s->dh - (ii[index[order][LAYER]] - s->dl);
        else
          layer = ii[index[order][LAYER]];
        plane = ii[index[order][PLANE]];

        output_sde_exec.main();

      }

  return;
}

};
```

A.15 pbmtojbg.sc

```
/*
 *  pbmtojbg - Portable Bitmap to JBIG converter
 *
 *  Markus Kuhn -- mkuhn@acm.org
 *
 *  £Id: pbmtojbg.c,v 1.8 1998-04-11 02:24:42+01 mgk25 Rel £
 */

#include <stddef.h>
#include <ctype.h>
#include "constant.sh"


import "std_include";
import "jbig";
import "jbg_enc_out";
import "jbg_split_planes";
import "jbg_enc_layers";
import "jbg_enc_lrlmax";
import "jbg_enc_free";
import "jbg_enc_lrange";
import "jbg_enc_options";
import "jbg_enc_init";

char *progname;                      /* global pointer to argv[0] */
unsigned long total_length = 0;      /* used for determining output file length */


/*
 * malloc() with exception handler
 */
void *checkedmalloc(size_t n)
{
  void *p;

  if ((p = malloc(n)) == NULL) {
    fprintf(stderr, "Sorry,␣not␣enough␣memory␣available!\n");
    exit(1);
  }

  return p;
}


/*
 * Read an ASCII integer number from file f and skip any PBM
 * comments which are encountered.
 */
static unsigned long getint(FILE *f)
{
  int c;
  unsigned long i;

  while ((c = getc(f)) != EOF && !isdigit(c))
    if (c == '#')
      while ((c = getc(f)) != EOF && !(c == 13 || c == 10)) ;
  if (c != EOF) {
    ungetc(c, f);
    fscanf(f, "%lu", &i);
  }

  return i;
}


/*
 * Callback procedure which is used by JBIG encoder to deliver the
 * encoded data. It simply sends the bytes to the output file.
 */
static void data_out(unsigned char *start, size_t len, void *file)
{
  fwrite(start, len, 1, (FILE *) file);
  total_length += len;
  return;
}


/*
 * Print usage message and abort
 */
static void usage(void)
{
  fprintf(stderr, "\n␣getting␣started␣....␣\n\n");
  exit(1);
}
```

```
behavior Main
{
  struct jbg_enc_state s;
  struct jbg_enc_state *s_ptr;
const char *fnin = "<stdin>", *fnout = "<stdout>";
  int i, j, c;
  int all_args = 0, files = 0;
  unsigned long x, y;
  unsigned long width, height, max, v;
  unsigned long bpl;
  int bpp, planes, encode_planes = -1;
  size_t bitmap_size;
  char type;
  unsigned char **bitmap, *p;
  unsigned char *image;

  int verbose = 0, delay_at = 0, use_graycode = 1;
  long mwidth = 640, mheight = 480;
  int dl = -1, dh = -1, d = -1, mx= -1;
  int long l0=-1;
  int options = JBG_TPDON | JBG_TPBON | JBG_DPON;
  int order = JBG_ILEAVE | JBG_SMID;
  FILE *fin, *fout;
  int const_nl =-1;



jbg_enc_out jbg_enc_out_exec(s_ptr);
jbg_split_planes jbg_split_planes_exec(width, height, planes, encode_planes, image, bitmap,
                          use_graycode);
jbg_enc_layers jbg_enc_layers_exec(s_ptr, d);
jbg_enc_lrlmax jbg_enc_lrlmax_exec(s_ptr, mwidth, mheight);
jbg_enc_free jbg_enc_free_exec(s_ptr);
jbg_enc_lrange jbg_enc_lrange_exec(s_ptr, dl, dh);
jbg_enc_options jbg_enc_options_exec(s_ptr, order, options, l0, mx, const_nl);
jbg_enc_init jbg_enc_init_exec(s_ptr, width, height, encode_planes, bitmap);

int main (int argc, char **argv)
{


  fin = stdin;
  fout = stdout;

  s_ptr = &s;
  /* parse command line arguments */
  progname = argv[0];
  for (i = 1; i < argc; i++) {
    if (! all_args && argv[i][0] == '-')
      if (argv[i][1] == '\0' && files == 0)
        ++files;
      else
        for (j = 1; j > 0 && argv[i][j]; j++)
          switch(tolower(argv[i][j])) {
          case '-' :
            all_args = 1;
            break;
          case 'v':
            verbose = 1;
            break;
          case 'b':
            use_graycode = 0;
            break;
          case 'c':
            delay_at = 1;
            break;
          case 'x':
            if (++i >= argc) usage();
            j = -1;
            mwidth = atol(argv[i]);
            break;
          case 'y':
            if (++i >= argc) usage();
            j = -1;
            mheight = atol(argv[i]);
            break;
          case 'o':
            if (++i >= argc) usage();
            j = -1;
            order = atoi(argv[i]);
            break;
          case 'p':
            if (++i >= argc) usage();
            j = -1;
            options = atoi(argv[i]);
            break;
          case 'l':
            if (++i >= argc) usage();
            j = -1;
            dl = atoi(argv[i]);
            break;
          case 'h':
            if (++i >= argc) usage();
            j = -1;
            dh = atoi(argv[i]);
            break;
          case 'q':
            d = 0;
            break;
          case 'd':
            if (++i >= argc) usage();
            j = -1;
            d = atoi(argv[i]);
            break;
          case 's':
            if (++i >= argc) usage();
            j = -1;
            l0 = atoi(argv[i]);
            break;
          case 't':
            if (++i >= argc) usage();
            j = -1;
            encode_planes = atoi(argv[i]);
            break;
          case 'm':
            if (++i >= argc) usage();
            j = -1;
            mx= atoi(argv[i]);
            break;
          default:
            usage();
          }
    else
      switch (files++) {
      case 0:
        if (argv[i][0] != '-' || argv[i][1] != '\0') {
          fnin = argv[i];
          fin = fopen(fnin, "rb");
          if (!fin) {
            fprintf(stderr, "Can't␣open␣input␣file␣'%s", fnin);
            perror("'");
          exit(1);
        }
      }
      break;
    case 1:
      fnout = argv[i];
      fout = fopen(fnout, "wb");
      if (!fout) {
        fprintf(stderr, "Can't␣open␣input␣file␣'%s", fnout);
        perror("'");
        exit(1);
      }
      break;
    default:
      usage();
    }
}

/* read PBM header */
while ((c = getc(fin)) != EOF && (isspace(c) || c == '#'))
  if (c == '#')
    while ((c = getc(fin)) != EOF && !(c == 13 || c == 10)) ;
if (c != 'P') {
  fprintf(stderr, "Input␣file␣'%s'␣does␣not␣look␣like␣a␣PBM␣file!\n", fnin);
  exit(1);
}
type = getc(fin);
width = getint(fin);
height = getint(fin);
if (type == '2' || type == '5' ||
    type == '3' || type == '6')
  max = getint(fin);
else
  max = 1;
for (planes = 0; 1UL << planes <= max; planes++);
bpp = (planes + 7) / 8;
if (encode_planes < 0 || encode_planes > planes)
  encode_planes = planes;
fgetc(fin);     /* skip line feed */

/* read PBM image data */
bpl = (width + 7) / 8;      /* bytes per line */
bitmap_size = bpl * (size_t) height;
bitmap = (unsigned char **) checkedmalloc(sizeof(unsigned char *) *
                                          encode_planes);
for (i = 0; i < encode_planes; i++)
  bitmap[i] = (unsigned char *) checkedmalloc(bitmap_size);
switch (type) {
case '1':
  /* PBM text format */
  p = bitmap[0];
  for (y = 0; y < height; y++)
    for (x = 0; x <= ((width-1) | 7); x++) {
      *p <<= 1;
      if (x < width)
        *p |= getint(fin) & 1;
      if ((x & 7) == 7)
        ++p;
    }
  break;
case '4':
  /* PBM raw binary format */
  fread(bitmap[0], bitmap_size, 1, fin);
  break;
case '2':
case '5':
    /* PGM */
    image = (unsigned char *) checkedmalloc(width * height * bpp);
    if (type == '2') {
      for (x = 0; x < width * height; x++) {
        v = getint(fin);
        for (j = 0; j < bpp; j++)
          image[x * bpp + (bpp - 1) - j] = v >> (j * 8);
      }
    } else
      fread(image, width * height, bpp, fin);
    jbg_split_planes_exec.main();
    free(image);
    break;
  default:
    fprintf(stderr, "Unsupported␣PBM␣type␣P%c!\n", type);
    exit(1);
  }
  if (ferror(fin)) {
    fprintf(stderr, "Problem␣while␣reading␣input␣file␣'%s", fnin);
    perror("");
    exit(1);
  }
  if (feof(fin)) {
    fprintf(stderr, "Unexpected␣end␣of␣input␣file␣'%s'!\n", fnin);
    exit(1);
  }

  /* Test the final byte in each image line for correct zero padding */
  if ((width & 7) && type == '4') {
    for (y = 0; y < height; y++)
      if (bitmap[0][y * bpl + bpl - 1] & ((1 << (8 - (width & 7))) - 1)) {
        fprintf(stderr, "Warning:␣No␣zero␣padding␣in␣last␣byte␣(0x%02x)␣of␣"
                "line␣%lu!\n", bitmap[0][y * bpl + bpl - 1], y + 1);
        break;
      }
  }

  /* Apply JBIG algorithm and write BIE to output file */

  /* initialize parameter struct for JBIG encoder*/
  s.data_out = data_out;
  s.file = fout;
  jbg_enc_init_exec.main();

  /* Select number of resolution layers either directly or based
   * on a given maximum size for the lowest resolution layer */
   if (d >= 0){

    jbg_enc_layers_exec.main();
   }else
    jbg_enc_lrlmax_exec.main();

  /* Specify a few other options (each is ignored if negative) */
  if (delay_at)
    options |= JBG_DELAY_AT;
  jbg_enc_lrange_exec.main();
  jbg_enc_options_exec.main();

  /* now encode everything and send it to data_out() */
  jbg_enc_out_exec.main();

  /* give encoder a chance to free its temporary data structures */
  jbg_enc_free_exec.main();

  /* check for file errors and close fout */
  if (ferror(fout) || fclose(fout)) {
      fprintf(stderr, "Problem while writing output file '%s'", fnout);
      perror("");
      exit(1);
    }

    /* In case the user wants to know all the gory details ... */
    if (verbose) {
      fprintf(stderr, "Information about the created JBIG bi-level image entity "
              "(BIE):\n\n");
      fprintf(stderr, "              input image size: %ld x %ld pixel\n",
              s.xd, s.yd);
      fprintf(stderr, "                    bit planes: %d\n", s.planes);
      if (s.planes > 1)
        fprintf(stderr, "                      encoding: %s code, MSB first\n",
                use_graycode ? "Gray" : "binary");
      fprintf(stderr, "                       stripes: %ld\n", s.stripes);
      fprintf(stderr, "   lines per stripe in layer 0: %ld\n", s.l0);
      fprintf(stderr, "  total number of diff. layers: %d\n", s.d);
      fprintf(stderr, "           lowest layer in BIE: %d\n", s.dl);
      fprintf(stderr, "          highest layer in BIE: %d\n", s.dh);
      fprintf(stderr, "             lowest layer size: %lu x %lu pixel\n",
              jbg_ceil_half(s.xd, s.d - s.dl), jbg_ceil_half(s.yd, s.d - s.dl));
      fprintf(stderr, "            highest layer size: %lu x %lu pixel\n",
              jbg_ceil_half(s.xd, s.d - s.dh), jbg_ceil_half(s.yd, s.d - s.dh));
      fprintf(stderr, "                   option bits:%s%s%s%s%s%s%s\n",
              s.options & JBG_LRLTWO  ? " LRLTWO" : "",
              s.options & JBG_VLENGTH ? " VLENGTH" : "",
              s.options & JBG_TPDON   ? " TPDON" : "",
              s.options & JBG_TPBON   ? " TPBON" : "",
              s.options & JBG_DPON    ? " DPON" : "",
              s.options & JBG_DPPRIV  ? " DPPRIV" : "",
              s.options & JBG_DPLAST  ? " DPLAST" : "");
      fprintf(stderr, "                    order bits:%s%s%s%s\n",
              s.order & JBG_HITOLO ? " HITOLO" : "",
              s.order & JBG_SEQ    ? " SEQ" : "",
              s.order & JBG_ILEAVE ? " ILEAVE" : "",
              s.order & JBG_SMID   ? " SMID" : "");
      fprintf(stderr, "           AT maximum x-offset: %d\n"
              "           AT maximum y-offset: %d\n", s.mx, s.my);
      fprintf(stderr, "         length of output file: %lu byte\n\n",
              total_length);
    }

        return 0;
}

  };
```

A.16 resolution_reduction.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";


behavior resolution_reduction(in struct jbg_enc_state *s,
                              in int plane,
                              in int higher_layer)
{
```

```
void main(void)
{
unsigned long hx, hy, lx, ly, hbpl, lbpl;
unsigned char *hp1, *hp2, *hp3, *lp;
unsigned long line_h1, line_h2, line_h3, line_l2;
unsigned long i, j;
int pix, k, l;

/* number of pixels in highres image */
hx = jbg_ceil_half(s->xd, s->d - higher_layer);
hy = jbg_ceil_half(s->yd, s->d - higher_layer);
/* number of pixels in lowres image */
lx = jbg_ceil_half(hx, 1);
ly = jbg_ceil_half(hy, 1);
/* bytes per line in highres and lowres image */
hbpl = (hx + 7) / 8;
lbpl = (lx + 7) / 8;
/* pointers to first image bytes */
hp2 = s->lhp[s->highres[plane]][plane];
hp1 = hp2 + hbpl;
hp3 = hp2 - hbpl;
lp = s->lhp[1 - s->highres[plane]][plane];

#ifdef DEBUG
fprintf(stderr, "resolution_reduction:␣plane␣=␣%d,␣higher_layer␣=␣%d\n",
        plane, higher_layer);
#endif

/*
 * Layout of the variables line_h1, line_h2, line_h3, which contain
 * as bits the high resolution neighbour pixels of the currently coded
 * lowres pixel /\:
 *                \/
 *
 *    76543210 76543210 76543210 76543210     line_h3
 *    76543210 76543210 765432/\ 76543210     line_h2
 *    76543210 76543210 765432\/ 76543210     line_h1
 *
 * Layout of the variable line_l2, which contains the low resolution
 * pixels near the currently coded pixel as bits. The lowres pixel
 * which is currently coded is marked as X:
 *
 *    76543210 76543210 76543210 76543210     line_l2
 *                              X
 */

  for (i = 0; i < ly; i++) {
    if (2*i + 1 >= hy)
      hp1 = hp2;
    pix = 0;
    line_h1 = line_h2 = line_h3 = line_l2 = 0;
    for (j = 0; j < lbpl * 8; j += 8) {
      *lp = 0;
      line_l2 |= i ? lp[-lbpl] : 0;
      for (k = 0; k < 8 && j + k < lx; k += 4) {
        if (((j + k) >> 2) < hbpl) {
          line_h3 |= i ? *hp3 : 0;
          ++hp3;
          line_h2 |= *(hp2++);
          line_h1 |= *(hp1++);
        }
        for (l = 0; l < 4 && j + k + l < lx; l++) {
          line_h3 <<= 2;
          line_h2 <<= 2;
          line_h1 <<= 2;
          line_l2 <<= 1;
          pix = s->res_tab[((line_h1 >> 8) & 0x007) |
                           ((line_h2 >> 5) & 0x038) |
                           ((line_h3 >> 2) & 0x1c0) |
                           (pix << 9) | ((line_l2 << 2) & 0xc00)];
          *lp = (*lp << 1) | pix;
        }
      }
      ++lp;
    }
    *(lp - 1) <<= lbpl * 8 - lx;
    hp1 += hbpl;
    hp2 += hbpl;
    hp3 += hbpl;
  }

#ifdef DEBUG
  {
    FILE *f;
    char fn[50];

    sprintf(fn, "dbg_d=%02d.pbm", higher_layer - 1);
    f = fopen(fn, "wb");
    fprintf(f, "P4\n%lu␣%lu\n", lx, ly);
    fwrite(s->lhp[1 - s->highres[plane]][plane], 1, lbpl * ly, f);
    fclose(f);
  }
#endif

  return;
}
};
```

A.17 int2dppriv.sc

```
/*
 * Convert the table which controls the deterministic prediction
 * process from the internal format into the representation required
 * for the 1728 byte long DPTABLE element of a BIH.
 *
 * The bit order of the DPTABLE format (see also ITU-T T.82 figure 13) is
 *
 * high res:   4  5  6     low res:  0  1
 *             7  8  9               2  3
 *            10 11 12
 *
 * were 4 table entries are packed into one byte, while we here use
 * internally an unpacked 6912 byte long table indexed by the following
 * bit order:
 *
 * high res:   7  6  5    high res:   8  7  6    low res:  1  0
 * (phase 0)   4  .  .    (phase 1)   5  4  .              3  2
 *             .  .  .                .  .  .
 *
 * high res:  10  9  8    high res:  11 10  9
 * (phase 2)   7  6  5    (phase 3)   8  7  6
 *             4  .  .                5  4  .
 */
behavior jbg_int2dppriv(in unsigned char *dptable,
                        in char *internal)
{
```

```
char *internaltmp;
void main(void)
{
  int i, j, k;
  int trans0[ 8] = { 1, 0, 3, 2, 7, 6, 5, 4 };
  int trans1[ 9] = { 1, 0, 3, 2, 8, 7, 6, 5, 4 };
  int trans2[11] = { 1, 0, 3, 2, 10, 9, 8, 7, 6, 5, 4 };
  int trans3[12] = { 1, 0, 3, 2, 11, 10, 9, 8, 7, 6, 5, 4 };

  internaltmp = internal;
  for (i = 0; i < 1728; dptable[i++] = 0);

#define FILL_TABLE1(offset, len, trans) \
  for (i = 0; i < len; i++) { \
    k = 0; \
    for (j = 0; j < 8; j++) \
      k |= ((i >> j) & 1) << trans[j]; \
    dptable[(i + offset) >> 2] |= \
      (internaltmp[k + offset] & 3) << ((3 - (i&3)) << 1); \
  }

  FILL_TABLE1(   0,  256, trans0);
  FILL_TABLE1( 256,  512, trans1);
  FILL_TABLE1( 768, 2048, trans2);
  FILL_TABLE1(2816, 4096, trans3);

  return;
}
};
```

A.18 encode_sde.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "sde_init";
import "sde_encode_lowest";
import "sde_encode_diff";
import "SDE_flush";



behavior encode_sde(in struct jbg_enc_state *s,
                    in unsigned long stripe,
                    in int layer,
                    in int plane)
{
  struct local_data *ld;

  sde_encode_lowest sde_encode_lowest_exec(ld, s, stripe, layer, plane);
  sde_init sde_init_exec(ld, s, stripe, layer, plane);
  sde_encode_diff sde_encode_diff_exec(ld, s, stripe, layer, plane);
  SDE_flush SDE_flush_exec(ld, s, stripe, layer, plane);


 /*
#ifdef DEBUG
  static long tp_lines, tp_exceptions, tp_pixels, dp_pixels;
  static long encoded_pixels;
 #endif

 #ifdef DEBUG
   if (stripe == 0)
     tp_lines = tp_exceptions = tp_pixels = dp_pixels = encoded_pixels = 0;
   fprintf(stderr, "encode_sde: s/d/p = %2ld/%2d/%2d\n",
           stripe, layer, plane);
 #endif
 */

   void main(void)
   {
   ld = (struct local_data*) malloc(sizeof(struct local_data));

   sde_init_exec.main();

   if (layer == 0) {
     sde_encode_lowest_exec.main();

    } else {

    sde_encode_diff_exec.main();
   }


 SDE_flush_exec.main();

#if 0
   if (stripe == s->stripes - 1)
     fprintf(stderr, "tp_lines␣=␣%ld,␣tp_exceptions␣=␣%ld,␣tp_pixels␣=␣%ld,␣"
             "dp_pixels␣=␣%ld,␣encoded_pixels␣=␣%ld\n",
             tp_lines, tp_exceptions, tp_pixels, dp_pixels, encoded_pixels);
#endif

   return;
 }

};
```

A.19 output_sde.sc

```
#include "constant.sh"
#include <assert.h>

import "jbig_head";
import "jbig";
import "encode_sde";
import "resolution_reduction";

/*
 * This function is called inside the three loops of jbg_enc_out() in
 * order to write the next SDE. It has first to generate the required
 * SDE and all SDEs which have to be encoded before this SDE can be
 * created. The problem here is that if we want to output a lower
 * resolution layer, we have to allpy the resolution reduction
 * algorithm in order to get it. As we try to safe as much memory as
 * possible, the resolution reduction will overwrite previous higher
 * resolution bitmaps. Consequently, we have to encode and buffer SDEs
 * which depend on higher resolution layers before we can start the
 * resolution reduction. All this logic about which SDE has to be
 * encoded before resolution reduction is allowed is handled here.
 * This approach might be a little bit more complex than alternative
 * ways to do it, but it allows us to do the encoding with the minimal
```

```
   * possible amount of temporary memory.
   */
behavior output_sde(in struct jbg_enc_state *s,
                     in unsigned long stripe,
                     in int layer,
                     in int plane)
 {
   int lfcl;       /* lowest fully coded layer */
   int lfcl_1;
   long i;
   unsigned long u;


 encode_sde encode_sde_exec(s, u, lfcl_1, plane);
 resolution_reduction resolution_reduction_exec(s, plane, lfcl_1);

 void main(void){

   assert(s->sde[stripe][layer][plane] != SDE_DONE);

   if (s->sde[stripe][layer][plane] != SDE_TODO) {
 #ifdef DEBUG
     fprintf(stderr, "writing␣SDE:␣s/d/p␣=␣%2lu/%2d/%2d\n",
             stripe, layer, plane);
 #endif
     jbg_buf_output(&s->sde[stripe][layer][plane], s->data_out, s->file);
     s->sde[stripe][layer][plane] = SDE_DONE;
     return;
   }

   /* Determine the smallest resolution layer in this plane for which
    * not yet all stripes have been encoded into SDEs. This layer will
    * have to be completely coded, before we can apply the next
    * resolution reduction step. */
   lfcl = 0;
   for (i = s->d; i >= 0; i--)
     if (s->sde[s->stripes - 1][i][plane] == SDE_TODO) {
       lfcl = i + 1;
       break;
     }
   if (lfcl > s->d && s->d > 0 && stripe == 0) {
     /* perform the first resolution reduction */
     lfcl_1 = s->d;
     resolution_reduction_exec.main();
   }
   /* In case HITOLO is not used, we have to encode and store the higher
    * resolution layers first, although we do not need them right now. */
   while (lfcl - 1 > layer) {
     for (u = 0; u < s->stripes; u++)
     {
       lfcl_1 = lfcl - 1;
       encode_sde_exec.main();
     }
     --lfcl;
     s->highres[plane] ^= 1;
     if (lfcl > 1)
     {
       lfcl_1 = lfcl - 1;
       resolution_reduction_exec.main();
     }
  }

/* added to satisfy the input conditions */
  u = stripe;
  lfcl_1 = layer;
  encode_sde_exec.main();
#ifdef DEBUG
    fprintf(stderr, "writing␣SDE:␣s/d/p␣=␣%2lu/%2d/%2d\n", stripe, layer, plane);
#endif
     jbg_buf_output(&s->sde[stripe][layer][plane], s->data_out, s->file);
     s->sde[stripe][layer][plane] = SDE_DONE;

     if (stripe == s->stripes - 1 && layer > 0 &&
         s->sde[0][layer-1][plane] == SDE_TODO) {
       s->highres[plane] ^= 1;
       if (layer > 1)
       {
         lfcl_1 = layer - 1;
         resolution_reduction_exec.main();
       }
     }


}
};
```

A.20 sde_encode_diff.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "determine_ATMOVE";
import "sde_diff_el_tp";
import "sde_diff_encode_line";


behavior sde_encode_diff(in struct local_data *ld,
                         in struct jbg_enc_state *s,
                         in unsigned long stripe,
                         in int layer,
                         in int plane)
{
int flag2;

determine_ATMOVE determine_ATMOVE_exec(ld, s, stripe, layer, plane, flag2);
sde_diff_el_tp sde_diff_el_tp_exec(ld, s, stripe, layer, plane);
sde_diff_encode_line sde_diff_encode_line_exec(ld, s, stripe, layer, plane);
 /*
    * Encode differential layer
    */

   void main(void)
   {

     flag2=1;
     for (ld->i = 0; ld->i < ld->hl && ld->y < ld->hy; ld->i++, ld->y++) {

     determine_ATMOVE_exec.main();
     sde_diff_el_tp_exec.main();
     sde_diff_encode_line_exec.main();


     } /* for (i = ...) */
   }
};
```

A.21 sde_encode_lowest.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "determine_ATMOVE";
import "sde_lowest_encode_line";
import "sde_lowest_tp";

behavior sde_encode_lowest ( in struct local_data *ld,
                             in struct jbg_enc_state *s,
                             in unsigned long stripe,
                             in int layer,
                             in int plane)
{
int flag;
int flag2;

determine_ATMOVE determine_ATMOVE_exec(ld, s, stripe,layer, plane,flag2);
sde_lowest_encode_line sde_lowest_encode_line_exec(ld, s, stripe,layer, plane);
sde_lowest_tp sde_lowest_tp_exec(ld, s, stripe,layer, plane,flag);

void main(void)
{
      unsigned long i;

      /*
       *  Encode lowest resolution layer
       */

     flag2=0;
     for (i = 0; i < ld->hl && ld->y < ld->hy; i++, ld->y++) {


       ld->i=i;
       determine_ATMOVE_exec.main();
       sde_lowest_tp_exec.main();
       if(flag){
         sde_lowest_encode_line_exec.main();
       }
     } /* for (i = ...) */
}
};
```

A.22 sde_diff_encode_line.sc

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";
import "arith_encode";
import "deterministic_prediction";
import "adaptive_template";
import "model_templates";
```

```
behavior sde_diff_encode_line(in struct local_data *ld,
                              in struct jbg_enc_state *s,
                              in unsigned long stripe,
                              in int layer,
                              in int plane)
{  int int1, int2, flag, options, at_determined, count, cx, tx;
   unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, * c_all, hx;
   unsigned *t, mx;
 struct jbg_arenc_state *par1;


deterministic_prediction  deterministic_prediction_exec(options, y, j, l1,l2,l3,
                          h1, h2, h3, flag);
adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx,
 count,flag);

arith_encode arith_encode_exec(par1, int1, int2);
model_templates model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

void main(void){

  /*
       * Layout of the variables line_h1, line_h2, line_h3, which contain
       * as bits the high resolution neighbour pixels of the currently coded
       * highres pixel X:
       *
       *             76543210 76543210 76543210 76543210     line_h3
       *             76543210 76543210 76543210 76543210     line_h2
       *    76543210 76543210 7654321X 76543210              line_h1
       *
       * Layout of the variables line_l1, line_l2, line_l3, which contain
       * the low resolution pixels near the currently coded pixel as bits.
       * The lowres pixel in which the currently coded highres pixel is
       * located is marked as Y:
       *
       *             76543210 76543210 76543210 76543210     line_l3
       *             76543210 7654321Y 76543210 76543210     line_l2
       *             76543210 76543210 76543210 76543210     line_l1
       */


       ld->line_h1 = ld->line_h2 = ld->line_h3 = ld->line_l1 = ld->line_l2 = ld->line_l3 = 0;
       if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
       if (ld->y > 1) {
         ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;
         ld->line_l3 = (long)*(ld->lp2 - ld->lbpl) << 8;
       }
       ld->line_l2 = (long)*ld->lp2 << 8;
       ld->line_l1 = (long)*ld->lp1 << 8;

       /* encode line */
       for (ld->j = 0; ld->j < ld->hx; ld->lp1++, ld->lp2++) {
         if ((ld->j >> 1) < ld->lbpl * 8 - 8) {
           if (ld->y > 1)
             ld->line_l3 |= *(ld->lp2 - ld->lbpl + 1);
           ld->line_l2 |= *(ld->lp2 + 1);
           ld->line_l1 |= *(ld->lp1 + 1);
         }
         do {
 /*
           assert(ld->hp - (s->lhp[s->highres[plane]][plane] +
                    (stripe * ld->hl + ld->i) * ld->hbpl)
                 == (ptrdiff_t) ld->j >> 3);
```

```
assert(ld->lp2 - (s->lhp[1-s->highres[plane]][plane] +
              (stripe * ld->ll + (ld->i>>1)) * ld->lbpl)
       == (ptrdiff_t) ld->j >> 4);
*/
ld->line_h1 |= *(ld->hp++);
if (ld->j < ld->hbpl * 8 - 8) {
  if (ld->y > 0) {
    ld->line_h2 |= *(ld->hp - ld->hbpl);
    if (ld->y > 1)
      ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl);
  }
}
do {
  ld->line_l1 <<= 1;  ld->line_l2 <<= 1;  ld->line_l3 <<= 1;
  if (ld->ltp && s->tp[ld->j >> 1] < 2) {
    /* pixel are typical and have not to be encoded */
    ld->line_h1 <<= 2;  ld->line_h2 <<= 2;  ld->line_h3 <<= 2;

    /*#ifdef DEBUG
    do {
      ++tp_pixels;
    } while (++(ld->j) & 1 && (ld->j) < hx);
    # else */
    (ld->j) += 2;
    /*              # endif */

  } else
    do {

      ld->line_h1 <<= 1;  ld->line_h2 <<= 1;  ld->line_h3 <<= 1;

      options=s->options;
      y=ld->y;
      j=ld->j;
      l1=ld->line_l1 ;
      l2=ld->line_l2 ;
      l3=ld->line_l3 ;
      h1=ld->line_h1 ;
      h2=ld->line_h2 ;
      h3=ld->line_h3 ;

      deterministic_prediction_exec.main();





      if (flag==1){
        continue;
      }
      else{

        y=ld->y;
        j=ld->j;
        l1=ld->line_l1 ;
        l2=ld->line_l2 ;
        l3=ld->line_l3 ;
        h1=ld->line_h1 ;
        h2=ld->line_h2 ;
        h3=ld->line_h3 ;
        tx=s->tx[plane];
        flag=5;

        model_templates_exec.main();
        ld->cx=cx;
```

```
                    par1=ld->se;
                    int1=ld->cx;
                    int2=(ld->line_h1 >> 8) & 1;

                    arith_encode_exec.main();
                  /*#ifdef DEBUG
                  encoded_pixels++;
                  #endif*/

                  /* diff_adaptive_template (ld, s);*/
                   at_determined=ld->at_determined;
                   j=ld->j;
                   h1=ld->line_h1;
                   h2=ld->line_h2;
                   t=&(ld->t);
                   c=ld->c;
                   c_all=&(ld->c_all);
                   mx=s->mx;
                   hx=ld->lx;
                   count=3;
                   flag=1;
                   adaptive_template_exec.main();

                  }






                } while (++(ld->j) & 1 && (ld->j) < ld->hx);
            } while ((ld->j) & 7 && (ld->j) < ld->hx);
          } while ((ld->j) & 15 && (ld->j) < ld->hx);
        } /* for (j = ...) */

        /* low resolution pixels are used twice */
        if (((ld->i) & 1) == 0) {
          ld->lp1 -= ld->lbpl;
          ld->lp2 -= ld->lbpl;
        }
}
};
```

A.23 sde_lowest_encode_line.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "sde_lowest_encode_pixel";

behavior sde_lowest_encode_line(in struct local_data *ld,
                                in struct jbg_enc_state *s,
                                in unsigned long stripe,
                                in int layer,
                                in int plane)
{

int flag_mt1, flag_mt2, flag_at;

sde_lowest_encode_pixel sde_lowest_encode_pixel_exec(ld, s, stripe, layer, plane, flag_mt1, flag_mt2, flag_

void main(void){
  /*
       * Layout of the variables line_h1, line_h2, line_h3, which contain
       * as bits the neighbour pixels of the currently coded pixel X:
       *
       *          76543210765432107654321076543210      line_h3
       *          76543210765432107654321076543210      line_h2
       *  76543210765432107654321X76543210              line_h1
       */

      ld->line_h1 = ld->line_h2 = ld->line_h3 = 0;
      if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
      if (ld->y > 1) ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;


      /* encode line */
      for (ld->j = 0; ld->j < ld->hx; ld->hp++) {
        ld->line_h1 |= *(ld->hp);
        if (ld->j < ld->hbpl * 8 - 8 && ld->y > 0) {
          ld->line_h2 |= *(ld->hp - ld->hbpl + 1);
          if (ld->y > 1)
            ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl + 1);
        }


       if (s->options & JBG_LRLTWO) {
          /* two_line_template */
                flag_mt1=1;
                flag_mt2=2;
                flag_at=5;
            sde_lowest_encode_pixel_exec.main();
        } else {
          /* three_line_template */
                flag_mt1=3;
                flag_mt2=4;
                flag_at=3;
            sde_lowest_encode_pixel_exec.main();
        } /* if (s->options & JBG_LRLTWO) */
      } /* for (j = ...) */
}
 };
```

A.24 sde_diff_el_tp.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "arith_encode";

behavior sde_diff_el_tp(in struct local_data *ld,
                        in struct jbg_enc_state *s,
                        in unsigned long stripe,
                        in int layer,
                        in int plane)
{int int1, int2;
 struct jbg_arenc_state *par1;

 arith_encode arith_encode_exec(par1, int1, int2);

void main(void){
  /* typical prediction */
```

```
    if (s->options & JBG_TPDON && (ld->i & 1) == 0) {
      ld->q1 = ld->lp1; ld->q2 = ld->lp2;
      ld->p0 = ld->p1 = ld->hp;
      if (ld->i < ld->hl - 1 && ld->y < ld->hy - 1)
        ld->p0 = ld->hp + ld->hbpl;
      if (ld->y > 1)
        ld->line_l3 = ( long)*(ld->q2 - ld->lbpl) << 8;
      else
        ld->line_l3 = 0;
      ld->line_l2 = (long)*(ld->q2) << 8;
      ld->line_l1 = (long)*(ld->q1) << 8;
      ld->ltp = 1;
      for (ld->j = 0; ld->j < ld->lx && ld->ltp; ld->q1++, ld->q2++) {
        if (ld->j < ld->lbpl * 8 - 8) {
          if (ld->y > 1)
            ld->line_l3 |= *(ld->q2 - ld->lbpl + 1);
          ld->line_l2 |= *(ld->q2 + 1);
          ld->line_l1 |= *(ld->q1 + 1);
        }
        do {
          if ((ld->j >> 2) < ld->hbpl) {
            ld->line_h1 = *(ld->p1++);
            ld->line_h0 = *(ld->p0++);
          }
          do {
            ld->line_l3 <<= 1;
            ld->line_l2 <<= 1;
            ld->line_l1 <<= 1;
            ld->line_h1 <<= 2;
            ld->line_h0 <<= 2;
            ld->cx = (((ld->line_l3 >> 15) & 0x007) |
                  ((ld->line_l2 >> 12) & 0x038) |
                  ((ld->line_l1 >> 9)  & 0x1c0));
            if (ld->cx == 0x000)
              if ((ld->line_h1 & 0x300) == 0 && (ld->line_h0 & 0x300) == 0)
                s->tp[ld->j] = 0;
              else {
                ld->ltp = 0;
                /*#ifdef DEBUG
                tp_exceptions++;
#endif*/
              }
            else if (ld->cx == 0x1ff)
              if ((ld->line_h1 & 0x300) == 0x300 && (ld->line_h0 & 0x300) == 0x300)
                s->tp[ld->j] = 1;
              else {
                ld->ltp = 0;
               /*#ifdef DEBUG
                tp_exceptions++;
#endif*/
              }
            else
              s->tp[ld->j] = 2;
          } while (++(ld->j) & 3 && (ld->j) < ld->lx);
        } while ((ld->j) & 7 && (ld->j) < ld->lx);
      } /* for (j = ...) */
      par1=ld->se;
      int1=TPDCX;
      int2=!ld->ltp;
      arith_encode_exec.main();

    /*#ifdef DEBUG
    (tp_lines) += ld->ltp;
#endif*/
   }
```

```
}
};
```

A.25 sde_lowest_tp.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "arith_encode";

/* self-defined functions */

behavior sde_lowest_tp(in struct local_data *ld,
                       in struct jbg_enc_state *s,
                       in unsigned long stripe,
                       in int layer,
                       in int plane,
                       out int flag)
{int int1, int2;
 struct jbg_arenc_state *par1;

 arith_encode arith_encode_exec(par1, int1, int2);


void main(void){
     flag=1;

  /* typical prediction */
      if (s->options & JBG_TPBON) {
        ld->ltp = 1;
        ld->p1 = ld->hp;
        if (ld->y > 0) {
          ld->q1 = ld->hp - ld->hbpl;
          while (ld->q1 < ld->hp && (ld->ltp = (*(ld->p1)++ == *(ld->q1)++)) != 0);
        } else
          while (ld->p1 < ld->hp + ld->hbpl && (ld->ltp = (*(ld->p1)++ == 0)) != 0);
        par1=ld->se;
        int1=((s->options & JBG_LRLTWO) ? TPB2CX : TPB3CX);
        int2=(ld->ltp == ld->ltp_old);
        arith_encode_exec.main();

        /*
#ifdef DEBUG
        tp_lines += ld->ltp;
#endif
*/
        ld->ltp_old = ld->ltp;
        if (ld->ltp) {
          /* skip next line */
          ld->hp += ld->hbpl;
          flag=0;
        }
      }

}
};
```

A.26 sde_lowest_encode_pixel.sc

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";
import "arith_encode";
import "adaptive_template";
import "model_templates";

behavior sde_lowest_encode_pixel(in struct local_data *ld,
                                 in struct jbg_enc_state *s,
                                 in unsigned long stripe,
                                 in int layer,
                                 in int plane,
                                 in int flag_mt1,
                                 in int flag_mt2,
                                 in int flag_at)
{
 int int1, int2, flag, options, at_determined, count, cx, tx;
    unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
    unsigned *t, mx;
struct jbg_arenc_state *par1;


adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx,
 count,flag);
arith_encode arith_encode_exec(par1, int1, int2);
model_templates model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);





void main(){


do {
            ld->line_h1 <<= 1;   ld->line_h2 <<= 1;   ld->line_h3 <<= 1;
            if (s->tx[plane]){
                 y=ld->y;
                  j=ld->j;
                  l1=ld->line_l1;
                  l2=ld->line_l2;
                  l3=ld->line_l3;
                  h1=ld->line_h1;
                  h2=ld->line_h2;
                  h3=ld->line_h3;
                  tx=s->tx[plane];
                  flag=flag_mt1;
                  model_templates_exec.main();
                  ld->cx=cx;

                  par1=ld->se;
                  int1=ld->cx;
                  int2=(ld->line_h1 >> 8) & 1;
              arith_encode_exec.main();
                /* cx=model_templates(ld->y, ld->j, ld->line_l1, ld->line_l2, ld->line_l3, ld->line_h1, ld-
              arith_encode(ld->se, cx, (ld->line_h1 >> 8) & 1); */
           }
           else{

               y=ld->y;
                  j=ld->j;
                  l1=ld->line_l1;
                  l2=ld->line_l2;
                  l3=ld->line_l3;
                  h1=ld->line_h1;
                  h2=ld->line_h2;
                  h3=ld->line_h3;
                  tx=s->tx[plane];
                  flag=flag_mt2;

              model_templates_exec.main();

              ld->cx=cx;

              par1=ld->se;
              int1=ld->cx;
              int2=(ld->line_h1 >> 8) & 1;
              arith_encode_exec.main();
         /*
                cx=model_templates_c(ld->y, ld->j, ld->line_l1, ld->line_l2, ld->line_l3, ld->line_h1, ld->

              arith_encode_c(ld->se, cx, (ld->line_h1 >> 8) & 1); */
            }
            /*
#ifdef DEBUG
            encoded_pixels++;
#endif
*/
            /* lowest_adaptive_template(ld, s->mx, 5); */
                at_determined=ld->at_determined;
                  j=ld->j;
                  h1=ld->line_h1;
                  h2=ld->line_h2;
                  t=&(ld->t);
                  c=ld->c;
                  c_all=&(ld->c_all);
                  mx=s->mx;
                  hx=ld->lx;
                  count=flag_at;
                  flag=0;
                  adaptive_template_exec.main();

          } while (++(ld->j) & 7 && (ld->j) < ld->hx);
        }
};
```

A.27 sde_init.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";

/*
 * The next functions implement the arithmedic encoder and decoder
 * required for JBIG. The same algorithm is also used in the arithmetic
 * variant of JPEG.
 */

#ifdef DEBUG
static long encoded_pixels = 0;
#endif
```

```
static void arith_encode_init(struct jbg_arenc_state *s, int reuse_st)
{
  int i;

  if (!reuse_st)
    for (i = 0; i < 4096; s->st[i++] = 0);
  s->c = 0;
  s->a = 0x10000L;
  s->sc = 0;
  s->ct = 11;
  s->buffer = -1;     /* empty */

  return;
}



behavior sde_init(in struct local_data *ld,
                  in struct jbg_enc_state *s,
                  in unsigned long stripe,
                  in int layer,
                  in int plane)
{

void main(void)
{
  /* return immediately if this stripe has already been encoded */
  if (s->sde[stripe][layer][plane] != SDE_TODO)
    return;

  ld->line_h0 = 0;
  ld->line_h1 = 0;
  ld->new_tx_line = -1;

  /* number of lines per stripe in highres image */
  ld->hl = s->l0 << layer;
  /* number of lines per stripe in lowres image */
  ld->ll = ld->hl >> 1;
  /* current line number in highres image */
  ld->y = stripe * ld->hl;
  /* number of pixels in highres image */
  ld->hx = jbg_ceil_half(s->xd, s->d - layer);
  ld->hy = jbg_ceil_half(s->yd, s->d - layer);
  /* number of pixels in lowres image */
  ld->lx = jbg_ceil_half(ld->hx, 1);
  ld->ly = jbg_ceil_half(ld->hy, 1);
  /* bytes per line in highres and lowres image */
  ld->hbpl = (ld->hx + 7) / 8;
  ld->lbpl = (ld->lx + 7) / 8;
  /* pointer to first image byte of highres stripe */
  ld->hp = s->lhp[s->highres[plane]][plane] + stripe * ld->hl * ld->hbpl;
  ld->lp2 = s->lhp[1 - s->highres[plane]][plane] + stripe * ld->ll * ld->lbpl;
  ld->lp1 = ld->lp2 + ld->lbpl;



  /* initialize arithmetic encoder */
  ld->se = s->s + plane;
  arith_encode_init(ld->se, stripe != 0);
  s->sde[stripe][layer][plane] = jbg_buf_init(&s->free_list);
  ld->se->byte_out = jbg_buf_write;
  ld->se->file = s->sde[stripe][layer][plane];

  /* initialize adaptive template movement algorithm */
  ld->c_all = 0;
```

```
    for (ld->t = 0; ld->t <= s->mx; ld->t++)
      ld->c[ld->t] = 0;
    if (stripe == 0)
      s->tx[plane] = 0;
    ld->new_tx = -1;
    ld->at_determined = 0;  /* we haven't yet decided the template move */
    if (s->mx == 0)
      ld->at_determined = 1;

    /* initialize typical prediction */
    ld->ltp = 0;
    if (stripe == 0)
      ld->ltp_old = 0;
    else {
      ld->ltp_old = 1;
      ld->p1 = ld->hp - ld->hbpl;
      if (ld->y > 1) {
        ld->q1 =ld->p1 - ld->hbpl;
        while (ld->p1 < ld->hp && (ld->ltp_old = (*(ld->p1)++ == *(ld->q1)++)) != 0);
      } else
        while (ld->p1 < ld->hp && (ld->ltp_old = (*(ld->p1)++ == 0)) != 0);
   }

  }

  };
```

A.28 sde_flush.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";

import "jbig";
import "arith_encode_flush";
import "add_atmove";

behavior SDE_flush(in struct local_data *ld,
                   in struct jbg_enc_state *s,
                   in unsigned long stripe,
                   in int layer,
                   in int plane)
{
struct jbg_arenc_state *local_se;

arith_encode_flush arith_encode_flush_exec(local_se);
 add_atmove add_atmove_exec(ld, s, stripe, layer, plane);

void main(){
  local_se = ld->se;
  arith_encode_flush_exec.main();

  jbg_buf_remove_zeros(s->sde[stripe][layer][plane]);
  jbg_buf_write(MARKER_ESC, s->sde[stripe][layer][plane]);
  jbg_buf_write(MARKER_SDNORM, s->sde[stripe][layer]['plane]);

 /* add ATMOVE*/
  add_atmove_exec.main();

}

};
```

A.29 model_templates.sc

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";



behavior model_templates(in unsigned long y,
                         in unsigned long j,
                         in unsigned long l1,
                         in unsigned long l2,
                         in unsigned long l3,
                         in unsigned long h1,
                         in unsigned long h2,
                         in unsigned long h3,
                         in int tx,
                         in int flag,
                         out int cx){

  void main(){
    switch(flag){
    case 1: {cx=(((h2 >> 10) & 0x3e0) |    ((h1 >> (4 + tx)) & 0x010) |
                ((h1 >>  9) & 0x00f));
             break;}
    case 2: {cx=(((h2 >> 10) & 0x3f0) | ((h1 >>  9) & 0x00f));
             break;}
    case 3: {cx= (((h3 >>  8) & 0x380) |    ((h2 >> 12) & 0x078) |
                ((h1 >> (6 + tx)) & 0x004) |((h1 >>  9) & 0x003));
             break;}
    case 4: {cx=(((h3 >>  8) & 0x380) |    ((h2 >> 12) & 0x07c) |
                ((h1 >>  9) & 0x003));
             break;}
    case 5: {if (tx)
                    cx = (((h1 >> 9)  & 0x003) |
                          ((h1 >> (4 + tx)) & 0x010) |
                          ((h2 >> 13) & 0x00c) |
                          ((h3 >> 11) & 0x020));
                  else
                    cx = (((h1 >> 9)  & 0x003) |
                          ((h2 >> 13) & 0x01c) |
                          ((h3 >> 11) & 0x020));
                  if ((j) & 1)
                    cx |= (((l2 >> 9)  & 0x0c0) |
                           ((l1 >> 7)  & 0x300)) | (1UL << 10);
                  else
                    cx |= (((l2 >> 10) & 0x0c0) |
                           ((l1 >> 8)  & 0x300));
                  cx |= (y & 1) << 11;
                  break;
             }
    }

  }
};
```

A.30 deterministic_prediction.sc

```
#include "constant.sh"
```

```
#include <assert.h>

import "jbig_head";
import "jbig";

behavior deterministic_prediction(in int options,
                                  in unsigned long y,
                                  in unsigned long j,
                                  in unsigned long l1,
                                  in unsigned long l2,
                                  in unsigned long l3,
                                  in unsigned long h1,
                                  in unsigned long h2,
                                  in unsigned long h3,
                                  out int result){

void main(void){

        result=0;

                /* deterministic prediction */
                if (options & JBG_DPON)
                  if ((y & 1) == 0) {
                    if (((j) & 1) == 0) {
                      /* phase 0 */
                      if (jbg_dptable[((l3 >> 16) & 0x003) |
                                      ((l2 >> 14) & 0x00c) |
                                      ((h1 >> 5)  & 0x010) |
                                      ((h2 >> 10) & 0x0e0)] < 2) {
                        /*#ifdef DEBUG
                        ++dp_pixels;
#endif*/
                        result=1;
                      }
                    } else {
                      /* phase 1 */
                      if (jbg_dptable[(((l3 >> 16) & 0x003) |
                                       ((l2 >> 14) & 0x00c) |
                                       ((h1 >> 5)  & 0x030) |
                                       ((h2 >> 10) & 0x1c0)) + 256] < 2) {
                        /*#ifdef DEBUG
                        ++dp_pixels;
#endif*/
                        result=1;
                      }
                    }
                  } else {
                    if (((j) & 1) == 0) {
                      /* phase 2 */
                      if (jbg_dptable[(((l3 >> 16) & 0x003) |
                                       ((l2 >> 14) & 0x00c) |
                                       ((h1 >> 5)  & 0x010) |
                                       ((h2 >> 10) & 0x0e0) |
                                       ((h3 >> 7) & 0x700)) + 768] < 2) {
                        /*#ifdef DEBUG
                        ++dp_pixels;
#endif*/
                        result=1;
                      }
                    } else {
                      /* phase 3 */
                      if (jbg_dptable[(((l3 >> 16) & 0x003) |
                                       ((l2 >> 14) & 0x00c) |
                                       ((h1 >> 5)  & 0x030) |
                                       ((h2 >> 10) & 0x1c0) |
                                       ((h3 >> 7)  & 0xe00)) + 2816] < 2) {
                                        /*#ifdef DEBUG
                                        ++dp_pixels;
#endif*/
                                        result=1;
                                    }
                                }
                            }


}

};
```

A.31 determine_ATMOVE.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";

behavior determine_ATMOVE(in struct local_data *ld,
                          in struct jbg_enc_state *s,
                          in unsigned long stripe,
                          in int layer,
                          in int plane,
                          in int flag)
{
void main(void)
{
   int ifcond;

  /* check whether it is worth to perform an ATMOVE*/
      if (!ld->at_determined && ld->c_all > 2048) {
        ld->cmin = ld->clmin = 0xffffffffL;
        ld->cmax = ld->clmax = 0;
        ld->tmax = 0;
        if(flag==1){
          ifcond=3;
        }
        else{
          ifcond=(s->options & JBG_LRLTWO) ? 5 : 3;
        }
        for (ld->t =ifcond ; ld->t <= s->mx; ld->t++) {
          if (ld->c[ld->t] > ld->cmax) ld->cmax = ld->c[ld->t];
          if (ld->c[ld->t] < ld->cmin) ld->cmin = ld->c[ld->t];
          if (ld->c[ld->t] > ld->c[ld->tmax]) ld->tmax = ld->t;
        }
        ld->clmin = (ld->c[0] < ld->cmin) ? ld->c[0] :ld->cmin;
        ld->clmax = (ld->c[0] > ld->cmax) ? ld->c[0] : ld->cmax;
        if (ld->c_all - ld->cmax < (ld->c_all >> 3) &&
            ld->cmax - ld->c[s->tx[plane]] > ld->c_all - ld->cmax &&
            ld->cmax - ld->c[s->tx[plane]] > (ld->c_all >> 4) &&
            /*                    ^ T.82 says here < !!! Typo ? */
            ld->cmax - (ld->c_all - ld->c[s->tx[plane]]) > ld->c_all - ld->cmax &&
            ld->cmax - (ld->c_all - ld->c[s->tx[plane]]) > (ld->c_all >> 4) &&
            ld->cmax - ld->cmin > (ld->c_all >> 2) &&
            (s->tx[plane] || ld->clmax - ld->clmin > (ld->c_all >> 3))) {
          /* we have decided to perform an ATMOVE*/
          ld->new_tx = ld->tmax;
          if (!(s->options & JBG_DELAY_AT)) {
            ld->new_tx_line = ld->i;
            s->tx[plane] = ld->new_tx;
          }

        }
        ld->at_determined = 1;
      }
      if(flag==1){
      if ((ld->i >> 1) >= ld->ll - 1 || (ld->y >> 1) >= ld->ly - 1)
        ld->lp1 = ld->lp2;
      }
}
};
```

A.32 add_atmove.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";

import "jbig";

behavior add_atmove(in struct local_data *ld,
                    in struct jbg_enc_state *s,
                    in unsigned long stripe,
                    in int layer,
                    in int plane)
{
void main(void)
{
  if (ld->new_tx != -1) {
    if (s->options & JBG_DELAY_AT) {
      /* ATMOVE will become active at the first line of the next stripe */
      s->tx[plane] = ld->new_tx;
      jbg_buf_write(MARKER_ESC, s->sde[stripe][layer][plane]);
      jbg_buf_write(MARKER_ATMOVE, s->sde[stripe][layer][plane]);
      jbg_buf_write(0, s->sde[stripe][layer][plane]);
      jbg_buf_write(0, s->sde[stripe][layer][plane]);
      jbg_buf_write(0, s->sde[stripe][layer][plane]);
      jbg_buf_write(0, s->sde[stripe][layer][plane]);
      jbg_buf_write(s->tx[plane], s->sde[stripe][layer][plane]);
      jbg_buf_write(0, s->sde[stripe][layer][plane]);
    } else {
      /* ATMOVE has already become active during this stripe
      * => we have to prefix the SDE data with an ATMOVE marker */
      ld->new_jbg_buf = jbg_buf_init(&s->free_list);
      jbg_buf_write(MARKER_ESC, ld->new_jbg_buf);
      jbg_buf_write(MARKER_ATMOVE, ld->new_jbg_buf);
      jbg_buf_write((ld->new_tx_line >> 24) & 0xff, ld->new_jbg_buf);
      jbg_buf_write((ld->new_tx_line >> 16) & 0xff, ld->new_jbg_buf);
      jbg_buf_write((ld->new_tx_line >> 8) & 0xff, ld->new_jbg_buf);
      jbg_buf_write(ld->new_tx_line & 0xff, ld->new_jbg_buf);
      jbg_buf_write(ld->new_tx, ld->new_jbg_buf);
      jbg_buf_write(0, ld->new_jbg_buf);
      jbg_buf_prefix(ld->new_jbg_buf, &s->sde[stripe][layer][plane]);
    }
  }
}
};
```

A.33 adaptive_template.sc

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";

behavior adaptive_template(in int at_determined,
                           in unsigned long j,
                           in unsigned long h1,
                           in unsigned long h2,
                           inout unsigned *t,
                           inout unsigned long *c,
                           inout unsigned long *c_all ,
                           in unsigned mx,
                           in unsigned long hx,
                           in int count,
                           in int flag){

void main(){
/* statistics for adaptive template changes */
                if (!at_determined && j >= mx && (flag || (j < hx-2))) {
                  c[0] += !(((h2 >> 6) ^ h1) & 0x100);
                  for (*t = count; *t <= mx; *t++)
                    c[*t] += !(((h1 >> *t) ^ h1) & 0x100);
                  ++*c_all;
                }

}
};
```

A.34 arith_encode.sc

```
#include "constant.sh"


#include <assert.h>

import "jbig_head";
import "jbig";


behavior arith_encode(in struct jbg_arenc_state *s,
                      in int cx,
                      in int pix)
{/*
   short jbg_lsz[];
   unsigned char jbg_nmps[], jbg_nlps[];*/
void main(void){
  register unsigned lsz, ss;
  register unsigned char *st;
  long temp;

#ifdef DEBUG
  ++encoded_pixels;
#endif

  assert(cx >= 0 && cx < 4096);
  st = s->st + cx;
  ss = *st & 0x7f;
  assert(ss < 113);
  lsz = jbg_lsz[ss];

#if 0
```

```
  fprintf(stderr, "pix␣=␣%d,␣cx␣=␣%d,␣mps␣=␣%d,␣st␣=␣%3d,␣lsz␣=␣0x%04x,␣"
          "a␣=␣0x%05lx,␣c␣=␣0x%08lx,␣ct␣=␣%2d,␣buf␣=␣0x%02x\n",
          pix, cx, !!(s->st[cx] & 0x80), ss, lsz, s->a, s->c, s->ct,
          s->buffer);
#endif

  if (((pix << 7) ^ *st) & 0x80) {
    /* encode the less probable symbol */
    if ((s->a -= lsz) >= lsz) {
      /* If the interval size (lsz) for the less probable symbol (LPS)
       * is larger than the interval size for the MPS, then exchange
       * the two symbols for coding efficiency, otherwise code the LPS
       * as usual: */
      s->c += s->a;
      s->a = lsz;
    }
    /* Check whether MPS/LPS exchange is necessary
     * and chose next probability estimator status */
    *st &= 0x80;
    *st ^= jbg_nlps[ss];
  } else {
    /* encode the more probable symbol */
    if ((s->a -= lsz) & 0xffff8000L)
      return;   /* A >= 0x8000 -> ready, no renormalization required */
    if (s->a < lsz) {
      /* If the interval size (lsz) for the less probable symbol (LPS)
       * is larger than the interval size for the MPS, then exchange
       * the two symbols for coding efficiency: */
      s->c += s->a;
      s->a = lsz;
    }
    /* chose next probability estimator status */
    *st &= 0x80;
    *st |= jbg_nmps[ss];
  }

  /* renormalization of coding interval */
  do {
    s->a <<= 1;
    s->c <<= 1;
    --s->ct;
    if (s->ct == 0) {
      /* another byte is ready for output */
      temp = s->c >> 19;
      if (temp & 0xffffff00L) {
        /* handle overflow over all buffered 0xff bytes */
        if (s->buffer >= 0) {
          ++s->buffer;
          (*s->byte_out)(s->buffer, s->file);
          if (s->buffer == MARKER_ESC)
            (*s->byte_out)(MARKER_STUFF, s->file);
        }
        for (; s->sc; --s->sc)
          (*s->byte_out)(0x00, s->file);
        s->buffer = temp & 0xff;  /* new output byte, might overflow later */
        assert(s->buffer != 0xff);
        /* can s->buffer really never become 0xff here? */
      } else if (temp == 0xff) {
        /* buffer 0xff byte (which might overflow later) */
        ++s->sc;
      } else {
        /* output all buffered 0xff bytes, they will not overflow any more */
        if (s->buffer >= 0)
          (*s->byte_out)(s->buffer, s->file);
        for (; s->sc; --s->sc) {
          (*s->byte_out)(0xff, s->file);
          (*s->byte_out)(MARKER_STUFF, s->file);
        }
        s->buffer = temp;   /* buffer new output byte (can still overflow) */
      }
      s->c &= 0x7ffffL;
      s->ct = 8;
    }
  } while (s->a < 0x8000);

}
};
```

A.35 arith_encode_flush.sc

```
#include "constant.sh"

import "jbig_head";

behavior arith_encode_flush(in struct jbg_arenc_state *s)
{
  unsigned long temp;

#ifdef DEBUG
  fprintf(stderr, "␣␣encoded␣pixels␣=␣%ld,␣a␣=␣%05lx,␣c␣=␣%08lx\n",
          encoded_pixels, s->a, s->c);
#endif

void main(void)
{
  /* find the s->c in the coding interval with the largest
   * number of trailing zero bits */
  if ((temp = (s->a - 1 + s->c) & 0xffff0000L) < s->c)
    s->c = temp + 0x8000;
  else
    s->c = temp;
  /* send remaining bytes to output */
  s->c <<= s->ct;
  if (s->c & 0xf8000000L) {
    /* one final overflow has to be handled */
    if (s->buffer >= 0) {
      (*s->byte_out)(s->buffer + 1, s->file);
      if (s->buffer + 1 == MARKER_ESC)
        (*s->byte_out)(MARKER_STUFF, s->file);
    }
    /* output 0x00 bytes only when more non-0x00 will follow */
    if (s->c & 0x7fff800L)
      for (; s->sc; --s->sc)
        (*s->byte_out)(0x00, s->file);
  } else {
    if (s->buffer >= 0)
      (*s->byte_out)(s->buffer, s->file);
    /* T.82 figure 30 says buffer+1 for the above line! Typo? */
    for (; s->sc; --s->sc) {
      (*s->byte_out)(0xff, s->file);
      (*s->byte_out)(MARKER_STUFF, s->file);
    }
  }
  /* output final bytes only if they are not 0x00 */
  if (s->c & 0x7fff800L) {
    (*s->byte_out)((s->c >> 19) & 0xff, s->file);
    if (((s->c >> 19) & 0xff) == MARKER_ESC)
      (*s->byte_out)(MARKER_STUFF, s->file);
    if (s->c & 0x7f800L) {
      (*s->byte_out)((s->c >> 11) & 0xff, s->file);
      if (((s->c >> 11) & 0xff) == MARKER_ESC)
        (*s->byte_out)(MARKER_STUFF, s->file);
    }
  }

  return;
}

};
```

A.36 jbg_enc_free.sc

```
#include "constant.sh"


import "jbig_head";
import "jbig";



behavior jbg_enc_free(in struct jbg_enc_state *s)
{
void main(void)
{
  unsigned long stripe;
  int layer, plane;

#ifdef DEBUG
  fprintf(stderr, "jbg_enc_free(%p)\n", s);
#endif

  /* clear buffers for SDEs */
  if (s->sde) {
    for (stripe = 0; stripe < s->stripes; stripe++) {
      for (layer = 0; layer < s->d + 1; layer++) {
        for (plane = 0; plane < s->planes; plane++)
          if (s->sde[stripe][layer][plane] != SDE_DONE &&
              s->sde[stripe][layer][plane] != SDE_TODO)
            jbg_buf_free(&s->sde[stripe][layer][plane]);
        checked_free(s->sde[stripe][layer]);
      }
      checked_free(s->sde[stripe]);
    }
    checked_free(s->sde);
  }

  /* clear free_list */
  jbg_buf_free(&s->free_list);

  /* clear memory for arithmetic encoder states */
  checked_free(s->s);

  /* clear memory for differential-layer typical prediction buffer */
  checked_free(s->tp);

  /* clear memory for adaptive template pixel offsets */
  checked_free(s->tx);

  /* clear lowres image buffers */
  if (s->lhp[1]) {
    for (plane = 0; plane < s->planes; plane++)
      checked_free(s->lhp[1][plane]);
    checked_free(s->lhp[1]);
```

```
   }


}
};
```

B SpecC Architecture Model for JBIG Encoder

B.1 jbg_enc_out.sc

```
#include "constant.sh"
import "std_include";
import "jbig_head";
import "jbig";
import "sw_component";
import "hw_component";
import "bus";
import "memory";
import "CSync_mem";



behavior jbg_enc_out(inout struct jbg_enc_state *s, in int not_done)
{

Bus Bus_exec;
Mem_Access Mem_Access_exec;

sw_component sw_component_exec(Mem_Access_exec, s, Bus_exec);
hw_component hw_component_exec(Mem_Access_exec, Bus_exec, not_done);

memory memory_exec(Mem_Access_exec, not_done);

void main(){

        par{
                sw_component_exec.main();
                hw_component_exec.main();
                memory_exec.main();
        }
}
};
```

B.2 sw_component.sc

```
#include "constant.sh"
import "std_include";
import "jbig_head";
import "jbig";
import "output_sde";
import "jbg_int2dppriv";
import "bus";

behavior sw_component(PE_part sync_mem,
                      inout struct jbg_enc_state *s,
                      IBus_SW ibus)
{
  long bpl;
  unsigned char bih[20];
  unsigned long xd, yd, y;
  long ii[3], is[3], ie[3];      /* generic variables for the 3 nested loops */
  unsigned long stripe;
  int layer, plane;
  int order;
  unsigned char *dpbuf;
  char *local_dppriv;

  output_sde output_sde_exec(sync_mem, s, stripe, layer, plane, ibus);
  jbg_int2dppriv jbg_int2dppriv_exec(dpbuf, local_dppriv);

void main(void)
{
  dpbuf = (unsigned char *) malloc(sizeof(char) * 1728);
  /* some sanity checks */
  s->order &= JBG_HITOLO | JBG_SEQ | JBG_ILEAVE | JBG_SMID;
  order = s->order & (JBG_SEQ | JBG_ILEAVE | JBG_SMID);
  if (index[order][0] < 0)
    s->order = order = JBG_SMID | JBG_ILEAVE;
  if (s->options & JBG_DPON && s->dppriv != jbg_dptable)
    s->options |= JBG_DPPRIV;
  if (s->mx > MX_MAX)
    s->mx = MX_MAX;
  s->my = 0;
  if (s->mx && s->mx < ((s->options & JBG_LRLTWO) ? 5U : 3U))
    s->mx = 0;
  if (s->d > 255 || s->d < 0 || s->dh > s->d || s->dh < 0 ||
      s->dl < 0 || s->dl > s->dh || s->planes < 0 || s->planes > 255)
    return;

  /* ensure correct zero padding of bitmap at the final byte of each line */
  if (s->xd & 7) {
    bpl = (s->xd + 7) / 8;      /* bytes per line */
    for (plane = 0; plane < s->planes; plane++)
      for (y = 0; y < s->yd; y++)
        s->lhp[0][plane][y * bpl + bpl - 1] &= ~((1 << (8 - (s->xd & 7))) - 1);
  }

  /* calculate number of stripes that will be required */
  s->stripes = ((s->yd >> s->d) +
               ((((1UL << s->d) - 1) & s->xd) != 0) + s->l0 - 1) / s->l0;

  /* allocate buffers for SDE pointers */
  if (s->sde == NULL) {
    s->sde = (struct jbg_buf ****)
      checked_malloc(s->stripes * sizeof(struct jbg_buf ***));
    for (stripe = 0; stripe < s->stripes; stripe++) {
      s->sde[stripe] = (struct jbg_buf ***)
        checked_malloc((s->d + 1) * sizeof(struct jbg_buf **));
      for (layer = 0; layer < s->d + 1; layer++) {
        s->sde[stripe][layer] = (struct jbg_buf **)
          checked_malloc(s->planes * sizeof(struct jbg_buf *));
        for (plane = 0; plane < s->planes; plane++)
          s->sde[stripe][layer][plane] = SDE_TODO;
      }
    }
  }

  /* output BIH */
  bih[0] = s->dl;
  bih[1] = s->dh;
  bih[2] = s->planes;
  bih[3] = 0;
  xd = jbg_ceil_half(s->xd, s->d - s->dh);
  yd = jbg_ceil_half(s->yd, s->d - s->dh);
  bih[4] = xd >> 24;
  bih[5] = (xd >> 16) & 0xff;
  bih[6] = (xd >> 8) & 0xff;
  bih[7] = xd & 0xff;
  bih[8] = yd >> 24;
  bih[9] = (yd >> 16) & 0xff;
  bih[10] = (yd >> 8) & 0xff;
  bih[11] = yd & 0xff;
  bih[12] = s->l0 >> 24;
  bih[13] = (s->l0 >> 16) & 0xff;
  bih[14] = (s->l0 >> 8) & 0xff;
  bih[15] = s->l0 & 0xff;
  bih[16] = s->mx;
  bih[17] = s->my;
  bih[18] = s->order;
  bih[19] = s->options & 0x7f;
  (* s->data_out) (bih, 20, s->file);
  if ((s->options & (JBG_DPON | JBG_DPPRIV | JBG_DPLAST)) ==
      (JBG_DPON | JBG_DPPRIV)) {
    /* write private table */
    local_dppriv = s->dppriv;
    jbg_int2dppriv_exec.main();
    (* s->data_out) (dpbuf, 1728, s->file);
  }

  /*
   * Generic loops over all SDEs. Which loop represents layer, plane and
   * stripe depends on the option flags.
   */

  /* start and end value vor each loop */
  is[index[order][STRIPE]] = 0;
  ie[index[order][STRIPE]] = s->stripes - 1;
  is[index[order][LAYER]] = s->dl;
  ie[index[order][LAYER]] = s->dh;
  is[index[order][PLANE]] = 0;
  ie[index[order][PLANE]] = s->planes - 1;

  for (ii[0] = is[0]; ii[0] <= ie[0]; ii[0]++)
    for (ii[1] = is[1]; ii[1] <= ie[1]; ii[1]++){
      for (ii[2] = is[2]; ii[2] <= ie[2]; ii[2]++) {
        stripe = ii[index[order][STRIPE]];
        if (s->order & JBG_HITOLO)
          layer = s->dh - (ii[index[order][LAYER]] - s->dl);
        else
          layer = ii[index[order][LAYER]];
        plane = ii[index[order][PLANE]];

        output_sde_exec.main();
      }
 }

  return;
}

};
```

B.3 hw_component.sc

```
#include "constant.sh"
#include <assert.h>

import "std_include";
import "jbig_head";
import "jbig";
import "CSync_mem";
import "bus";

```

```
behavior hw_component(PE_part sync_mem,
                    IBus_HW ibus,
                    in int not_done)
{

void main(void){
  register unsigned lsz, ss;
  register unsigned char st;
  long temp;
  struct jbg_arenc_state *s;
  int cx;
  int pix, plane, i;
  int databuffer[100], bufferindex;
  int addr;

  s = (struct jbg_arenc_state *) malloc(sizeof(struct jbg_arenc_state));

  while(not_done){
        /* communicate with the coldfire */
        plane=ibus.write_receive(ADDR_PLANE);
        cx=ibus.write_receive(ADDR_CX);
        pix=ibus.write_receive(ADDR_PIX);
        ibus.receive_begin();

        bufferindex=0;
        for (i=0; i<100; i++){
                databuffer[i]=0;
        }

        /* communicate with memory, read memory */
        st=(char)sync_mem.read_send(plane,cx);
        s->c=(unsigned long)sync_mem.read_send(plane, ADDR_C);
        s->a=(unsigned long)sync_mem.read_send(plane, ADDR_A);
        s->sc=sync_mem.read_send( plane, ADDR_SC);
        s->ct=(int)sync_mem.read_send(plane, ADDR_CT);
        s->buffer=(int)sync_mem.read_send(plane, ADDR_BUFFER);

  assert(cx >= 0 && cx < 4096);
  /* st = s->st + cx; */
  ss = st & 0x7f;
  assert(ss < 113);
  lsz = jbg_lsz[ss];

  if (((pix << 7) ^ st) & 0x80) {
     /* encode the less probable symbol */
   if ((s->a -= lsz) >= lsz) {
     /* If the interval size (lsz) for the less probable symbol (LPS)
      * is larger than the interval size for the MPS, then exchange
      * the two symbols for coding efficiency, otherwise code the LPS
      * as usual: */
     s->c += s->a;
     s->a = lsz;
   }
   /* Check whether MPS/LPS exchange is necessary
    * and chose next probability estimator status */
   st &= 0x80;
   st ^= jbg_nlps[ss];
 } else {
   /* encode the more probable symbol */
   if ((s->a -= lsz) & 0xffff8000L){
       sync_mem.write_send((long)s->c, plane, ADDR_C);
       sync_mem.write_send((long)s->a, plane, ADDR_A);
       sync_mem.write_send(s->sc, plane, ADDR_SC);
       sync_mem.write_send((long)s->ct, plane, ADDR_CT);
       sync_mem.write_send((long)s->buffer, plane, ADDR_BUFFER);
       sync_mem.write_send((long)st,plane,cx);
      ibus.send_end();
      ibus.read_receive(0, ADDR_COUNT);
      continue; /* A >= 0x8000 -> ready, no renormalization required */
      }
  if (s->a < lsz) {
    /* If the interval size (lsz) for the less probable symbol (LPS)
     * is larger than the interval size for the MPS, then exchange
     * the two symbols for coding efficiency: */
    s->c += s->a;
    s->a = lsz;
  }
  /* chose next probability estimator status */
  st &= 0x80;
  st |= jbg_nmps[ss];
}

/* renormalization of coding interval */
do {
  s->a <<= 1;
  s->c <<= 1;
  --s->ct;
  if (s->ct == 0) {
    /* another byte is ready for output */
    temp = s->c >> 19;
    if (temp & 0xffffff00L) {
      /* handle overflow over all buffered 0xff bytes */
      if (s->buffer >= 0) {
        ++s->buffer;
        /* (*s->byte_out)(s->buffer, s->file); */
        databuffer[bufferindex]=s->buffer;
        bufferindex++;

        if (s->buffer == MARKER_ESC) {
          /* (*s->byte_out)(MARKER_STUFF, s->file); */
          databuffer[bufferindex]=MARKER_STUFF;
          bufferindex++;
          }
      }
      for (; s->sc; --s->sc){
        /* (*s->byte_out)(0x00, s->file); */
        databuffer[bufferindex]=0x00;
        bufferindex++;
      }
      s->buffer = temp & 0xff;  /* new output byte, might overflow later */
      assert(s->buffer != 0xff);
      /* can s->buffer really never become 0xff here? */
    } else if (temp == 0xff) {
      /* buffer 0xff byte (which might overflow later) */
      ++s->sc;
    } else {
      /* output all buffered 0xff bytes, they will not overflow any more */
      if (s->buffer >= 0){
        /* (*s->byte_out)(s->buffer, s->file); */
        databuffer[bufferindex]=s->buffer;
        bufferindex++;
      }
      for (; s->sc; --s->sc) {
        /* (*s->byte_out)(0xff, s->file); */
        databuffer[bufferindex]=0xff;
        bufferindex++;

        /* (*s->byte_out)(MARKER_STUFF, s->file); */
        databuffer[bufferindex]=MARKER_STUFF;
        bufferindex++;
      }
      s->buffer = temp;    /* buffer new output byte (can still overflow) */
```

```
       }
       s->c &= 0x7ffffL;
       s->ct = 8;
     }
   } while (s->a < 0x8000);

  sync_mem.write_send((long)s->c,plane, ADDR_C);
  sync_mem.write_send((long)s->a,plane, ADDR_A);
  sync_mem.write_send(s->sc, plane, ADDR_SC);
  sync_mem.write_send((long)s->ct, plane, ADDR_CT);
  sync_mem.write_send((long)s->buffer, plane, ADDR_BUFFER);
  sync_mem.write_send((long)st,plane,cx);

  ibus.send_end();

         ibus.read_receive(bufferindex, ADDR_COUNT);
         for(i=0; i<bufferindex; i++){
                 ibus.read_receive(databuffer[i], ADDR_FILE_BUFFER);
         }

}
}
};
```

B.4 CSync_SH.sc

```
import "std_include";
import "jbig_head";
import "jbig";
/*----------------------------*/

interface Syn_Int_SW{
        void write_send(int data);
        int read_send();
};

interface Syn_Int_HW{
        int write_receive();
        void read_receive(int data);

};

channel CSyncInt()
        implements Syn_Int_SW, Syn_Int_HW

{
/* SW active HW to do the HW part */
        bool hw_valid =false;
        event sent, received;
        int buffer;

        void write_send (int data){
                /* printf(" send ..."); */
                hw_valid=true;
                buffer=data;
                notify (sent);
                if(hw_valid) wait(received);

        }

        int write_receive (){
           int result;
                /* printf(" receive ..."); */
                if (!hw_valid) wait(sent);
                result=buffer;
                hw_valid=false ;
                notify(received );

                return result ;
        }

        int read_send(){
            int result ;
                hw_valid=true;
                notify(sent);
                if(hw_valid) wait(received );
                result=buffer;
                return result ;
        }


        void read_receive(int data){
                if (!hw_valid) wait(sent);
                hw_valid=false ;
                buffer=data;
                notify(received );
        }

};

/*----handshake---*/
interface Syn_HS_master{
        void receive_end();
        void send_begin();

};

interface Syn_HS_slave{
        void receive_begin();
        void send_end();

};


channel CSyncHS()
        implements Syn_HS_slave, Syn_HS_master

{
/* SW active HW to do the HW part */
        bool hw_valid =false ;
        event sent, received ;

        void send_begin (){
                hw_valid=true;
                notify (sent);
        }

        void receive_begin(){
                if (!hw_valid) wait(sent);
        }

        void send_end (){
                hw_valid=false ;
                notify(received );
        }
        void receive_end (){
                if(hw_valid) wait(received );
        }
```

```
};
```

B.5 CSync_mem.sc

```
#include "constant.sh"

import "std_include";

typedef struct jbg_mem_unit {
  unsigned char st[CHAR_ARRAY];     /* probability status for contexts, MSB = MPS */
  unsigned long c;                 /* 4096 *
                                   * layout as in Table 23                    */
  unsigned long a;        /*4100 */
  long sc;          /* 4104 */
  int ct;   /* 4108 */
  int buffer;                /* 4112 */
}jbg_mem_unit;

interface Mem_part{
        /* common */
        int write_receive_addr();
        int write_receive_data_type();
        int write_receive_rw();

        /* memory write */
        long write_receive_data();

        /* memory read */

        void read_receive_data(long data);


        };

interface PE_part{

        /* memory write */
        void write_send (long data, int adds1, int adds2);

        /* memory read */
        long read_send (int adds, int adds2);
};

channel Mem_Access()
        implements Mem_part, PE_part

{
/* SW active HW to do the HW part */
        bool mem_valid =false;
        event send_adds, recv_adds, send_data_type, recv_data_type, send_data, recv_data;
        int sub_addr;
        int addr;
        int nrw;
        long buffer;



        void write_send (long data, int adds1, int adds2){
                mem_valid=true;
                buffer=data;
                sub_addr=adds1;
                addr=adds2;
                nrw=1;

                notify (send_adds);
                if (mem_valid) wait (recv_adds);

                mem_valid=true;
                notify (send_data);
                if(mem_valid) wait(recv_data);

        }

        long read_send (int adds1, int adds2){
                mem_valid=true;
                sub_addr=adds1;

                addr=adds2;
                nrw=0;

                notify (send_adds);
                if (mem_valid) wait (recv_adds);



                mem_valid=true;
                notify (send_data);
                if(mem_valid) wait(recv_data);

                return buffer;

        }


    /*********commone used********/
        int write_receive_addr(){
                if (!mem_valid)  wait(send_adds);
                return sub_addr;
        }


        int write_receive_data_type(){
                return addr;
        }

        int write_receive_rw(){
                notify (recv_adds);
                mem_valid=false;

                return nrw;

        }


    /***--- receive  long ---***/
        long write_receive_data(){
                if (!mem_valid) wait(send_data);
                notify (recv_data);
                mem_valid=false;
                return buffer;

        }

        void read_receive_data(long data){
                if (!mem_valid) wait(send_data);
                notify (recv_data);
                mem_valid=false;
```

```
                        buffer=data;
                }
};
```

B.6 memory.sc

```
#include "constant.sh"

import "std_include";
import "CSync_mem";


static void mem_unit_init(int reuse, jbg_mem_unit *m)
{
        int i;
        if(!reuse)
        {
          for(i=0; i<CHAR_ARRAY; i++)
                  m->st[i] = 0;
        }
        m->c = 0;
        m->a = 0x10000L;
        m->sc = 0;
        m->ct = 11;
        m->buffer = -1;
}

behavior memory(Mem_part sync, in int not_done)
{

void main(){

        int i;
        int rw;
        int sub_addr, addr; /* sub_addr is for inside the array, addr is for distingush different */
        jbg_mem_unit blocks[6];
        int reuse;

        while(not_done){
                sub_addr=sync.write_receive_addr();
                addr=sync.write_receive_data_type();
                rw=sync.write_receive_rw();
                switch (addr){
                        case UNIT_INIT:
                                reuse=sync.write_receive_data();
                                mem_unit_init(reuse, &blocks[sub_addr]);
                                break;

                        case ADDR_CT: /* write int data value */
                                if(rw==1){
                                        blocks[sub_addr].ct=(int)sync.write_receive_data();
                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].ct);
                                }
                                break;

                        case ADDR_BUFFER: /* write int data value */
                                if(rw==1){
                                        blocks[sub_addr].buffer=(int)sync.write_receive_data();
                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].buffer);
                                }
                                break;

                        case ADDR_C: /* write unsigned long data value */
                                if(rw==1){
                                        blocks[sub_addr].c=sync.write_receive_data();
                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].c);
                                }
                                break;

                        case ADDR_A: /* write unsigned long data value */
                                if(rw==1){
                                        blocks[sub_addr].a=sync.write_receive_data();
                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].a);
                                }
                                break;

                        case ADDR_SC:  /* write long data value */
                                if(rw==1){
                                        blocks[sub_addr].sc=sync.write_receive_data();
                                }else{
                                        sync.read_receive_data(blocks[sub_addr].sc);
                                }
                                break;

                        default:        /* for char addr is address in the table from 0 to 4095 */
                                if(rw==1){
                                         blocks[sub_addr].st[addr]=sync.write_receive_data();
                                }else{
                                        sync.read_receive_data(blocks[sub_addr].st[addr]);
                                }
                                break;
                        }

        }
}
};
```

B.7 bus.sc

```
#include "constant.sh"

import "std_include";
import "jbig_head";
import "jbig";
import "CSync_SH";

interface IBus_SW
{
        void write_send(int data, int addr);
        int read_send(int addr);

        void receive_end();
        void send_begin();
};

interface IBus_HW
{
        int write_receive(int addr);
        void read_receive(int data, int addr);

        void receive_begin();
        void send_end();


};

channel Bus()
        implements IBus_SW, IBus_HW
{

        CSyncInt plane;
        CSyncInt cx;
        CSyncInt pix;
        CSyncInt count;
        CSyncInt file_buffer ;
        CSyncHS handshake;

        void write_send(int data, int addr)
        {
                switch(addr){
                        case ADDR_PLANE:  return plane.write_send(data);
                                          break;
                        case ADDR_CX:     return cx.write_send(data);
                                          break;
                        case ADDR_PIX:    return pix.write_send(data);
                                          break;
                }
        }


        int write_receive(int addr){
                switch(addr){
                        case ADDR_PLANE:  return plane.write_receive();
                                          break;
                        case ADDR_CX:     return cx.write_receive();
                                          break;
                        case ADDR_PIX:    return pix.write_receive();
                                          break;
                }
        }

        int read_send(int addr){
                switch(addr){
                        case ADDR_COUNT:  return count.read_send();
                                          break;
                        case ADDR_FILE_BUFFER:   return file_buffer.read_send();
                                          break;
                }
        }

        void read_receive(int data, int addr){
                switch(addr){
                        case ADDR_COUNT:  return count.read_receive(data);
                                          break;
                        case ADDR_FILE_BUFFER:   return file_buffer.read_receive(data);
                                          break;
                }
        }

        void receive_end(){
                handshake.receive_end();
        }
        void send_begin(){
                handshake.send_begin();
        }

        void receive_begin(){
                handshake.receive_begin();
        }

        void send_end(){
                handshake.send_end();
        }

};
```

B.8 output_sde.sc

```
#include "constant.sh"
#include <assert.h>

import "jbig_head";
import "jbig";
import "encode_sde";
import "resolution_reduction";
import "bus";
import "CSync_mem";


behavior output_sde(PE_part sync_mem,
                    in struct jbg_enc_state *s,
                    in unsigned long stripe,
                    in int layer,
                    in int plane,
                    IBus_SW ibus)
{
  int lfcl;      /* lowest fully coded layer */
  int lfcl_1;
  long i;
  unsigned long u;


  encode_sde encode_sde_exec(sync_mem, s, u, lfcl_1, plane, ibus);
  resolution_reduction resolution_reduction_exec(s, plane, lfcl_1);

void main(void){

  assert(s->sde[stripe][layer][plane] != SDE_DONE);

  if (s->sde[stripe][layer][plane] != SDE_TODO) {

    jbg_buf_output(&s->sde[stripe][layer][plane], s->data_out, s->file);
    s->sde[stripe][layer][plane] = SDE_DONE;
    return;
  }

  /* Determine the smallest resolution layer in this plane for which
   * not yet all stripes have been encoded into SDEs. This layer will
   * have to be completely coded, before we can apply the next
   * resolution reduction step. */
  lfcl = 0;
  for (i = s->d; i >= 0; i--)
    if (s->sde[s->stripes - 1][i][plane] == SDE_TODO) {
      lfcl = i + 1;
      break;
   }
```

```
    if (lfcl > s->d && s->d > 0 && stripe == 0) {
      /* perform the first resolution reduction */
      lfcl_1 = s->d;
      resolution_reduction_exec.main();
    }
    /* In case HITOLO is not used, we have to encode and store the higher
     * resolution layers first, although we do not need them right now. */
    while (lfcl - 1 > layer) {
      for (u = 0; u < s->stripes; u++)
      {
        lfcl_1 = lfcl - 1;
        encode_sde_exec.main();
      }
      --lfcl;
      s->highres[plane] ^= 1;
      if (lfcl > 1)
      {
        lfcl_1 = lfcl - 1;
        resolution_reduction_exec.main();
      }
    }

  /* added to satisfy the input conditions */
    u = stripe;
    lfcl_1 = layer;
    encode_sde_exec.main();

    jbg_buf_output(&s->sde[stripe][layer][plane], s->data_out, s->file);
    s->sde[stripe][layer][plane] = SDE_DONE;

    if (stripe == s->stripes - 1 && layer > 0 &&
        s->sde[0][layer-1][plane] == SDE_TODO) {
      s->highres[plane] ^= 1;
      if (layer > 1)
      {
        lfcl_1 = layer - 1;
        resolution_reduction_exec.main();
      }
    }

  }
};
```

B.9 encode_sde.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "sde_init";
import "sde_encode_lowest";
import "sde_encode_diff";
import "SDE_flush";
import "bus";
import "CSync_mem";


behavior encode_sde(PE_part sync_mem,
                    in struct jbg_enc_state *s,
                    in unsigned long stripe,
                    in int layer,
                    in int plane,
                    IBus_SW ibus)
{
    struct local_data *ld;
    struct jbg_buf *file;

    sde_encode_lowest sde_encode_lowest_exec(ld, s, sync_mem,  stripe, layer, plane, file, ibus);
    sde_init sde_init_exec(ld, s, sync_mem, stripe, layer, plane);
    sde_encode_diff sde_encode_diff_exec(ld, s, sync_mem,  stripe, layer, plane, file, ibus);
    SDE_flush SDE_flush_exec(ld, s, stripe, layer, sync_mem, plane, file);


  void main(void)
  {
    ld = (struct local_data*) malloc(sizeof(struct local_data));

    sde_init_exec.main();
    file = s->sde[stripe][layer][plane];

    if (layer == 0) {
      sde_encode_lowest_exec.main();
     } else {
     sde_encode_diff_exec.main();
   }

    SDE_flush_exec.main();

   return;
  }
  };
```

B.10 sde_init.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "CSync_mem";

/*
 * The next functions implement the arithmedic encoder and decoder
 * required for JBIG. The same algorithm is also used in the arithmetic
 * variant of JPEG.
 */

behavior sde_init(in struct local_data *ld,
                  in struct jbg_enc_state *s,
                  PE_part sync_mem,
                  in unsigned long stripe,
                  in int layer,
                  in int plane)
{
  int reuse;

 void main(void)
 {
  /* return immediately if this stripe has already been encoded */
  if (s->sde[stripe][layer][plane] != SDE_TODO)
    return;

  ld->line_h0 = 0;
  ld->line_h1 = 0;
  ld->new_tx_line = -1;

  /* number of lines per stripe in highres image */
  ld->hl = s->l0 << layer;
  /* number of lines per stripe in lowres image */
  ld->ll = ld->hl >> 1;
  /* current line number in highres image */
  ld->y = stripe * ld->hl;
  /* number of pixels in highres image */
  ld->hx = jbg_ceil_half(s->xd, s->d - layer);
  ld->hy = jbg_ceil_half(s->yd, s->d - layer);
  /* number of pixels in lowres image */
  ld->lx = jbg_ceil_half(ld->hx, 1);
  ld->ly = jbg_ceil_half(ld->hy, 1);
  /* bytes per line in highres and lowres image */
  ld->hbpl = (ld->hx + 7) / 8;
  ld->lbpl = (ld->lx + 7) / 8;
  /* pointer to first image byte of highres stripe */
  ld->hp = s->lhp[s->highres[plane]][plane] + stripe * ld->hl * ld->hbpl;
  ld->lp2 = s->lhp[1 - s->highres[plane]][plane] + stripe * ld->ll * ld->lbpl;
  ld->lp1 = ld->lp2 + ld->lbpl;

  if(stripe != 0)
   reuse = 1;
  else
   reuse = 0;

  sync_mem.write_send((long)reuse, plane, UNIT_INIT);
  s->sde[stripe][layer][plane] = jbg_buf_init(&s->free_list);

  /* initialize adaptive template movement algorithm */
  ld->c_all = 0;
  for (ld->t = 0; ld->t <= s->mx; ld->t++)
    ld->c[ld->t] = 0;
  if (stripe == 0)
    s->tx[plane] = 0;
  ld->new_tx = -1;
  ld->at_determined = 0;  /* we haven't yet decided the template move */
  if (s->mx == 0)
    ld->at_determined = 1;

  /* initialize typical prediction */
  ld->ltp = 0;
  if (stripe == 0)
    ld->ltp_old = 0;
  else {
    ld->ltp_old = 1;
    ld->p1 = ld->hp - ld->hbpl;
    if (ld->y > 1) {
      ld->q1 =ld->p1 - ld->hbpl;
      while (ld->p1 < ld->hp && (ld->ltp_old = (*(ld->p1)++ == *(ld->q1)++)) != 0);
    } else
      while (ld->p1 < ld->hp && (ld->ltp_old = (*(ld->p1)++ == 0)) != 0);
  }
}
};
```

B.11 SDE_flush.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "arith_encode_flush";
import "add_atmove";
import "CSync_mem";

behavior SDE_flush(in struct local_data *ld,
                   in struct jbg_enc_state *s,
                   in unsigned long stripe,
                   in int layer,
                   PE_part sync_mem,
                   in int plane,
                   in struct jbg_buf *file)
{
  arith_encode_flush arith_encode_flush_exec(sync_mem, plane, file);
  add_atmove add_atmove_exec(ld, s, stripe, layer, plane);

void main(){
  arith_encode_flush_exec.main();

  jbg_buf_remove_zeros(s->sde[stripe][layer][plane]);
  jbg_buf_write(MARKER_ESC, s->sde[stripe][layer][plane]);
  jbg_buf_write(MARKER_SDNORM, s->sde[stripe][layer][plane]);

  add_atmove_exec.main();

}

};
```

B.12 sde_encode_lowest.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "determine_ATMOVE";
import "sde_lowest_encode_line";
import "sde_lowest_tp";
import "bus";
import "CSync_mem";


behavior sde_encode_lowest ( in struct local_data *ld,
                             in struct jbg_enc_state *s,
                             PE_part sync_mem,
                             in unsigned long stripe,
                             in int layer,
                             in int plane,
                             in struct jbg_buf *file,
                             IBus_SW ibus)
{
  int flag;
  int flag2;

  determine_ATMOVE determine_ATMOVE_exec(ld, s, stripe, layer, plane, flag2);
  sde_lowest_encode_line sde_lowest_encode_line_exec(ld, s, sync_mem, stripe, layer, plane, file, ibus);
  sde_lowest_tp sde_lowest_tp_exec(sync_mem, ld, s, stripe, layer, plane, flag, file);

void main(void)
{
    unsigned long i;

  flag2=0;
```

```
     for (i = 0; i < ld->hl && ld->y < ld->hy; i++, ld->y++) {
       ld->i=i;
       determine_ATMOVE_exec.main();
       sde_lowest_tp_exec.main();
       if(flag){
         sde_lowest_encode_line_exec.main();
       }
     } /* for (i = ...) */
 }
 };
```

B.13 sde_encode_diff.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "determine_ATMOVE";
import "sde_diff_el_tp";
import "sde_diff_encode_line";
import "bus";
import "CSync_mem";


behavior sde_encode_diff(inout struct local_data *ld,
                         in struct jbg_enc_state *s,
                         PE_part sync_mem,
                         in unsigned long stripe,
                         in int layer,
                         in int plane,
                         in struct jbg_buf *file,
                         IBus_SW ibus)
{
int flag2;

determine_ATMOVE determine_ATMOVE_exec(ld, s, stripe, layer, plane, flag2);
sde_diff_el_tp sde_diff_el_tp_exec(sync_mem, ld, s, stripe, layer, plane, file);
sde_diff_encode_line sde_diff_encode_line_exec(ld, s, sync_mem, stripe, layer, plane, file, ibus);

void main(void)
{
    flag2=1;
    for (ld->i = 0; ld->i < ld->hl && ld->y < ld->hy; ld->i++, ld->y++)
    {
        determine_ATMOVE_exec.main();
        sde_diff_el_tp_exec.main();
        sde_diff_encode_line_exec.main();
    } /* for (i = ...) */
 }
};
```

B.14 sde_lowest_encode_line.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "sde_lowest_encode_pixel";
import "bus";

behavior sde_lowest_encode_line(in struct local_data *ld,
                                        in struct jbg_enc_state *s,
                                        PE_part sync_mem,
                                        in unsigned long stripe,
                                        in int layer,
                                        in int plane,
                                        in struct jbg_buf *file,
                                        IBus_SW ibus)
 {
   int flag_mt1, flag_mt2, flag_at;

   sde_lowest_encode_pixel sde_lowest_encode_pixel_exec(ld, s, sync_mem, stripe, layer, plane, flag_mt1, fla

 void main(void){

       ld->line_h1 = ld->line_h2 = ld->line_h3 = 0;
       if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
       if (ld->y > 1) ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;

       /* encode line */
       for (ld->j = 0; ld->j < ld->hx; ld->hp++) {
         ld->line_h1 |= *(ld->hp);
         if (ld->j < ld->hbpl * 8 - 8 && ld->y > 0) {
           ld->line_h2 |= *(ld->hp - ld->hbpl + 1);
           if (ld->y > 1)
             ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl + 1);
         }

        if (s->options & JBG_LRLTWO) {
           /* two_line_template */
                 flag_mt1=1;
                 flag_mt2=2;
                 flag_at=5;
             sde_lowest_encode_pixel_exec.main();
         } else {
           /* three_line_template */
                 flag_mt1=3;
                 flag_mt2=4;
                 flag_at=3;
             sde_lowest_encode_pixel_exec.main();
         } /* if (s->options & JBG_LRLTWO) */
      } /* for (j = ...) */
}
 };
```

B.15 sde_lowest_tp.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "arith_encode";

/* self-defined functions */

behavior sde_lowest_tp(PE_part sync_mem,
                         in struct local_data *ld,
                        in struct jbg_enc_state *s,
                        in unsigned long stripe,
                        in int layer,
                        in int plane,
                        out int flag,
                        in struct jbg_buf *file)
{
 int int1, int2;
```

```
  arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);

 void main(void){
      flag=1;

   /* typical prediction */
      if (s->options & JBG_TPBON) {
        ld->ltp = 1;
        ld->p1 = ld->hp;
        if (ld->y > 0) {
          ld->q1 = ld->hp - ld->hbpl;
          while (ld->q1 < ld->hp && (ld->ltp = (*(ld->p1)++ == *(ld->q1)++)) != 0);
        } else
          while (ld->p1 < ld->hp + ld->hbpl && (ld->ltp = (*(ld->p1)++ == 0)) != 0);
        int1=((s->options & JBG_LRLTWO) ? TPB2CX : TPB3CX);
        int2=(ld->ltp == ld->ltp_old);
        arith_encode_exec.main();

        ld->ltp_old = ld->ltp;
        if (ld->ltp) {
          /* skip next line */
          ld->hp += ld->hbpl;
          flag=0;
        }
      }

}
 };
```

B.16 sde_diff_el_tp.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "arith_encode";

behavior sde_diff_el_tp(PE_part sync_mem,
                        in struct local_data *ld,
                        in struct jbg_enc_state *s,
                        in unsigned long stripe,
                        in int layer,
                        in int plane,
                        in struct jbg_buf *file)
{int int1, int2;
 struct jbg_arenc_state *par1;

 arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);

void main(void){
  /* typical prediction */
      if (s->options & JBG_TPDON && (ld->i & 1) == 0) {
        ld->q1 = ld->lp1; ld->q2 = ld->lp2;
        ld->p0 = ld->p1 = ld->hp;
        if (ld->i < ld->hl - 1 && ld->y < ld->hy - 1)
          ld->p0 = ld->hp + ld->hbpl;
        if (ld->y > 1)
          ld->line_l3 = ( long)*(ld->q2 - ld->lbpl) << 8;
        else
          ld->line_l3 = 0;
        ld->line_l2 = (long)*(ld->q2) << 8;
        ld->line_l1 = (long)*(ld->q1) << 8;
        ld->ltp = 1;
        for (ld->j = 0; ld->j < ld->lx && ld->ltp ; ld->q1++, ld->q2++) {
          if (ld->j < ld->lbpl * 8 - 8) {
            if (ld->y > 1)
              ld->line_l3 |= *(ld->q2 - ld->lbpl + 1);
            ld->line_l2 |= *(ld->q2 + 1);
            ld->line_l1 |= *(ld->q1 + 1);
          }
          do {
            if ((ld->j >> 2) < ld->hbpl) {
              ld->line_h1 = *(ld->p1++);
              ld->line_h0 = *(ld->p0++);
            }
            do {
              ld->line_l3 <<= 1;
              ld->line_l2 <<= 1;
              ld->line_l1 <<= 1;
              ld->line_h1 <<= 2;
              ld->line_h0 <<= 2;
              ld->cx = (((ld->line_l3 >> 15) & 0x007) |
                        ((ld->line_l2 >> 12) & 0x038) |
                        ((ld->line_l1 >> 9)  & 0x1c0));
              if (ld->cx == 0x000)
                if ((ld->line_h1 & 0x300) == 0 && (ld->line_h0 & 0x300) == 0)
                  s->tp[ld->j] = 0;
                else {
                  ld->ltp = 0;
                  /*#ifdef DEBUG
                  tp_exceptions++;
#endif*/
                }
              else if (ld->cx == 0x1ff)
                if ((ld->line_h1 & 0x300) == 0x300 && (ld->line_h0 & 0x300) == 0x300)
                  s->tp[ld->j] = 1;
                else {
                  ld->ltp = 0;
                  /*#ifdef DEBUG
                  tp_exceptions++;
#endif*/
                }
              else
                s->tp[ld->j] = 2;
            } while (++(ld->j) & 3 && (ld->j) < ld->lx);
          } while ((ld->j) & 7 && (ld->j) < ld->lx);
        } /* for (j = ...) */
        par1=ld->se;
        int1=TPDCX;
        int2=!ld->ltp;
        arith_encode_exec.main();

        /*#ifdef DEBUG
        (tp_lines) += ld->ltp;
#endif*/
      }

}
};
```

B.17 sde_diff_encode_line.sc

```
#include "constant.sh"

#include <assert.h>
```

```
import "jbig_head";
import "jbig";
import "arith_encode";
import "deterministic_prediction";
import "adaptive_template";
import "model_templates";
import "bus";

behavior sde_diff_encode_line(inout struct local_data *ld,
                              in struct jbg_enc_state *s,
                              PE_part sync_mem,
                              in unsigned long stripe,
                              in int layer,
                              in int plane,
                              in struct jbg_buf *file,
                              IBus_SW ibus)
{
   int int1, int2, flag, options, at_determined, count, cx, tx;
   unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
   unsigned *t, mx;

   int kk, number_of_bytes;
   int codes[6];
   int addr;


   deterministic_prediction  deterministic_prediction_exec(options, y, j, l1,l2,l3, h1, h2, h3, flag);
   adaptive_template  adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx, count,flag);

   arith_encode  arith_encode_exec(sync_mem, plane, int1, int2, file);
   model_templates  model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

void main(void){

      ld->line_h1 = ld->line_h2 = ld->line_h3 = ld->line_l1 = ld->line_l2 = ld->line_l3 = 0;
      if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
      if (ld->y > 1) {
        ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;
        ld->line_l3 = (long)*(ld->lp2 - ld->lbpl) << 8;
      }
      ld->line_l2 = (long)*ld->lp2 << 8;
      ld->line_l1 = (long)*ld->lp1 << 8;

      /* encode line */
      for (ld->j = 0; ld->j < ld->hx; ld->lp1++, ld->lp2++) {
        if ((ld->j >> 1) < ld->lbpl * 8 - 8) {
          if (ld->y > 1)
            ld->line_l3 |= *(ld->lp2 - ld->lbpl + 1);
          ld->line_l2 |= *(ld->lp2 + 1);
          ld->line_l1 |= *(ld->lp1 + 1);
        }
        do {
          ld->line_h1 |= *(ld->hp++);
          if (ld->j < ld->hbpl * 8 - 8) {
            if (ld->y > 0) {
              ld->line_h2 |= *(ld->hp - ld->hbpl);
              if (ld->y > 1)
                ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl);
            }
          }
          do {
            ld->line_l1 <<= 1;  ld->line_l2 <<= 1;  ld->line_l3 <<= 1;
            if (ld->ltp && s->tp[ld->j >> 1] < 2) {
              /* pixel are typical and have not to be encoded */
              ld->line_h1 <<= 2;  ld->line_h2 <<= 2;  ld->line_h3 <<= 2;
              (ld->j) += 2;
} else
  do {
    ld->line_h1 <<= 1;   ld->line_h2 <<= 1;   ld->line_h3 <<= 1;

    options=s->options;
    y=ld->y;
    j=ld->j;
    l1=ld->line_l1;
    l2=ld->line_l2;
    l3=ld->line_l3;
    h1=ld->line_h1;
    h2=ld->line_h2;
    h3=ld->line_h3;

    deterministic_prediction_exec.main();

    if (flag==1){
      continue;
    }
    else{
      y=ld->y;
      j=ld->j;
      l1=ld->line_l1;
      l2=ld->line_l2;
      l3=ld->line_l3;
      h1=ld->line_h1;
      h2=ld->line_h2;
      h3=ld->line_h3;
      tx=s->tx[plane];
      flag=5;

      model_templates_exec.main();
      ld->cx=cx;

      int1=ld->cx;
      int2=(ld->line_h1 >> 8) & 1;

      ibus.write_send(plane, ADDR_PLANE);
      ibus.write_send(int1, ADDR_CX);
      ibus.write_send(int2, ADDR_PIX);
      ibus.send_begin();
      ibus.receive_end();

      number_of_bytes = ibus.read_send(ADDR_COUNT);
      for(kk=0; kk<number_of_bytes; kk++){
            codes[kk] = ibus.read_send(ADDR_FILE_BUFFER);
      }

      /* write codes to sde[plane][layer][stripe] */
      for(kk=0; kk<number_of_bytes; kk++)
      jbg_buf_write(codes[kk], file);

      at_determined=ld->at_determined;
      j=ld->j;
      h1=ld->line_h1;
      h2=ld->line_h2;
      t=&(ld->t);
      c=ld->c;
      c_all=&(ld->c_all);
      mx=s->mx;
      hx=ld->lx;
      count=3;
      flag=1;
      adaptive_template_exec.main();
    }
  } while (++(ld->j) & 1 && (ld->j) < ld->hx);
```

```
          } while ((ld->j) & 7 && (ld->j) < ld->hx);
        } while ((ld->j) & 15 && (ld->j) < ld->hx);
      } /* for (j = ...) */

      /* low resolution pixels are used twice */
      if (((ld->i) & 1) == 0) {
        ld->lp1 -= ld->lbpl;
        ld->lp2 -= ld->lbpl;
      }
  }
  };
```

B.18 sde_lowest_encode_pixel.sc

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";
import "arith_encode";
import "adaptive_template";
import "model_templates";
import "bus";
import "CSync_mem";


behavior sde_lowest_encode_pixel(in struct local_data *ld,
                                 in struct jbg_enc_state *s,
                                 PE_part sync_mem,
                                 in unsigned long stripe,
                                 in int layer,
                                 in int plane,
                                 in int flag_mt1,
                                 in int flag_mt2,
                                 in int flag_at,
                                 in struct jbg_buf *file,
                                 IBus_SW ibus)
{
  int int1, int2, flag, options, at_determined, count, cx, tx;
  unsigned long y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
  unsigned *t, mx;
  struct jbg_arenc_state *par1;
  int kk, number_of_bytes;
  int codes[6];

  adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx, count, flag);
  arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);
  model_templates model_templates_exec(y, j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

 void main(){
      do {
              ld->line_h1 <<= 1;  ld->line_h2 <<= 1;  ld->line_h3 <<= 1;
              if (s->tx[plane]){
                   y=ld->y;
                    j=ld->j;
                    l1=ld->line_l1;
```

```
        l2=ld->line_l2;
        l3=ld->line_l3;
        h1=ld->line_h1;
        h2=ld->line_h2;
        h3=ld->line_h3;
        tx=s->tx[plane];
        flag=flag_mt1;
        model_templates_exec.main();
        ld->cx=cx;

        par1=ld->se;
        int1=ld->cx;
        int2=(ld->line_h1 >> 8) & 1;

        ibus.write_send(plane, ADDR_PLANE);
        ibus.write_send(int1, ADDR_CX);
        ibus.write_send(int2, ADDR_PIX);
        ibus.send_begin();
        ibus.receive_end();
 }
 else{

     y=ld->y;
       j=ld->j;
       l1=ld->line_l1;
       l2=ld->line_l2;
       l3=ld->line_l3;
       h1=ld->line_h1;
       h2=ld->line_h2;
       h3=ld->line_h3;
       tx=s->tx[plane];
       flag=flag_mt2;

  model_templates_exec.main();

  ld->cx=cx;

  par1=ld->se;
  int1=ld->cx;
  int2=(ld->line_h1 >> 8) & 1;


    ibus.write_send(plane, ADDR_PLANE);
      ibus.write_send(int1, ADDR_CX);
      ibus.write_send(int2, ADDR_PIX);
      ibus.send_begin();
      ibus.receive_end();

}

number_of_bytes = ibus.read_send(ADDR_COUNT);
for(kk=0; kk<number_of_bytes; kk++){
  codes[kk] = ibus.read_send(ADDR_FILE_BUFFER);
}

/* write codes to sde[plane][layer][stripe] */
for(kk=0; kk<number_of_bytes; kk++)
   jbg_buf_write(codes[kk], file);


/* lowest_adaptive_template(ld,s->mx,5); */
    at_determined=ld->at_determined;
      j=ld->j;
      h1=ld->line_h1;
      h2=ld->line_h2;
      t=&(ld->t);
```

```
                    c=ld->c;
                    c_all =&(ld->c_all );
                    mx=s->mx;
                    hx=ld->lx;
                    count=flag_at;
                    flag=0;
                    adaptive_template_exec.main();

            } while (++(ld->j) & 7 && (ld->j) < ld->hx);
          }
};
```

B.19 arith_encode.sc

```
#include "constant.sh"


#include <assert.h>


import "jbig_head";
import "jbig";
import "CSync_mem";


behavior arith_encode(PE_part sync_mem,
                      in int plane,
                      in int cx,
                      in int pix,
                      in struct jbg_buf *file
                      )
{
void main(void){
  register unsigned lsz, ss;
  register unsigned char *st;
  long temp;

  /* local copies of shared variables */
  unsigned char local_st;
  unsigned long local_c;
  unsigned long local_a;
  long local_sc;
  int local_ct;
  int local_buffer;

  /* load local copies */


  local_st = (char) sync_mem.read_send(plane, cx);
  local_c = (unsigned long) sync_mem.read_send(plane, ADDR_C);
  local_a = (unsigned long) sync_mem.read_send(plane, ADDR_A);
  local_sc = sync_mem.read_send(plane, ADDR_SC);
  local_ct = (int)sync_mem.read_send(plane, ADDR_CT);
  local_buffer = (int)sync_mem.read_send(plane, ADDR_BUFFER);

  ss = local_st & 0x7f;
  lsz = jbg_lsz[ss];
  if (((pix << 7) ^ local_st) & 0x80) {
    /* encode the less probable symbol */
    if ((local_a -= lsz) >= lsz) {
      /* If the interval size (lsz) for the less probable symbol (LPS)
       * is larger than the interval size for the MPS, then exchange
       * the two symbols for coding efficiency, otherwise code the LPS
       * as usual: */
      local_c += local_a;
      local_a = lsz;
    }
    /* Check whether MPS/LPS exchange is necessary
     * and chose next probability estimator status */
    local_st &= 0x80;
    local_st ^= jbg_nlps[ss];
  } else {
    /* encode the more probable symbol */
    if ((local_a -= lsz) & 0xffff8000L)
    {
      sync_mem.write_send((long)local_a, plane, ADDR_A);
      return;    /* A >= 0x8000 -> ready, no renormalization required */
    }

    if (local_a < lsz) {
      /* If the interval size (lsz) for the less probable symbol (LPS)
       * is larger than the interval size for the MPS, then exchange
       * the two symbols for coding efficiency: */
      local_c += local_a;
      local_a = lsz;
    }
    /* chose next probability estimator status */
    local_st &= 0x80;
    local_st |= jbg_nmps[ss];
  }

  /* renormalization of coding interval */
  do {
    local_a <<= 1;
    local_c <<= 1;
    --local_ct;
    if (local_ct == 0) {
      /* another byte is ready for output */
      temp = local_c >> 19;
      if (temp & 0xffffff00L) {
        /* handle overflow over all buffered 0xff bytes */
        if (local_buffer >= 0) {
          ++local_buffer;
          jbg_buf_write(local_buffer, file);
          if (local_buffer == MARKER_ESC)
            jbg_buf_write(MARKER_STUFF, file);
        }
        for (; local_sc; --local_sc)
          jbg_buf_write(0x00, file);
        local_buffer = temp & 0xff;   /* new output byte, might overflow later */
        assert(local_buffer != 0xff);
        /* can s->buffer really never become 0xff here? */
      } else if (temp == 0xff) {
        /* buffer 0xff byte (which might overflow later) */
        ++local_sc;
      } else {
        /* output all buffered 0xff bytes, they will not overflow any more */
        if (local_buffer >= 0)
          jbg_buf_write(local_buffer, file);
        for (; local_sc; --local_sc) {
          jbg_buf_write(0xff, file);
          jbg_buf_write(MARKER_STUFF, file);
        }
        local_buffer = temp;   /* buffer new output byte (can still overflow) */
      }
      local_c &= 0x7ffffL;
      local_ct = 8;
    }
  } while (local_a < 0x8000);

  /* store local copies */
  sync_mem.write_send((long)local_c,  plane, ADDR_C);
  sync_mem.write_send((long)local_a, plane, ADDR_A);
  sync_mem.write_send((long)local_ct, plane, ADDR_CT);
  sync_mem.write_send((long)local_buffer, plane, ADDR_BUFFER);
  sync_mem.write_send(local_sc, plane, ADDR_SC);
  sync_mem.write_send((long)local_st, plane, cx);

}
};
```

B.20 arith_encode_flush.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "CSync_mem";

behavior arith_encode_flush(PE_part sync_mem, in int plane, in struct jbg_buf *file)
{
  unsigned long temp;

void main(void)
{
  unsigned long local_c;
  unsigned long local_a;
  long local_sc;
  int local_ct;
  int local_buffer;

  local_c = (unsigned long)sync_mem.read_send(plane, ADDR_C);
  local_a = (unsigned long)sync_mem.read_send(plane,ADDR_A);
  local_sc = sync_mem.read_send(plane,ADDR_SC);
  local_ct = (int)sync_mem.read_send(plane,ADDR_CT);
  local_buffer = (int)sync_mem.read_send(plane,ADDR_BUFFER);

  if ((temp = (local_a - 1 + local_c) & 0xffff0000L) < local_c)
    local_c = temp + 0x8000;
  else
    local_c = temp;
  /* send remaining bytes to output */
  local_c <<= local_ct;
  if (local_c & 0xf8000000L) {
    /* one final overflow has to be handled */
    if (local_buffer >= 0) {
      jbg_buf_write(local_buffer + 1, file);
      if (local_buffer + 1 == MARKER_ESC)
        jbg_buf_write(MARKER_STUFF, file);
    }
    /* output 0x00 bytes only when more non-0x00 will follow */
    if (local_c & 0x7fff800L)
      for (; local_sc; --local_sc)
        jbg_buf_write(0x00, file);
  } else {
    if (local_buffer >= 0)
```

```
      jbg_buf_write(local_buffer, file);
    /* T.82 figure 30 says buffer+1 for the above line! Typo? */
    for (; local_sc; --local_sc) {
      jbg_buf_write(0xff, file);
      jbg_buf_write(MARKER_STUFF, file);
    }
  }
  /* output final bytes only if they are not 0x00 */
  if (local_c & 0x7fff800L) {
    jbg_buf_write((local_c >> 19) & 0xff, file);
    if (((local_c >> 19) & 0xff) == MARKER_ESC)
      jbg_buf_write(MARKER_STUFF, file);
    if (local_c & 0x7f800L) {
      jbg_buf_write((local_c >> 11) & 0xff, file);
      if (((local_c >> 11) & 0xff) == MARKER_ESC)
        jbg_buf_write(MARKER_STUFF, file);
    }
  }

 sync_mem.write_send((long)local_c, plane, ADDR_C);
 sync_mem.write_send((long)local_a, plane, ADDR_A);
 sync_mem.write_send((long)local_ct, plane, ADDR_CT);
 sync_mem.write_send((long)local_buffer, plane, ADDR_BUFFER);
 sync_mem.write_send(local_sc, plane, ADDR_SC);

  return;
}
};
```

C SpecC Communication Model for JBIG Encoder

Note: The Communication Model is not completely debugged at this moment.

C.1 jbg_enc_out.sc

```
#include "constant.sh"
import "std_include";
import "jbig_head";
import "jbig";
import "sw_component";
import "hw_component";
import "memory";
import "channel";



behavior jbg_enc_out(inout struct jbg_enc_state *s, in int not_done)
{
int maddr;
unsigned bit[0:0] mrwb;
event mtsb;
unsigned bit[0:0] mwdataoe;
WORD32 mrwdata;
event mtab;

WORD32 mem_maddr;
unsigned bit[0:0] mem_mrwb;
event mem_mtsb;
unsigned bit[0:0] mem_mwdataoe;
WORD32 mem_mrwdata;
event mem_mtab;

sw_component  sw_component_exec( s, maddr, mrwb, mtsb, mrwdata,mwdataoe,
              mtab,mem_maddr, mem_mrwb, mem_mtsb, mem_mrwdata,
              mem_mwdataoe, mem_mtab);
hw_component  hw_component_exec( maddr, mrwb, mtsb, mrwdata,mwdataoe,
              mtab,mem_maddr, mem_mrwb, mem_mtsb, mem_mrwdata,
              mem_mwdataoe, mem_mtab , not_done);

memory memory_exec(mem_maddr, mem_mrwb, mem_mtsb, mem_mrwdata,
       mem_mwdataoe,mem_mtab, not_done);


void main(){

         par{
                  sw_component_exec.main();
                  hw_component_exec.main();
                  memory_exec.main();
         }
}
};
```

C.2 bus.sc

```
#include "constant.sh"
#include "typedef.sh"
import "std_include";
import "channel";

```

```
/*--------------------------*/
interface IBus_SW
{
        void write_send(int data, int addr);
        int read_send(int addr);

        void receive_end();
        void send_begin();

};

interface IBus_HW
{
        int write_receive(int addr);
        void read_receive(int data, int addr);

        void receive_begin();
        void send_end();

};




/* The following is for the interface of software */

channel timing_bus_sw(
                    out WORD32 maddr,
                    out unsigned bit[0:0] mrwb,
                    event mtsb_n,
                    inout WORD32 mrwdata,
                    out unsigned bit[0:0] mwdataoe,
                    IRecvEvent intC
                    )
        implements IBus_SW
{


void write_send(int data, int addr){


        /* send signal to bus after the clock_cycle and signel_delay */
        /* waitfor(CLOCK_CYCLE_SW); */
        waitfor(SIGNEL_DELAY);
        maddr=addr;
        mrwb=0;
        mrwdata=data;
        mwdataoe=1;
        /* printf("!!! Now mrwb=%d, mwdataoe=%d, maddr=%d, in master\n",(int)mrwb, (int)mwdataoe, maddr); *
        notify(mtsb_n);
        /* printf("1: notify start signal\n"); */



        /* wait another clock */
        waitfor (CLOCK_CYCLE_SW);
        waitfor(SIGNEL_DELAY);

        /*wait mtab */
        intC.recv();
        /* printf("4: master finish work\n"); */
        /* wait for another clock */
        waitfor (CLOCK_CYCLE_SW);
        waitfor (SIGNEL_DELAY);

        mrwb=1;
        mwdataoe=0;


}


int read_send(int addr){
        int result;

        /* send signal to bus after the clock_cycle and signel_delay */
        /* waitfor (CLOCK_CYCLE_SW); */
        waitfor (SIGNEL_DELAY);

        maddr=addr;
        mrwb=1;
        notify (mtsb_n);
        /* printf("1: notify start signal\n"); */

        mwdataoe=0;

        waitfor (CLOCK_CYCLE_SW);

        waitfor (SIGNEL_DELAY);

        intC.recv ();
         /* printf("4: master finish work\n"); */

        /* cycle finish */
        /*receive_sw_clock ();*/

        waitfor (CLOCK_CYCLE_SW);
        waitfor (SIGNEL_DELAY);
        result=mrwdata;
        /* printf(" data[%d]=%d\n\n", addr, mrwdata); */
        mrwb=1;
        mwdataoe=0;

        return mrwdata;


}

void send_begin(){
        notify (mtsb_n);
}


void receive_end(){
        intC.recv ();
}


};
```

```
/* The following is for the interface of software */

channel timing_bus_hw(
                    in WORD32 maddr,
                    in unsigned bit[0:0] mrwb,
                    inout WORD32 mrwdata,
                    in unsigned bit[0:0] mwdataoe,
                    event mtab,
                    IRecvEvent intC
                    )
          implements IBus_HW
{

int write_receive(int addr){

          /* receive mtsb_n */
          intC.recv();
           /* printf("2: receive start signal\n");*/

          waitfor(CLOCK_CYCLE_HW);

          /* hw can implement computation in one clock cycle */
          waitfor(CLOCK_CYCLE_HW);
          if(maddr==addr && mrwb==0 && mwdataoe==1){
                     /* printf("good protocol\n");
                    printf("---Y Now mrwb=%d, mwdataoe=%d, maddr=%d, addr=%d in slave\n", (int)mrwb, (int)mwda

          }
          else{
                    /*  printf("wrong protocol\n");
                    printf("---N Now mrwb=%d, mwdataoe=%d, maddr=%d, addr=%d in slave\n", (int)mrwb, (int)mwda

          }

          notify(mtab);
          /* printf("3: reactive master\n");
          printf(" data[%d]=%d\n", addr, mrwdata); */

          return mrwdata;

 }

void read_receive(int data, int addr){
          /* receive mtsb_n */
          intC.recv();
          /* printf("2: receive start signal\n"); */

          /* receive_hw_clock(); */
          waitfor(CLOCK_CYCLE_HW);

          /* hw can implement computation in one clock cycle */
          /*receive_hw_clock();*/
          waitfor(CLOCK_CYCLE_HW);

          if(maddr==addr && mrwb==1){
                    mrwdata=data;
          }
          else{
                    /* printf("wrong protocol, read\n"); */
          }
          notify(mtab);
          /* printf("3: reactive master\n"); */
```

```
        /* keep data on bus for a while */
        waitfor (CLOCK_CYCLE_SW);
        waitfor (SIGNEL_DELAY);

  }


  void receive_begin(){
        intC.recv();
  }


  void send_end(){
        notify(mtab);
  }


  };
```

C.3 mbus.sc

```
/* This is used to model the SpecC memory model.
        It includes three parts: PE part, Memory part. interface between PE and mem part. The last part is
        added for the protocol matching */


#include "constant.sh"
#include "typedef.sh"
import "std_include";
import "channel";



typedef struct jbg_mem_unit {
  unsigned char st[CHAR_ARRAY];     /* probability status for contexts, MSB = MPS */
  unsigned long c;                 /* 4096 *
                                   * layout as in Table 23              */
  unsigned long a;       /*4100 */
  long sc;          /* 4104 */
  int ct;  /* 4108 */
  int buffer;                 /* 4112 */
}jbg_mem_unit;

/*-------------------------*/
interface PE_part
{

        /* memory write */
        void write_send (long data, int adds1, int adds2);

        /* memory read */
        long read_send (int adds1, int adds2);


};

interface MBus_mem
{
        /* common */
        int write_receive_addr ();
        int write_receive_data_type ();
        int write_receive_rw ();

        /* memory write */
        long write_receive_data ();

        /* memory read */

        void read_receive_data(long data);


};

interface MBus_transducer
{
        void transducer();
};







/* The following is for the interface of software */

channel timing_bus_pe(
                out WORD32 maddr,
                out unsigned bit[0:0] mrwb,
                event mtsb_n,
                inout WORD32 mrwdata,
                out unsigned bit[0:0] mwdataoe,
                IRecvEvent mtab_intC
                )
        implements PE_part

{



void write_send (long data, int adds1, int adds2){

        int addr;

        /* send signal to bus after the clock_cycle and signel_delay */
        /* waitfor(CLOCK_CYCLE_SW); */

        waitfor(SIGNEL_DELAY);
        addr=adds1*10000+adds2; /* adds1 is array, adds2 is in-array address. */


        maddr=addr;
        mrwb=0;
        mrwdata=data;
        mwdataoe=1;

        notify(mtsb_n);




        /* wait another clock
        waitfor (CLOCK_CYCLE_SW);
        waitfor(SIGNEL_DELAY);*/

        /*wait mtab */
        mtab_intC.recv();

        /* wait for another clock */
        waitfor (REST_DELAY);
        /*
        waitfor(SIGNEL_DELAY);
        mrwb=1;
        mwdataoe=0; */




    }



    long read_send (int adds1, int adds2){
        long result;
        int addr;

        /* send signal to bus after the clock_cycle and signel_delay */
        /* waitfor (CLOCK_CYCLE_SW); */
        waitfor(SIGNEL_DELAY);

        addr=adds1*10000+adds2; /* adds1 is array, adds2 is in-array address. */

        maddr=addr;
        mrwb=1;
        notify(mtsb_n);

        mwdataoe=0;



        mtab_intC.recv();


        /* cycle finish */
        /*receive_sw_clock();*/

        waitfor (REST_DELAY);
        result=mrwdata;


        /*
        waitfor(SIGNEL_DELAY);
        mrwb=1;
        mwdataoe=0;*/

        return result;


    }



    };


    channel timing_bus_transducer(in WORD32 maddr,
                                  in unsigned bit[0:0] mrwb,
                                  inout WORD32 mrwdata,
                                  IRecvEvent mtsb_intC,
                                  event mtab,
                                  out WORD32 addr,
                         out unsigned bit[0:0] n_cs,
                         out unsigned bit[0:0] n_oe,
                         out unsigned bit[0:0] n_we,
                         inout WORD32 data
                         )
 implements MBus_transducer
 {

         void transducer(){

                         mtsb_intC.recv();
                 addr=maddr;
                 n_cs=1;

                 waitfor(REST_DELAY);

                 if(mrwb==1){ /*read */

                         waitfor(SIGNEL_DELAY);
                         n_oe=0;
                         n_we=1;
                         notify(mtab);
                         waitfor(10);
                         mrwdata=data;
                         waitfor(1);
                 }
                 else{

                         waitfor(SIGNEL_DELAY);
                         n_oe=1;
                         n_we=0;
                         notify(mtab);
                         data=mrwdata;
                         waitfor(REST_DELAY);
                 }


                 n_oe=1;
                 n_we=1;

         }
};







/* The following is for the interface of software */

channel timing_bus_mem(
                   in WORD32 addr,
                   inout unsigned bit[0:0] n_cs,
                   inout unsigned bit[0:0] n_oe,
                   in unsigned bit[0:0] n_we,
                   inout WORD32 bus_data
                   )
        implements MBus_mem
{
        int write_receive_addr(){
                int result;


                while(n_oe==1 && n_we==1){
                        waitfor(1);
                }


                result=addr/10000;
                return result;
        }

        int write_receive_data_type(){
                int result;

                result=addr % 10000;

                return result;
        }

        int write_receive_rw(){
                int result;


                if(n_oe==0){
                        result=0;
                }
                else{
                        result=1;
                }

                return result;
        }

        long write_receive_data(){
                waitfor(REST_DELAY);

                n_we=1;
                n_oe=1;
                return (long)bus_data;
        }

        void read_receive_data(long mem_data){
                bus_data=(WORD32)mem_data;
                waitfor(REST_DELAY);
                n_we=1;
                n_oe=1;
        }

};
```

C.4 memory.sc

```
#include "constant.sh"
#include <assert.h>

import "std_include";
import "memory_main";
import "memory_transducer";

behavior memory(
                        in WORD32 maddr,
                        in unsigned bit[0:0] mrwb,
                        event mtsb,
                        inout WORD32 mrwdata,
                        in unsigned bit[0:0] mwdataoe,
                        event  mtab,
                        in int not_done
                        )
{
bit[0:0] n_cs=1;
bit[0:0] n_oe=1;
bit[0:0] n_we=1;
WORD32 data;
WORD32 addr;


memory_transducer memory_transducer_exec(maddr, mrwb,mtsb, mrwdata,mtab,addr, n_cs, n_oe, n_we, data);
memory_main memory_main_exec(addr, n_cs, n_oe, n_we, data, not_done);

        void main(void){

                        par{
                                memory_main_exec.main();
                                memory_transducer_exec.main();
                        }


        }
};
```

C.5 memory_main.sc

```
import "std_include";
import "mbus";
import "abstract_memory";
import "channel";

behavior memory_main(in WORD32 addr,
                     in unsigned bit[0:0] n_cs,
                     in unsigned bit[0:0] n_oe,
                     in unsigned bit[0:0] n_we,
                     inout WORD32 data,
                     in int not_done)
{

timing_bus_mem mem_bus(addr,n_cs , n_oe,n_we,data);
abstract_memory abstract_memory_exec(mem_bus, not_done);

        void main(void){
                abstract_memory_exec.main();
        }
};
```

C.6 abstract_memory.sc

```
#include "constant.sh"

import "std_include";
import "mbus";

```

```
static void mem_unit_init(int reuse, jbg_mem_unit *m)
{

        int i;
        if(!reuse)
        {
          for(i=0; i<CHAR_ARRAY; i++)
                   m->st[i] = 0;
        }
        m->c = 0;
        m->a = 0x10000L;
        m->sc = 0;
        m->ct = 11;
        m->buffer = -1;
}

behavior abstract_memory(MBus_mem sync, in int not_done)
{

void main(){

        int i;
        int rw;
        int sub_addr, addr; /* sub_addr is for inside the array, addr is for distingush different */
        jbg_mem_unit blocks[6];
        int reuse;

        while(not_done){
                sub_addr=sync.write_receive_addr();
                addr=sync.write_receive_data_type();
                rw=sync.write_receive_rw();

                switch (addr){
                        case UNIT_INIT:
                                reuse=sync.write_receive_data();
                                mem_unit_init(reuse, &blocks[sub_addr]);
                                break;

                        case ADDR_CT: /* write int data value */
                                if(rw==1){
                                        blocks[sub_addr].ct=(int)sync.write_receive_data();
                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].ct);
                                }
                                break;

                        case ADDR_BUFFER: /* write int data value */
                                if(rw==1){
                                        blocks[sub_addr].buffer=(int)sync.write_receive_data();
                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].buffer);
                                }
                                break;

                        case ADDR_C: /* write unsigned long data value */
                                if(rw==1){
                                        blocks[sub_addr].c=sync.write_receive_data();

                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].c);

                                }

                                break;
                        case ADDR_A: /* write unsigned long data value */
                                if(rw==1){
                                        blocks[sub_addr].a=sync.write_receive_data();

                                }else{
                                        sync.read_receive_data((long)blocks[sub_addr].a);

                                }
                                break;

                        case ADDR_SC:  /* write long data value */
                                if(rw==1){
                                        blocks[sub_addr].sc=sync.write_receive_data();
                                }else{
                                        sync.read_receive_data(blocks[sub_addr].sc);
                                }
                                break;

                        default:        /* for char addr is address in the table from 0 to 4095 */
                                if(rw==1){
                                         blocks[sub_addr].st[addr]=sync.write_receive_data();
                                }else{
                                        sync.read_receive_data(blocks[sub_addr].st[addr]);
                                }
                                break;
                        }

        }

}
};
```

C.7 memory_transducer.sc

```
import "std_include";
import "mbus";
import "channel";
import "memory_transducer_main";


behavior memory_transducer( in WORD32 maddr,
                            in unsigned bit[0:0] mrwb,
                            event mtsb,
                            inout WORD32 mrwdata,
                            event  mtab,
                            out WORD32 addr,
                            out unsigned bit[0:0] n_cs,
                            out unsigned bit[0:0] n_oe,
                            out unsigned bit[0:0] n_we,
                            inout WORD32 data)
{

CEvent mtsb_intC;

timing_bus_transducer t_bus(maddr, mrwb, mrwdata, mtsb_intC, mtab, addr, n_cs, n_oe, n_we, data);
memory_tranducer_main memory_transducer_main_exec(t_bus);
event_handler event_handler_e(mtsb_intC);
```

```


        void main(void){
                try{
                        memory_transducer_main_exec.main();
                }
                interrupt(mtsb){
                        event_handler_e.main();
                }
        }
};
```

C.8 memory_transducer_main.sc

```
import "std_include";
import "mbus";
import "channel";

behavior memory_tranducer_main(MBus_transducer t_bus){

        void main(){
                while(1){
                        t_bus.transducer();
                }
        }
};
```

C.9 sw_component.sc

```
#include "constant.sh"
import "std_include";
import "jbig_head";
import "jbig";
import "bus";
import "sw_main";
import "channel";

import "mbus";

behavior sw_component(
                        inout struct jbg_enc_state *s,
                        out WORD32 maddr,
                        out unsigned bit[0:0] mrwb,
                        event mtsb_n,
                        inout WORD32 mrwdata,
                        out unsigned bit[0:0] mwdataoe,
                        event  mtab,
                       out WORD32 mem_maddr,
                        out unsigned bit[0:0] mem_mrwb,
                        event mem_mtsb,
                        inout WORD32 mem_mrwdata,
                        out unsigned bit[0:0] mem_mwdataoe,
                        event  mem_mtab
                        )
{


  CEvent intC, mem_mtab_intC;

  timing_bus_pe mbus(mem_maddr, mem_mrwb, mem_mtsb, mem_mrwdata,mem_mwdataoe, mem_mtab_intC);
  timing_bus_sw  ibus(maddr, mrwb, mtsb_n, mrwdata,mwdataoe, intC);
  sw_main  sw_main_e(mbus, s, ibus);
  event_handler  event_handler_e(intC);
  event_handler  event_handler_mem(mem_mtab_intC);



        void  main(void)
        {
                try{
                        sw_main_e.main();
                }
                interrupt(mtab){
                        event_handler_e.main();
                }
                interrupt(mem_mtab){
                        event_handler_mem.main();
                }

        }
};

```

C.10 hw_component.sc

```
#include "constant.sh"
#include <assert.h>

import "std_include";
import "jbig_head";
import "jbig";
import "bus";
import "hw_component_main";

import "mbus";

behavior hw_component(
                      in WORD32 maddr,
                    in unsigned bit[0:0] mrwb,
                    event mtsb,
                    inout WORD32 mrwdata,
                    in unsigned bit[0:0] mwdataoe,
                    event  mtab,
                    out WORD32 mem_maddr,
                       out unsigned bit[0:0] mem_mrwb,
                       event mem_mtsb,
                       inout WORD32 mem_mrwdata,
                       out unsigned bit[0:0] mem_mwdataoe,
                       event  mem_mtab,
                         in int not_done

                    )
{

CEvent intC, mem_mtab_intC;

event_handler  event_handler_e(intC);
event_handler  event_handler_mem(mem_mtab_intC);

timing_bus_pe mbus(mem_maddr, mem_mrwb, mem_mtsb, mem_mrwdata,mem_mwdataoe, mem_mtab_intC);
timing_bus_hw ibus(maddr, mrwb,mrwdata,mwdataoe,mtab,intC);
hw_component_main hw_component_main_exec( mbus, ibus, not_done);

        void main(void){

                try{
                        hw_component_main_exec.main();
                }
                interrupt(mtsb){
                        event_handler_e.main();
                }
                interrupt(mem_mtab){
                        event_handler_mem.main();
                }

        }
};
```

C.11 hw_component_main.sc

```
#include "constant.sh"
#include <assert.h>

import "std_include";
import "jbig_head";
import "jbig";
import "mbus";
import "bus";


behavior hw_component_main(PE_part sync_mem,
                           IBus_HW ibus,
                           in int not_done
                           )
{

void main(void){
  register unsigned lsz, ss;
  register unsigned char st;
  long temp;
  struct jbg_arenc_state *s;
  int cx;
  int pix, plane, i;
  int databuffer[100], bufferindex;
  int addr;

  s = (struct jbg_arenc_state *) malloc(sizeof(struct jbg_arenc_state));

  while(not_done){
        /* communicate with the coldfire */
        plane=ibus.write_receive(ADDR_PLANE);
        cx=ibus.write_receive(ADDR_CX);
        pix=ibus.write_receive(ADDR_PIX);
        ibus.receive_begin();

        bufferindex=0;
        for (i=0; i<100; i++){
                databuffer[i]=0;
        }

        /* communicate with memory, read memory */
        st=(char)sync_mem.read_send(plane,cx);
        s->c=(unsigned long)sync_mem.read_send(plane, ADDR_C);
        s->a=(unsigned long)sync_mem.read_send(plane, ADDR_A);
  void main(void){

    assert(s->sde[stripe][layer][plane] != SDE_DONE);

    if (s->sde[stripe][layer][plane] != SDE_TODO) {

      jbg_buf_output(&s->sde[stripe][layer][plane], s->data_out, s->file);
      s->sde[stripe][layer][plane] = SDE_DONE;
      return;
    }

    /* Determine the smallest resolution layer in this plane for which
     * not yet all stripes have been encoded into SDEs. This layer will
     * have to be completely coded, before we can apply the next
     * resolution reduction step. */
    lfcl = 0;
    for (i = s->d; i >= 0; i--)
      if (s->sde[s->stripes - 1][i][plane] == SDE_TODO) {
        lfcl = i + 1;
        break;
      }
    if (lfcl > s->d && s->d > 0 && stripe == 0) {
      /* perform the first resolution reduction */
      lfcl_1 = s->d;
      resolution_reduction_exec.main();
    }
    /* In case HITOLO is not used, we have to encode and store the higher
     * resolution layers first, although we do not need them right now. */
    while (lfcl - 1 > layer) {
      for (u = 0; u < s->stripes; u++)
      {
        lfcl_1 = lfcl - 1;
        encode_sde_exec.main();
      }
      --lfcl;
      s->highres[plane] ^= 1;
      if (lfcl > 1)
      {
        lfcl_1 = lfcl - 1;
        resolution_reduction_exec.main();
      }
    }

  /* added to satisfy the input conditions */
    u = stripe;
    lfcl_1 = layer;
    encode_sde_exec.main();

    jbg_buf_output(&s->sde[stripe][layer][plane], s->data_out, s->file);
    s->sde[stripe][layer][plane] = SDE_DONE;

    if (stripe == s->stripes - 1 && layer > 0 &&
        s->sde[0][layer-1][plane] == SDE_TODO) {
      s->highres[plane] ^= 1;
      if (layer > 1)
      {
        lfcl_1 = layer - 1;
        resolution_reduction_exec.main();
      }
    }

 }
};
```

```
 * required for JBIG. The same algorithm is also used in the arithmetic
 * variant of JPEG.
 */

behavior sde_init(in struct local_data *ld,
                  in struct jbg_enc_state *s,
                  PE_part sync_mem,
                  in unsigned long stripe,
                  in int layer,
                  in int plane)
{
  int reuse;

  void main(void)
  {
   /* return immediately if this stripe has already been encoded */
   if (s->sde[stripe][layer][plane] != SDE_TODO)
     return;

   ld->line_h0 = 0;
   ld->line_h1 = 0;
   ld->new_tx_line = -1;

   /* number of lines per stripe in highres image */
   ld->hl = s->l0 << layer;
   /* number of lines per stripe in lowres image */
   ld->ll = ld->hl >> 1;
   /* current line number in highres image */
   ld->y = stripe * ld->hl;
   /* number of pixels in highres image */
   ld->hx = jbg_ceil_half(s->xd, s->d - layer);
   ld->hy = jbg_ceil_half(s->yd, s->d - layer);
   /* number of pixels in lowres image */
   ld->lx = jbg_ceil_half(ld->hx, 1);
   ld->ly = jbg_ceil_half(ld->hy, 1);
   /* bytes per line in highres and lowres image */
   ld->hbpl = (ld->hx + 7) / 8;
   ld->lbpl = (ld->lx + 7) / 8;
   /* pointer to first image byte of highres stripe */
   ld->hp = s->lhp[s->highres[plane]][plane] + stripe * ld->hl * ld->hbpl;
   ld->lp2 = s->lhp[1 - s->highres[plane]][plane] + stripe * ld->ll * ld->lbpl;
   ld->lp1 = ld->lp2 + ld->lbpl;

   if(stripe != 0)
    reuse = 1;
   else
    reuse = 0;

   sync_mem.write_send((long)reuse, plane, UNIT_INIT);
   s->sde[stripe][layer][plane] = jbg_buf_init(&s->free_list);

   /* initialize adaptive template movement algorithm */
   ld->c_all = 0;
   for (ld->t = 0; ld->t <= s->mx; ld->t++)
     ld->c[ld->t] = 0;
   if (stripe == 0)
     s->tx[plane] = 0;
   ld->new_tx = -1;
   ld->at_determined = 0;  /* we haven't yet decided the template move */
   if (s->mx == 0)
     ld->at_determined = 1;

   /* initialize typical prediction */
   ld->ltp = 0;
   if (stripe == 0)
     ld->ltp_old = 0;
    else {
      ld->ltp_old = 1;
      ld->p1 = ld->hp - ld->hbpl;
      if (ld->y > 1) {
        ld->q1 =ld->p1 - ld->hbpl;
        while (ld->p1 < ld->hp && (ld->ltp_old = (*(ld->p1)++ == *(ld->q1)++)) != 0);
      } else
        while (ld->p1 < ld->hp && (ld->ltp_old = (*(ld->p1)++ == 0)) != 0);
    }
  }
};
```

C.16 SDE_flush.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "arith_encode_flush";
import "add_atmove";
import "mbus";

behavior SDE_flush(in struct local_data *ld,
                   in struct jbg_enc_state *s,
                   in unsigned long stripe,
                   in int layer,
                   PE_part sync_mem,
                   in int plane,
                   in struct jbg_buf *file)
{
  arith_encode_flush arith_encode_flush_exec(sync_mem, plane, file);
  add_atmove add_atmove_exec(ld, s, stripe, layer, plane);

void main(){
  arith_encode_flush_exec.main();

  jbg_buf_remove_zeros(s->sde[stripe][layer][plane]);
  jbg_buf_write(MARKER_ESC, s->sde[stripe][layer][plane]);
  jbg_buf_write(MARKER_SDNORM, s->sde[stripe][layer][plane]);

  add_atmove_exec.main();

}

};
```

C.17 sde_encode_lowest.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "determine_ATMOVE";
import "sde_lowest_encode_line";
import "sde_lowest_tp";
import "bus";
import "mbus";

```

```
    for (ld->i = 0; ld->i < ld->hl && ld->y < ld->hy; ld->i++, ld->y++)
    {
        determine_ATMOVE_exec.main();
        sde_diff_el_tp_exec.main();
        sde_diff_encode_line_exec.main();
    } /* for (i = ...) */
  }
};
```

C.19 sde_lowest_encode_line.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "sde_lowest_encode_pixel";
import "bus";

behavior sde_lowest_encode_line(in struct local_data *ld,
                                in struct jbg_enc_state *s,
                                PE_part sync_mem,
                                in unsigned long stripe,
                                in int layer,
                                in int plane,
                                in struct jbg_buf *file,
                                IBus_SW ibus)
{
  int flag_mt1, flag_mt2, flag_at;

  sde_lowest_encode_pixel sde_lowest_encode_pixel_exec(ld, s, sync_mem, stripe, layer, plane, flag_mt1, fla

void main(void){

      ld->line_h1 = ld->line_h2 = ld->line_h3 = 0;
      if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
      if (ld->y > 1) ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;

      /* encode line */
      for (ld->j = 0; ld->j < ld->hx; ld->hp++) {
        ld->line_h1 |= *(ld->hp);
        if (ld->j < ld->hbpl * 8 - 8 && ld->y > 0) {
          ld->line_h2 |= *(ld->hp - ld->hbpl + 1);
          if (ld->y > 1)
            ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl + 1);
        }

       if (s->options & JBG_LRLTWO) {
          /* two_line_template */
                flag_mt1=1;
                flag_mt2=2;
                flag_at=5;
            sde_lowest_encode_pixel_exec.main();
        } else {
          /* three_line_template */
                flag_mt1=3;
                flag_mt2=4;
                flag_at=3;
            sde_lowest_encode_pixel_exec.main();
        } /* if (s->options & JBG_LRLTWO) */
      } /* for (j = ...) */
}
};
```

C.20 sde_lowest_tp.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "arith_encode";

/* self-defined functions */

behavior sde_lowest_tp(PE_part sync_mem,
                         in struct local_data *ld,
                        in struct jbg_enc_state *s,
                        in unsigned long stripe,
                        in int layer,
                        in int plane,
                        out int flag,
                        in struct jbg_buf *file)
{
 int int1, int2;
 arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);

void main(void){
      flag=1;

   /* typical prediction */
      if (s->options & JBG_TPBON) {
        ld->ltp = 1;
        ld->p1 = ld->hp;
        if (ld->y > 0) {
          ld->q1 = ld->hp - ld->hbpl;
          while (ld->q1 < ld->hp && (ld->ltp = (*(ld->p1)++ == *(ld->q1)++)) != 0);
        } else
          while (ld->p1 < ld->hp + ld->hbpl && (ld->ltp = (*(ld->p1)++ == 0)) != 0);
        int1=((s->options & JBG_LRLTWO) ? TPB2CX : TPB3CX);
        int2=(ld->ltp == ld->ltp_old);
        arith_encode_exec.main();

        ld->ltp_old = ld->ltp;
        if (ld->ltp) {
          /* skip next line */
          ld->hp += ld->hbpl;
          flag=0;
        }
      }

}
};
```

C.21 sde_diff_el_tp.sc

```
#include "constant.sh"
/*
#include <assert.h>
*/
import "jbig_head";
import "jbig";
import "arith_encode";

behavior sde_diff_el_tp(PE_part sync_mem,
                        in struct local_data *ld,
                        in struct jbg_enc_state *s,
                        in unsigned long stripe,
                        in int layer,
                          in int plane,
                          in struct jbg_buf *file )
{int int1, int2;
 struct jbg_arenc_state *par1;

 arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);

void main(void){
  /* typical prediction */
      if (s->options & JBG_TPDON && (ld->i & 1) == 0) {
        ld->q1 = ld->lp1; ld->q2 = ld->lp2;
        ld->p0 = ld->p1 = ld->hp;
        if (ld->i < ld->hl - 1 && ld->y < ld->hy - 1)
          ld->p0 = ld->hp + ld->hbpl;
        if (ld->y > 1)
          ld->line_l3 = ( long)*(ld->q2 - ld->lbpl) << 8;
        else
          ld->line_l3 = 0;
        ld->line_l2 = (long)*(ld->q2) << 8;
        ld->line_l1 = (long)*(ld->q1) << 8;
        ld->ltp = 1;
        for (ld->j = 0; ld->j < ld->lx && ld->ltp; ld->q1++, ld->q2++) {
          if (ld->j < ld->lbpl * 8 - 8) {
            if (ld->y > 1)
              ld->line_l3 |= *(ld->q2 - ld->lbpl + 1);
            ld->line_l2 |= *(ld->q2 + 1);
            ld->line_l1 |= *(ld->q1 + 1);
          }
          do {
            if ((ld->j >> 2) < ld->hbpl) {
              ld->line_h1 = *(ld->p1++);
              ld->line_h0 = *(ld->p0++);
            }
            do {
              ld->line_l3 <<= 1;
              ld->line_l2 <<= 1;
              ld->line_l1 <<= 1;
              ld->line_h1 <<= 2;
              ld->line_h0 <<= 2;
              ld->cx = (((ld->line_l3 >> 15) & 0x007) |
                    ((ld->line_l2 >> 12) & 0x038) |
                    ((ld->line_l1 >> 9)  & 0x1c0));
              if (ld->cx == 0x000)
                if ((ld->line_h1 & 0x300) == 0 && (ld->line_h0 & 0x300) == 0)
                  s->tp[ld->j] = 0;
                else {
                  ld->ltp = 0;
                  /*#ifdef DEBUG
                  tp_exceptions++;
#endif*/
                }
              else if (ld->cx == 0x1ff)
                if ((ld->line_h1 & 0x300) == 0x300 && (ld->line_h0 & 0x300) == 0x300)
                  s->tp[ld->j] = 1;
                else {
                  ld->ltp = 0;
                  /*# ifdef DEBUG
                  tp_exceptions++;
#endif*/
                }
              else
                s->tp[ld->j] = 2;
            } while (++(ld->j) & 3 && (ld->j) < ld->lx);
          } while ((ld->j) & 7 && (ld->j) < ld->lx);
        } /* for (j = ...) */
        par1=ld->se;
        int1=TPDCX;
        int2=!ld->ltp;
        arith_encode_exec.main();

        /*#ifdef DEBUG
        (tp_lines) += ld->ltp;
#endif*/
      }

  }
};
```

C.22 sde_diff_encode_line.sc

```
#include "constant.sh"

#include <assert.h>

import "jbig_head";
import "jbig";
import "arith_encode";
import "deterministic_prediction";
import "adaptive_template";
import "model_templates";
import "bus";

behavior sde_diff_encode_line(inout struct local_data *ld,
                              in struct jbg_enc_state *s,
                              PE_part sync_mem,
                              in unsigned long stripe,
                              in int layer,
                              in int plane,
                              in struct jbg_buf *file,
                              IBus_SW ibus)
{
   int int1, int2, flag, options, at_determined, count, cx, tx;
   unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
   unsigned *t, mx;

   int kk, number_of_bytes;
   int codes[6];
   int addr;


   deterministic_prediction  deterministic_prediction_exec(options, y, j, l1,l2,l3, h1, h2, h3, flag);
   adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx, count,flag);

   arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);
   model_templates model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

 void main(void){

      ld->line_h1 = ld->line_h2 = ld->line_h3 = ld->line_l1 = ld->line_l2 = ld->line_l3 = 0;
      if (ld->y > 0) ld->line_h2 = (long)*(ld->hp - ld->hbpl) << 8;
      if (ld->y > 1) {
        ld->line_h3 = (long)*(ld->hp - ld->hbpl - ld->hbpl) << 8;
        ld->line_l3 = (long)*(ld->lp2 - ld->lbpl) << 8;
      }
      ld->line_l2 = (long)*ld->lp2 << 8;
      ld->line_l1 = (long)*ld->lp1 << 8;

      /* encode line */
      for (ld->j = 0; ld->j < ld->hx; ld->lp1++, ld->lp2++) {
        if ((ld->j >> 1) < ld->lbpl * 8 - 8) {
```

```
    if (ld->y > 1)
      ld->line_l3 |= *(ld->lp2 - ld->lbpl + 1);
    ld->line_l2 |= *(ld->lp2 + 1);
    ld->line_l1 |= *(ld->lp1 + 1);
  }
  do {
    ld->line_h1 |= *(ld->hp++);
    if (ld->j < ld->hbpl * 8 - 8) {
      if (ld->y > 0) {
        ld->line_h2 |= *(ld->hp - ld->hbpl);
        if (ld->y > 1)
          ld->line_h3 |= *(ld->hp - ld->hbpl - ld->hbpl);
      }
    }
    do {
      ld->line_l1 <<= 1;  ld->line_l2 <<= 1;  ld->line_l3 <<= 1;
      if (ld->ltp && s->tp[ld->j >> 1] < 2) {
        /* pixel are typical and have not to be encoded */
        ld->line_h1 <<= 2;  ld->line_h2 <<= 2;  ld->line_h3 <<= 2;
        (ld->j) += 2;
      } else
        do {
          ld->line_h1 <<= 1;  ld->line_h2 <<= 1;  ld->line_h3 <<= 1;

          options=s->options;
          y=ld->y;
          j=ld->j;
          l1=ld->line_l1;
          l2=ld->line_l2;
          l3=ld->line_l3;
          h1=ld->line_h1;
          h2=ld->line_h2;
          h3=ld->line_h3;

          deterministic_prediction_exec.main();

          if (flag==1){
            continue;
          }
          else{
            y=ld->y;
            j=ld->j;
            l1=ld->line_l1;
            l2=ld->line_l2;
            l3=ld->line_l3;
            h1=ld->line_h1;
            h2=ld->line_h2;
            h3=ld->line_h3;
            tx=s->tx[plane];
            flag=5;

            model_templates_exec.main();
            ld->cx=cx;

            int1=ld->cx;
            int2=(ld->line_h1 >> 8) & 1;

            ibus.write_send(plane, ADDR_PLANE);
            ibus.write_send(int1, ADDR_CX);
            ibus.write_send(int2, ADDR_PIX);
            ibus.send_begin();
            ibus.receive_end();

            number_of_bytes = ibus.read_send(ADDR_COUNT);
            for(kk=0; kk<number_of_bytes; kk++){
                  codes[kk] = ibus.read_send(ADDR_FILE_BUFFER);
                  }

                  /* write codes to sde[plane][layer][stripe] */
                  for(kk=0; kk<number_of_bytes; kk++)
                  jbg_buf_write(codes[kk], file);

                  at_determined=ld->at_determined;
                  j=ld->j;
                  h1=ld->line_h1;
                  h2=ld->line_h2;
                  t=&(ld->t);
                  c=ld->c;
                  c_all=&(ld->c_all);
                  mx=s->mx;
                  hx=ld->lx;
                  count=3;
                  flag=1;
                  adaptive_template_exec.main();
                }
              } while (++(ld->j) & 1 && (ld->j) < ld->hx);
          } while ((ld->j) & 7 && (ld->j) < ld->hx);
        } while ((ld->j) & 15 && (ld->j) < ld->hx);
      } /* for (j = ...) */

      /* low resolution pixels are used twice */
      if (((ld->i) & 1) == 0) {
        ld->lp1 -= ld->lbpl;
        ld->lp2 -= ld->lbpl;
      }
}
};
```

C.23 sde_lowest_encode_pixel.sc

```
#include "constant.sh"

#include <assert.h>


import "jbig_head";
import "jbig";
import "arith_encode";
import "adaptive_template";
import "model_templates";
import "bus";
import "mbus";


behavior sde_lowest_encode_pixel(in struct local_data *ld,
                                 in struct jbg_enc_state *s,
                                 PE_part sync_mem,
                                 in unsigned long stripe,
                                 in int layer,
                                 in int plane,
                                 in int flag_mt1,
                                 in int flag_mt2,
                                 in int flag_at,
                              in struct jbg_buf *file,
                              IBus_SW ibus)
{
  int int1, int2, flag, options, at_determined, count, cx, tx;
  unsigned long  y, j, l1, l2, l3, h1, h2, h3, *c, *c_all, hx;
  unsigned *t, mx;
  struct jbg_arenc_state *par1;
  int kk, number_of_bytes;
  int codes[6];

  adaptive_template adaptive_template_exec(at_determined, j, h1, h2, t, c, c_all, mx, hx, count,flag);
  arith_encode arith_encode_exec(sync_mem, plane, int1, int2, file);
  model_templates model_templates_exec(y,j, l1, l2, l3, h1, h2, h3, tx, flag, cx);

void main(){
    do {
          ld->line_h1 <<= 1;  ld->line_h2 <<= 1;  ld->line_h3 <<= 1;
          if (s->tx[plane]){
              y=ld->y;
               j=ld->j;
               l1=ld->line_l1;
               l2=ld->line_l2;
               l3=ld->line_l3;
               h1=ld->line_h1;
               h2=ld->line_h2;
               h3=ld->line_h3;
               tx=s->tx[plane];
               flag=flag_mt1;
               model_templates_exec.main();
               ld->cx=cx;

               par1=ld->se;
               int1=ld->cx;
               int2=(ld->line_h1 >> 8) & 1;

               ibus.write_send(plane, ADDR_PLANE);
               ibus.write_send(int1, ADDR_CX);
               ibus.write_send(int2, ADDR_PIX);
               ibus.send_begin();
               ibus.receive_end();
          }
          else{

             y=ld->y;
               j=ld->j;
               l1=ld->line_l1;
               l2=ld->line_l2;
               l3=ld->line_l3;
               h1=ld->line_h1;
               h2=ld->line_h2;
               h3=ld->line_h3;
               tx=s->tx[plane];
               flag=flag_mt2;

           model_templates_exec.main();

           ld->cx=cx;

           par1=ld->se;
           int1=ld->cx;
           int2=(ld->line_h1 >> 8) & 1;


             ibus.write_send(plane, ADDR_PLANE);
               ibus.write_send(int1, ADDR_CX);
               ibus.write_send(int2, ADDR_PIX);
```

```
                    ibus.send_begin();
                    ibus.receive_end();

                }

                number_of_bytes = ibus.read_send(ADDR_COUNT);
                for(kk=0; kk<number_of_bytes; kk++){
                  codes[kk] = ibus.read_send(ADDR_FILE_BUFFER);
                }

                /* write codes to sde[plane][layer][stripe] */
                for(kk=0; kk<number_of_bytes; kk++)
                   jbg_buf_write(codes[kk], file);


                /* lowest_adaptive_template(ld, s->mx, 5); */
                    at_determined=ld->at_determined;
                      j=ld->j;
                      h1=ld->line_h1;
                      h2=ld->line_h2;
                      t=&(ld->t);
                      c=ld->c;
                      c_all=&(ld->c_all);
                      mx=s->mx;
                      hx=ld->lx;
                      count=flag_at;
                      flag=0;
                      adaptive_template_exec.main();

              } while (++(ld->j) & 7 && (ld->j) < ld->hx);
            }
  };
```

C.24 arith_encode.sc

```
#include "constant.sh"


#include <assert.h>


import "jbig_head";
import "jbig";
import "mbus";


behavior arith_encode(PE_part sync_mem,
                      in int plane,
                      in int cx,
                      in int pix,
                      in struct jbg_buf *file
                      )
{
void main(void){
  register unsigned lsz, ss;
  register unsigned char *st;
  long temp;
```

```
/* local copies of shared variables */
unsigned char local_st;
unsigned long local_c;
unsigned long local_a;
long local_sc;
int local_ct;
int local_buffer;

/* load local copies */


local_st =(char) sync_mem.read_send(plane, cx);
local_c = (unsigned long)sync_mem.read_send(plane, ADDR_C);
local_a = (unsigned long)sync_mem.read_send(plane, ADDR_A);
local_sc = sync_mem.read_send(plane, ADDR_SC);
local_ct = (int)sync_mem.read_send(plane, ADDR_CT);
local_buffer = (int) sync_mem.read_send(plane, ADDR_BUFFER);

ss = local_st & 0x7f;
lsz = jbg_lsz[ss];

if (((pix << 7) ^ local_st) & 0x80) {
  /* encode the less probable symbol */
  if ((local_a -= lsz) >= lsz) {
    /* If the interval size (lsz) for the less probable symbol (LPS)
     * is larger than the interval size for the MPS, then exchange
     * the two symbols for coding efficiency, otherwise code the LPS
     * as usual: */
    local_c += local_a;
    local_a = lsz;
  }
  /* Check whether MPS/LPS exchange is necessary
   * and chose next probability estimator status */
  local_st &= 0x80;
  local_st ^= jbg_nlps[ss];
} else {
  /* encode the more probable symbol */
  if ((local_a -= lsz) & 0xffff8000L)
  {
    sync_mem.write_send((long)local_a, plane, ADDR_A);
    return;    /* A >= 0x8000 -> ready, no renormalization required */
  }

  if (local_a < lsz) {
    /* If the interval size (lsz) for the less probable symbol (LPS)
     * is larger than the interval size for the MPS, then exchange
     * the two symbols for coding efficiency: */
    local_c += local_a;
    local_a = lsz;
  }
  /* chose next probability estimator status */
  local_st &= 0x80;
  local_st |= jbg_nmps[ss];
}

/* renormalization of coding interval */
do {
  local_a <<= 1;
  local_c <<= 1;
  --local_ct;
  if (local_ct == 0) {
    /* another byte is ready for output */
    temp = local_c >> 19;
    if (temp & 0xffffff00L) {
      /* handle overflow over all buffered 0xff bytes */
```

```
        if (local_buffer >= 0) {
          ++local_buffer;
          jbg_buf_write(local_buffer, file);
          if (local_buffer == MARKER_ESC)
            jbg_buf_write(MARKER_STUFF, file);
        }
        for (; local_sc; --local_sc)
          jbg_buf_write(0x00, file);
        local_buffer = temp & 0xff; /* new output byte, might overflow later */
        assert(local_buffer != 0xff);
        /* can s->buffer really never become 0xff here? */
      } else if (temp == 0xff) {
        /* buffer 0xff byte (which might overflow later) */
        ++local_sc;
      } else {
        /* output all buffered 0xff bytes, they will not overflow any more */
        if (local_buffer >= 0)
          jbg_buf_write(local_buffer, file);
        for (; local_sc; --local_sc) {
          jbg_buf_write(0xff, file);
          jbg_buf_write(MARKER_STUFF, file);
        }
        local_buffer = temp;   /* buffer new output byte (can still overflow) */
      }
      local_c &= 0x7ffffL;
      local_ct = 8;
    }
  } while (local_a < 0x8000);

  /* store local copies */
  sync_mem.write_send((long)local_c,  plane, ADDR_C);
  sync_mem.write_send((long)local_a, plane, ADDR_A);
  sync_mem.write_send((long)local_ct, plane, ADDR_CT);
  sync_mem.write_send((long)local_buffer, plane, ADDR_BUFFER);
  sync_mem.write_send((long)local_sc, plane, ADDR_SC);
  sync_mem.write_send((long)local_st, plane, cx);

}
};
```

C.25 arith_encode_flush.sc

```
#include "constant.sh"

import "jbig_head";
import "jbig";
import "mbus";

behavior arith_encode_flush(PE_part sync_mem, in int plane, in struct jbg_buf *file)
{
  unsigned long temp;

void main(void)
{
  unsigned long local_c;
  unsigned long local_a;
  long local_sc;
  int local_ct;
  int local_buffer;

  local_c = (unsigned long) sync_mem.read_send(plane, ADDR_C);
  local_a = (unsigned long)sync_mem.read_send(plane,ADDR_A);
  local_sc = sync_mem.read_send(plane,ADDR_SC);
  local_ct = (int)sync_mem.read_send(plane,ADDR_CT);
```

```
    local_buffer = (int)sync_mem.read_send(plane,ADDR_BUFFER);

    if ((temp = (local_a - 1 + local_c) & 0xffff0000L) < local_c)
      local_c = temp + 0x8000;
    else
      local_c = temp;
    /* send remaining bytes to output */
    local_c <<= local_ct;
    if (local_c & 0xf8000000L) {
      /* one final overflow has to be handled */
      if (local_buffer >= 0) {
        jbg_buf_write(local_buffer + 1, file);
        if (local_buffer + 1 == MARKER_ESC)
          jbg_buf_write(MARKER_STUFF, file);
      }
      /* output 0x00 bytes only when more non-0x00 will follow */
      if (local_c & 0x7fff800L)
        for (; local_sc; --local_sc)
          jbg_buf_write(0x00, file);
    } else {
      if (local_buffer >= 0)
        jbg_buf_write(local_buffer, file);
      /* T.82 figure 30 says buffer+1 for the above line! Typo? */
      for (; local_sc; --local_sc) {
        jbg_buf_write(0xff, file);
        jbg_buf_write(MARKER_STUFF, file);
      }
    }
    /* output final bytes only if they are not 0x00 */
    if (local_c & 0x7fff800L) {
      jbg_buf_write((local_c >> 19) & 0xff, file);
      if (((local_c >> 19) & 0xff) == MARKER_ESC)
        jbg_buf_write(MARKER_STUFF, file);
      if (local_c & 0x7f800L) {
        jbg_buf_write((local_c >> 11) & 0xff, file);
        if (((local_c >> 11) & 0xff) == MARKER_ESC)
          jbg_buf_write(MARKER_STUFF, file);
      }
    }

  sync_mem.write_send((long)local_c, plane, ADDR_C);
  sync_mem.write_send((long)local_a, plane, ADDR_A);
  sync_mem.write_send((long)local_ct, plane, ADDR_CT);
  sync_mem.write_send((long)local_buffer, plane, ADDR_BUFFER);
  sync_mem.write_send((long)local_sc, plane, ADDR_SC);

   return;
}
};
```