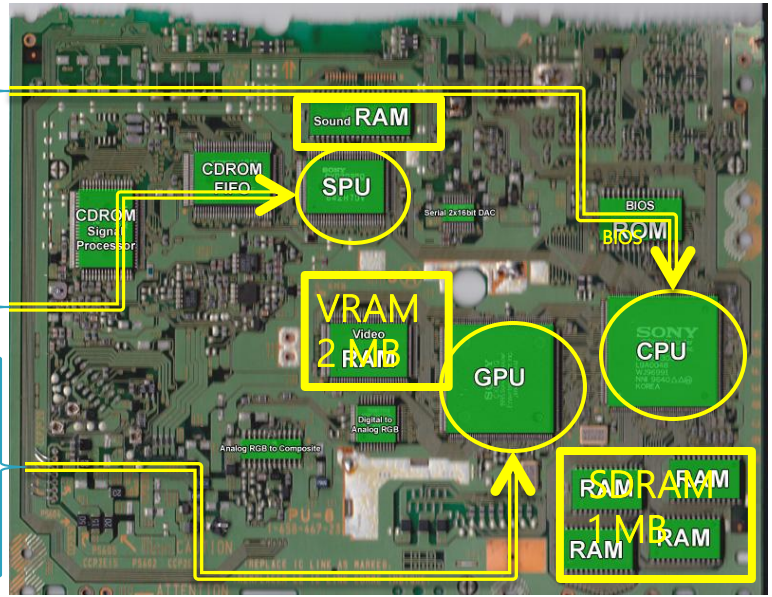
FPGA : PS1'S MDEC

laxer3a

Playstation 1 Spec

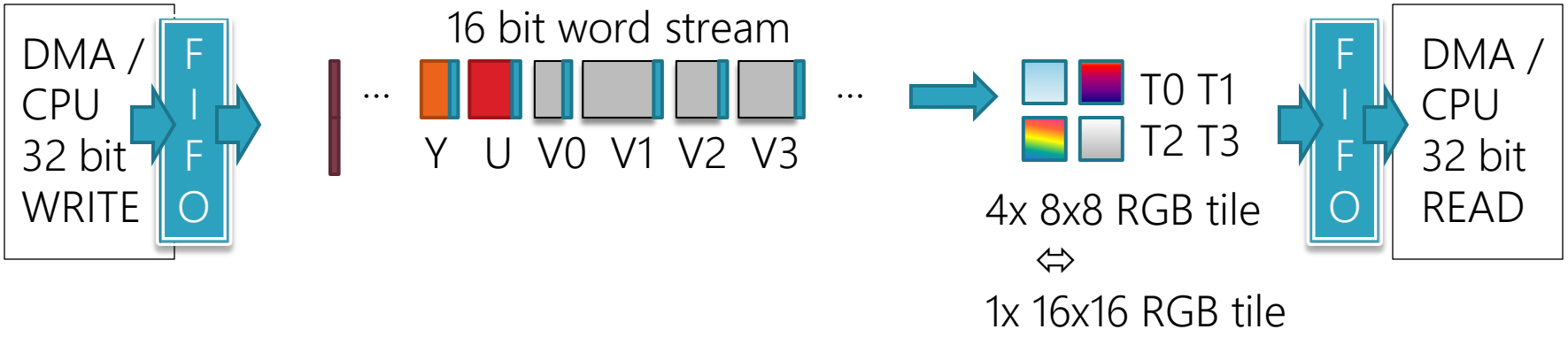
- CPU : MIPS R3000A @ 33.8 Mhz
  - 5 kb L1 Cache (4K I$, 1K D$)
- GTE : 25 Coprocessor CPU instructions.
  - Attached to the CPU, no bus. 64 x 32 bit visible registers.
- MDEC
  - RLE Encoded IDCT Matrices decoder = JPEG Decoder.→ Video decoder.
- AUDIO
  - 24 Channel ADPCM, 512 KB RAM, Digital Effects.
- GPU
  - Triangle/line rasterization/data transfer.
  - Texture, Vertex color (multiply only)
  - Support blending, support 1 bit stencil. / Dithering (RGB888->555) option.
- Others
  - DMA, CD Rom, IO, etc..



What is MDEC ?

- Motion DECoder, like a 'JPEG decoder'.
  - Each frame is like a decoded JPEG, except no Huffman.
  - A frame is a list of Macro Block of 8x8.
    - For color image, there is U/Y/V macro blocks in specific order.
  - A macro block is a list of 16 bit data with a end code (FE00)
  - Complete spec : http://problemkaputt.de/psx-spx.htm#macroblockdecodermdec
- Only 3 commands (using 3 bit) :
  - 0,4..7 = Invalid (NOP)
  - 1=Decode Macro Block.
  - 2=Set Quantization Table
  - 3=Set Scale Table

Stream of 16 bit word…



32 bit, Command type 1 : send block to decode, number of block, etc…

End of single block = FE00

Note : support also stream of Y block only for monochrome 8/4 bit output.

Registers

- 1F801820 [Write] : MDEC0 - INPUT
  - Command port, write 32 bit word stream.
- 1F801820 [Read] : MDEC0 - OUTPUT
  - Macroblock data, 32 bit word : always 8x8 pixel (24bit:48 word, 16 bit: 32 word, 8 bit: 16 word, 4 bit: 8 word)
- 1F801824 [Read] : MDEC1 Status Register
  - Data-Out Fifo Empty / Data-In Fifo Full.
  - Command busy/decode status
  - DMA related flags (In/Out)
  - Data Output Depth (4/8/24/16 bit), Signed/Unsigned
    - Fill 0 or 1 to bit 15 in 15 bit mode.
- 1F801824 [Write] : MDEC1 Status Register
  - Reset MDEC
  - Enable/Disable In/Out requests.

➔ Only 2 Registers !

MDEC Work

RLE Decode → IDCT Pass → YUV 2 RGB → Format Conversion

16 bit stream

SCALE	COEF
OFFSET	COEF
OFFSET	COEF
OFFSET	COEF
OFFSET	COEF
6 BIT	10 BIT

…

END (FE00)

n = prev n + 1 + offset

if SCALE=0 → Full block mode

$2 * COEF * 0$

$2 * COEF * n$

if SCALE≠0 → RLE block mode

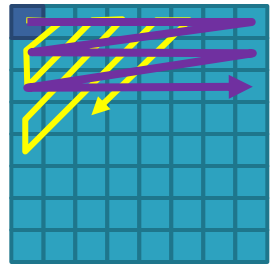
$COEF * 0$

$SCALE * COEF * \frac{n}{8}$

HW Implementation

```
n=0
n * 16      /8
n!=0
n * 16      /8

n=0
n * 8       /8
n!=0
n * Scale   /8
```

-2048
+2047
Clamp



Matrix to decompress.

(12 bit per entry, 0 filled on start)

Quantization Table

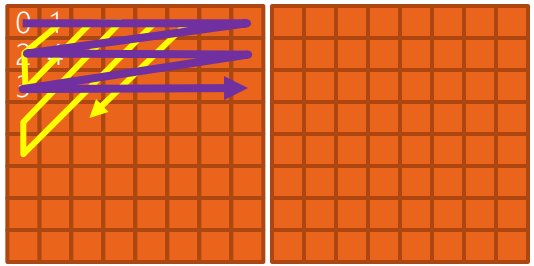


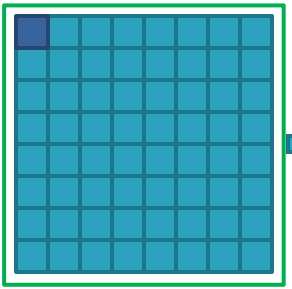
Table 0 : UV Block

Table 1 : Y Block

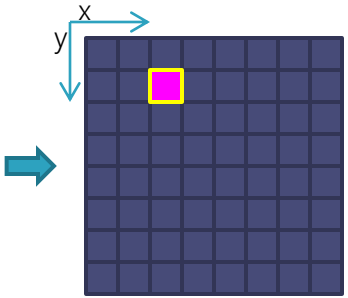
MDEC Work

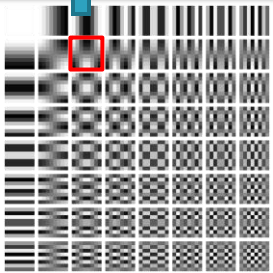
RLE Decode → IDCT Pass → YUV 2 RGB → Format Conversion

Inverse Discrete Cosine Transform (kind of Fourier transform)



```
for tblx = 0..7 {
  for tbly = 0..7 {

    sum = 0;
    for x = 0..7 {
      for y = 0..7 {
        sum += Matrix[x,y] * CosTbl[tblx,tbly][x,y]
    }}
    dest[tblx,tbly] = sum

}}
```

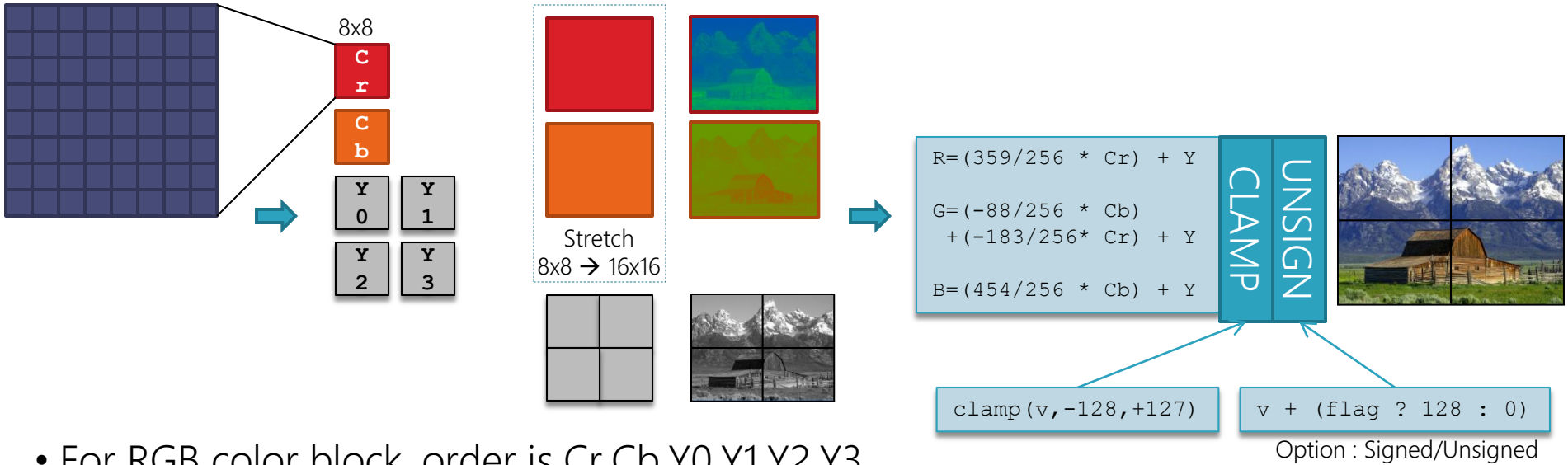




8x8 table of 8x8 COS (Scale Table)

MDEC Work

RLE Decode → IDCT Pass → YUV 2 RGB → Format Conversion



- For RGB color block, order is Cr,Cb,Y0,Y1,Y2,Y3
- By storing ONLY Cr,Cb we can convert to RGB pixel on the fly.
  - Read from Cr,Cb + current Y value.
- Output in valid order of tile and X,Y coordinate (scan order)

MDEC Work

- Y Block only : 4 bit/8 bit format out.
- Cr,Cb,Y0/1/2/3 : 24 bit / 15 bit RGB out.
  - In 15 bit mode, 16th bit = 0 or 1 setup possible
    - Use in GPU as masking / stencil bit.
  - Signed/Unsigned conversion done at YUV 2 RGB

Happy end ?

- Problem :
  - Screen is 320x240@30 fps → 6 matrix per 16x16 pixels. → 20x15x6 block / frame → 54000 block / sec.
  - Basic brute force : 8x8x8x8 computation per block.
    - 221,184,000 compute per sec.
    - Chip frequency is 33.8 Mhz !!!
    - Need **a minimum x6.6** performance up compare to SIMPLE implementation (1 clock, 1 operation).

Be smarter…

- Checked for FAST IDCT ?
  - Oh yeah, give me 'Yukihiro Arai & Masayuki Nakajima 1988' paper !

    (By the way, f**k scientific journal…

    Research is funded by taxes, science should not be behind paywall)
  - Optimized for pipelining, high speed, sin table is hard coded by data path.
    - Can NOT change the table like in PSX specs. Complex 'fat' design, not small unlikely in the PSX.

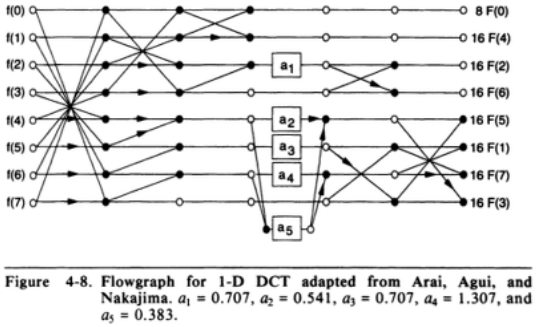
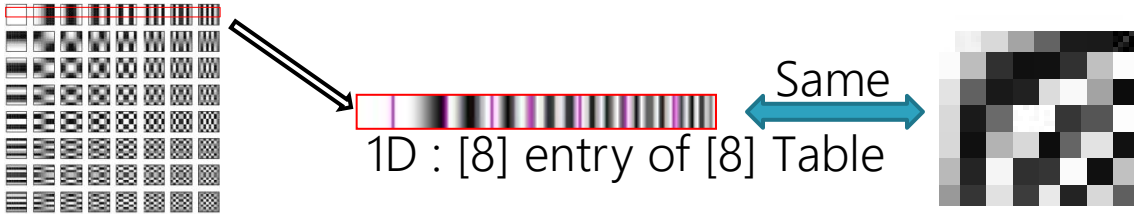


Figure 4-8. Flowgraph for 1-D DCT adapted from Arai, Agui, and Nakajima. $a_1 = 0.707$, $a_2 = 0.541$, $a_3 = 0.707$, $a_4 = 1.307$, and $a_5 = 0.383$.

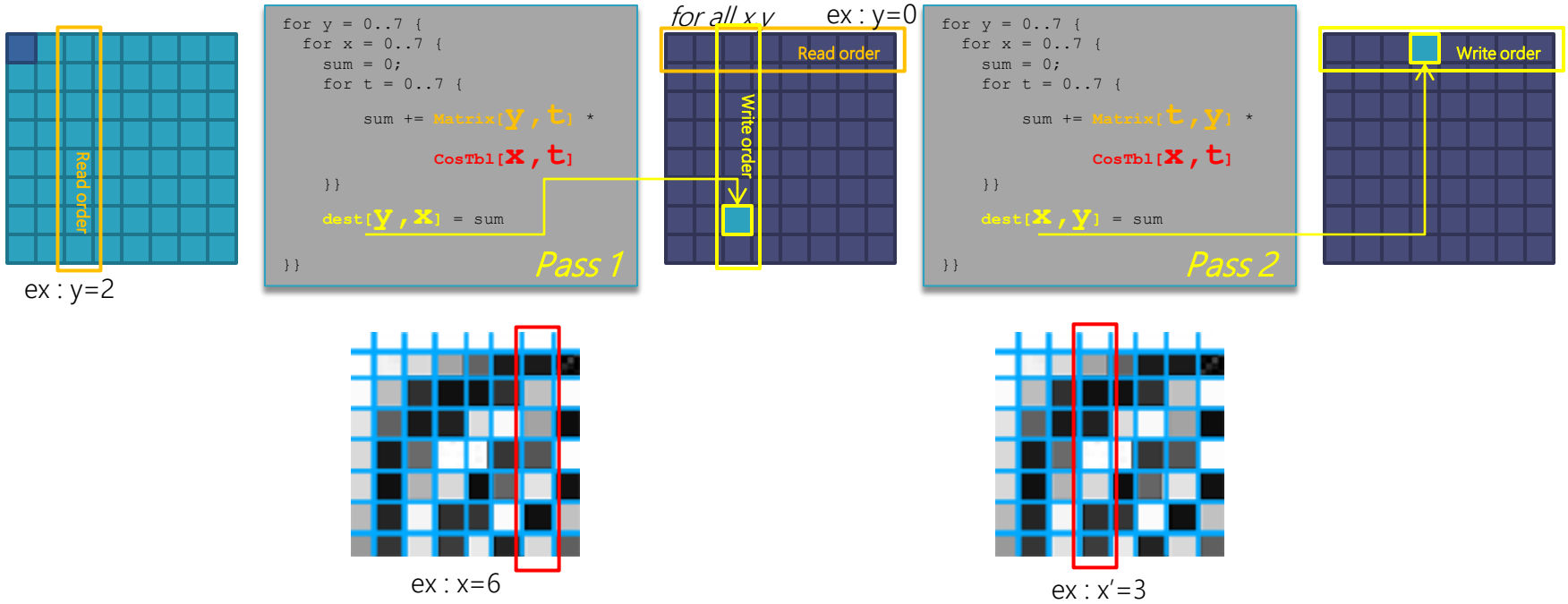
- Well, we know at least that IDCT is 1D separable filter.
  - Basically, can get the SAME result with 2 pass (1D) 8x8x8 computation ➔ x4 performance for same computation cost. Still need a 1.64 times improvement (well… 2x most likely, isn't it ?)
  - [!] And now we notice that the command n.3, scale table send 64 entries (8x8) : 1D TABLE !

2D : [8x8] Entry of [8x8] Table



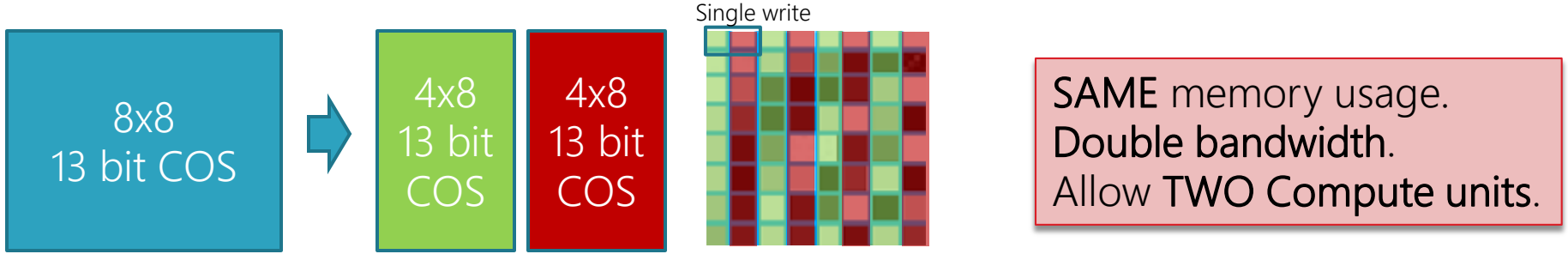
IDCT 1D Separable filter

- 2x 1D Pass instead of 1x 2D pass =



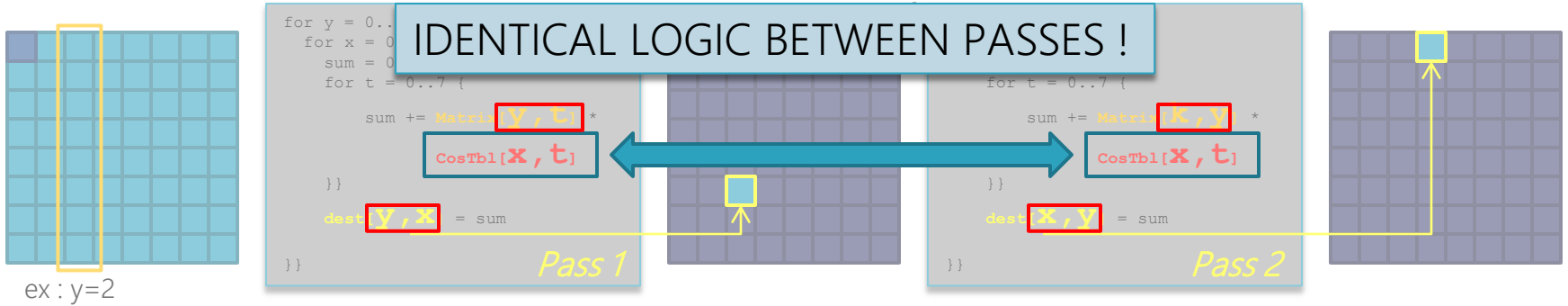
Let’s get x2 performance !

- Trick 1 : Not much of a trick but…
  - When we LOAD the COS table from CPU to MDEC, we send 32 bit.
  - So we send 2 COS entry at once → Let’s store COS table splitted in TWO.
    - 8x8 -> 2x 4x8 tables in parallel.
  - ODD/EVEN access will drive parallelism.



SAME memory usage.
Double bandwidth.
Allow **TWO Compute units**.

Let’s get x2 performance !



ex : y=2

Multiple possible loop implementation, BUT
- Use COMMON TABLE ADRESSING IS KEY.
- Keep correct X,Y scan order at end of pass 2.

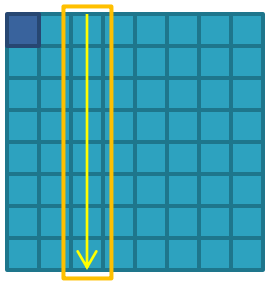
DIFFERENCES

ex : x=6

ex : x’=3

Let’s get x2 performance !

coef Table
12bit x 8x8



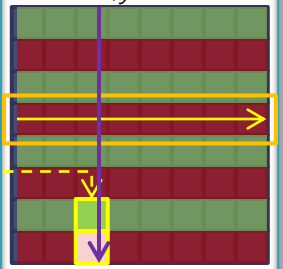
ex : y=2

Rewrite PASS 1

```
for y = 0..7 {
  for x = 0..6 step 2 {
    sumA = 0; sumB = 0;
    for t = 0..7 {
      sumA+=Matrix[y,t]*CosTbl[x,t]
      sumB+=Matrix[y,t]*CosTbl[x+1,t]
    }}
    dest[y,x]     = sumA
    dest[y,x+1] = sumB
}}
```

Pass 1

for all x,y



13 bit TEMP :
2x 4x8 RAM
instead of
1x 8x8 RAM

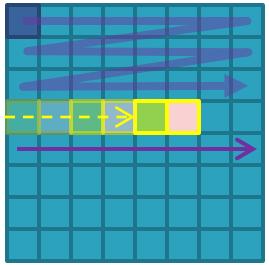
PARALLEL WRITE
BUT SINGLE READ

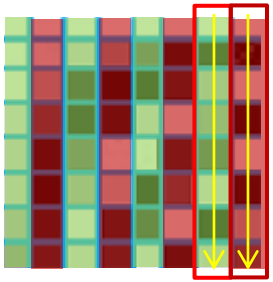
Rewrite PASS 2

```
for y = 0..7 {
  for x = 0..6 step 2 {
    sumA = 0; sumB = 0;
    for t = 0..7 {
      sumA+=Matrix[t,y]*CosTbl[x,t]
      sumB+=Matrix[t,y]*CosTbl[x+1,t]
    }}
    dest[x,y]     = sumA
    dest[x+1,y] = sumB
}}
```

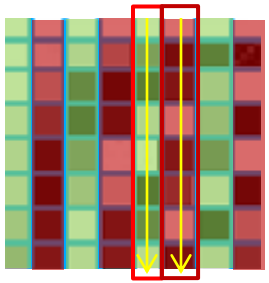
Pass 2

Out / Storage



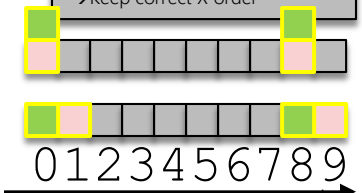


ex : x=6



ex : x=4

Problem :
→ 2 write at the same time
Solution :
→ 2 write every 8 cycle.
→Buffer 1 write, push next cycle.
→Keep correct X order

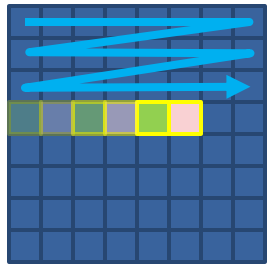


Read Order
Write Order

Let’s get x2 performance !

- We time-shift the double output, because in some cases we do NOT store the values (process on the fly), want to avoid processing 2 values at the same time. (See Y0/Y1/Y2/Y3 block)

- Notice how we output X,Y in correct scan order at end of second pass too.

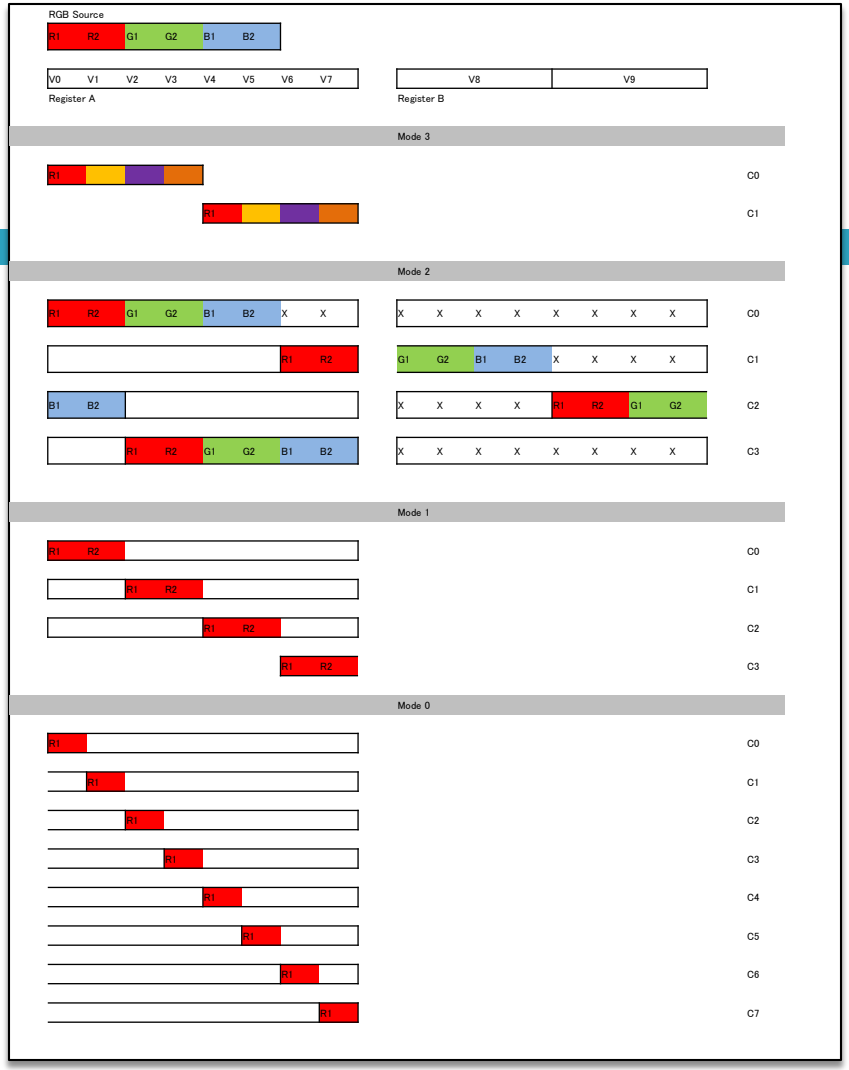


Let’s get x2 performance !

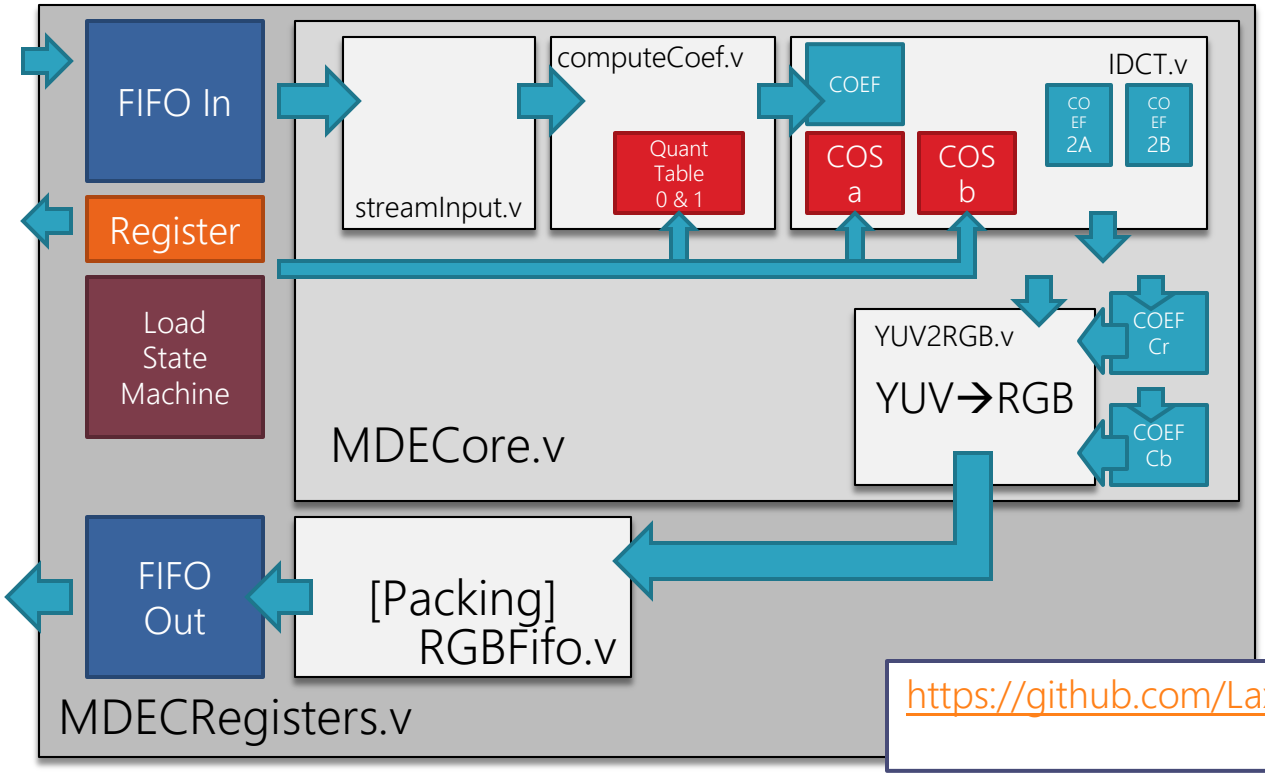
- Now can perform max of 66K block per second.
- Cost is just an ADDER and MUL added + cheap logic.

Packing pixels…

- 8x8 [4/8/16/24 bit] pixels into 32 bit FIFO word.
- Temp register A & B +
- State Machine
  - Pixel Format
  - Current Step



My Real Implementation Design

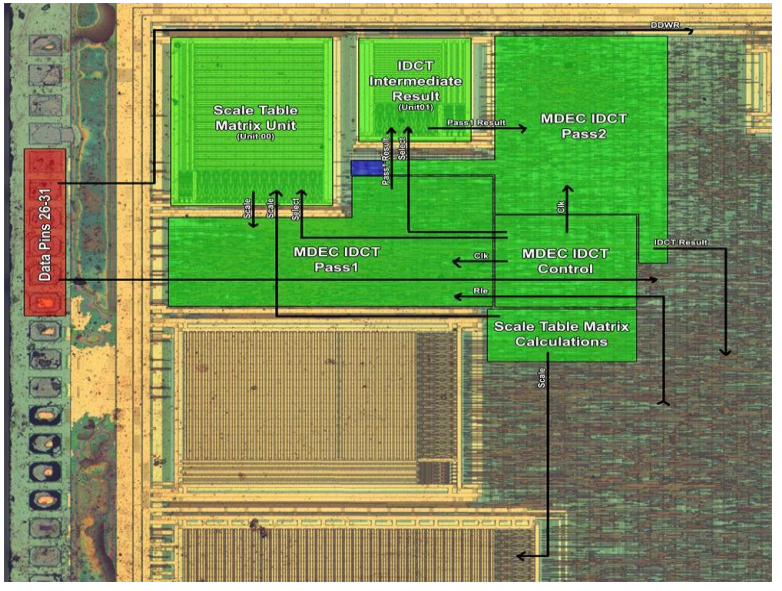


https://github.com/Laxer3a/MDEC/tree/master/hdlMDEC

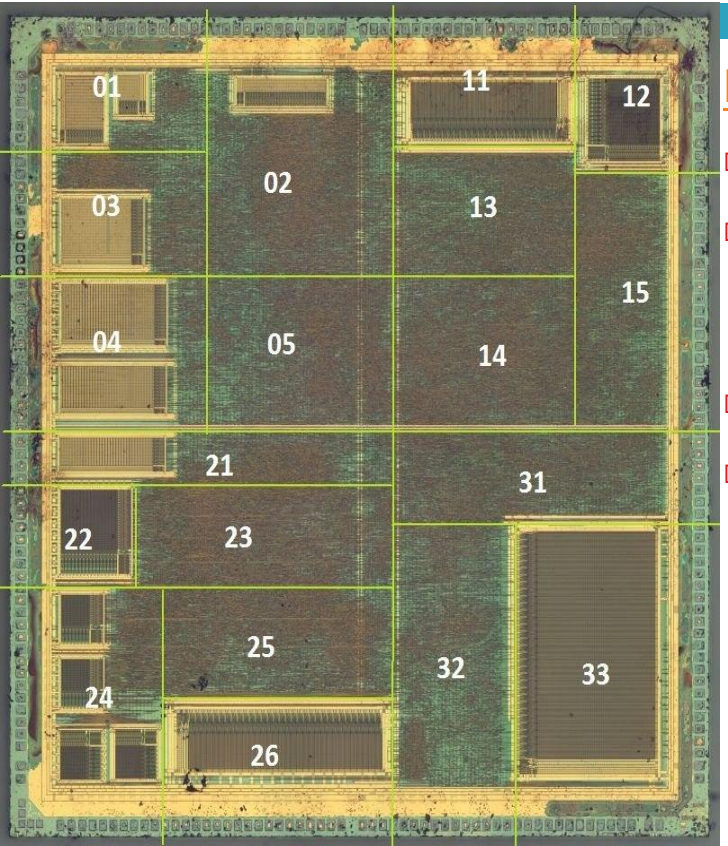
Here comes Russia !

http://psxdev.ru/

- MDEC Topic in english from No$, famous emulator writer. http://board.psxdev.ru/topic/9/
- http://forum.emu-russia.net/viewtopic.php?f=13&t=4106
- (English) http://www.psxdev.net/forum/viewtopic.php?f=70&t=551

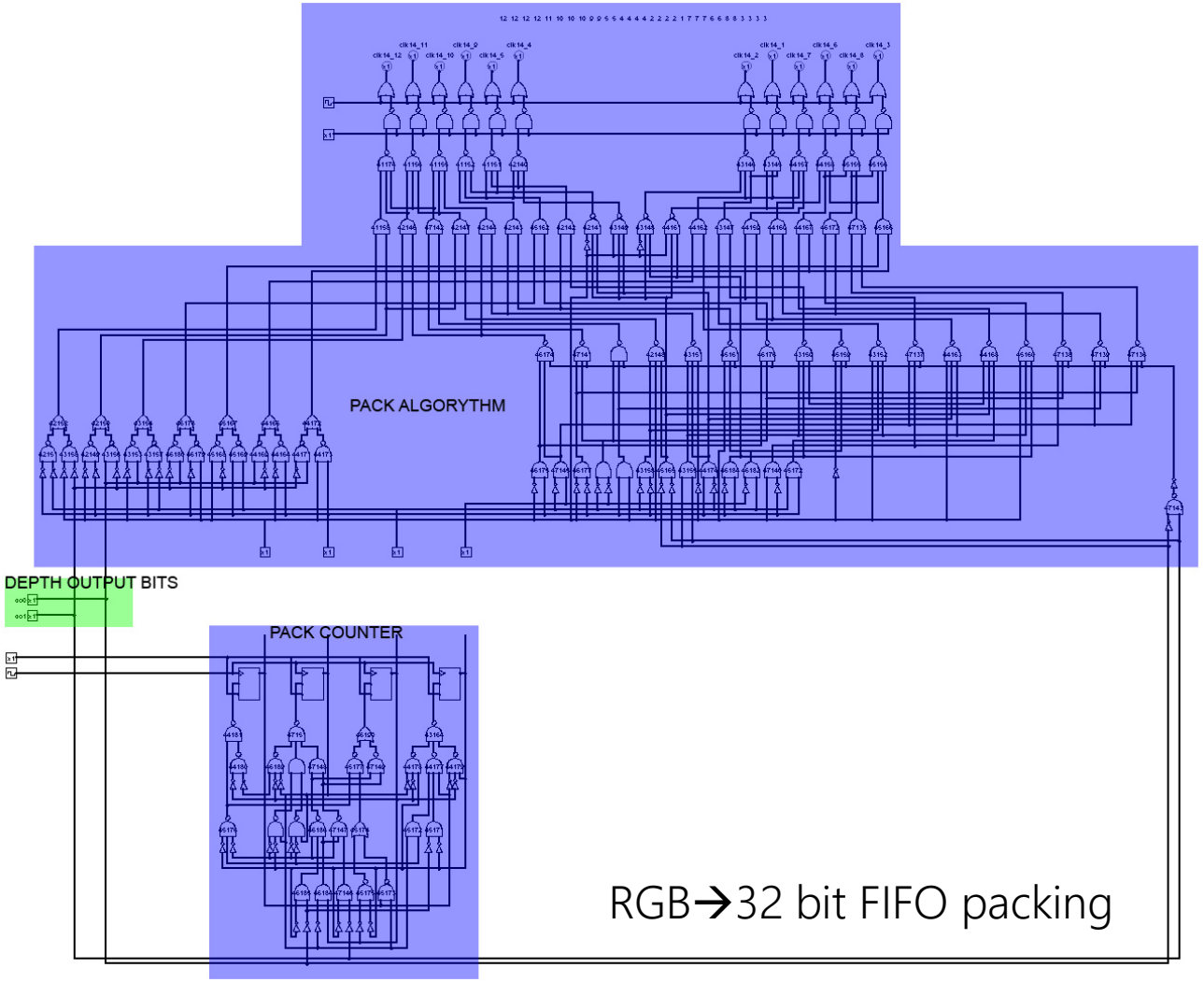


Reverse Engineering result :



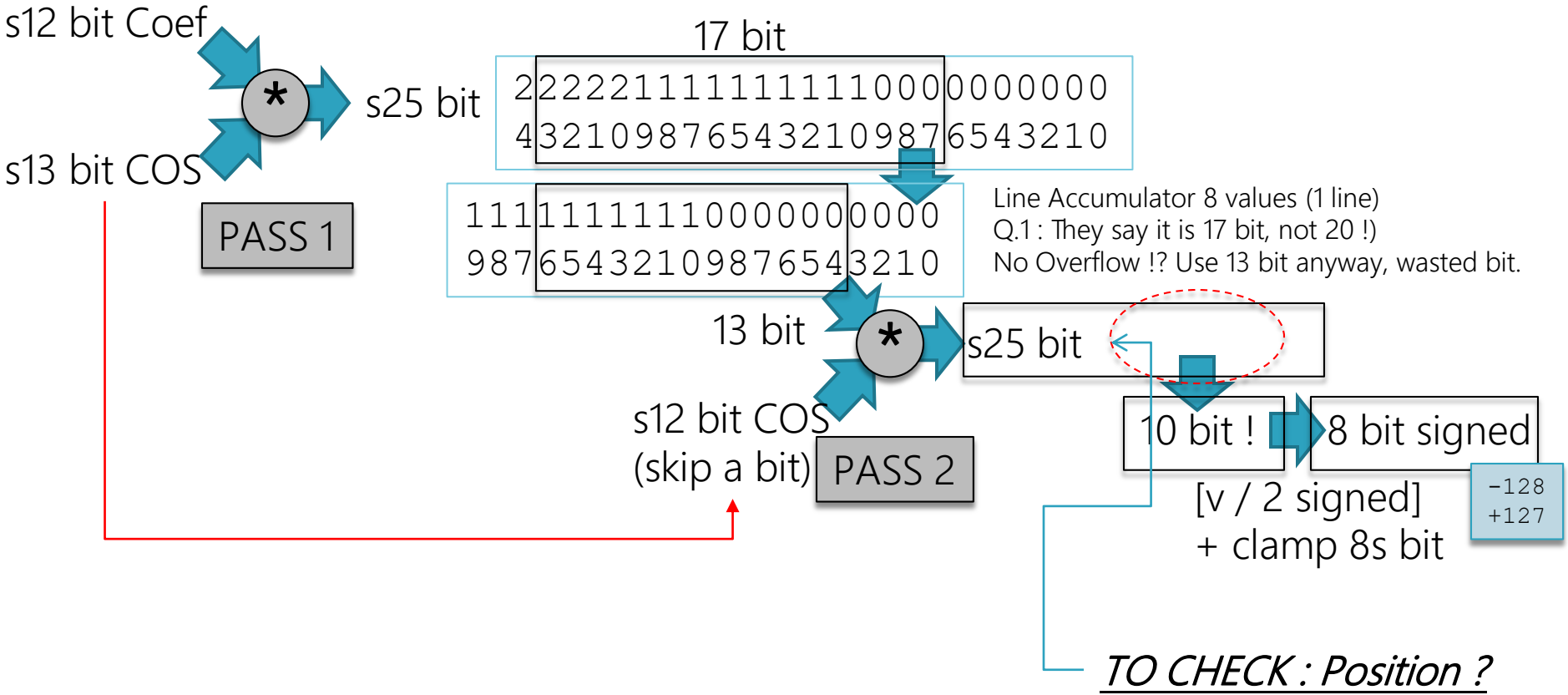
http://www.psxdev.net/forum/viewtopic.php?f=70&t=551

- Unit 03 : MDEC Input FIFO (32x32 bit)
- RLE Decoded + Compute coef. matrix values clamped to signed 12 bit. + **[round to next EVEN number closer to zero, except -1]**
- Scale Factor = 0 → Linear scan, not zigzag.
- YUV→RGB (Akari - 5/10/2014)
  - Const → 10 bit, Value Signed 8 bit,
  - G=(-88/256 * Cb)+(-183/256 * Cr) + Y
  - R=359/256 * Cr + Y
  - B=454/256 * Cb + Y



RGB→32 bit FIFO packing

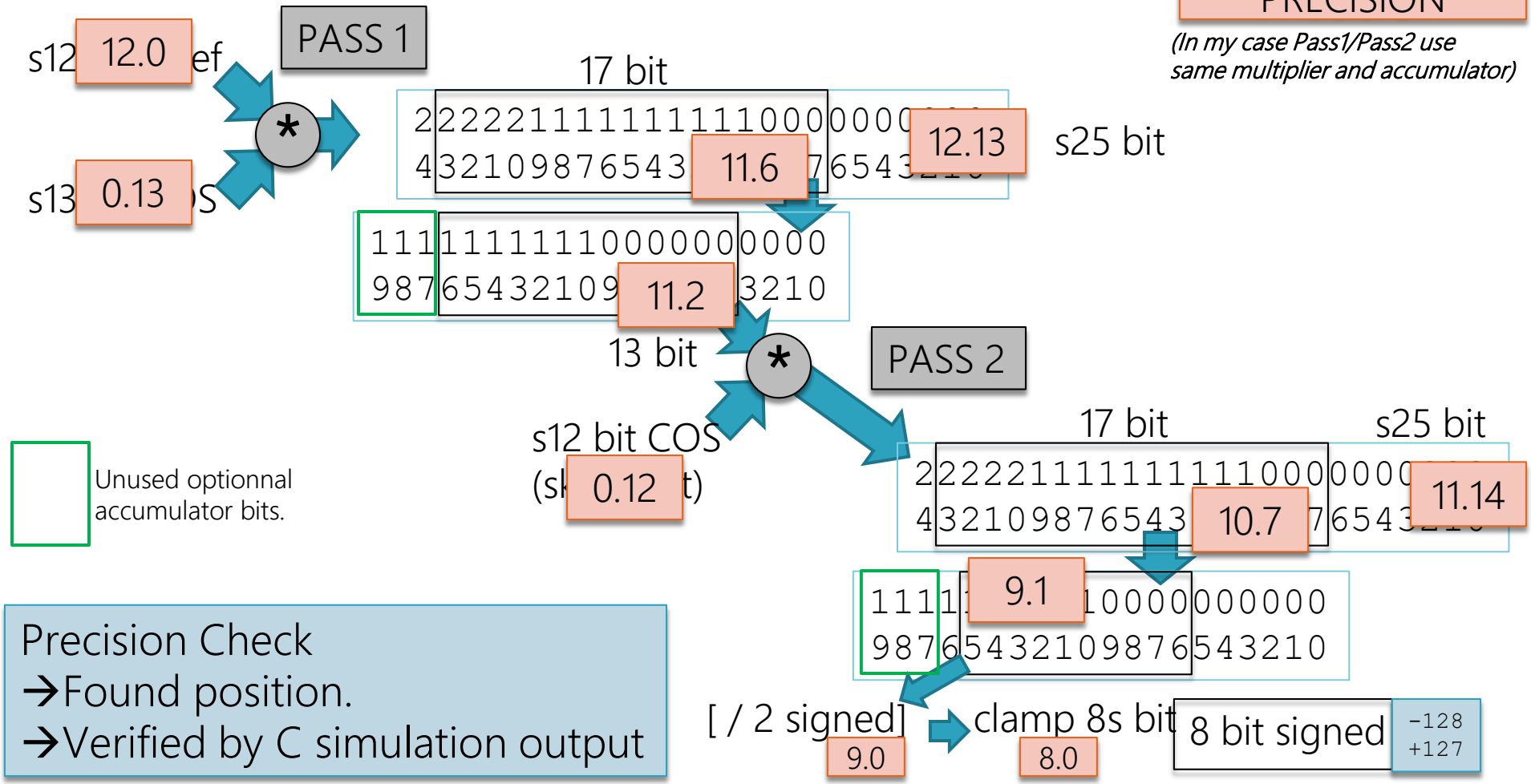
http://board.psxdev.ru/topic/9/



Complete IDCT Compute Path Precision

FIXED POINT PRECISION

(In my case Pass1/Pass2 use same multiplier and accumulator)



Precision Check
→Found position.
→Verified by C simulation output

Remaining work ?

- Read the log file in GitHub, it will show what has been done and what has to be done.

(don’t want to document here as it changes)

Appendix

Specification from No$PSX

as of 31/08/2019

MDEC I/O Ports

1F801820h - MDEC0 - MDEC Command/Parameter Register (W)

```
  31-0  Command or Parameters
```

Used to send command word, followed by parameter words to the MDEC (usually, only the command word is written to this register, and the parameter words are transferred via DMA0).

1F801820h.Read - MDEC Data/Response Register (R)

```
  31-0  Macroblock Data (or Garbage if there's no data available)
```

The data is always output as a 8x8 pixel bitmap, so, when manually reading from this register and using colored 16x16 pixel macroblocks, the data from four 8x8 blocks must be re-ordered accordingly (usually, the data is received via DMA1, which is doing the re-ordering automatically). For monochrome 8x8 macroblocks, no re-ordering is needed (that works with DMA1 too).

1F801824h - MDEC1 - MDEC Status Register (R)

```
  31    Data-Out Fifo Empty (0=No, 1=Empty)
  30    Data-In Fifo Full   (0=No, 1=Full, or Last word received)
  29    Command Busy  (0=Ready, 1=Busy receiving or processing parameters)
  28    Data-In Request  (set when DMA0 enabled and ready to receive data)
  27    Data-Out Request (set when DMA1 enabled and ready to send data)
  26-25 Data Output Depth  (0=4bit, 1=8bit, 2=24bit, 3=15bit)      ;CMD.28-27
  24    Data Output Signed (0=Unsigned, 1=Signed)                  ;CMD.26
  23    Data Output Bit15  (0=Clear, 1=Set) (for 15bit depth only) ;CMD.25
  22-19 Not used (seems to be always zero)
  18-16 Current Block (0..3=Y1..Y4, 4=Cr, 5=Cb) (or for mono: always 4=Y)
  15-0  Number of Parameter Words remaining minus 1  (FFFFh=None)  ;CMD.Bit0-15
```

Reset → status=80040000h

If there's data in the output fifo, then the Current Block bits are always set to the current output block number (ie. Y1..Y4; or Y for mono) (this information is apparently passed to the DMA1 controller, so that it knows if and how it must re-order the data in RAM). If the output fifo is empty, then the bits indicate the currently processsed incoming block (ie. Cr,Cb,Y1..Y4; or Y for mono).

1F801824h - MDEC1 - MDEC Control/Reset Register (W)

```
  31    Reset MDEC (0=No change, 1=Abort any command, and set status=80040000h)
  30    Enable Data-In Request  (0=Disable, 1=Enable DMA0 and Status.bit28)
  29    Enable Data-Out Request (0=Disable, 1=Enable DMA1 and Status.bit27)
  28-0  Unknown/Not used - usually zero
```

The data requests are required to be enabled for using DMA (and for reading the request status flags by software). The Data-Out request acts a bit strange: It gets set when a block is available, but, it gets cleared after reading the first some words of that block (nethertheless, one can keep reading the whole block, until the fifo-empty flag gets set).

DMA
MDEC decompression uses a lot of DMA channels,

1) DMA3 (CDROM)   to send compressed data from CDROM to RAM
2) DMA0 (MDEC.In) to send compressed data from RAM to MDEC
3) DMA1 (MDEC.Out) to send uncompressed macroblocks from MDEC to RAM
4) DMA2 (GPU)   to send uncompressed macroblocks from RAM to GPU

DMA0 and DMA1 should be usually used with a blocksize of 20h words. If necessary, the parameters for the MDEC(1) command should be padded with FE00h halfwords to match the 20h words (40h halfwords) DMA blocksize.

MDEC Commands

MDEC(1) - Decode Macroblock(s)

```
  31-29 Command (1=decode_macroblock)
  28-27 Data Output Depth  (0=4bit, 1=8bit, 2=24bit, 3=15bit)      ;STAT.26-25
  26    Data Output Signed (0=Unsigned, 1=Signed)                  ;STAT.24
  25    Data Output Bit15  (0=Clear, 1=Set) (for 15bit depth only) ;STAT.23
  24-16 Not used (should be zero)
  15-0  Number of Parameter Words (size of compressed data)
```

This command is followed by one or more Macroblock parameters (usually, all macroblocks for the whole image are sent at once).

MDEC(2) - Set Quant Table(s)

```
  31-29 Command (2=set_iqtab)
  28-1  Not used (should be zero)  ;Bit25-28 are copied to STAT.23-26 though
  0     Color   (0=Luminance only, 1=Luminance and Color)
```

The command word is followed by 64 unsigned parameter bytes for the Luminance Quant Table (used for Y1..Y4), and if Command.Bit0 was set, by another 64 unsigned parameter bytes for the Color Quant Table (used for Cb and Cr).

MDEC(3) - Set Scale Table

```
  31-29 Command (3=set_scale)
  28-0  Not used (should be zero)  ;Bit25-28 are copied to STAT.23-26 though
```

The command is followed by 64 signed halfwords with 14bit fractional part, the values should be usually/always the same values (based on the standard JPEG constants, although, MDEC(3) allows to use other values than that constants).

MDEC(0) - No function

This command has no function. Command bits 25-28 are reflected to Status bits 23-26 as usually. Command bits 0-15 are reflected to Status bits 0-15 (similar as the "number of parameter words" for MDEC(1), but without the "minus 1" effect, and without actually expecting any parameters).

MDEC(4..7) - Invalid

These commands act identical as MDEC(0).

MDEC Decompression

```
decode_colored_macroblock ;MDEC(1) command (at 15bpp or 24bpp depth)

  rl_decode_block(Crblk,src,iq_uv)                ;Cr (low resolution)
  rl_decode_block(Cbblk,src,iq_uv)                ;Cb (low resolution)
  rl_decode_block(Yblk,src,iq_y), yuv_to_rgb(0,0)  ;Y1 (and upper-left Cr,Cb)
  rl_decode_block(Yblk,src,iq_y), yuv_to_rgb(0,8)  ;Y2 (and upper-right Cr,Cb)
  rl_decode_block(Yblk,src,iq_y), yuv_to_rgb(8,0)  ;Y3 (and lower-left Cr,Cb)
  rl_decode_block(Yblk,src,iq_y), yuv_to_rgb(8,8)  ;Y4 (and lower-right Cr,Cb)


decode_monochrome_macroblock ;MDEC(1) command (at 4bpp or 8bpp depth)

  rl_decode_block(Yblk,src,iq_y), y_to_mono        ;Y


rl_decode_block(blk,src,qt)

  for i=0 to 63, blk[i]=0, next i   ;initially zerofill all entries (for skip)
 @@skip:
  n=[src], src=src+2, k=0           ;get first entry, init dest addr k=0
  if n=FE00h then @@skip            ;ignore padding (FE00h as first halfword)
  q_scale=(n SHR 10) AND 3Fh        ;contains scale value (not "skip" value)
  val=signed10bit(n AND 3FFh)*qt[k] ;calc first value (without q_scale/8) (?)
 @@lop:
  if q_scale=0 then val=signed10bit(n AND 3FFh)*2   ;special mode without qt[k]
  val=minmax(val,-400h,+3FFh)            ;saturate to signed 11bit range
  val=val*scalezag[i]                    ;<-- for "fast_idct_core" only
  if q_scale>0 then blk[zagzig[k]]=val   ;store entry (normal case)
  if q_scale=0 then blk[k]=val           ;store entry (special, no zigzag)
  n=[src], src=src+2                     ;get next entry (or FE00h end code)
  k=k+((n SHR 10) AND 3Fh)+1             ;skip zerofilled entries
  val=(signed10bit(n AND 3FFh)*qt[k]*q_scale+4)/8  ;calc value for next entry
  if k<=63 then jump @@lop          ;should end with n=FE00h (that sets k>63)
  idct_core(blk)
  return (with "src" address advanced)
```

real_idct_core(blk) ;low level "idct_core" version
Low level code with 1024 multiplications, using the scaletable from the MDEC(3) command. Computes dst=src*scaletable (using normal matrix maths, but with "src" being diagonally mirrored, ie. the matrices are processed column by column, instead of row by column), repeated with src/dst exchanged.

```
  src=blk, dst=temp_buffer
  for pass=0 to 1
    for x=0 to 7
      for y=0 to 7
        sum=0
        for z=0 to 7
          sum=sum+src[y+z*8]*(scaletable[x+z*8]/8)
        next z
        dst[x+y*8]=(sum+0fffh)/2000h              ;<-- or so?
      next y
    next x
    swap(src,dst)
  next pass
```

The "(sum+0fffh)/2000h" part is meant to strip fractional bits, and to round-up the result if the fraction was BIGGER than 0.5. The hardware appears to be working roughly like that, still the results aren't perfect. Maybe the real hardware is doing further roundings in other places, possibly stripping some fractional bits before summing up "sum", possibly stripping different amounts of bits in the two "pass" cycles, and possibly keeping a final fraction passed on to the y_to_mono stage.

yuv_to_rgb(xx,yy)

```
  for y=0 to 7
    for x=0 to 7
      R=[Crblk+((x+xx)/2)+((y+yy)/2)*8], B=[Cbblk+((x+xx)/2)+((y+yy)/2)*8]
      G=(-0.3437*B)+(-0.7143*R), R=(1.402*R), B=(1.772*B)
      Y=[Yblk+(x)+(y)*8]
      R=MinMax(-128,127,(Y+R))
      G=MinMax(-128,127,(Y+G))
      B=MinMax(-128,127,(Y+B))
      if unsigned then BGR=BGR xor 808080h  ;aka add 128 to the R,G,B values
      dst[(x+xx)+(y+yy)*16]=BGR
    next x
  next y
```

Note: The exact fixed point resolution for "yuv_to_rgb" is unknown. And, there's probably also some 9bit limit (similar as in "y_to_mono").

y_to_mono

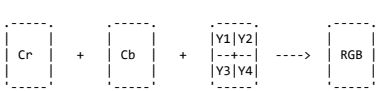
```
  for i=0 to 63
    Y=[Yblk+i]
    Y=Y AND 1FFh                  ;clip to signed 9bit range
    Y=MinMax(-128,127,Y)          ;saturate from 9bit to signed 8bit range
    if unsigned then Y=Y xor 80h  ;aka add 128 to the Y value
    dst[i]=Y
  next i
```

set_iqtab ;MDEC(2) command

```
  iqtab_core(iq_y,src), src=src+64       ;luminance quant table
  if command_word.bit0=1
    iqtab_core(iq_uv,src), src=src+64    ;color quant table (optional)
  endif
```

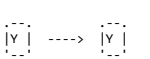
```
iqtab_core(iq,src) ;src = 64 unsigned paramter bytes

  for i=0 to 63, iq[i]=src[i], next i

Note: For "fast_idct_core" one could precalc "iq[i]=src[i]*scalezag[i]", but that would conflict with the RLE saturation/rounding steps (though those steps aren't actually required, so a very-fast decoder could omit them).

scalefactor[0..7] = cos((0..7)*90'/8) ;for [1..7]: multiplied by sqrt(2)

  1.000000000, 1.387039845, 1.306562965, 1.175875602,
  1.000000000, 0.785694958, 0.541196100, 0.275899379

zigzag[0..63] =
  0 ,1 ,5 ,6 ,14,15,27,28,
  2 ,4 ,7 ,13,16,26,29,42,
  3 ,8 ,12,17,25,30,41,43,
  9 ,11,18,24,31,40,44,53,
  10,19,23,32,39,45,52,54,
  20,22,33,38,46,51,55,60,
  21,34,37,47,50,56,59,61,
  35,36,48,49,57,58,62,63

scalezag[0..63] (precalulated factors, for "fast_idct_core")
  for y=0 to 7
   for x=0 to 7
    scalezag[zigzag[x+y*8]] = scalefactor[x] * scalefactor[y] / 8
   next x
  next y


zagzig[0..63] (reversed zigzag table)
  for i=0 to 63, zagzig[zigzag[i]]=i, next i

set_scale_table: ;MDEC(3) command
This command defines the IDCT scale matrix, which should be usually/always:
  5A82 5A82 5A82 5A82 5A82 5A82 5A82 5A82
  7D8A 6A6D 471C 18F8 E707 B8E3 9592 8275
  7641 30FB CF04 89BE 89BE CF04 30FB 7641
  6A6D E707 8275 B8E3 471C 7D8A 18F8 9592
  5A82 A57D A57D 5A82 5A82 A57D A57D 5A82
  471C 8275 18F8 6A6D 9592 E707 7D8A B8E3
  30FB 89BE 7641 CF04 CF04 7641 89BE 30FB
  18F8 B8E3 6A6D 8275 7D8A 9592 471C E707

Note that the hardware does actually use only the upper 13bit of those 16bit values. The values are choosen like so,
  +s0  +s0  +s0  +s0  +s0  +s0  +s0  +s0
  +s1  +s3  +s5  +s7  -s7  -s5  -s3  -s1
  +s2  +s6  -s6  -s2  -s2  -s6  +s6  +s2
  +s3  -s7  -s1  -s5  +s5  +s1  +s7  -s3
  +s4  -s4  -s4  +s4  +s4  -s4  -s4  +s4
  +s5  -s1  +s7  +s3  -s3  -s7  +s1  -s5
  +s6  -s2  +s2  -s6  -s6  +s2  -s2  +s6
  +s7  -s5  +s3  -s1  +s1  -s3  +s5  -s7

whereas, s0..s7 = scalefactor[0..7], multiplied by sqrt(2) (ie. by 1.414), and multiplied by 4000h (ie. with 14bit fractional part).
```

Colored Macroblocks (16x16 pixels) (in 15bpp or 24bpp depth mode)
Each macroblock consists of six blocks: Two low-resolution blocks with color information (Cr,Cb) and four full-resolution blocks with luminance (grayscale) information (Y1,Y2,Y3,Y4). The color blocks are zoomed from 8x8 to 16x16 pixel size, merged with the luminance blocks, and then converted from YUV to RGB format.



Native PSX files are usually containing vertically arranged Macroblocks (eg. allowing to send them to the GPU as 16x240 portion) (JPEG-style horizontally arranged Macroblocks would require to send the data in 16x16 pixel portions to the GPU) (something like 320x16 won't work, since that'd require to wrap from the bottom of the first macroblock to the top of the next macroblock).

Monochrome Macroblocks (8x8 pixel) (in 4bpp or 8bpp depth mode)
Each macroblock consist of only one block: with luminance (grayscale) information (Y), the data comes out as such (it isn't converted to RGB).



The output is an 8x8 bitmap (not 16x16), so it'd be send to the GPU as 8x8 pixel rectangle, or multiple blocks at once as 8x240 pixel rectangle. Since the data isn't RGB, it should be written to Texture memory (and then it can be forwarded to the frame buffer in form of a texture with monochrome 15bit palette with 32 grayscales). Alternately, one could convert the 8bpp image to 24bpp by software (this would allow to use 256 grayscales).

Blocks (8x8 pixels)
An (uncompressed) block consists of 64 values, representing 8x8 pixels. The first (upper-left) value is an absolute value (called "DC" value), the remaining 63 values are relative to the DC value (called "AC" values). After decompression and zig-zag reordering, the data in unfiltered horizontally and vertically (IDCT conversion, ie. the relative "AC" values are converted to absolute "DC" values).

MDEC vs JPEG
The MDEC data format is very similar to the JPEG file format, the main difference is that JPEG uses Huffman compressed blocks, whilst MDEC uses Run-Length (RL) compressed blocks.
The (uncompressed) blocks are same as in JPEGs, using the same zigzag ordering, AC to DC conversion, and YUV to RGB conversion (ie. the MDEC hardware can be also used to decompress JPEGs, when handling the file header and huffman decompression by software).
Some other differences are that MDEC has only 2 fixed-purpose quant tables, whilst JPEGs <can> use up to 4 general-purpose quant tables. Also, JPEGs <can> use other color resolutions than the 8x8 color info for 16x16 pixels. Whereas, JPEGs <can> do that stuff, but most standard JPEG files aren't actually using 4 quant tables, nor higher color resolution.

Run-Length compressed Blocks
Within each block the DCT information and RLE compressed data is stored:

```
  DCT              ;1 halfword
  RLE,RLE,RLE,etc. ;0..63 halfwords
  EOB              ;1 halfword
```

DCT (1st value)
DCT data has the quantization factor and the Direct Current (DC) reference.

```
  15-10 Q    Quantization factor (6 bits, unsigned)
  9-0   DC   Direct Current reference (10 bits, signed)
```

Contains the absolute DC value (the upper-left value of the 8x8 block).

RLE (Run length data, for 2nd through 64th value)

```
  15-10 LEN  Number of zero AC values to be inserted (6 bits, unsigned)
  9-0   AC   Relative AC value (10 bits, signed)
```

Example: AC values "000h,000h,123h" would be compressed as "(2 shl 10)+123h".

EOB (End Of Block)
Indicates the end of a 8x8 pixel block, causing the rest of the block to be padded with zero AC values.

```
  15-0  End-code (Fixed, FE00h)
```

EOB isn't required if the block was already fully defined (up to including blk[63]), however, most games seem to append EOB to all blocks (although it's just acting as dummy/padding value in case of fully defined blocks).

Need to check implementation. EOB on full block does nothing too ?
Both spec OK ?

Dummy halfwords
Data is sent in units of words (or, when using DMA, even in units of 32-words), which is making it neccessary to send some dummy halfwords (unless the compressed data size should match up the transfer unit). The value FE00h can be used as dummy value: When FE00h appears at the begin of a new block, or after the end of block, then it is simply ignored by the hardware (if it occurs elsewhere, then it acts as EOB end code, as described above).

Our implementation timing…