The port of MGI DVDMax software from Linux to Solaris

Nasser M. Abbasi

Sept 18, 2000 page compiled on July 2, 2015 at 5:05pm

Contents

List of Figures

1 Introduction

DVDMax was MGI Inc. software for playing DVD's.

Current version on windows is version 5.0. A linux version is currently under development and in final stages.

The purpose of this report is to describes the design and implementation of the DVDMax software on Linux to help in the port of the software to SUN Solaris operating system.

The port of the DVDMax software to Solaris will be based on the Linux version.

2 Software requirements

The following is from the statement of work for DVDMax on Solaris

> The features of the sunDVD player will be the same as the features listed in the product description SoftDVDMax, entitled "SoftDVD Max Reviewer's Guide (july 1999)" and "MGI SoftDVD Max specifications (July 1999)" copies of which are attached to this statement of work.

Examining the above mentioned document, I come up with the following points.

1. Auto detect DVD drive.
2. Allow window size reduction.
3. Parental control panel. (works if supported on DVD disc).
4. Supports following audio setting:
   (a) Stereo 2 channel output.
   (b) Surround Sound. 2 channel Dolby pro-logic output.
   (c) 3D audio.
   (d) Use DirectSound (windows specific I assume).
   (e) Generate S/PDIF output. (Sony/Phillips Digital Interface format).
   (f) speed up decoding via subsampling.
   (g) Karoake vocals (if dvd disc supported)
5. Support 16:9 and 4:3 aspect ratios. These are the configurations
   (a) Auto. Display 4:3 video in 4:3 screen. Display 16:9 video in 16:9 screen.
   (b) letter box. Allows display of 16:9 in 4:3 screen.
   (c) Pan and Scan. Display 16:9 to 4:3 using pan and Scan process.
   (d) Wide screen. Display 4:3 video in 16:9 window.
6. Advanced features.
   (a) Automatic adjust of resolution for optimum performance.
   (b) Allows hardware motion compensation acceleration (if graphic controller supports it)
   (c) Deinterlacing.
   (d) Auto detect interlaced and non-interlaced content and select Bob or Weave.

3 Main port issues

These are the main port issues that I see right now for a successfully port of the Linux based DVDMax to Solaris.

1. Build issues. Initially, the same build process used on Linux will used on Solaris, making only the necessary changes to get a complete build. The goal is to make minimal changes to the build process initially in order to speed the port. The current build process on Linux is based on heavy use of *GNU* tools such as:

   (a) bison
   (b) cvs (for source control)
   (c) flex
   (d) autoconf
   (e) m4
   (f) gawk
   (g) gcc
   (h) libtool

   All the above tools needs to be installed on Solaris before starting to build DVDMax.

   To minimize changes to the build process, and to speed the port process, gcc will be used initially.

   After a successfully completion of the port, gcc can be replaced with SUN cc compiler along with any changes to cc commands and options used in the current Makefiles.

   There will be changes needed in the makefiles in order to build Sparc assembler routines using VIS instructions. Those are the modules that will replace the current intel assembler modules. My current understanding is that the Solaris build will target *UltraSparc* with *v9a* extensions, so the option *-Xarch=v9a* will be used to assembler the Sparc assembler modules.

   The resulting executables will only run on Solaris with 64-bit Kernels on UltraSprac hardware. [1]

   Another changes in the current Makefiles that needs to be made are those to link against the Solaris *libdvd* and, if needed, against *MediaLib* libraries. The Makefiles that come with the examples of how to use the above libraries will be used to help in making the changes.

   We also need to find a way to make the build system easier to use on multiple operating systems. One possible way is to use autoconf macro to guess the OS name, and based on the OS name, generate the correct compile and link options, and any compile time variables, such as *-DSOLARIS* into the compile command and have those automatically generated when the *configure* command is run. This needs to be investigated more.

2. User interface issues. The current user interface on linux uses *Qt* libraries. It also uses the *skin* technology to modify the look of the user interface dynamically.

   On Solaris *Motif* will be used instead on *Qt*. But with latest announcement by SUN that they will adopt *GNOME* as the default desktop for feature Solaris releases, it seems that it will make more sense to use *GTK+* for the user interface. [2].

   There will also be an impact on the build process here for build the user interface, a new Makefiles will be needed to build the user interface on Solaris since different libraries are used than on Linux.

3. DVD Authentication software (CSS). On Linux, *DeCSS* is used for the authentication of the DVD and used many Linux ioctls() and linux specific header files such as `<linux/cdrom.h>`

   On Solaris a new implementation will be used. This needs to be coordinated and integrated into the Solaris DVDMax build process. My understanding is that a third party will write this software.

[1] verify these requirements with SUN.
[2] discuses the Motif/GTK+ options with SUN to verify, also verify if current skin used on Linux will work with GTK+ on Solaris

4. DVD Keys. It is expected that SUN will supplies these keys. How/when? How will this affect the build? I need to understand more about this.

5. Audio output. Now, on linux, OSS is used to output audio, and it used Linux specific ioctls and header files. Need to understand how will audio be output on Solaris. Direct interface to `/dev/audio` How will audio control be done? Note that `/dev/audio` can only be opened by one process at a time. Also Solaris does support asynchronous IO on `/dev/audio`. It is expected that audio output on Solaris will be a new implementation from that of Linux, but this needs to be analyzed more.

6. Video output. `libdvd` will be used on Solaris for display of video bitstream. This will be a new implementation from that on Linux. Initially, I will try to use the X11 implementation that exist already on Linux as is on Solaris to see that will work, and make changes to libdvd after that.

   Linux implementation of display have an *MMX* module for doing color space conversion from YUV to RGB.

7. MPEG-2 decoder. This is the most difficult software component to port to Solaris/Sparc. MPEG-2 decoder implementation on Linux, which is based on the windows version, is heavily dependent on intel assembler and *MMX* instructions. This is done to achieve the maximum performance.

   The Solaris implementation will use *VIS* sparc instructions to achieve the maximum performance on Sparc hardware. Hence a conversion of the Intel assembler and *MMX* to Sparc assembler and *VIS* will be the most critical work that needs to be completed successfully.

   I outline in details the current Linux design and implementation of MPEG-2 decoder below with the interfaces to the Intel assembler in order to help with this conversion. It is expected that SUN will do the actual conversion based on these interfaces. [3]

8. Standard integer types include file. On linux, those are defined in /usr/include/stdint.h while on Solaris, those are obtained by including /usr/include/inttypes.h. An `#ifdefined` will be used to include the correct header file at compile time.

9. UDF file system interface for reading DVD discs. The C module dvd2/src/common/udf.c is the interface used by DVDMax to access the UDF file system on the DVD disc. It is possible that some changes will be needed when porting this to Solaris. udf.c uses large file programming environments and uses such calls as *fsetpos64* to position itself in the DVD disc file system.

   udf.c is used by the DVDDisc filter and the IFO implementation.

10. Threads yielding issues. The Linux based code uses *usleep()* to yield, because linuxthreads does not have a pthread_yield(). Solaris has a pthread_yield(), so it might be more efficient to use on Solaris that to force thread context switching than calling usleep().

11. The timing synchronization code in *dvd/src/filters/syncmaster/mpeg2sync.c* needs to examined to make sure it is correct under Solaris/Sparc and that there is nothing Linux specific in that code.

Design of DVDMax on Linux

4 High level Architecture

The DVDMax software on linux is based on individual components, called *filters*. Each filter has what is called *pins*. These are software connection that connect the input of one filter to the output of another filter.

Each filter takes some input, process the input, and sends the output to one of its output pins. A filter can have more than one output pin. For example, the *demux* filter have a number of output pins.

A plugin shared library is loaded at run time by the main program of DVDMax, each plugin create one type of filter. So there is one-to-one mapping between the plugin and the filter it creates. For example, the dvddisk.so

[3] any one knows of an Intel/MMX to Sparc/VIS conversion software? :)

plugin creates the filter that reads from the dvd disc and write to the demux filter. The actual methods that implement the work of the filter are in the plugin, the filter is the data structure that is used to interface to those methods. The filter also contains pointers to other variables important for the working of the plugin, such as buffers and flags.

Each plugin contains one pthread thread than continuously runs reading input and generating output. [4]

Figure 1 shows the architecture of DVDMax on Linux and Solaris. The grayed components indicates an assembler/MMX intel modules.

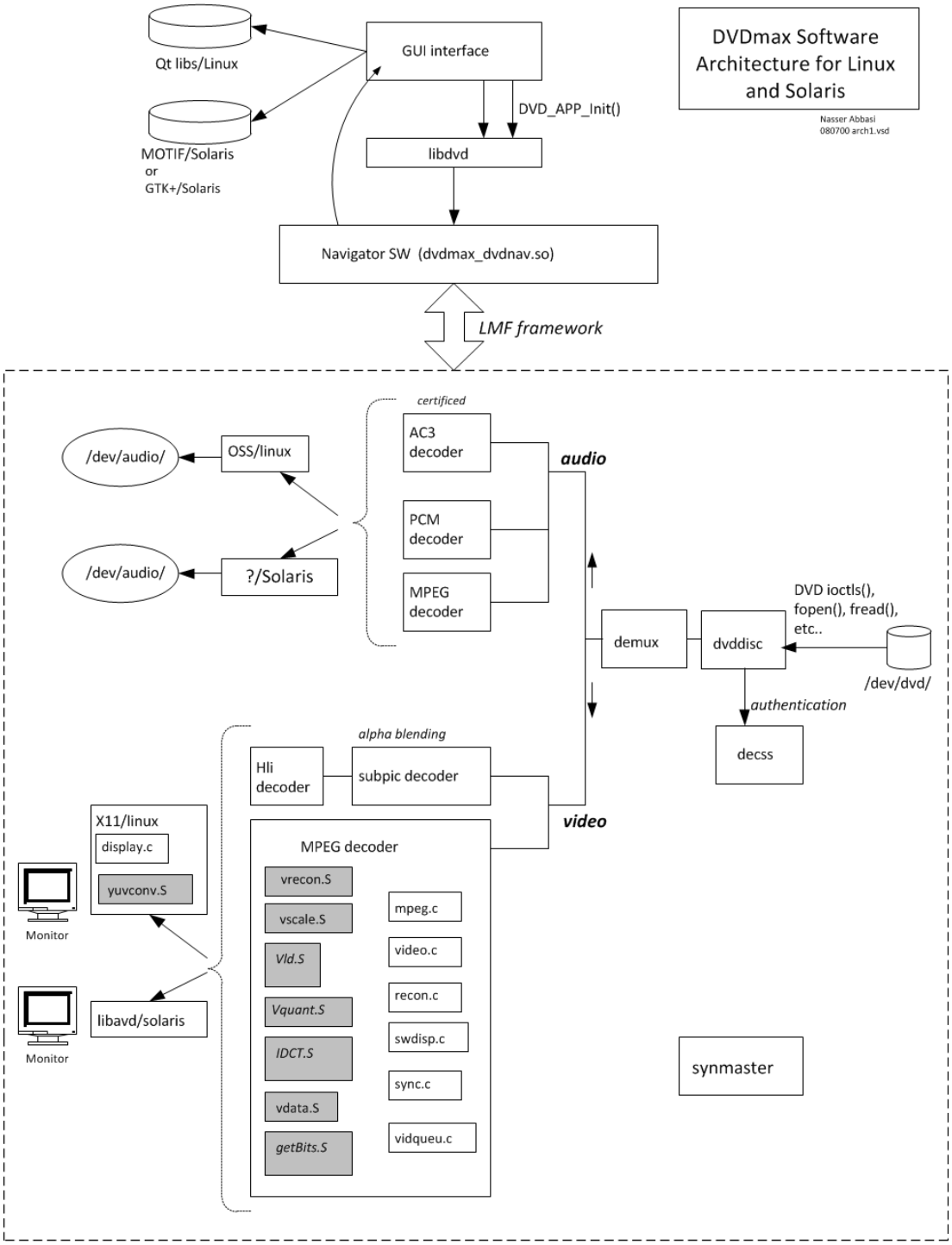


Figure 1: Software architecture of DVDMax on Linux and Solaris. arch1.vsd

The unit of data exchanged between filters is *LMFpacket struct*. Figure 2 on the next page shows how an

[4]On Linux, the Audio output filter, based on OSS, does not have a thread in it

LMFPacket is send from the one of *demux* filter output pins that is connected to the mpeg-2 decoder input pin.

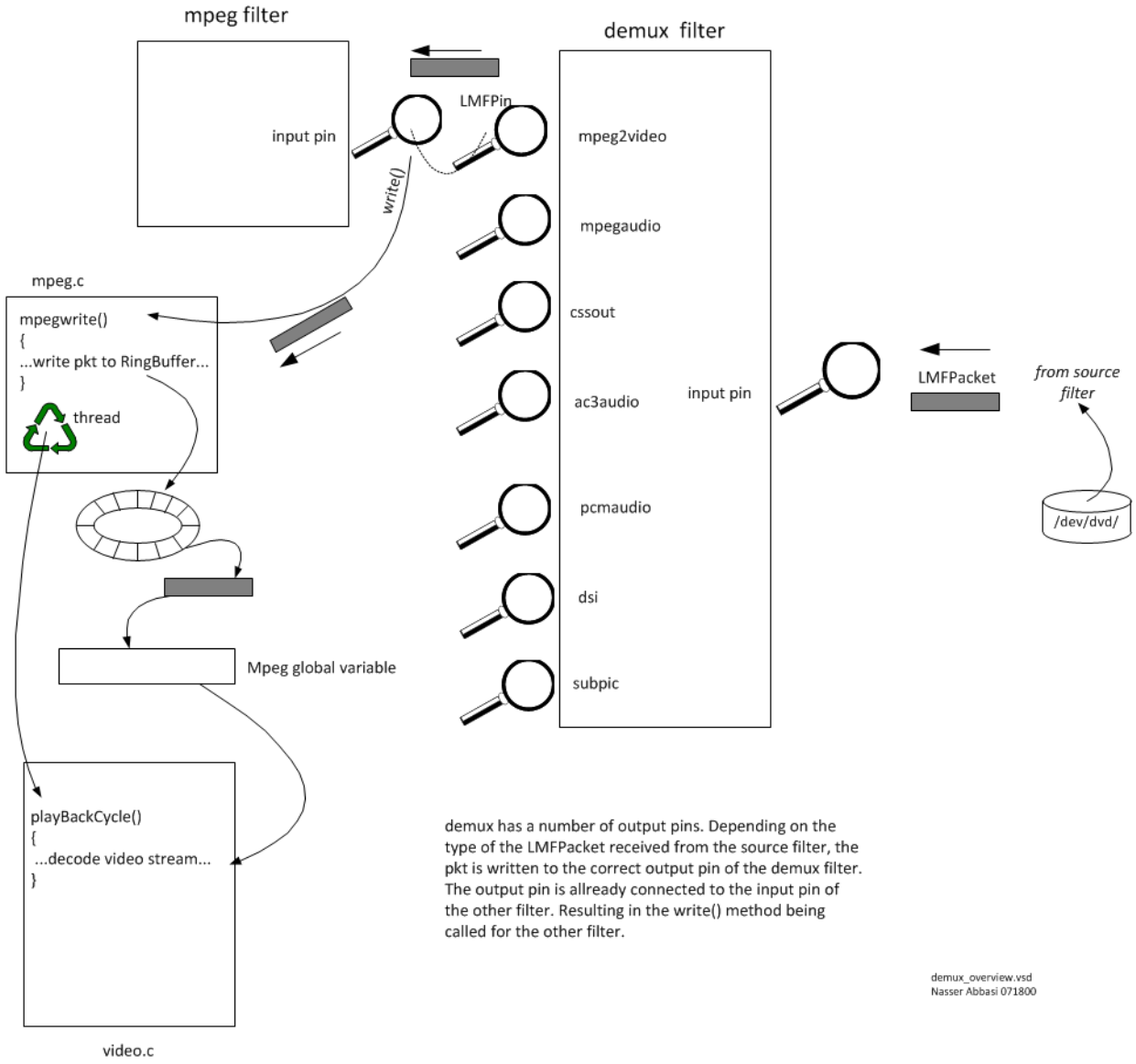


Figure 2: Internal data flow between filters in DVDMax demux_overview.vsd

5 Loading of a plugin and creation of a filter object

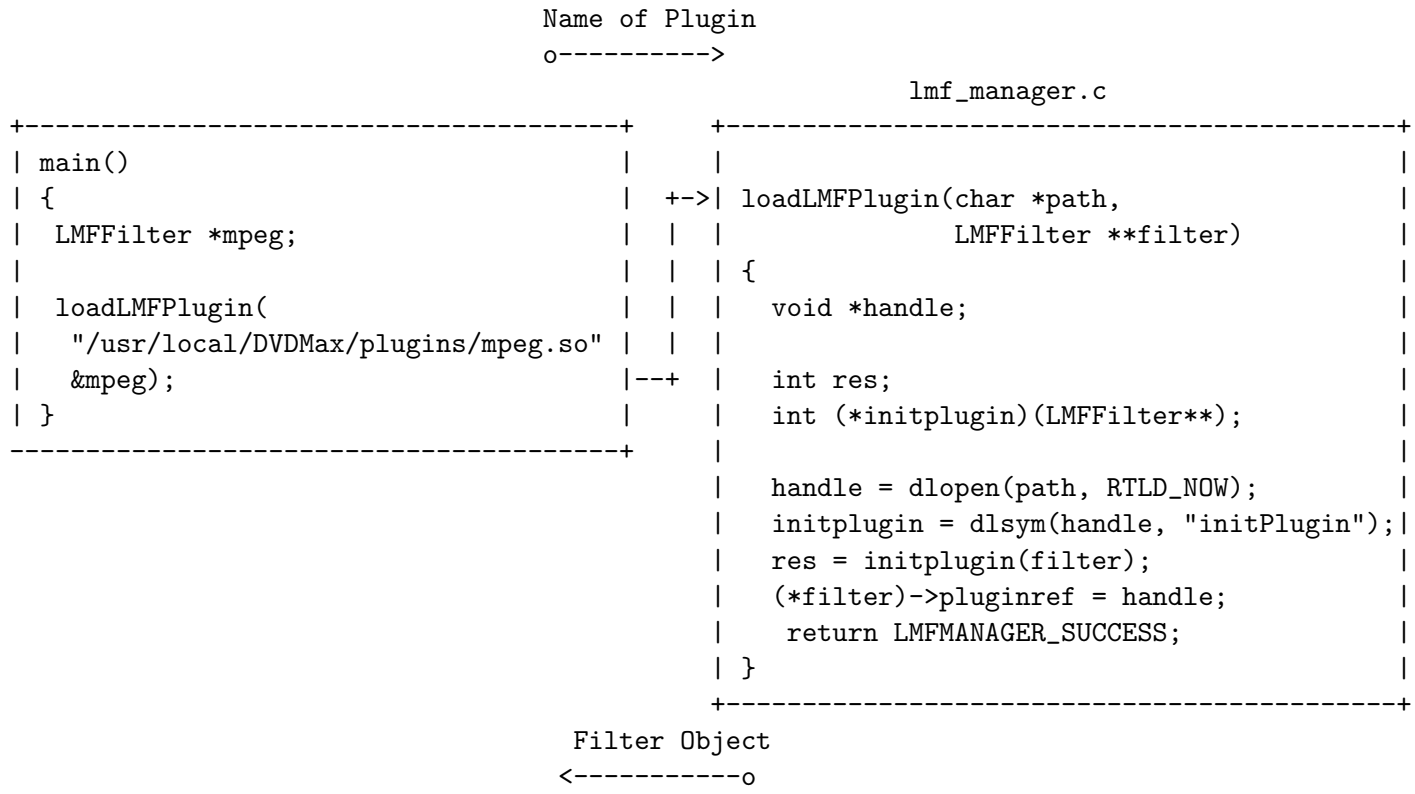
A user program main() starts by a call to lmf_manager.c/loadLMFPlugin(), passing it a string name of the plugin (the sharable module) to load. When this call returns, it return back a pointer to a filter object associated with this plugin.

loadLMFPlugin() uses dlopen() to load the sharable library to memory, then dlsym() is used to obtain a pointer to the function called 'initPlugin' in that module. So each plugin must have such an entry point.

Then initPlugin() is then called, which creates the filter object. The filter is then returned to the caller. Notice that the handle to the plugin sharable library is saved in the filter object, so it is not lost.

Each filter has an initPlugin(), which calls lmf_filter.c/createLMFFilter() to create the Filter structure.

Then another call to lmf_filter.c/initLMFFilter() is made to do any initialization to the filter struct and which will setup the command table in the filter, which contains pointers to default functions in the lmf_filter.c module.



The plugin simply allocates a filter struct from the heap. The filter struct contains a table of function pointers. These functions are inside the plugin and the filter is passed around during the calls. The filter struct contains pointers to any data buffers used by the plugin.

6 MPEG-2 decoder design and dataflow

The mpeg decoder is a filter, it contains a pthread thread, it reads its input from a ring buffer, each entry in the ring buffer is a pointer to data of type *LMFPacket*. The ring buffer is written to by the demux filter via the connection between the output pin of the demux filter to the input pin of the mpeg filter. The ring buffer is protected by a pthread mutex against concurrency access by more than one thread at a time.

The mpeg thread runs all the time, it calls video.c/PLayBackCycle(), which in turn calls getvideo.c/requestConsecutiveVideoBytes(), which in turn calls mpeg.c/GetVideoInputBytesFromFile() which removes bytes from the ring buffer, and write them to the global MPEG buffer.

So, when it returns, video.c has stream data in the global MPEG buffer to process.

Figure 3 on page 25 shows the mpeg filter and related data structures involved.

6.1 Assembler modules used by mpeg-2 decoder

The mpeg decoder contains a number of assembler modules. They are:

1. mmxGetBits.S: MMX optimized version of the bit streaming routines. Services offered are functions to consume, show, initialize and swap bits.
2. mmxRecon.S: mmx version of software motion comp reconstruction
3. vdata.S: Defines data structures used by assembly language modules. Optimized for 4-way, 16kbyte data cache.
4. vidct.S: Computes IDCT on 8x8 array of DCT coefficients. Optimized for Pentium MMX.
5. vld.S: Decodes MPEG Variable Length Code blocks into 8x8 arrays. Optimized for Pentium II.
6. vquant.S Dequantizes, scales, and clamps output arrays from VLD. Requires Pentium MMX or Pentium II
7. vrecon.S Video reconstruction and averaging routines. Requires Pentium MMX or Pentium II
8. vscale.S Scales IDCT results to suit formats required by software and hardware motion comp. Optimized for Pentium MMX.

6.2 general logic flow in the mpeg-2 decoder

Life starts when the runfilter() thread is started, this is the mpeg-2 internal thread, which continues to loop calling playbackcycle() in video.c.

When playbackcycle() is called, it calls GetVideoStartCode() in video.c which in turn looks into the global vd variable, if it needs video stream data to process, it calls requestConsecutiveVideoBytes() in getvideo.c to get the required number of bytes.

requestConsecutiveVideoBytes() will call GetVideoInputBytesFromFile() in mpeg.c to move the required bytes from the mpeg-2 ring buffer to the global vd variable. If the Ringbuffer is empty, It will block waiting.

During the decoding of the bit stream. many calls are made to MmxGetBits.S (MMX instructions) for parsing video bit stream.

When GetVideoStartCode() returns, playBackCycle() continues by doing a large switch statement on the picture code (in the global vd.i.startcode). Looking at one case, when a picture() start code is detected, picture() is called (in video.c).

picture() parses the picture stream, it parses each macroblock, there is a LOOP over all macroblocks, each time it needs to decode a macroblock, it calls SampleProcessMacroBlock() in samplemc.c. This assumes that samplemc.c is the modules used to driver the actual decoding at the assembler level. There are two main C modules for doing this, one is samplemc.c which the diagram below is based on, and another one called fastmc.c which I'll will into in more details later on. These C modules interface to the assembler modules for doing the actual decoding in assembler.

SampleProcessMacroBlock() finds the type of the block. Motion compensation is first done by making calls to 'recon()' C routine, which ends up calling 'recon_comp()' in recon.c, in this file, there is a #if USE_MMX_FOR_RECON to decide if motion compensation is done using MMX or plain C. If MMX is to be used, MMX routine in recon.S is called.

When motion compensation is done, MMX routines in Vscale.S are called to decode the blocks. Either IntraVldIdctEightBitOutput() is called, or NonIntraVldIdctNineBitSun() is called.

MMX instructions in Vscale.S calls _intraVld or _NonIntraVld MMX routine in vld.S to Decodes MPEG Variable Length Code blocks into 8x8 arrays. After the routines in Vld.S return, Vscale.S calls _IntraQuant or _NonIntraQuant MMX routines in vquant.S to Dequantizes, scales, and clamps output arrays from VLD.

vquant.S MMX routines in turn jump to the idct MMX routine in idct.S to Computes IDCT on 8x8 array of DCT coefficients.

When this is all done, and when end of picture is reached, then SampleEndingPictureMC() in samplemc.c is called. This in turn called QueueForDecodeAndDisplay() in vidqueue.c to queue the decoded frame. This ends up calling SampleRenderingFunctionMC() in sampelmc.c which call DecodedYCrCbToDisplay() in swdisp.c to display the frame.

In swdisp.c, there is a queue where the decoded frame is send to the output filter (X11 filter for example). Which will actually display the picture to the display. Notice that color mapping conversion is done in the output filter and not by the mpeg decoder.

Figure below shows the main dataflow in the mpeg-2 filter.

6.3 Decoding macroblocks

The core of the decoder is in decoding macrobloacks. This is in samplemc.c, in the function SampleProcessMacroBlock() or in fastmc.c in the function FastSoftwareProcessMacroBlockMC() depending on how the build was done.

Figure **??** on page 27 shows the algorithm used.

6.4 Assembler interfaces in mpeg decoder

The decoding process goes through these steps

1. Motion compensation for non Intra blocks.
2. Variable length decoding VLD.
3. Dequantizes, scales, and clamps output arrays from VLD.
4. IDCT.
5. color mapping conversion from YUV to RGB before display.

The mpeg is divided in two main section, the C modules does the high level processing, such as reading the bit stream, locating the macroblocks, deciding on the type of the picture and type of prediction needed. Once the macroblock is found and needs to be decoded, the assembler routines are called to do the process. The interface between the C modules and the assembler modules can be looked at as being the fastmc.c module, or the samplemc.c modules depending on the build parameter used (only one of those can be used).

Figure **??** on page 28 illustrate the above. It shows that the C modules share C based global variables, and that the assembler modules share assembler based data buffers and tables. Also, the assembler routines have access to the C based buffers.

6.4.1 mmxGetBits.S interface

mmxGetBits is used to obtain, examine, and skips bits in the video bit stream. It is the main interface to access the video bit stream during the decoding process.

The bit stream is accessed via global pointer *vd.i.puDword*, 2 MMX registers are used to store the top 128 bits in the bit stream. [5]. The symbolic names of these 2 MMX registers is FIRST and SECOND.

Another MMX register, with a symbolic name of COUNT is used to store the number of bits consumed in the FIRST register. The value in the COUNT register is saved in memory in the variable *vd. i.bitsUsedInDword*.

Another MMX register with symbolic name of SOURCE is used to contain the address the top of the bit stream, and is advanced by 8 bytes at a time. The value of this register is saved in memory in the variable *vd.i.puDword*.

The C interface to the mmxGetBits.S is as follows

```
signature: unsigned int GetVideoBitsSmall(int numberOfBits)
semantics:  returns back the number of bits requested from the video
           stream, and consumes them. internally it updates the MMX
           registers COUNT and FIRST and SECOND, and SOURCE.

           Notice that a maximum of 32 bits can be returned per call,
           since this is the sizeof unsigned int.


signature: unsigned int NextVideoBitsSmall(int numberOfBits)
semantics: Similar to  GetVideoBitsSmall() function, expect the bits
           returned are not consumed, i.e. the COUNT MMX register is not
           advanced. Also, the SOURCE, FIRST and SECOND MMX registers are
           not modified. This is like a 'peek' call, just to see the bits
           in the bit stream, without advancing.

           Notice that a maximum of 32 bits can be returned per call,
           since this is the sizeof unsigned int.


signature: void SkipVideoBitsSmall(int numberOfBits)
semantics: Skips the numberOfBits bits in the bitstream.
           No bits are returned, but bits are consumed. the MMX registers
           COUNT, FIRST, SECOND, and SOURCE are all modified.


 signature: void initGetBits(unsigned int *ptr)
 semantics:  This is the first call to use to initialize the mmxGetBits.S module.
            It will initialize the MMX registers COUNT, FIRST, SECOND and
            SOURCE to the correct values.
            The argument ptr have the value of vd.i.puDword<<2


 signature: void InputToMmx()
 semantics: This call is used to load MMX registers from the values
            stored in memory. after this call returns, the MMX registers
            FIRST, SECOND, COUNT and SOURCE contains the correct values
```

[5] Each MMX register is 64 bit long

```
                  as before. The variables vd.i.puDword and vd.i.bitsUsedInDword
                  are read and used to updated the content of the MMX registers.


        signature: void MmxToInput()
        semantics: This call is made to store the content of SOURCE register
                   (after dividing by 4 and adding 2) back in vd.i.puDword
                   and to store the content of MMX register COUNT into
                   vd.i.bitsUsedInDword
```

This is a high level version of mmxGetBits. Lets call this cGetBits.c It will have the same interface as the Mmx based functions. The purpose of this is to help show what the GetBits MMX code does, this is not meant to be working code that will compile as is.

```
static unsigned char  first[8];   /* first is MMX register in assembler*/
static unsigend char  second[8];  /* second is MMX register in assembler*/
static int count;                 /* count is another MMX register in assmbler code*/
static unsigned char *src;        /* src points to current top of buffer of bit stream*/

static _initGetBits(char *ptr)
{
   int i;
   count=0;

   for(i=0;i<7;i++)
      first[i]=ptr[i];

   ptr=ptr+8;

   for(i=0;i<7;i++)
      second[i]=ptr[i];

   src=ptr;
}



static loadBits()
{
    src= src+8;

   for(i=0;i<7;i++)
      second[i]=src[i];

   //left shift second by count INTO lower first
   // note: count here is the overflow
}

static _skipBits()
{
   count = count +n;
   // left shift first by n
   // left shift second by n into first lower first
   if ( count >= 64 )
```

```
   {
      count = count - 64;
      loadBits();
   }
}

/* gets and consumes the numberOfBits bits from the bitstream. */
unsigned int GetVideoBitsSmall(int numberOfBits)
{
   unsigned int result= /* top most numberOfBits from first */
   // shift left first by numberofbits
   //shift left second by numberOfBits INTO lower first
   count = count + numberOfBits;
   if (count >=64)
   {
      count = count - 64;
      loadBits();
   }

   return result;
}


/* Returns the numberOfBits bits from the bitstream without consuming them. */
unsigned int NextVideoBitsSmall(int numberOfBits)
{
   return top-most numberOfBits from first;
}

/* Skips the numberOfBits bits in the bitstream. */
void SkipVideoBitsSmall(int numberOfBits)
{
    count = count + numberOfBits;
    //left shift first by numberOfBits
    //left shift second by numberOfBits INTO lower first
    if(count >= 64)
    {
       count = count - 64;
       loadBits()
    }
}

void MmxToInput()
{
   vd.i.puDword = (src/4) -2 ;  // check on this
   vd.i.bitsUsedInDword = count;
}

void InputToMmx()
{
   _initGetBits( (vd.i.puDword)<<2 );
   _skipBits(vd.i.bitsUsedInDword)
```

```
}

void initGetBits()
{
   _initGetBits( (vd.i.puDword)<<2 );
}
```

Figure **??** on page 29 shows detailed walkthough of the initGetBits MMX code. Figure **??** on page 30 shows detailed walkthough of the InputToMmx MMX code. Figure **??** on page 31 shows the rest of the walkthough of the InputToMmx MMX code. Figure **??** on page 32 shows summary of InputToMmx MMX code.

Figure **??** on page 33 shows detailed walkthough of the MmxToInput MMX code, MmxToInput() basically takes the output of the opertation in MMX registers, and update the vd.i.puDword and vd.i.bitsUsedInDword.

Figure **??** on page 34 is a walk though of GetVideoBitsSmall(), it takes as an argument the number of bits to return from the video stream, and the return value will contains those bits. Since unsigned int is used for the return value, only 32 bites can be returned per each call. MMX registers COUNT, FIRST and SECOND are updated as needed for next call.

6.4.2 vrecon.S interface

The vrecon.S MMX module contains the code for doing video reconstruction and averaging routines. the C interfaces to the entry points in this module is declared in *vrecon.h* This MMX module is called from the *fastmc.c module*. Total number of lines in vrecon.S module is about 1130 including comments.

These are C interfaces from *vrecon.h*

```
void    MMX_Recon_no_motion_f();
void    MMX_Recon_full();
void    MMX_Recon_right_half();
void    MMX_Recon_down_half();
void    MMX_Recon_rightdown_half();
void    MMX_Recon_full_fb();
void    MMX_Recon_right_half_f_full_b();
void    MMX_Recon_down_half_f_full_b();
void    MMX_Recon_full_f_right_half_b();
void    MMX_Recon_full_f_down_half_b();
void    MMX_Recon_right_half_f_right_half_b();
void    MMX_Recon_down_half_f_right_half_b();
void    MMX_Recon_right_half_f_down_half_b();
void    MMX_Recon_down_half_f_down_half_b();
void    MMX_Recon_rightdown_half_f_full_b();
void    MMX_Recon_full_f_rightdown_half_b();
void    MMX_Recon_rightdown_half_f_right_half_b();
void    MMX_Recon_rightdown_half_f_down_half_b();
void    MMX_Recon_right_half_f_rightdown_half_b();
void    MMX_Recon_down_half_f_rightdown_half_b();
void    MMX_Recon_rightdown_half_f_rightdown_half_b();
```

I'll will look at one function from the above, the *MMX_Recon_no_motion_f()* to show the interface to it.

```
        MMX_Recon_no_motion_f(int motionCompStride,
                              unsigned char *from1,
                              unsigned char *to,
                              int lineStride,
```

```
                                int nYlines);

This MMX code will copy 8 bytes at a time from where 'from1' points to, to buffer
pointed to by 'to'. It does this for nYLines number of times.
```

There is a C version of vrecon.S that exist. It is commented out code sections in the same file vrecon.S, so it is possible to initially use that for Solaris. See the C code (all written as macros) for more description of what the assembler does.

6.4.3 vscale.S interface

The vscale.S module is called after motion compensation. There are number of interface to this module, however, only two are used. One for non-intra blocks, and one for intra blocks.

These are the entry points to the vscale.S module:

```
.globl IntraVldIdctSevenBitShiftedOutput
.globl NonIntraVldIdctEightBitShiftedSum
.globl IntraVldIdctEightBitOutput
.globl IntraVldIdctEightBitSignedOutput
.globl NonIntraVldIdctNineBitSum
.globl NonIntraVldIdctEightBitShiftedOutput
.globl NonIntraVldIdctSixteenBitOutput
```

For intra blocks, *IntraVldIdctSevenBitShiftedOutput* is called.

For non-intra *NonIntraVldIdctEightBitShiftedSum* is called.

vscale.S does not access C based structures, but will access the 8x8 IDCTbuffer defined in data.S assembler module.

6.4.4 vld.S interface

For intra blocks, the entry point is *_intraVld* and for non-intra blocks, the entry point is *_NonIntraVld*. These MMX entry points read from C based global structures and read and write to assembler based buffers such as the IDCTbuffer, VLCTable0, VLCtable1, VLCTable2, VLCTable3, DCluma buffer, DCShift.

The C based fields in C structures that this assembler code reads are fields in a structure of type struct MPEG_VIDEO_VLD_VARIABLES_STRUCT declared in vld.h and type struct MPEG_VIDEO_INPUT_DATA_VARIABLES declared in video.h.

The C fields read from struct MPEG_VIDEO_VLD_VARIABLES_STRUCT are: flag mc_intraBlockIsLumFlag, flag dPictureFlag, intra_vlc_format, macroblockIsIntraFlag, mpeg2IfNotZero.

The C fields written into struct MPEG_VIDEO_VLD_VARIABLES_STRUCT are: vldLimitOverflowFlag.

The C fields read from struct MPEG_VIDEO_INPUT_DATA_VARIABLES are: bitsUsedInDword, puDword, bitsUsedInDword.

The C fields written into struct MPEG_VIDEO_INPUT_DATA_VARIABLES are: bitsUsedInDword.

6.4.5 vquant.S

Dequantizes, scales, and clamps output arrays from VLD.

There are two main entries into this assembler module. For Intra blocks it is *_IntraQuant* and for non-intra blocks, it is *_NonIntraQuant*.

This assemble module access C based global variables, and assembler based buffers and tables.

The C based variables accessed are fields in global variable vld, which is of type MPEG_VIDEO_VLD_VARIABLES_STRUCT. The variable vld itself is a field in a larger variable, vd, of type MPEG_VIDEO_DECODER_VARIABLES_TYPE that is allocated in the module video.c.

The C based fields in struct MPEG_VIDEO_VLD_VARIABLES_STRUCT accessed in vld.S are: intra_dc_precision (which can be 1,2,4 or 8), mc_pDcPredictor (ptr to block), psIntraQuantMatrix (ptr to Intra quantizing matrix), intraQuantMatrixScale, psNonIntraQuantMatrix (ptr to non-intra quantizing matrix), nonIntraQuantMatrixScale, quantizer_scale(global quantizer scale).

Assembler based buffers accessed are located in data.S module, and they are: IdctColumnMask, 8x8 IDCTBuffer (read/write access).

6.4.6 vidct.S interface

Computes IDCT on 8x8 array of DCT coefficients.

The entry point in this module is idct, which is called to do idct on the 8x8 IDCTBufger.

The C based variables read in this module are fields in structure of type `MPEG_VIDEO_VLD_VARIABLES_STRUCT`, and these are: `mc_IdctblockDestinationStride`, and `mc_pucBlockDestination`.

The assembler based buffer accessed are: 8x8 IDCTBuffer. This contains the dct coefficients, to perform idct on. the DCSTEP and various other assembler based constants, all of these are defined in data.S.

idct stores 16 bits final results in MMX registers, then it calls the output routine, which clamps the results, scales them to a specified precision, and stores or sums the results into 8-bit, pre-configured C based buffers.

The output routine for intra blocks is called Out7BitIntra in the assembler module vscale.S, and the output routine for non-intra blocks is called Sum8BitNonIntra in the assembler module vscale.S.

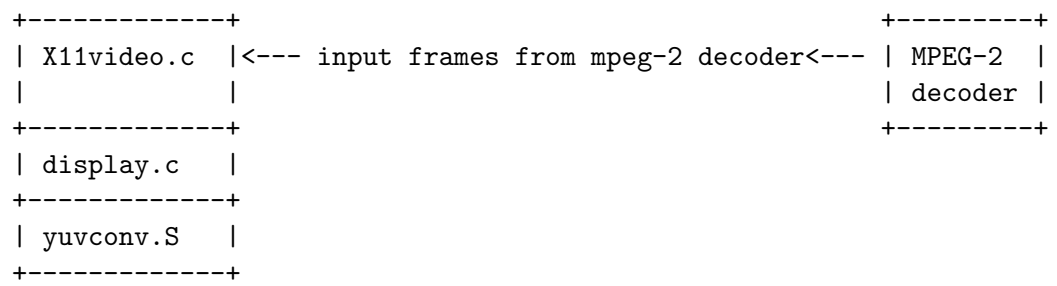
7 Video output filter design

The X11 output filter takes as input a decoded picture frames from the mpeg-2 decoder (or subpic decoder), and will display the frame. Currently, the output filter will do YUV to RGB conversion using an MMX module.

The X11 output filter is located in `dvd2/src/filters/sink/video/x11/` directory in the source tree.

The main filter is implemented in the file x11video.c. Other supporting files are display.c which has functions that are called from X11video.c to actually display the frames, and yuvconv.S, which is an MMX modules that does the YUV to RGB conversion. There are two versions of YUV to RGB conversions, one with alpha blending and one without.

The MMX code is called from the display.c module to do the conversion before displaying.



The video output filter is build in one of 5 modes. Once selects the mode to build the filter in by manually editing the file build.h in the same directory, and setting the variable `OUTPUT_MODE` to the mode needed. Looking at build.h we see:

```
/* the different types of output modes.  Selected with OUTPUT_MODE */
#define OUTPUT_MODE_VIDMEM      2
#define OUTPUT_MODE_XSHM        3
#define OUTPUT_MODE_SDL         4 /* not implemented fully (requires addition
                                     of -lSDL in the makefile) */
#define OUTPUT_MODE_DGA         5
#define OUTPUT_MODE_I810        6

/* the mode selected */
#define OUTPUT_MODE OUTPUT_MODE_XSHM
```

7.1 VIDMEM mode for video display

If OUTPUT_MODE_VIDMEM is selected, the the device /dev/agpgrat is opened and used to write to. The device is opened, then queried using an IOCTL call to compute the video memory size. Then the device is memory maped using mmap() call for the calculated size.

The result of the mmap() call is to return a memory pointer which is the mapped memory the the device accessable memory to write the frame to. This memory address is used in the function DisplayFrame() by the x86 instructions to move the frame buffer to the device /dev/agpgrat/ mapped buffer as shown below

```
dest=vidmem;   /* vidmem is address of mapped graphic device memory */

if(vfbd->blendframe!= ((void *)0) ){
    CALL_YUV_ALPHA(yval,
                   cbval,
                   crval,
                   dest,      /* address of mapped memory */
                   vfbd);
```

```
}
else
{
    asm("movl %3,%%edi\n" "call yuv_convert" :: "S" (yval),
         "c" (cbval),
         "d" (crval),
         "m" (dest),      /* address of mapped memory */
         "a" (bobFlag) : "%esi","%ecx","%edx","%edi","%eax"); ;
}
```

7.2 DGA mode for video display

If OUTPUT_MODE_DGA is used, the file /usr/X11R6/include/X11/extensions/xf86dga.h is included and the build is linked to shared library xf86dga.so. The process of using direct graphics calls is initialized using the following sequence of calls to function in the xf86dga.so library.

```
    display=XOpenDisplay(((void *)0) );
    screen= (((_XPrivDisplay) display )->default_screen) ;

    int min,maj,flags;

    XF86DGAQueryVersion(display,&min,&maj);

    XF86DGAQueryDirectVideo(display,screen,&flags);

    XF86DGAGetVideo(display,screen,(char**)&vidmem,&vidmem_width,&vidmem_size,&ram_size);

    XF86DGAGetViewPortSize(display,screen,&sizex,&sizey);

    XF86DGADirectVideo(display,screen,0x0002 );
}
```

To output a frame using direct graphics mode, after calling YUV to RGB conversion, calls to XF86DGASetViewPort() are made as shown

```
dest=whichpage ? page2 : page1;

if(vfbd->blendframe!= ((void *)0) ){
    CALL_YUV_ALPHA(yval,cbval,crval,dest,vfbd);
}else{
    asm("movl %3,%%edi\n" "call yuv_convert" ::
    "S" (yval),
    "c" (cbval),
    "d" (crval),
    "m" (dest),
    "a" (bobFlag) : "%esi","%ecx","%edx","%edi","%eax"); ;
}

if(whichpage==0){
    XF86DGASetViewPort(display,screen,0,0);
    whichpage=1;
}else{
```

```
    XF86DGASetViewPort(display,screen,0,vidmem_height);
    whichpage=0;
}
```

The screen is closed in DGA mode by making a call to XF86DGADirectVideo(display,screen,0).

7.3 intel I810 mode for video display

The intel i810 graphic card has the following features (obtained from the net http://www.xfree86.org/4.0/i810.html).

- Full support for 8, 15, 16, and 24 bit pixel depths.
- Hardware cursor support to reduce sprite flicker.
- Hardware accelerated 2D drawing engine support for 8, 15, 16 and 24 bit pixel depths.
- Support for high resolution video modes up to 1600x1200.
- Fully programmable clock supported.
- Robust text mode restore for VT switching.

Hardware acceleration is not possible when using the framebuffer in 32 bit per pixel format, and this mode is not supported by this driver.

Interlace modes cannot be supported.

This driver currently only works for Linux/ix86, and normal use requires the agpgart.o kernel module, included in Linux kernels 2.3.42 and higher.

This mode requires mgilib.h which I was not able to find in the source tree. (ask Ben on that).

some of the functions called in the mgilib when running in i810 mode are:

mgiGetDriverInfo(), mgiMapDriverInfo(), mgiStartOverlay(), mgiCloseOverlay().

(Need to find more information on this mgilib).

7.4 Shared memory mode for video display

In this mode, we use functions as defined in X11 extention `/usr/X11R6/include/X11/extensions/XShm.h`

To load the display window in XSHM mode we create a shared memory segment and map it to the window created as shown:

```
XSizeHints hint;
int screen;
XEvent xev;

display=XOpenDisplay(((void *)0) );
screen= (((_XPrivDisplay) display )->default_screen) ;
gc= ((&((_XPrivDisplay)  display  )->screens[  screen  ]) ->default_gc) ;

window=XCreateSimpleWindow(...);
XSetStandardProperties(display,window,
    "DVDMax",
    "DVDMax",
    0L ,
    ((void *)0) ,
    0,
    &hint);
```

```
        XSelectInput(display,window,(1L<<17) );
        XMapWindow(display,window);

        do{
            XNextEvent(display,&xev);
        }while((xev.type!= 19 ) || (xev.xmap.event!=window));

        XSelectInput(display,window,(1L<<17)  | (1L<<2) );
        printf("window created\n");
    }
    int pixtype;
    int datalen;

    CompletionType = XShmGetEventBase(display) + 0 ;
    pixtype=XShmPixmapFormat(display);

    image = XShmCreateImage(
                display,
                0L ,
                16,
                pixtype,
                ((void *)0) ,
                &shminfo,
                720 , 480 );

    datalen=image->bytes_per_line*(image->height+2);
    shminfo.shmid = shmget(((__key_t) 0) , datalen, 01000 |0777);
    shminfo.shmaddr = shmat(shminfo.shmid,((void *)0) ,0);

    image->data=shminfo.shmaddr;
    shminfo.readOnly= 0 ;
    XShmAttach(display,&shminfo)
}
```

To close the display in XSHM mode we detach from the shared memory segment and then use X call to destroy the display.

```
    XShmDetach(display,&shminfo);
    XDestroyWindow(display,window);
    shmdt(shminfo.shmaddr);
```

To display a frame in XSHM mode we make a call to XShmPutImage() followed by a call to XSync().

```
    unsigned char *crval = vfbd->baseframe->planes[2 ];
    unsigned char *cbval = vfbd->baseframe->planes[1 ];
    unsigned char *yval = vfbd->baseframe->planes[0 ];

    unsigned char *dest;

    dest=image->data;

    if(vfbd->blendframe!= ((void *)0) ){
```

```
        CALL_YUV_ALPHA(yval,cbval,crval,dest,vfbd);
    }else{
        asm("movl %3,%%edi\n" "call yuv_convert" :: "S" (yval),
        "c" (cbval), "d" (crval), "m" (dest), "a" (bobFlag) :
        "%esi","%ecx","%edx","%edi","%eax"); ;
    }

    XShmPutImage(
        display,
        window,
        gc,
        image,
        0,0,
        0,0,
        image->width,
        image->height,
        0
    );
    XSync(display,0 );
```

7.5 SDL mode for video display

The SDL functions used when running in this mode are:

```
int SDL_Init(Uint32 flags); void SDL_Quit(void); int SDL_LockSO(void);
void SDL\_Quite(); SDL\_UpdateRect(...);
```

To load the display window in SDL we do

```
SDL_Init(0x0020);
screen = SDL_SetVideoMode(720 ,480 +2,16,0x00000000 | 0x00000002 );
```

To display a frame in SDL mode, do

```
dest=(unsigned char *)screen->pixels;

if(vfbd->blendframe!= ((void *)0) ){
     CALL_YUV_ALPHA(yval,cbval,crval,dest,vfbd);
}else{
     asm("movl %3,%%edi\n" "call yuv_convert" ::
         "S" (yval), "c" (cbval), "d" (crval),
          "m" (dest), "a" (bobFlag) : "%esi","%ecx","%edx","%edi","%eax"); ;
}
SDL_UpdateRect(screen,0,0,720 ,480 );
```

The SDL functions are implemeted in the directory dvd2/src/filters/sink/video/sdl/ in the source tree.
Building DVDMax
These are the steps I did to build DVDMax on Solaris.

1. download the following packages from my web site at MGI and install using the commands

```
gunzip file.gz
pkgadd -d file
```

where *file.gz* is any one of the following

- cvs-1.10.7-sol8-sparc-local.gz
- bison-1.28-sol8-sparc-local.gz
- flex-2.5.4a-sol8-sparc-local.gz
- autoconf-2.13-sol8-sparc-local.gz
- m4-1.4-sol8-sparc-local.gz
- gawk-3.0.4-sol8-sparc-local.gz
- make-3.79-sol8-sparc-local.gz
- pkg SMCgcc for gcc, this is already installed on Solaris 8 before building. If we later decide to go with Sun CC, then need to remove this package. If unable to find the solaris SMCgcc already installed on Solaris8, use gcc-2.95.2-sol8-sparc-local.gz (I did build with gcc-2.95.2).

2. download `fileutils-4.0i_build_solaris8.tar.gz`. I've already pre-build fileutils for Solaris8/sparc with my changes to the install.c to ignore -C option (which is used by our Makefiles, this is until I find out how to make autoconf not generate -C option for install)..

   Simply gunzip and tar xf the above, and run 'make install' from the top level directory. If for some reason you get an error `/usr/local/bin/install` not found, then from the top level directory of fileutils do this

   ```
   cp /src/ginstall /usr/local/bin/install
   make install
   ```

3. download `libtool-1.3.5_build_solaris8.tar.gz`. I've already prebuild this for solaris8/sparc. simply extract and run `make install` from its top level directory.

4. download `qt-2.2.0-beta2_build_solaris8.tar.gz`, I've already pre-build this for Solaris8/sparc, so simply extract only (i.e. gunzip followed by tar xf). No need to install this. The only thing needed is to set an env. variables to point to where it is located.

   Assume this is installed in `/home/nabbasi/data/QT_downloads/qt-2.2.0-beta2` then add this to your .bashrc (for bash):

   ```
   QTDIR=/home/nabbasi/data/QT_downloads/qt-2.2.0-beta2
   PATH=$QTDIR/bin:$PATH
   if [ $MANPATH ]
   then
           MANPATH=$QTDIR/man:$MANPATH
   else
           MANPATH=$QTDIR/man:
   fi
   if [ $LD_LIBRARY_PATH ]
   then
           LD_LIBRARY_PATH=$QTDIR/lib:$LD_LIBRARY_PATH
   else
           LD_LIBRARY_PATH=$QTDIR/lib
   fi
   LIBRARY_PATH=$LD_LIBRARY_PATH
   if [ $CPLUS_INCLUDE_PATH ]
   then
           CPLUS_INCLUDE_PATH=$QTDIR/include:$CPLUS_INCLUDE_PATH
   else
           CPLUS_INCLUDE_PATH=$QTDIR/include
   fi
   export QTDIR PATH MANPATH LD_LIBRARY_PATH LIBRARY_PATH
   export CPLUS_INCLUDE_PATH
   ```

To build QT yourself, do this

```
1. make sure QTDIR path is first set correctly to the QT your
   are building as shown above.
2. cd $QTDIR; ./configure
   NOte the build type.
3. cd configs
   and edit the correct config that matches the solaris build type
   (default should be solaris-cc-shared and do the following
   changes (notes difference between original and changed file.)
   basically, I've used -fPIC instead of -KPIC (since building with
   gcc not Sun Compiler for now). and to fix a problem in Qt
   build on Solaris, use the '-isystem' instead of '-I' as shown

#diff solaris-cc-shared solaris-cc-shared.orig
6c6
< SYSCONF_CXXFLAGS_X11  = -isystem /usr/openwin/include
---
> SYSCONF_CXXFLAGS_X11  = -I/usr/openwin/include
84,87c84,85
< #SYSCONF_CXXFLAGS_LIB = -KPIC
< SYSCONF_CXXFLAGS_LIB  = -fPIC
< #SYSCONF_CFLAGS_LIB   = -KPIC
< SYSCONF_CFLAGS_LIB    = -fPIC
---
> SYSCONF_CXXFLAGS_LIB  = -KPIC
> SYSCONF_CFLAGS_LIB    = -KPIC
89,92c87,88
< #SYSCONF_CXXFLAGS_SHOBJ       = -KPIC
< SYSCONF_CXXFLAGS_SHOBJ        = -fPIC
< #SYSCONF_CFLAGS_SHOBJ = -KPIC
< SYSCONF_CFLAGS_SHOBJ  = -fPIC
---
> SYSCONF_CXXFLAGS_SHOBJ        = -KPIC
> SYSCONF_CFLAGS_SHOBJ  = -KPIC

4. Now, simply do 'make' from top level, go get a cup of
   coffee, and in about 1-2 hrs, Qt will be build.
```

5. download kde-1.1.2-1-Solaris-7-Sparc.tar.gz. This is a pre-build KDE for Solaris. It is only needed to make our GUI configuration happy.(it is OK that it is for Solaris 7), Simply extract it to some directory.

   Assume you extracted it to /export/home/kde-1.1.2-1-Solaris-7-Sparc/ then do this

```
cd /opt
ln -s /export/home/kde-1.1.2-1-Solaris-7-Sparc  kde

and edit .bashrc and add this

KDEDIR=/opt/kde
export KDEDIR

and add $KDEDIR/bin to your PATH also.
```

6. Edit your .bashrc and have this PATH (notice, /usr/local/bin is first)

```
PATH=/usr/local/bin:/usr/bin:/usr/sbin:/sbin:/bin:/sbin:/usr/ccs/bin:/usr/openwin/bin:$KDEDIR/b
LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib
export PATH
export LD_LIBRARY_PATH
```

```
we add /usr/local/lib since during the build this where our libraries go, and linker
needs to find them.
```

7. Ok, now we have all the needed software. Use CVS to obtains the source tree.

8. Build the tree using these steps [6]

   (a) add some symbolic links

```
cd /usr/loca/bin
ln -s gcc CC
ln -s gcc cc
ln -s flex flex++
ln -s flex lex
```

   (b) set a CC env. variable as follows

```
export CC=''gcc -DSOLARIS''
```

   (c) `cd dvd2/src/filters/sink/video/x11` and edit the build.h file as needed to specify the video mode output to be used. (I used XSHM)

```
 /* the mode selected */
#define OUTPUT_MODE OUTPUT_MODE_XSHM
```

   (d)

```
cd lmf; rm config.cache; ./configure; make uninstall; make; make install
```

   (e)

```
cd sconv; rm config.cache; ./configure; make uninstall; make; make install
```

   (f) For DVD2, need to do make twice:

```
cd dvd2; rm config.cache; ./configure; make uninstall; make -i; make -i install; mak
```

   (g) Make sure X11 is running in 16 bit depth (since this is what DVDmax on linux based source now supports). On Linux[7] this is done as follows

```
startx -- -bpp 16
```

   (h) set up the skin directory:

[6] You do not need to be root unless root permission is needed to write to /usr/local. I had my /usr/local write allowed by everyone so I do not need to keep switching to root to install sw.

[7] find how to do this on CDE/Solaris

```
cp ./dvd2/src/app/frontend/skin.tar.gz  $HOME
cd
gunzip skin.tar.gz
tar xf skin.tar
```

(i)
```
su; export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/local/lib; cd dvd2/src/test
./ftest12
```

8 Current status of Solaris build

As of 090500, result of Solaris build shows these plugins being build (9 plugins)

```
cd /usr/local/DVDMax/plugins
SunOS>ls -l *.so

-rwxr-xr-x    1 nabbasi  staff       99028 Sep  5 00:55 dvdmax_demux.so
-rwxr-xr-x    1 nabbasi  staff       72184 Sep  5 00:55 dvdmax_dvddisc.so
-rwxr-xr-x    1 nabbasi  staff      650248 Sep  5 00:55 dvdmax_dvdnav.so
-rwxr-xr-x    1 nabbasi  staff       48420 Sep  5 00:55 dvdmax_hli.so
-rwxr-xr-x    1 nabbasi  staff       33684 Sep  5 00:55 dvdmax_mpeg2sync.so
-rwxr-xr-x    1 nabbasi  staff       39024 Sep  5 00:55 dvdmax_pcm.so
-rwxr-xr-x    1 nabbasi  staff       93396 Sep  5 00:55 dvdmax_rawfile.so
-rwxr-xr-x    1 nabbasi  staff       95572 Sep  5 00:55 dvdmax_subpic.so
-rwxr-xr-x    1 nabbasi  staff      111880 Sep  5 00:55 libdvdapp.so
```

On Linux, complete build shows 14 plugins.

```
cd /usr/local/DVDMax/plugins
nabbasi>ls -l *.so
-rwxr-xr-x    1 nabbasi  users      209711 Sep  4 23:38 dvdmax_ac3_filter.so
-rwxr-xr-x    1 nabbasi  users      113375 Sep  4 23:38 dvdmax_decss.so
-rwxr-xr-x    1 nabbasi  users       79181 Sep  4 23:38 dvdmax_demux.so
-rwxr-xr-x    1 nabbasi  users       69295 Sep  4 23:38 dvdmax_dvddisc.so
-rwxr-xr-x    1 nabbasi  users      418122 Sep  4 23:38 dvdmax_dvdnav.so
-rwxr-xr-x    1 nabbasi  users       44292 Sep  4 23:38 dvdmax_hli.so
-rwxr-xr-x    1 nabbasi  users      271365 Sep  4 23:38 dvdmax_mpeg.so
-rwxr-xr-x    1 nabbasi  users       43113 Sep  4 23:38 dvdmax_mpeg2sync.so
-rwxr-xr-x    1 nabbasi  users       71574 Sep  4 23:38 dvdmax_oss.so
-rwxr-xr-x    1 nabbasi  users       48213 Sep  4 23:38 dvdmax_pcm.so
-rwxr-xr-x    1 nabbasi  users       74610 Sep  4 23:38 dvdmax_rawfile.so
-rwxr-xr-x    1 nabbasi  users       79182 Sep  4 23:38 dvdmax_subpic.so
-rwxr-xr-x    1 nabbasi  users      112248 Sep  4 23:38 dvdmax_x11video.so
-rwxr-xr-x    1 nabbasi  users      113059 Sep  4 23:38 libdvdapp.so
nabbasi>
```

plugins failed to link on Solaris are: ac3, decss, mpeg, oss, x11.

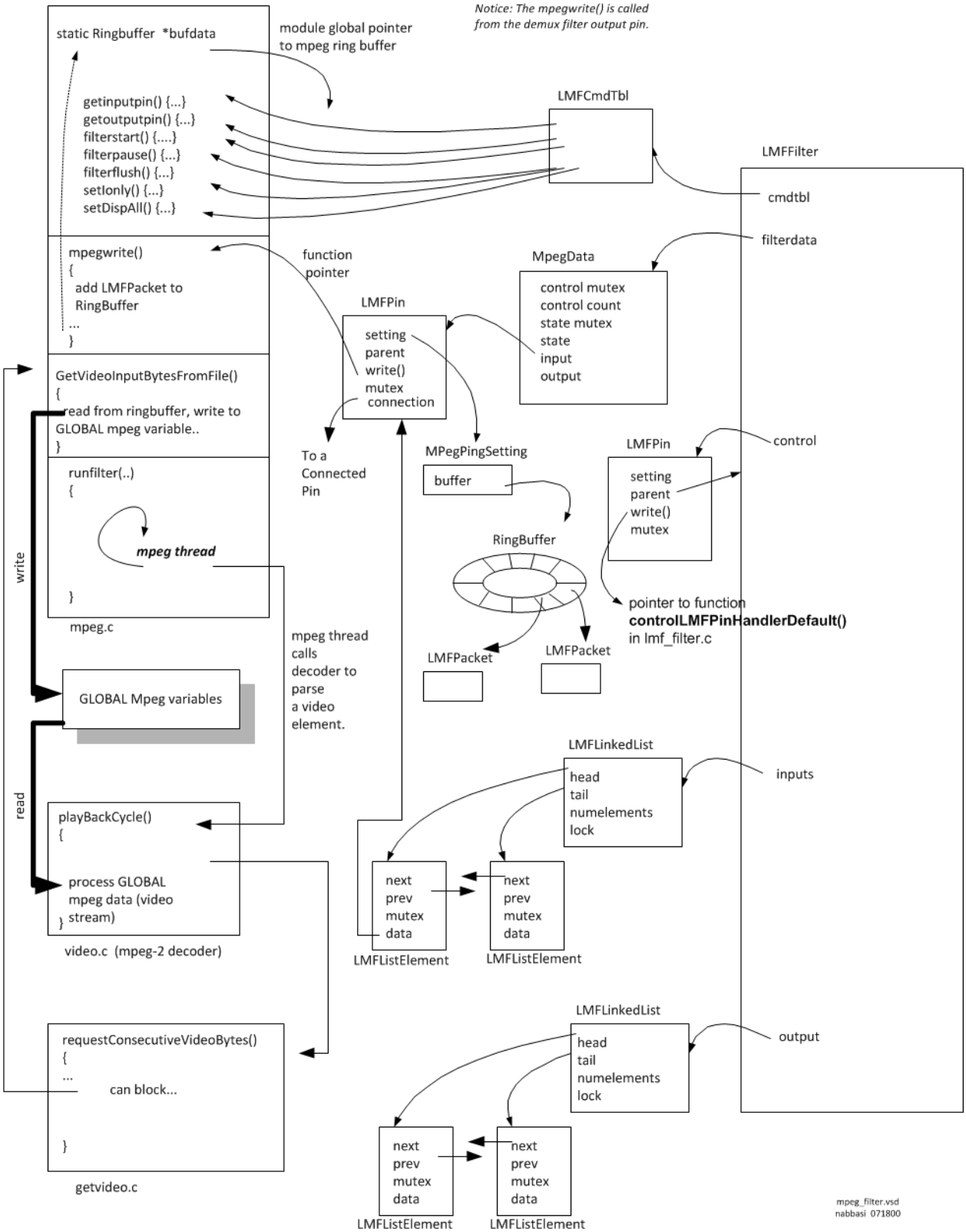


Figure 3: Mpeg filter internals. mpeg filter.vsd

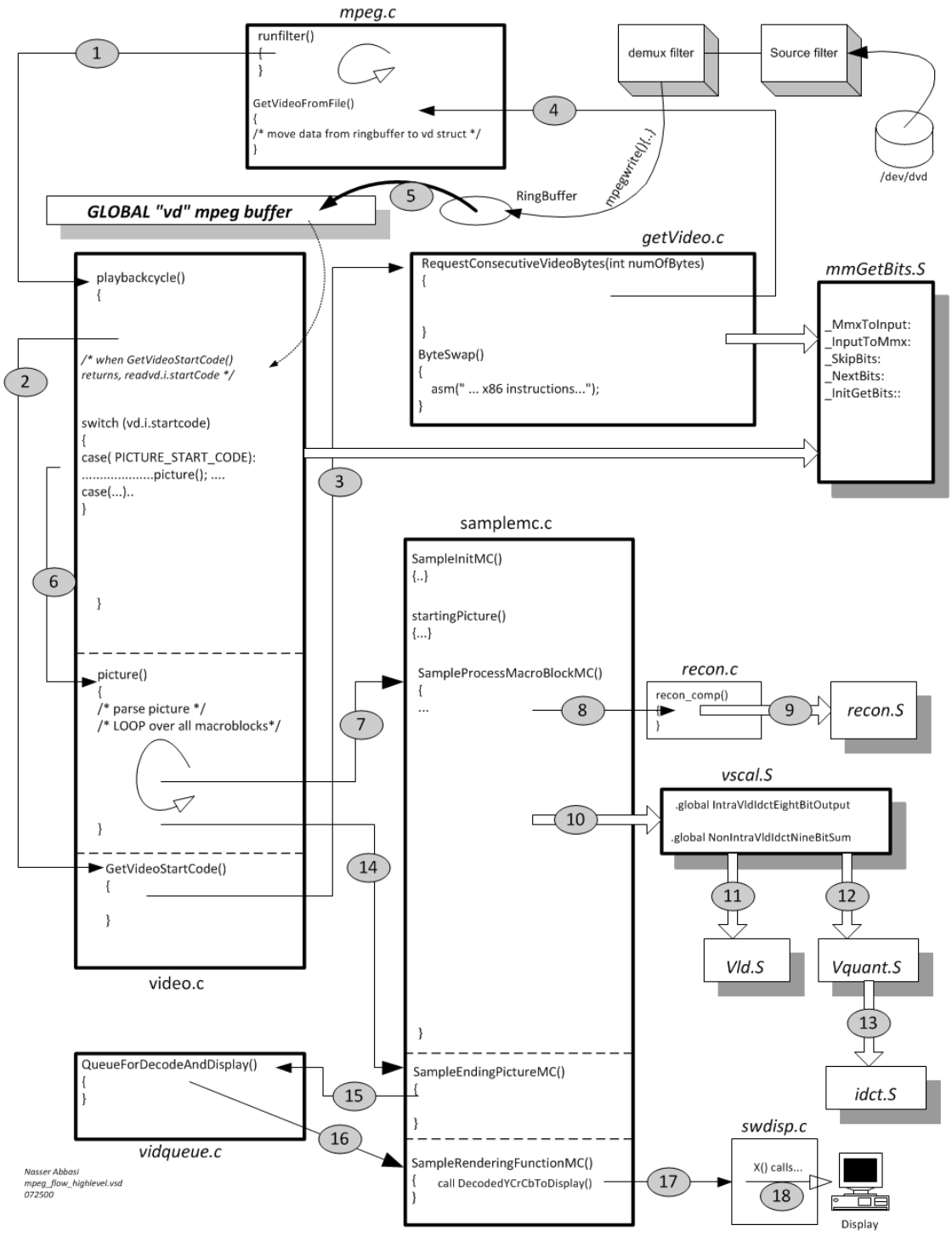


Figure 4: mpeg-2 filter main logic

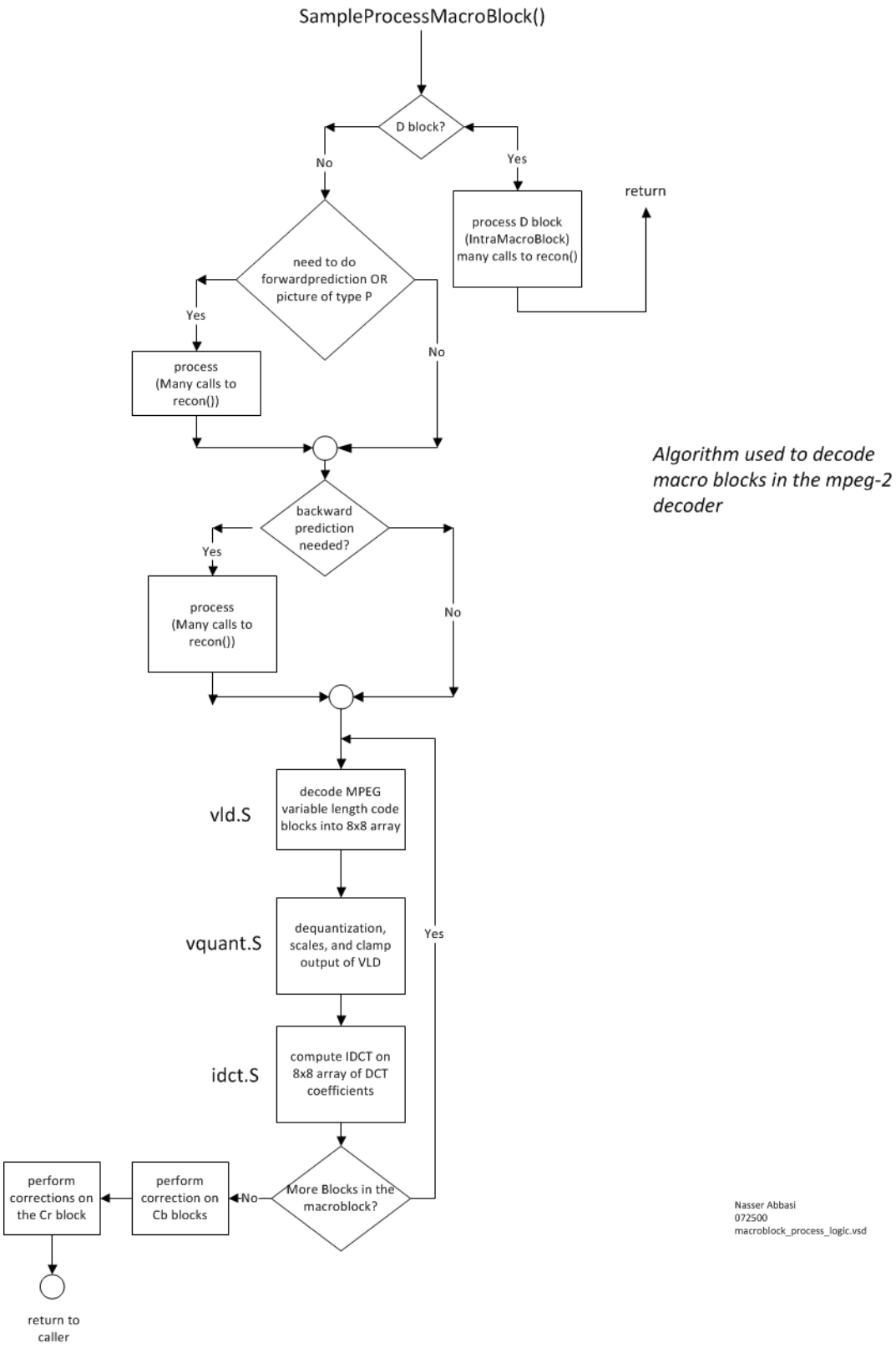


Figure 5: macroblock decoding using samplemc.c as the driver

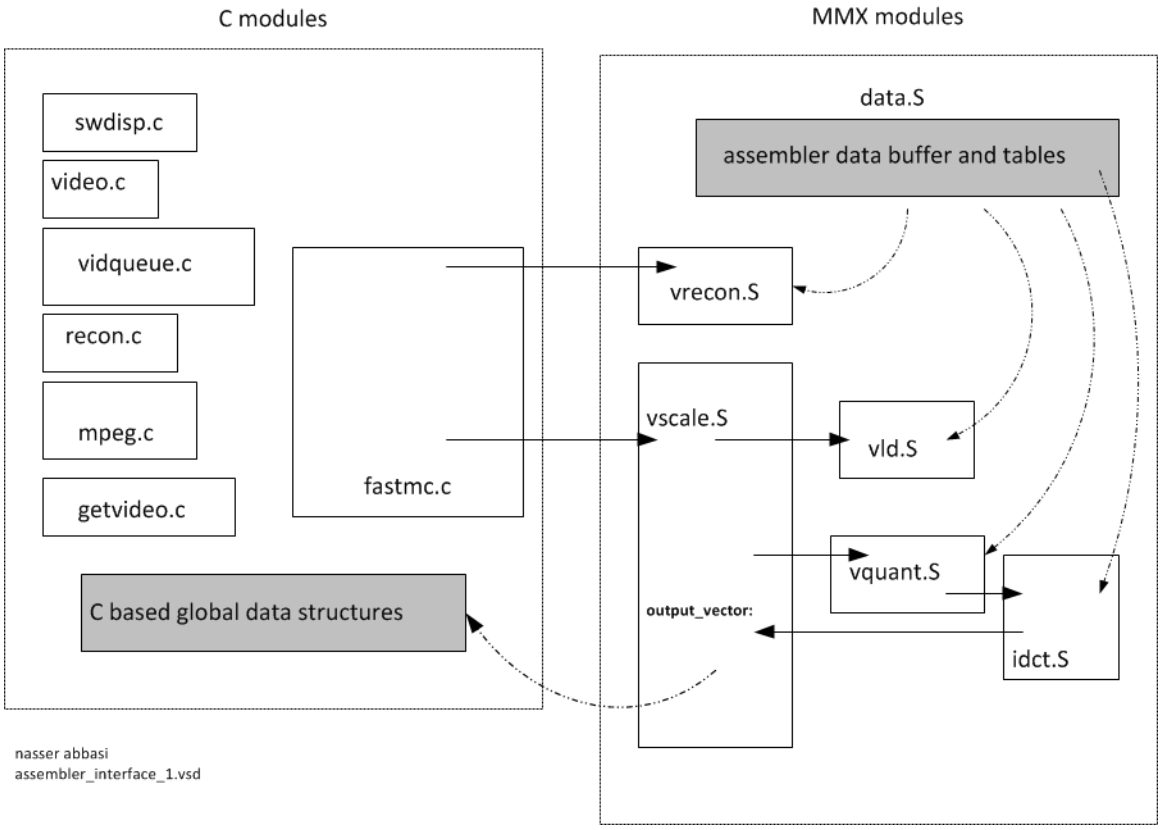


Figure 6: High level diagram showing the C and Assembler modules used in MPEG-2 decoder and global buffers

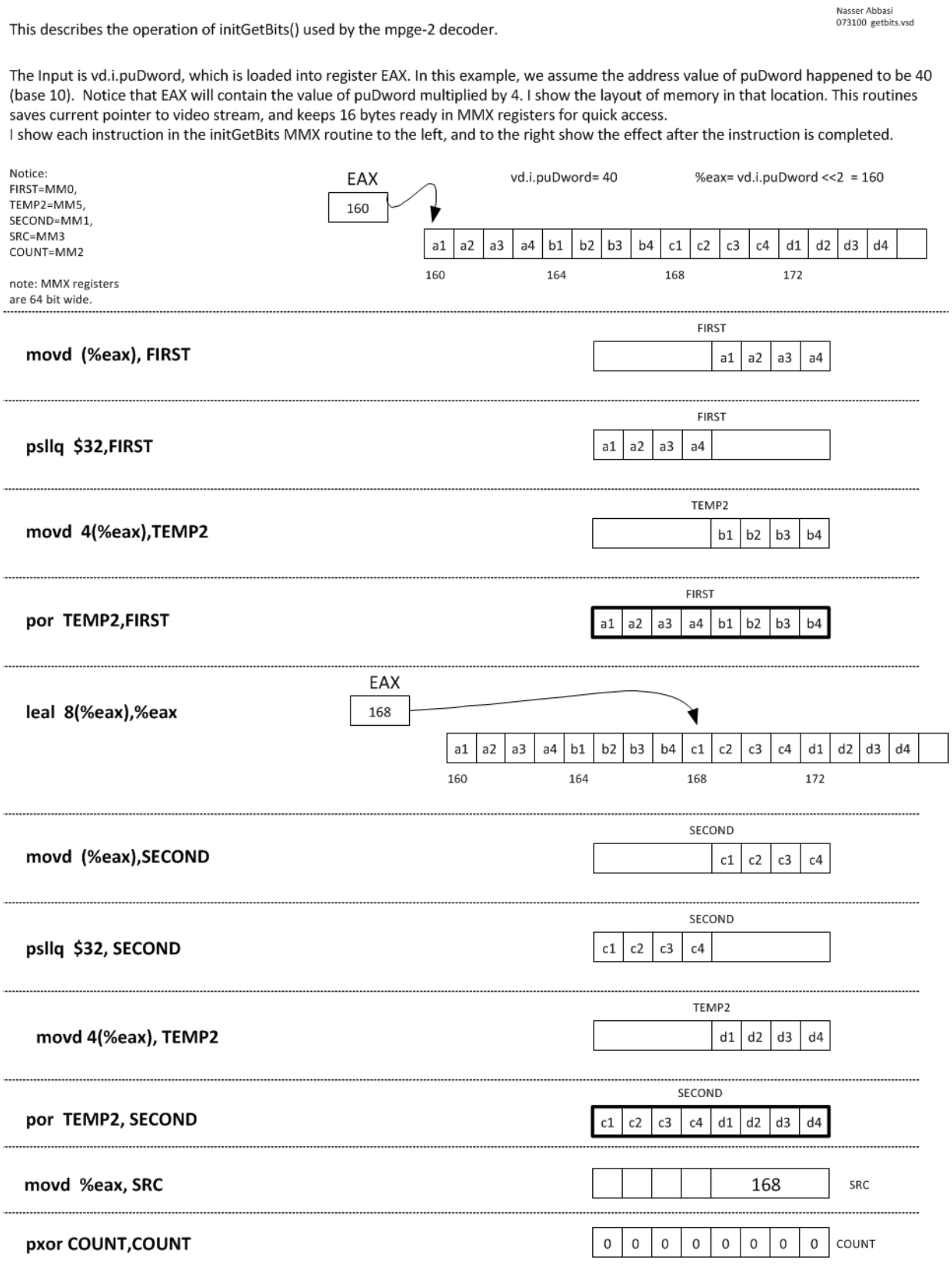


Figure 7: walk though initGetBits MMX code used in GetBits.S

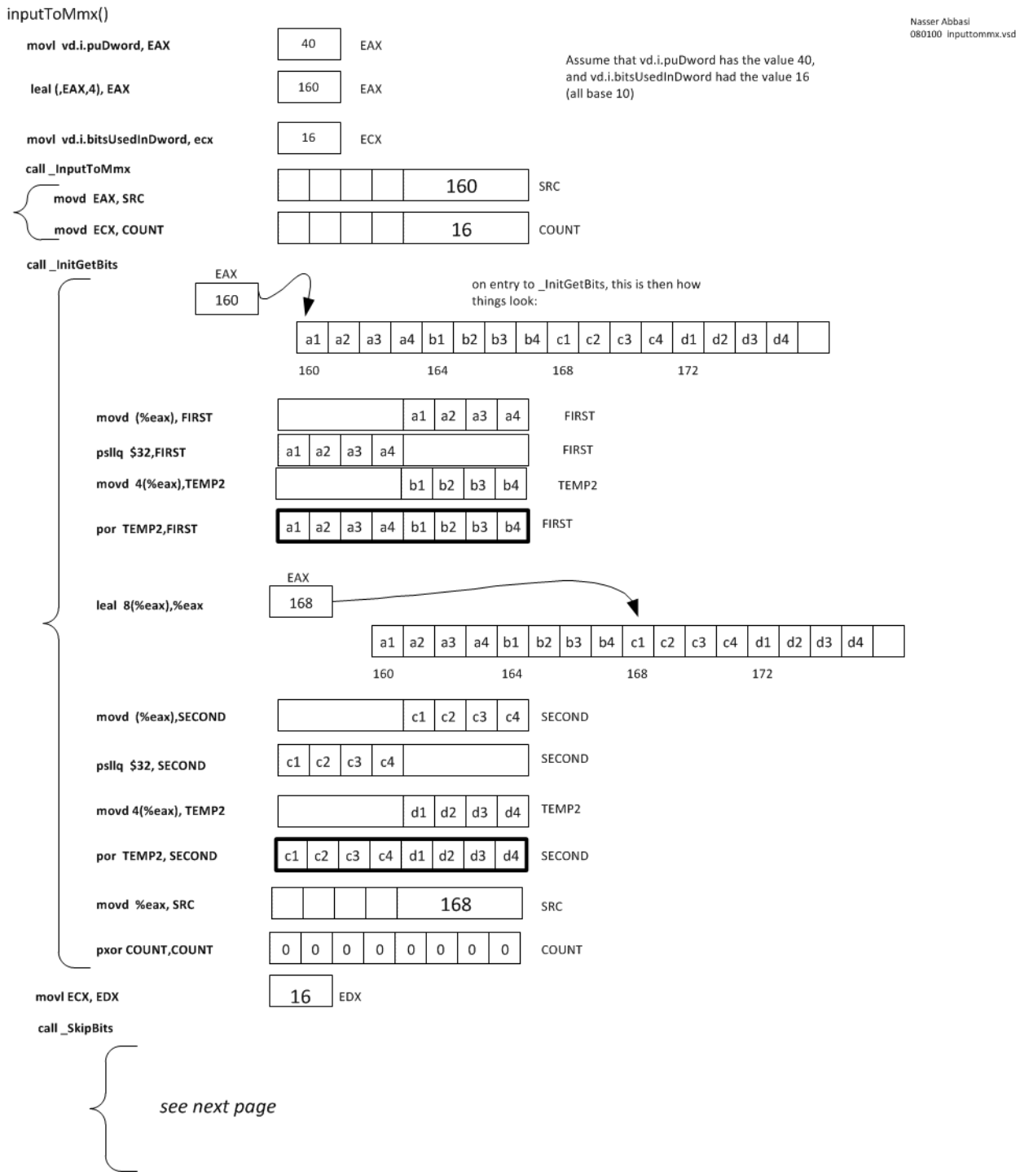


Figure 8: walk though InptToMmx MMX code used in GetBits.S

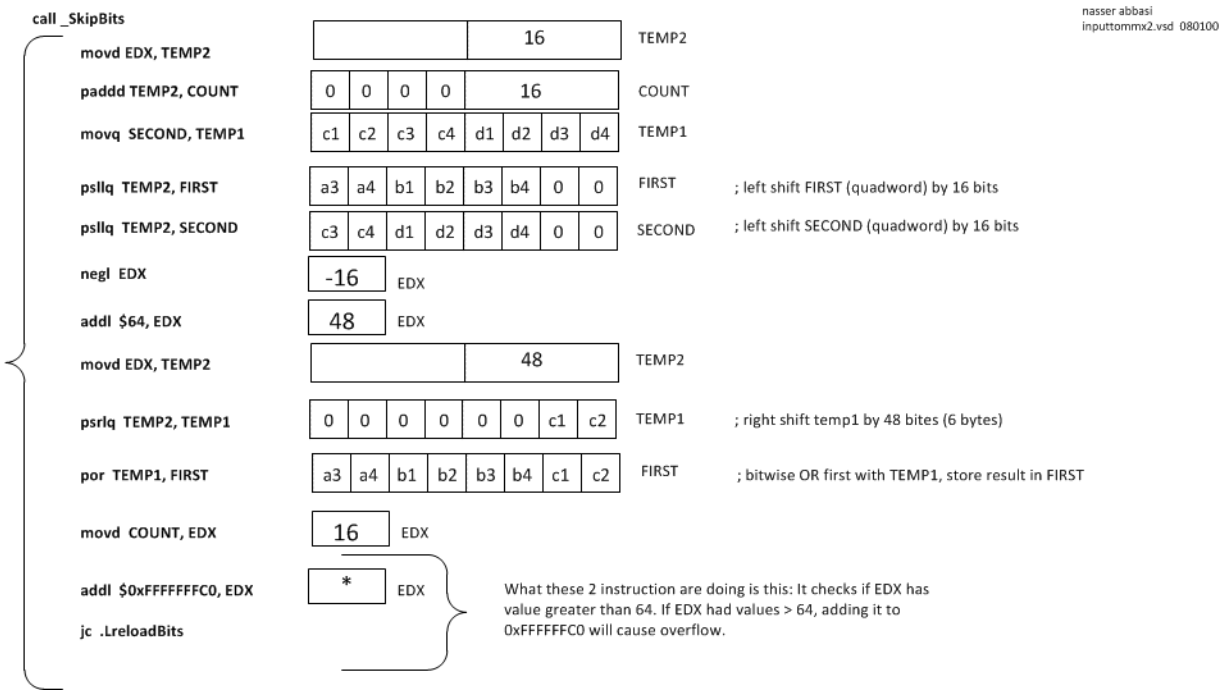


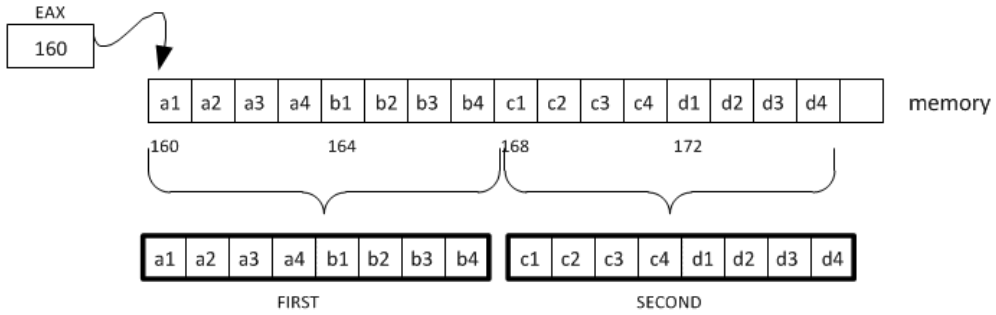
Figure 9: follow up of walk though InptToMmx MMX code used in GetBits.S

Nasser Abbasi
080100 inputtommx_summary.vsd

This is a summary of the inputToMmx() routine.

inputToMmx() loads 2 MMX registers with with data from the video input stream (pointed to by vd.i.puDword
On entry, there is a count in vd.i.bitsUsedInDword, whic tells how many bits from pointer to bit stream are allready consumed.
This is a summary of input and output. Assume puDword is 40 and bitsUsedInDword is 16.

After initGetBits, this is the picture:



After inputToMmx() returns, this is the pictire:

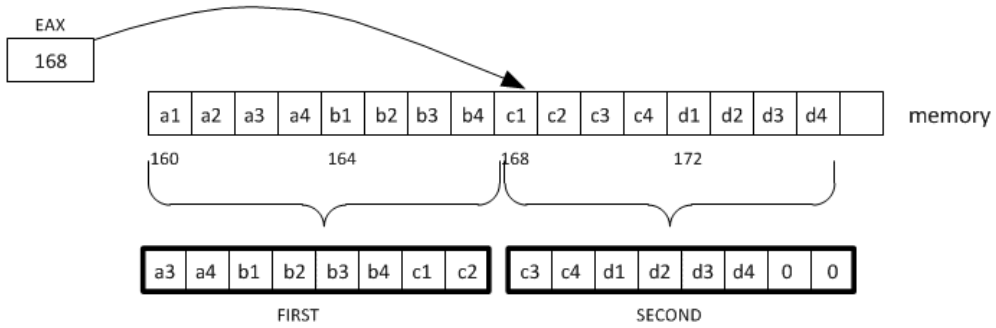


Figure 10: summary of InptToMmx MMX code used in GetBits.S

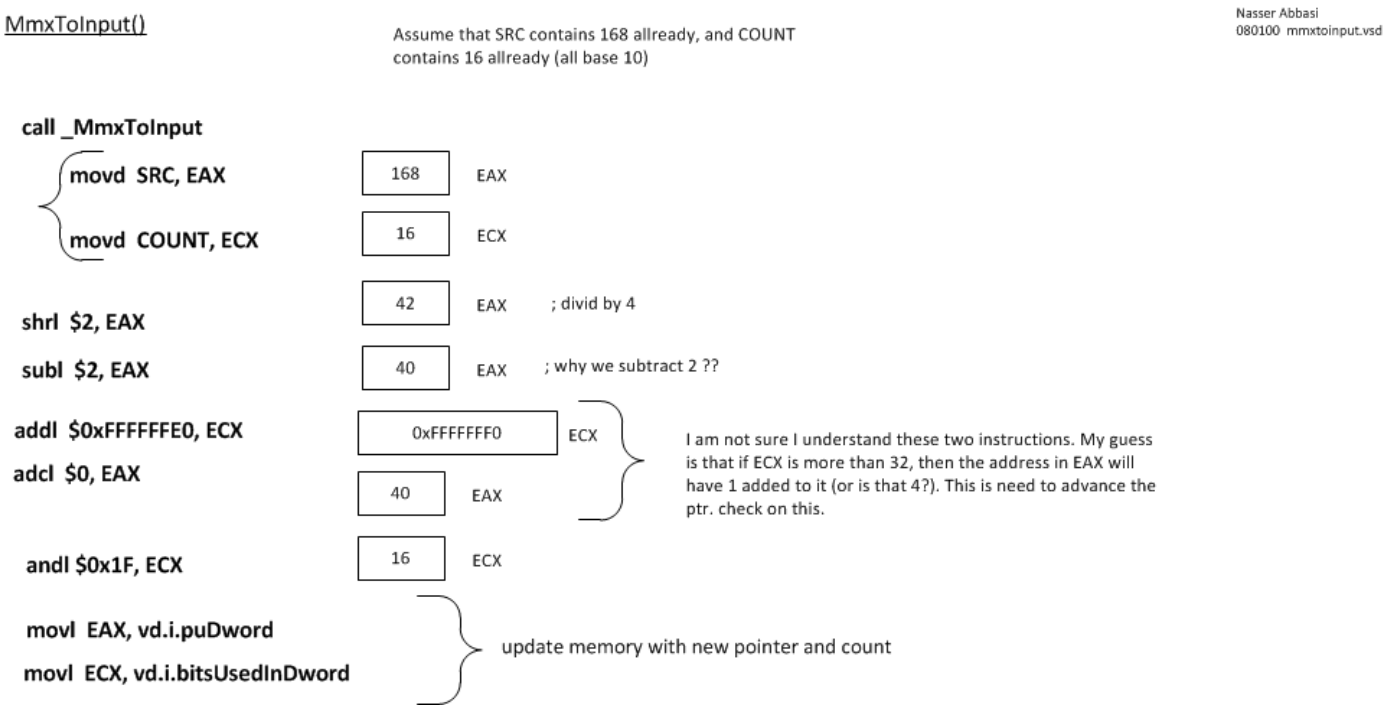


Figure 11: MmxToInput walk though MMX code used in GetBits.S

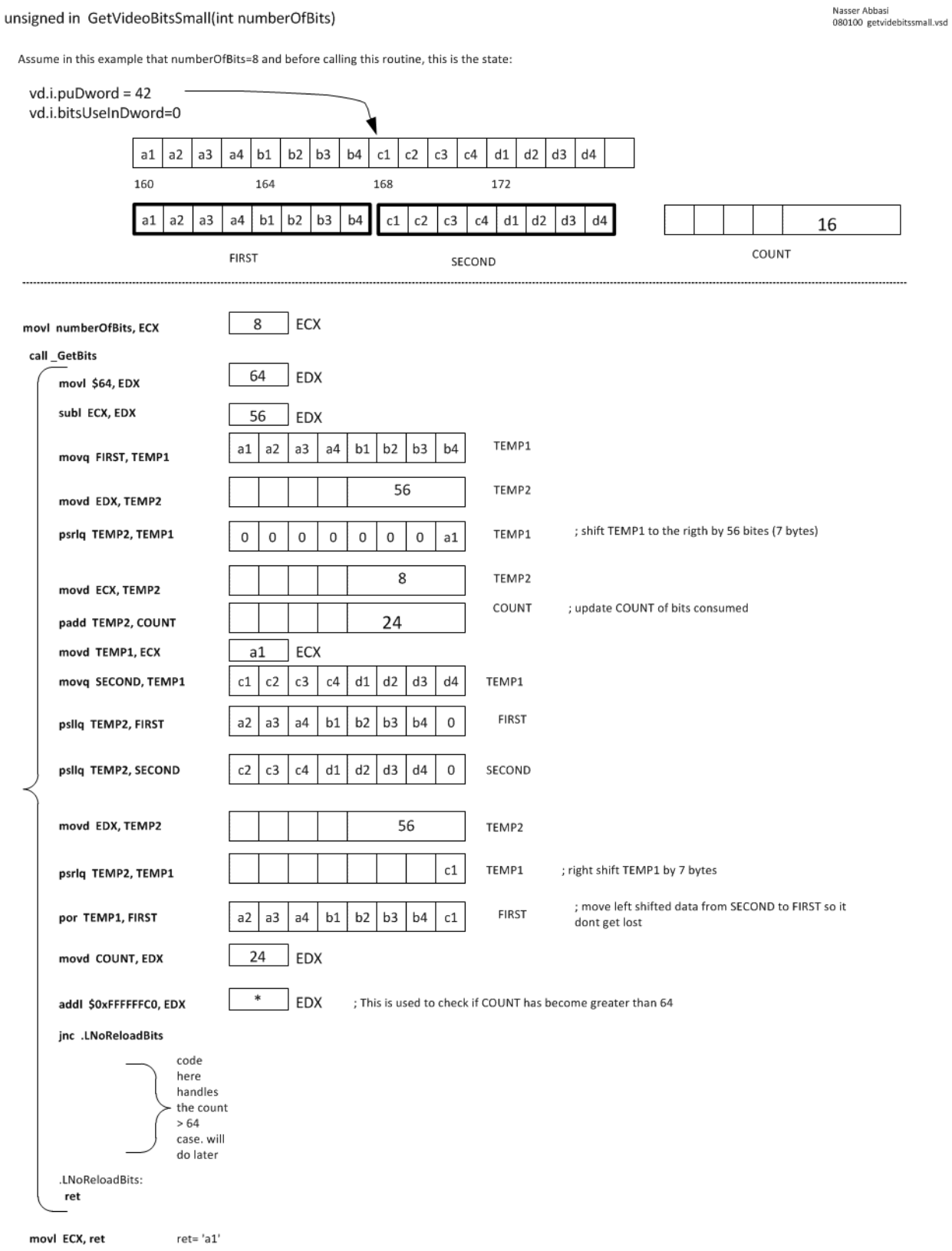


Figure 12: GetVideoBitsSmall walk though MMX code used in GetBits.S