The lab system and JPEG acceleration

The Need for Speed

or Power

or Cost

Per Karlström
Olle Seger

Some tips about arrays/memories

In our FPGA memories can be synthesized using:

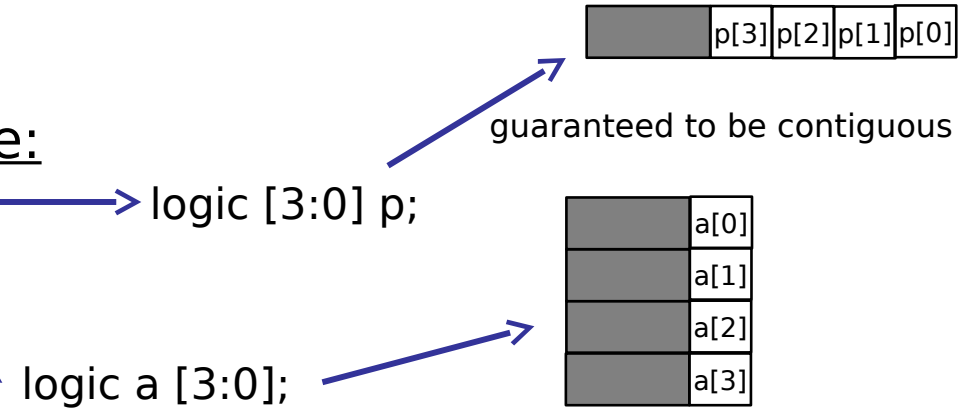
- F/Fs: asynch read, synch write, 1 x 1
- Distributed using LUTs: asynch read, synch write, 16 x 1
- BRAMs: synch read, synch write, 512 x 32, 1024 x 16, ...

Memories can be designed:

- Using templates (BRAMs)
- Inferred (distributed)

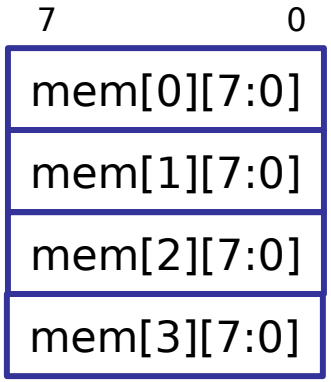
Arrays in SV can be:

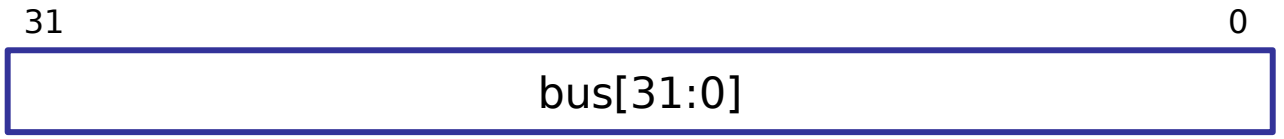
- packed → logic [3:0] p;
- unpacked → logic a [3:0];



Arrays unpacked

```
wire [31:0] bus;
reg [7:0] mem [0:3];  // a 4-byte memory
…
assign bus[31:24] = mem[3];
```

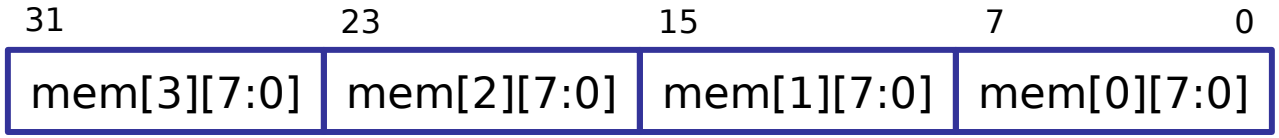


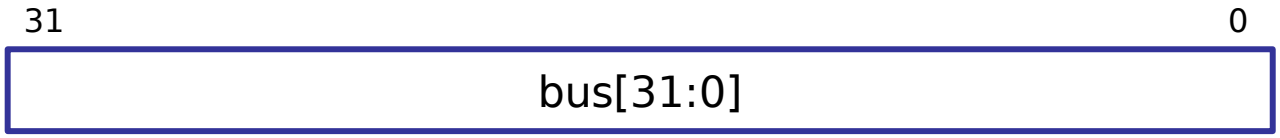


Arrays packed

```
wire [31:0] bus;                    // a packed array
reg [3:0][7:0] mem;                 // so is this
                                    // both are contiguous

assign bus = mem;
assign bus[31:16] = mem[3:2];
```



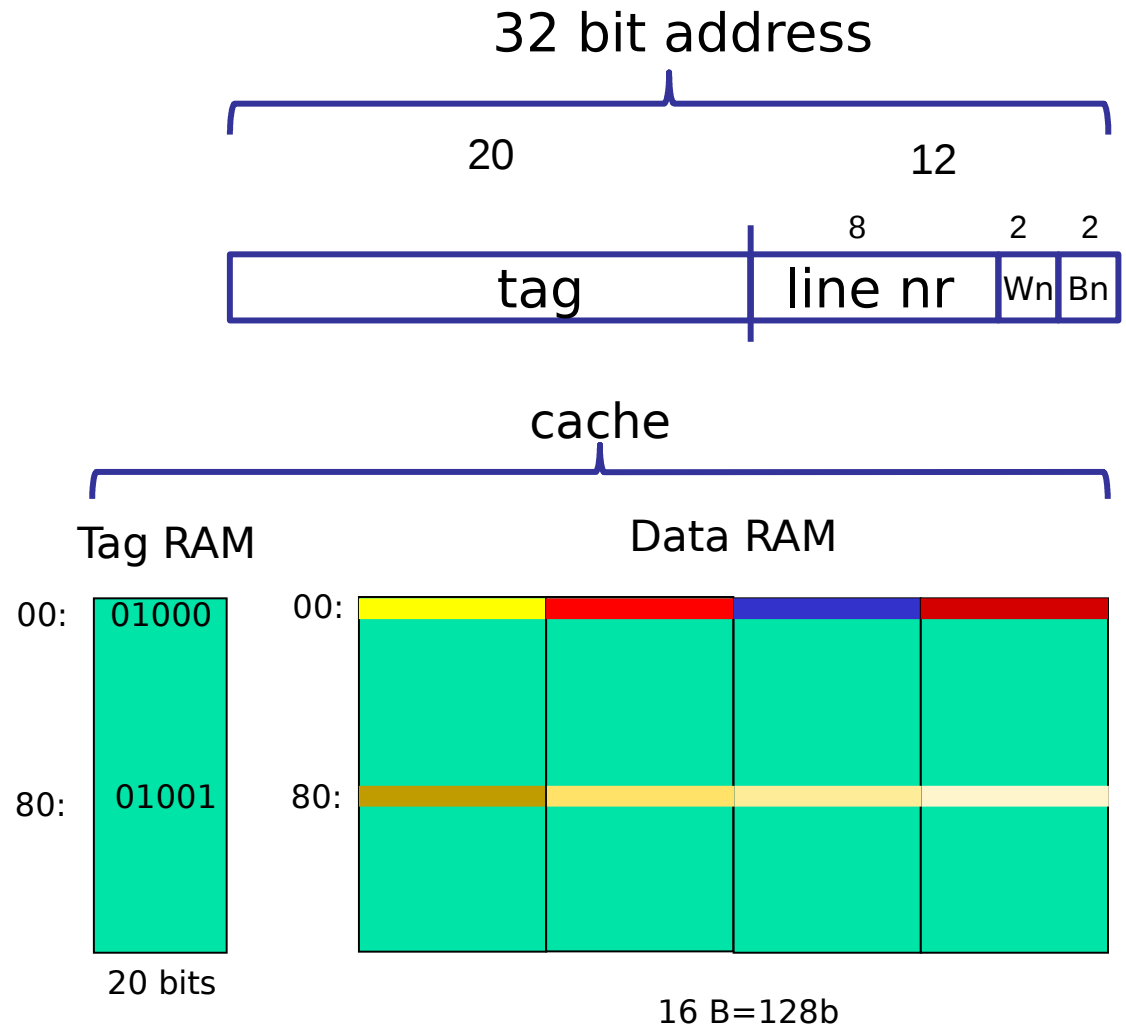


Caches

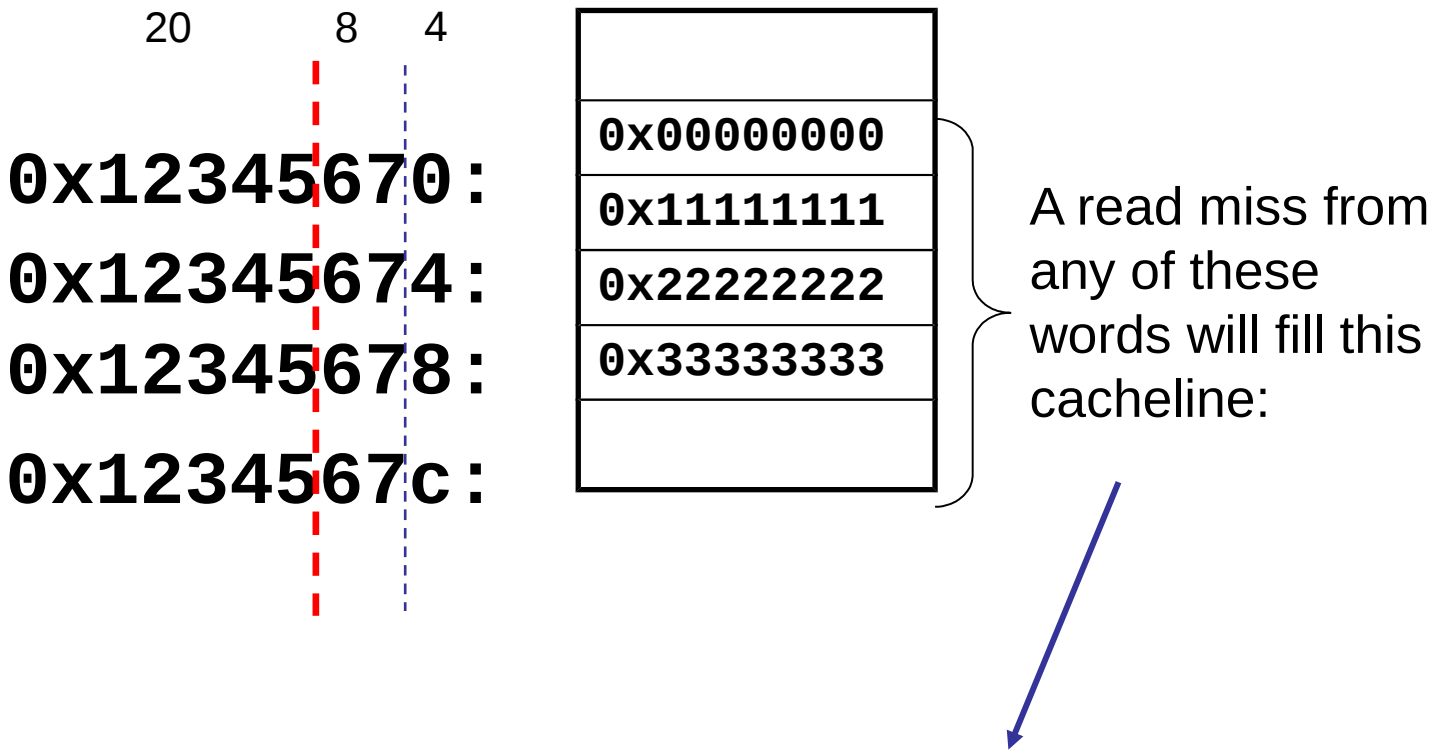
- Are essential! Without them we can forget
  about 1 CPI!
- We want to fetch 1 instruction every clock cycle
  (internal mem 3 CK, external mem 4 CK)
- **Size**: depending on the FPGA
  we have 120 x 2KB block RAMs
  => 8kB each IC,DC
  Type: direct mapped (or set associative)

4kB cache example

⇒ more than 1 word is fetched on a cache miss
⇒ a cache line is 4 words = 16 bytes
⇒ same size of data RAM, but 256x128

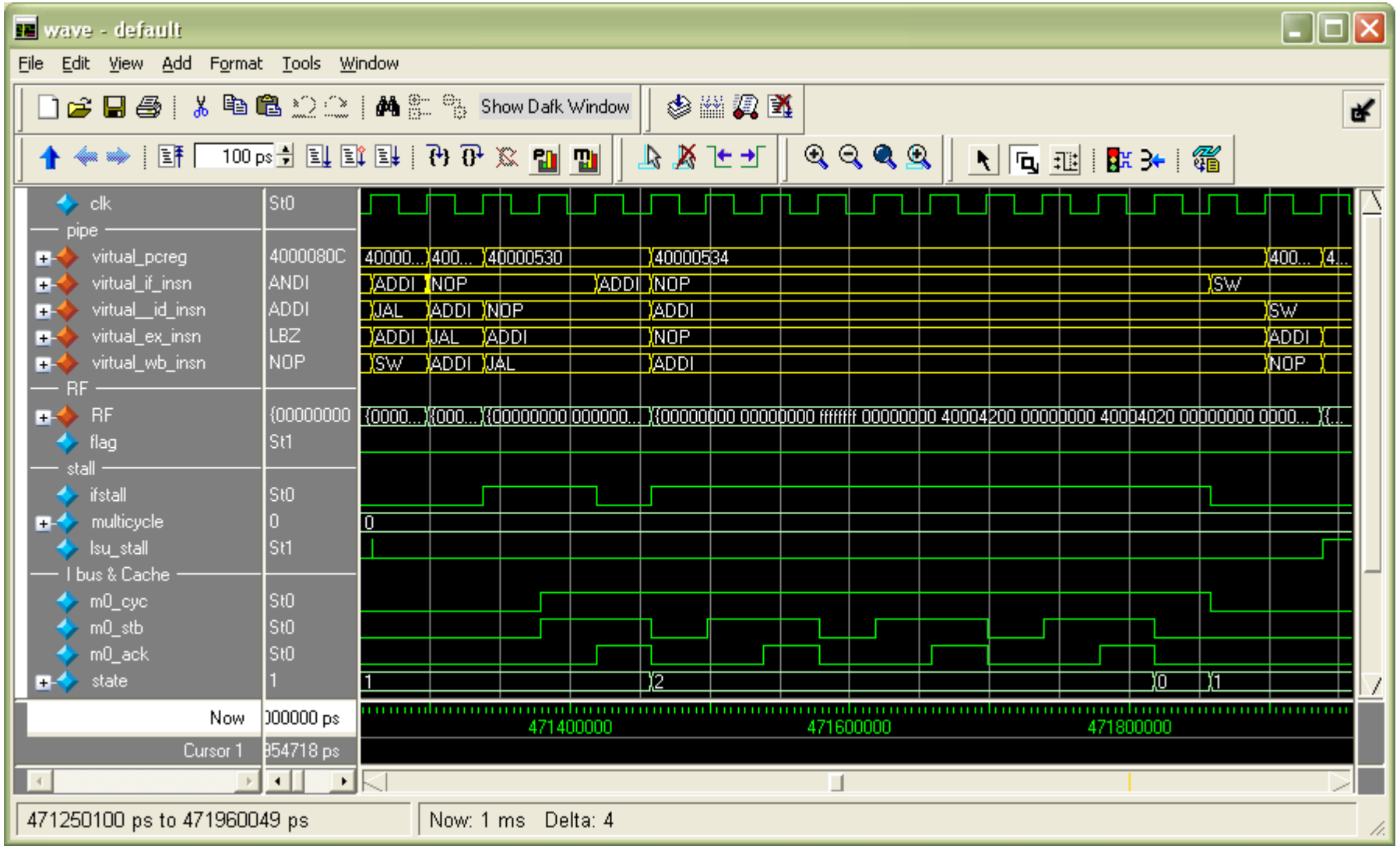


A closer look at a cacheline



An IC cache miss

```
40000530:    9c 21 ff e4    l.addi r1,r1,0xffffffe4
40000534:    d4 01 48 04    l.sw 0x4(r1),r9
40000538:    d4 01 50 08    l.sw 0x8(r1),r10
```



Cache policy

Cacheline = 4 words = 16B

Instruction cache

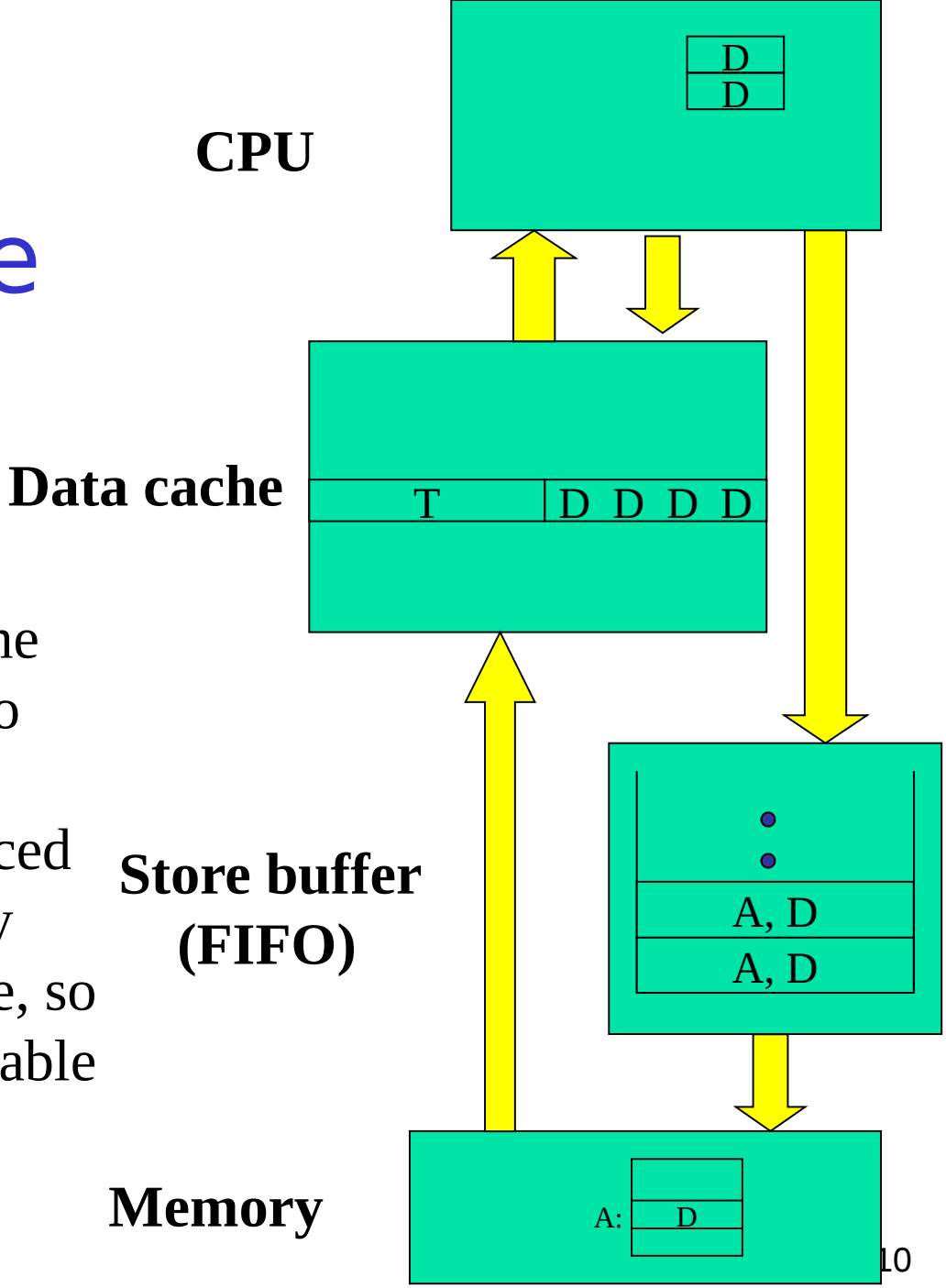
	hit	miss
read	read from cache	fill (replace) cacheline from memory

Data cache

	hit	miss
read	read from cache	fill (replace) cacheline from memory
write	write to cache write thru to memory	write to memory only

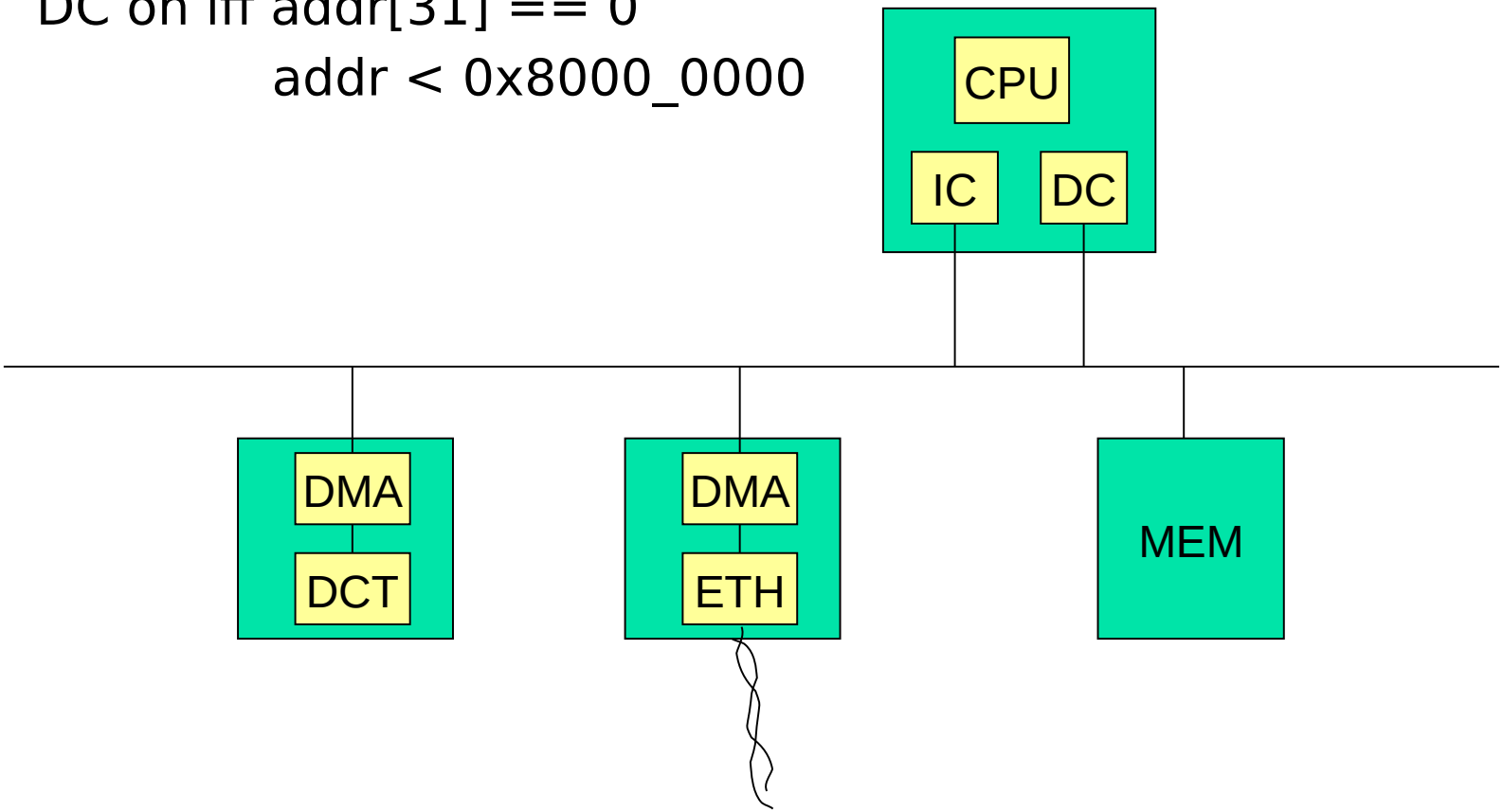
or1200 store buffer

- In a write-through data cache every write is equivalent to a cache miss!
- A store (write) buffer is placed between CPU and memory
- Writes are placed in a queue, so that the data cache is available on the next clock cycle

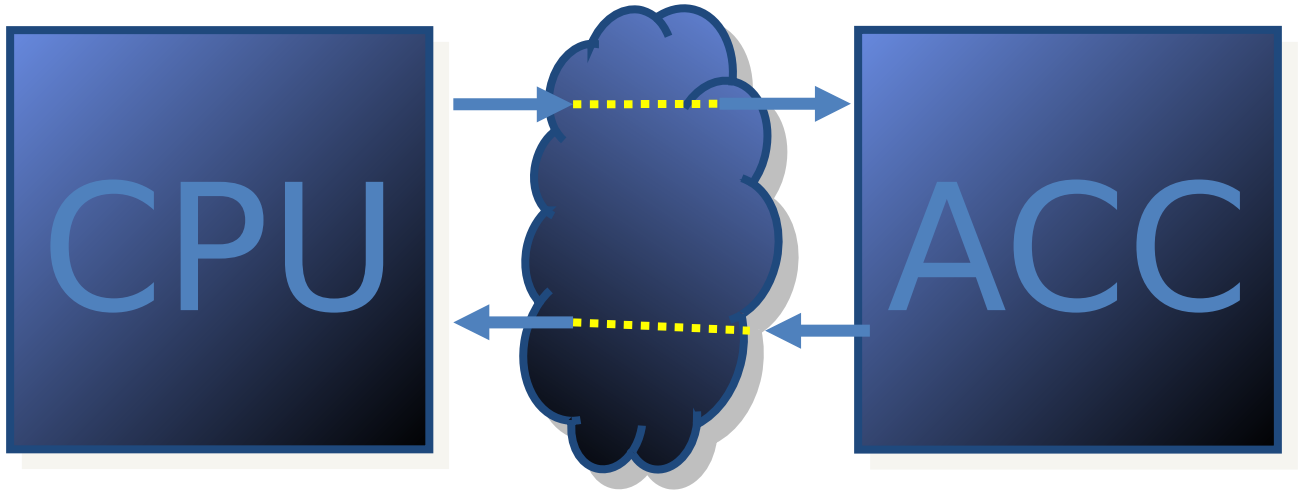


Watch out!

- Caches can be incoherent when using DMA.
- Parts of memory should be non-cacheable
  - IC on for all addresses
  - DC on iff addr[31] == 0
    addr < 0x8000_0000

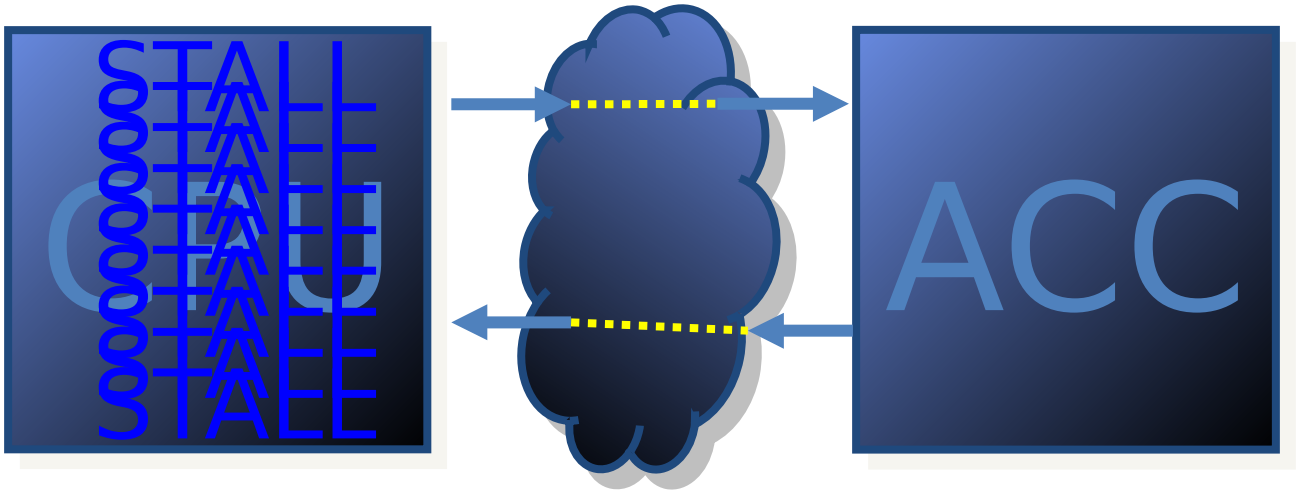


Accelerators, single cycle



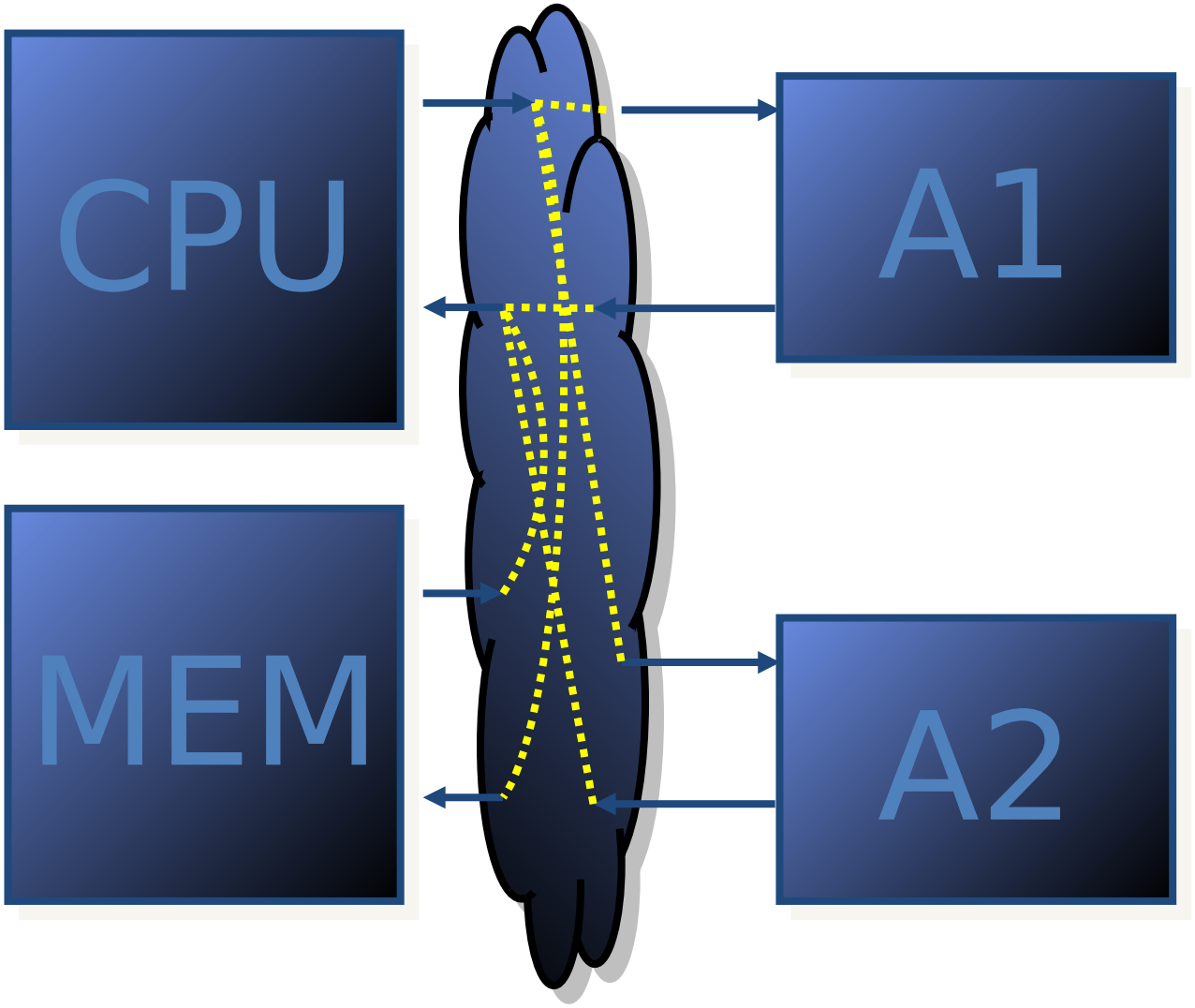
1 cycle

Accelerators, multi cycle

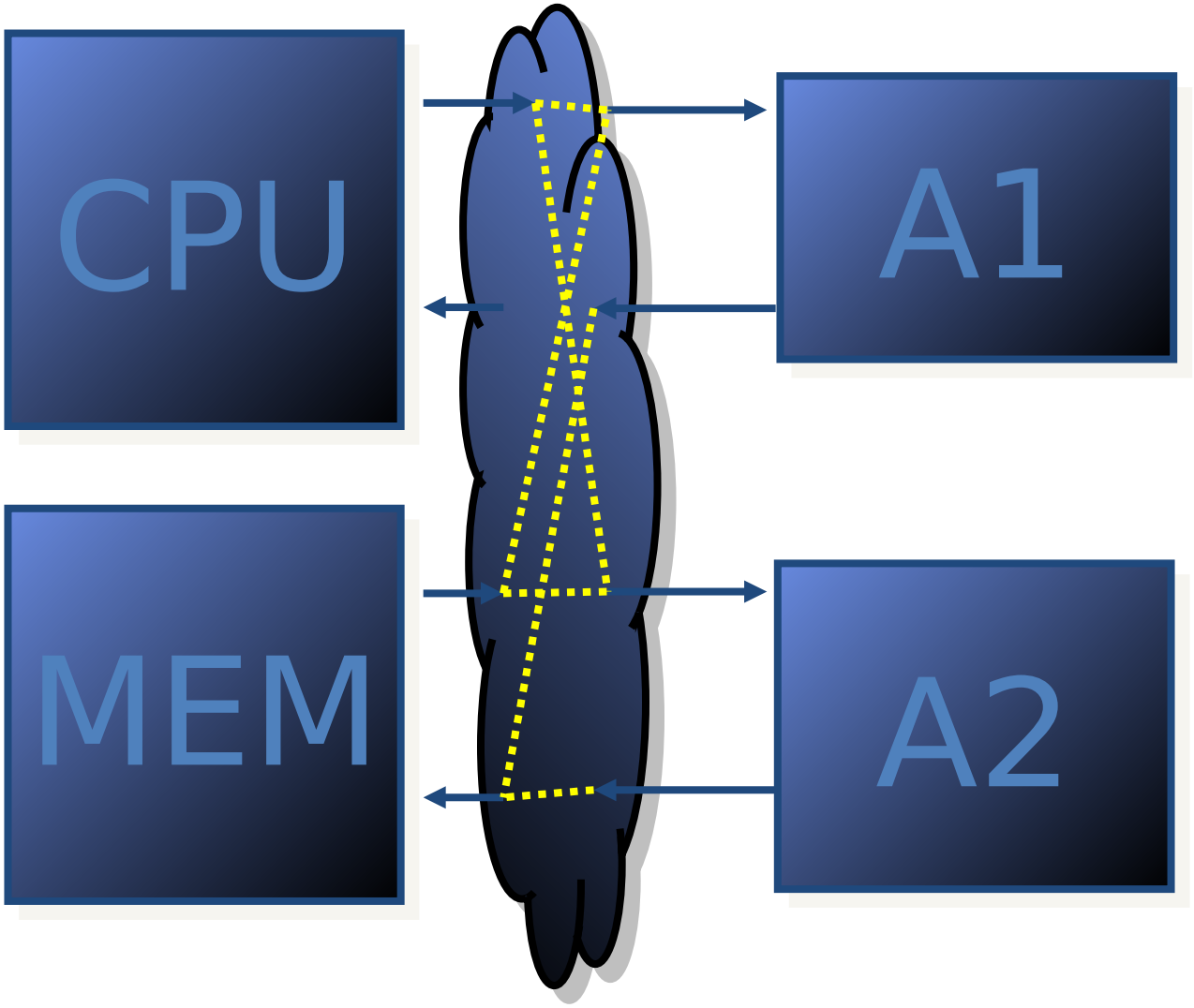


n cycles

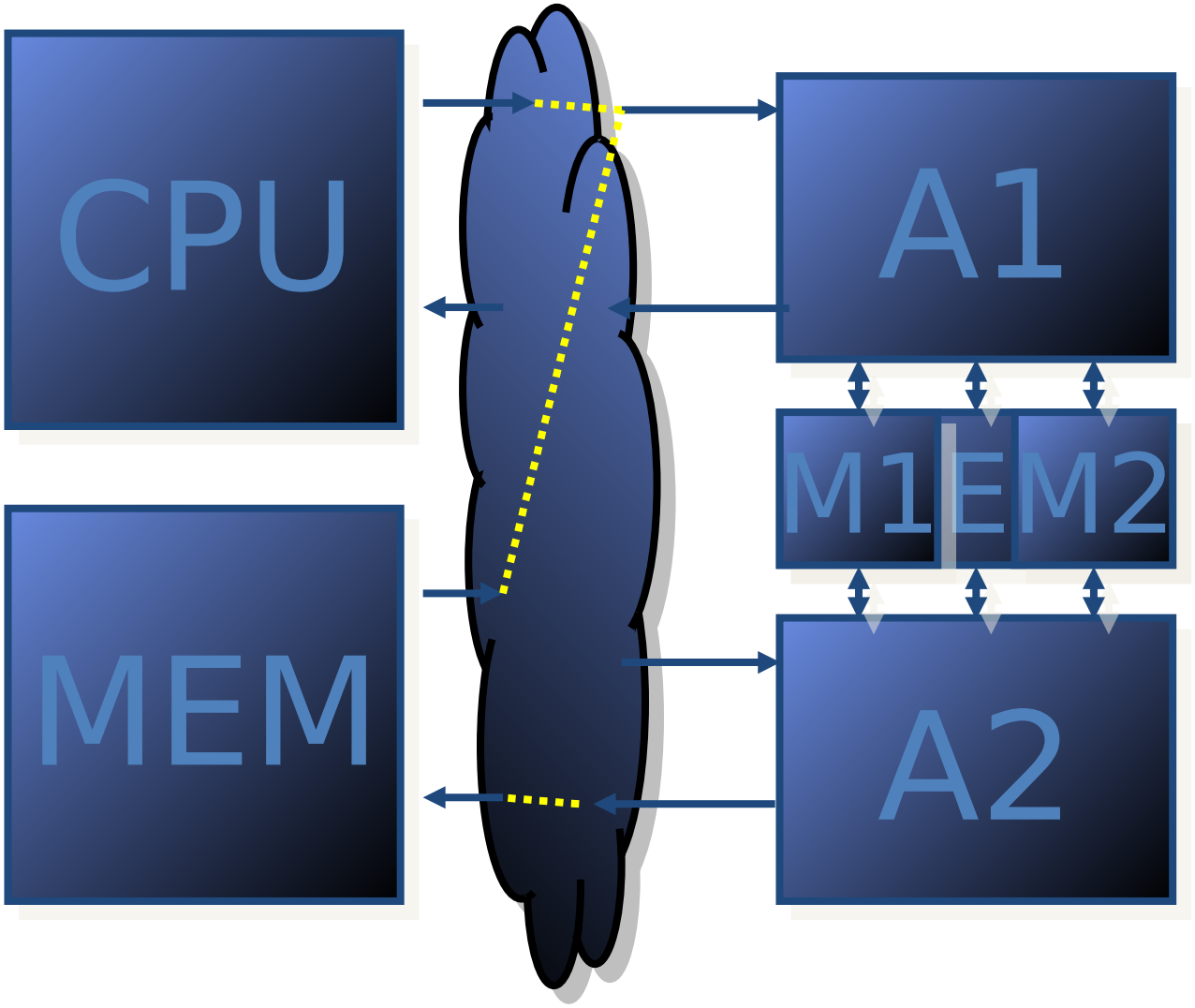
Accelerators - Through CPU



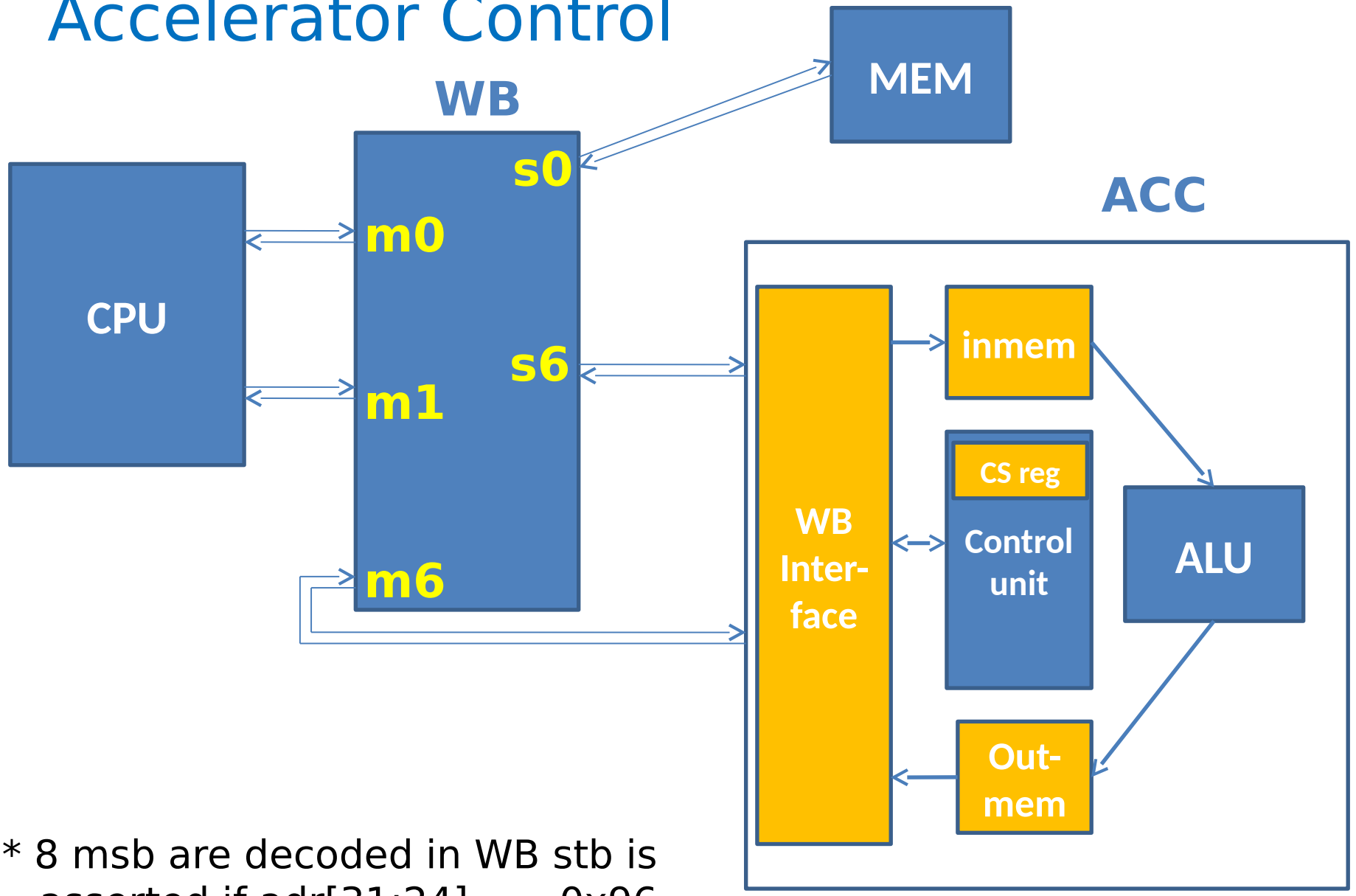
Accelerators - DMA



Accelerators - Extra Memory



Accelerator Control



* 8 msb are decoded in WB stb is asserted if adr[31:24] == 0x96
* Extra address decoding in ACC

JPEG Introduction

Joint Photographers Expert Group

Remove things we cannot see

encode

decode

Raw image
Camera

Color conversion

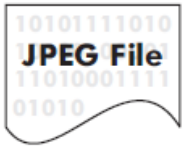
Down sampling

DCT

Quantization

RLE

Huffman encoding



Huffman decoding

RLD

De-quantization

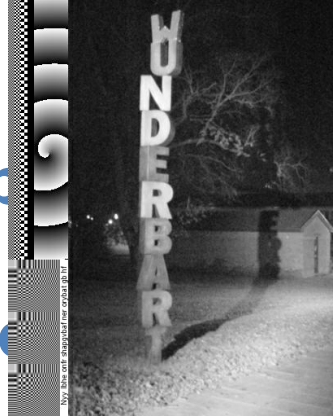
IDCT

Up sampling

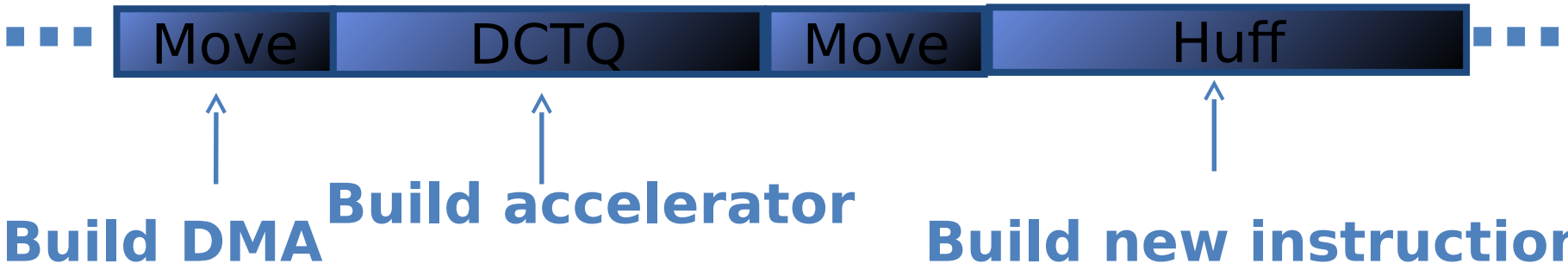
Color conversion

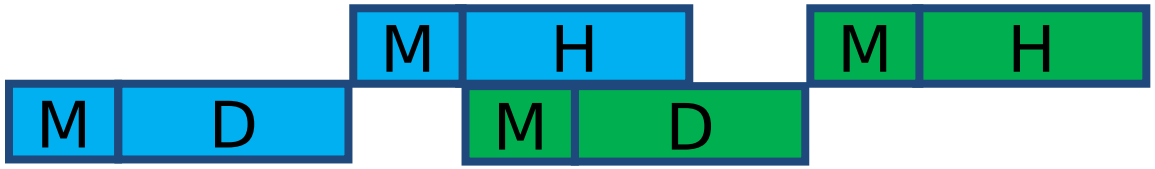
Raw image
Display

Problem



JPEG compression of testbild.raw 512x400 p
JPEG works on 8x8 blocks => 3200 blocks
Unacc JPEG takes more than 32 000 000 clo
=> 1 block takes more 10 000 clocks





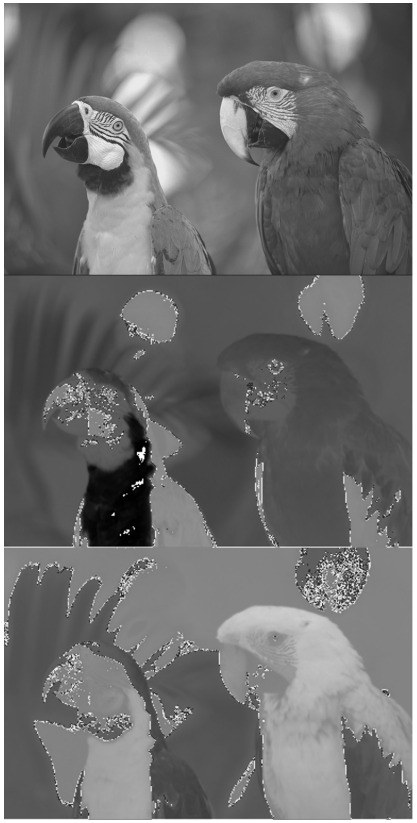
Color Conversion

$$Y = 0.299R + 0.587G + 0.144B$$
$$Cb = -0.1687R - 0.3313G + 0.5B + 2^{Ps-1}$$
$$Cr = 0.5R - 0.4187G - 0.0813B + 2^{Ps-1}$$

Y

Cb

Cr



R

G

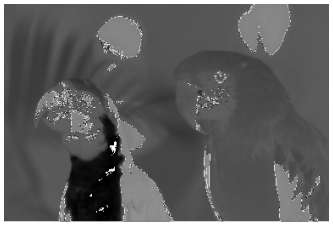
B

Y = luminance,
Cb/Cr = chrominance

Resampling

Y

Cb



Cr

Data Reduction
50%

8-point 1-D DCT/IDCT

$$T(k) = c(k)\sum_{x=0}^{7} v(x)\cos\left[\frac{(2x+1)k\pi}{16}\right], \quad k = 0 \cdots 7$$

$$v(x) = \sum_{k=0}^{7} c(k)T(k)\cos\left[\frac{(2x+1)k\pi}{16}\right], \quad x = 0 \cdots 7$$

$$c(0) = \sqrt{\frac{1}{8}}$$

$$c(k) = \frac{1}{2}, k \neq 0$$

$$C(x;k) = \cos\left[\frac{(2x+1)k\pi}{16}\right]$$

coord freq

8x8-point 2-D DCT/IDCT

$$T(k,l) = c(k,l)\sum_{x=0}^{7}\sum_{y=0}^{7} v(x,y)\cdot C(y;l)C(x;k), \quad k,l = 0\cdots 7$$

$$v(x,y) = \sum_{k=0}^{7}\sum_{l=0}^{7} c(k,l)T(k,l)\cdot C(y;l)C(x;k), \quad x,y = 0\cdots 7$$

$$c(0,0) = \frac{1}{8} \quad k = l = 0$$

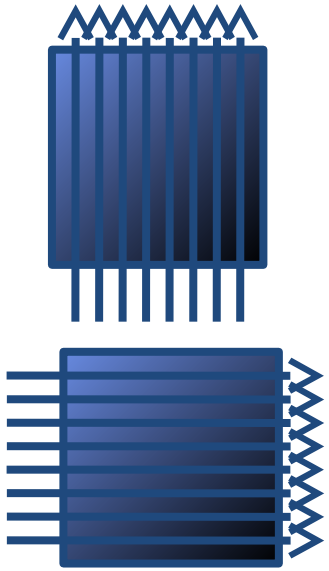
$$c(k,l) = \frac{1}{4} \qquad else$$

Simplifications

1) Separation in x and y

$$T(k,l) = c(k,l)\sum_{x=0}^{7}\left[\sum_{y=0}^{7} v(x,y)C(y;l)\right]\cdot C(x;k)$$

$$= c(k,l)\sum_{x=0}^{7} \mathrm{B}(x,l)\cdot C(x;k)$$

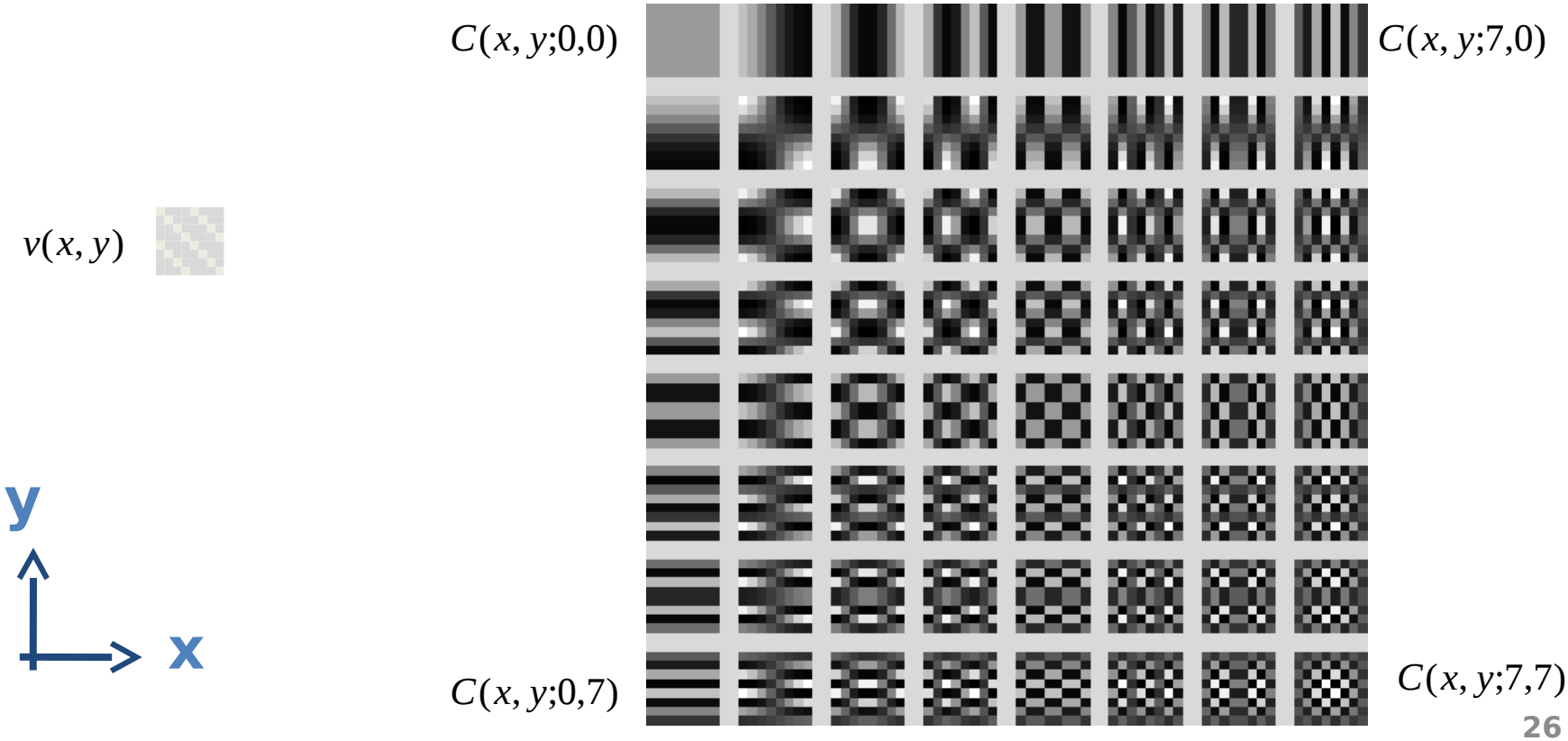


2) 1-D DCT can be simplified for N=8

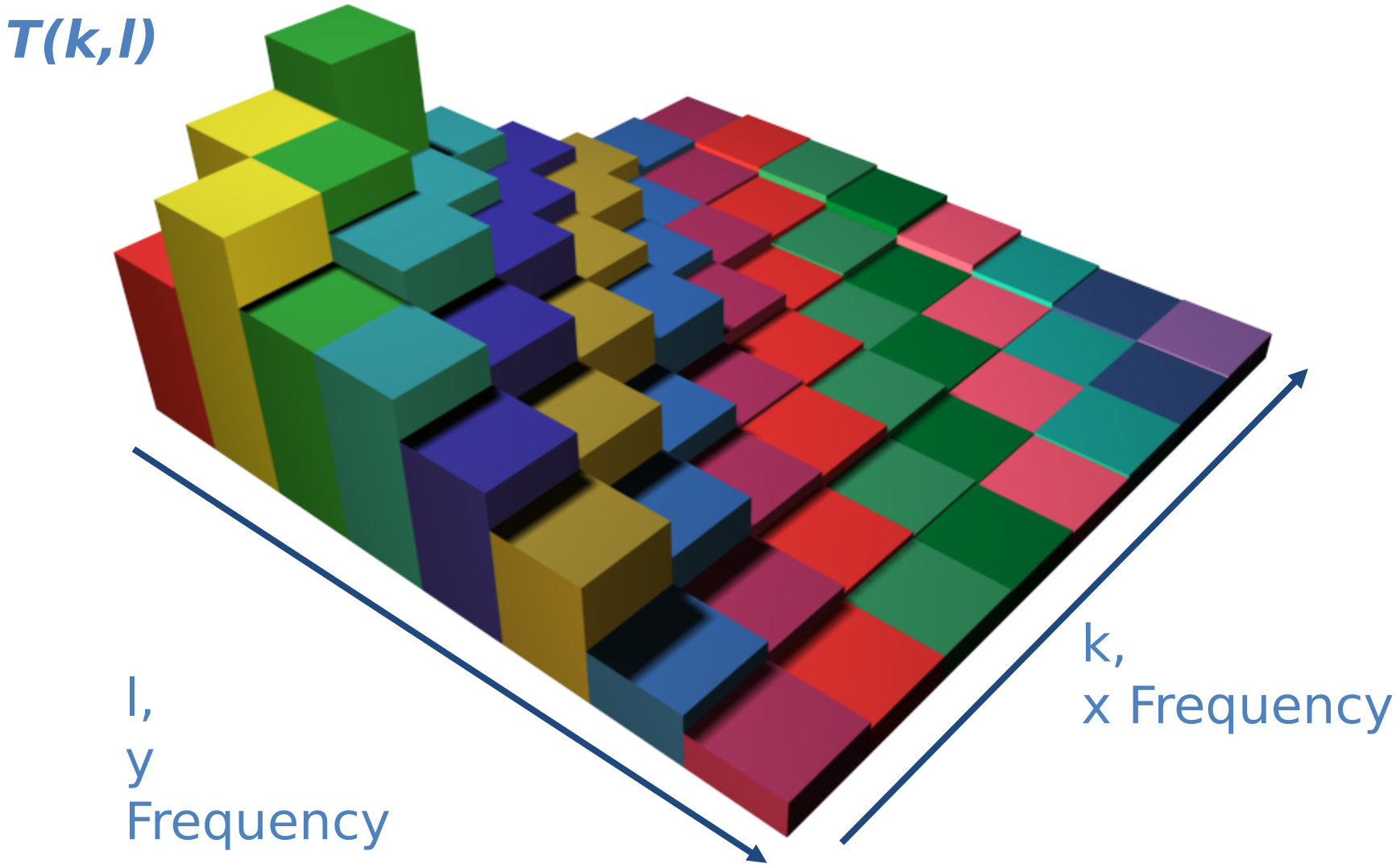
Meaning of the transform

$$v(x,y) = \sum_{k=0}^{7}\sum_{l=0}^{7} T(k,l)\cdot c(k,l)C(y;l)C(x;k)$$

$$= \sum_{k=0}^{7}\sum_{l=0}^{7} T(k,l)\cdot C(x,y;k,l)$$



Quantization



Data Reduction

$$Y = \begin{bmatrix} 162 & 162 & 162 & 161 & 162 & 157 & 163 & 161 \\ 162 & 162 & 162 & 161 & 162 & 157 & 163 & 161 \\ 162 & 162 & 162 & 161 & 162 & 157 & 163 & 161 \\ 162 & 162 & 162 & 161 & 162 & 157 & 163 & 161 \\ 162 & 162 & 162 & 161 & 162 & 157 & 163 & 161 \\ 164 & 164 & 158 & 155 & 161 & 159 & 159 & 160 \\ 160 & 160 & 163 & 158 & 160 & 162 & 159 & 156 \\ 159 & 159 & 155 & 157 & 158 & 159 & 156 & 157 \end{bmatrix}$$

$$\mathcal{DCT}_2(Y) = \begin{bmatrix} 259 & 5 & 3 & 0 & 0 & -1 & -5 & 6 \\ 8 & -1 & 1 & -5 & 2 & 3 & -4 & 3 \\ -5 & 0 & -2 & 2 & -1 & 0 & 2 & -2 \\ 2 & 1 & 2 & 1 & -1 & -1 & 0 & 1 \\ -1 & -1 & 0 & -1 & 2 & 1 & -1 & -1 \\ 1 & 0 & -2 & 0 & -2 & 1 & 2 & 1 \\ -2 & 0 & 3 & 2 & 2 & -2 & -1 & -1 \\ 1 & 0 & -2 & -2 & -1 & 2 & 1 & 1 \end{bmatrix}$$

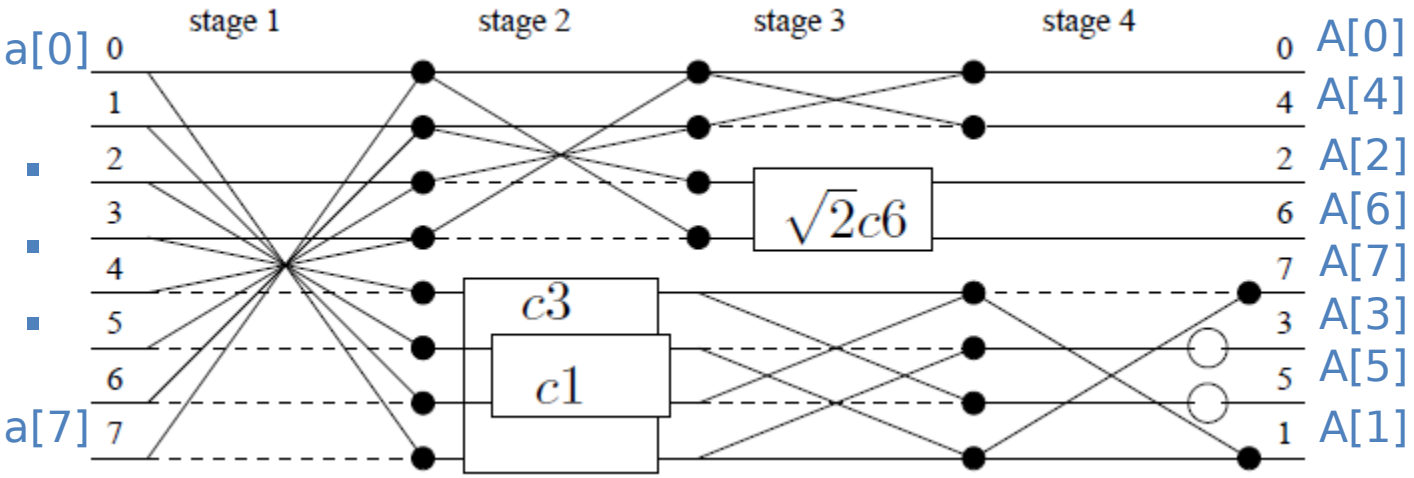
$$Q_L = \begin{bmatrix} 16 & 11 & 10 & 16 & 24 & 40 & 51 & 61 \\ 12 & 12 & 14 & 19 & 26 & 58 & 60 & 55 \\ 14 & 13 & 16 & 24 & 40 & 57 & 69 & 56 \\ 14 & 17 & 22 & 29 & 51 & 87 & 80 & 62 \\ 18 & 22 & 37 & 56 & 68 & 109 & 103 & 77 \\ 24 & 35 & 55 & 64 & 81 & 104 & 113 & 92 \\ 49 & 64 & 78 & 87 & 103 & 121 & 120 & 101 \\ 72 & 92 & 95 & 98 & 112 & 100 & 103 & 99 \end{bmatrix}$$

$$Y = \begin{bmatrix} 16 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

Loefflers algorithm

$$A[u] = c[u] \cdot \sum_{x=0}^{7} a[x] \cos\left(\frac{2\pi}{32}(2x+1)u\right)$$

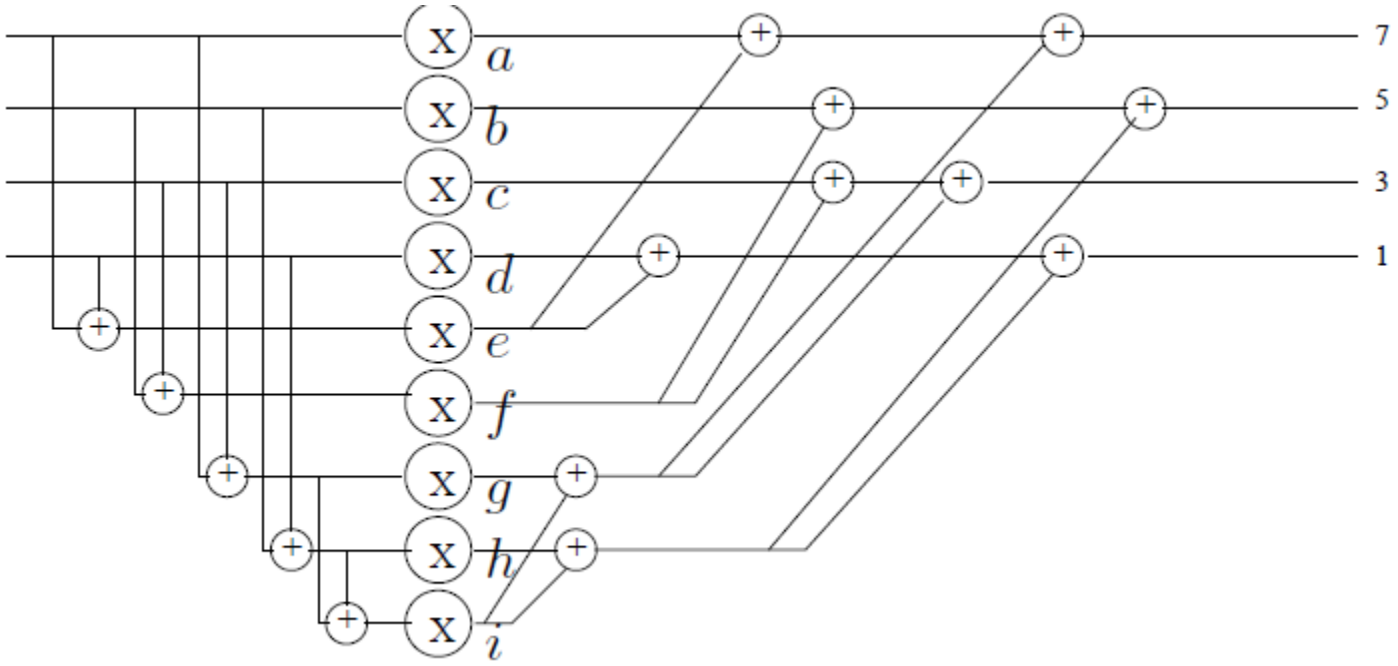
1-D 8-point DCT can be simplified



cn => $\begin{cases} x_{out} = x_{in} \cdot \cos n\pi/16 + y_{in} \cdot \sin n\pi/16 \\ y_{out} = -x_{in} \cdot \sin n\pi/16 + y_{in} \cdot \cos n\pi/16 \end{cases}$

◯ => multiplication with $\sqrt{2}$

Final modification



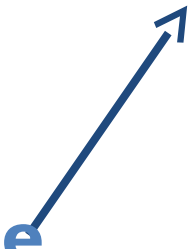
$k_3(k_1x + k_2y) = k_3k_1\ x + k_3k_2\ y$

precompute

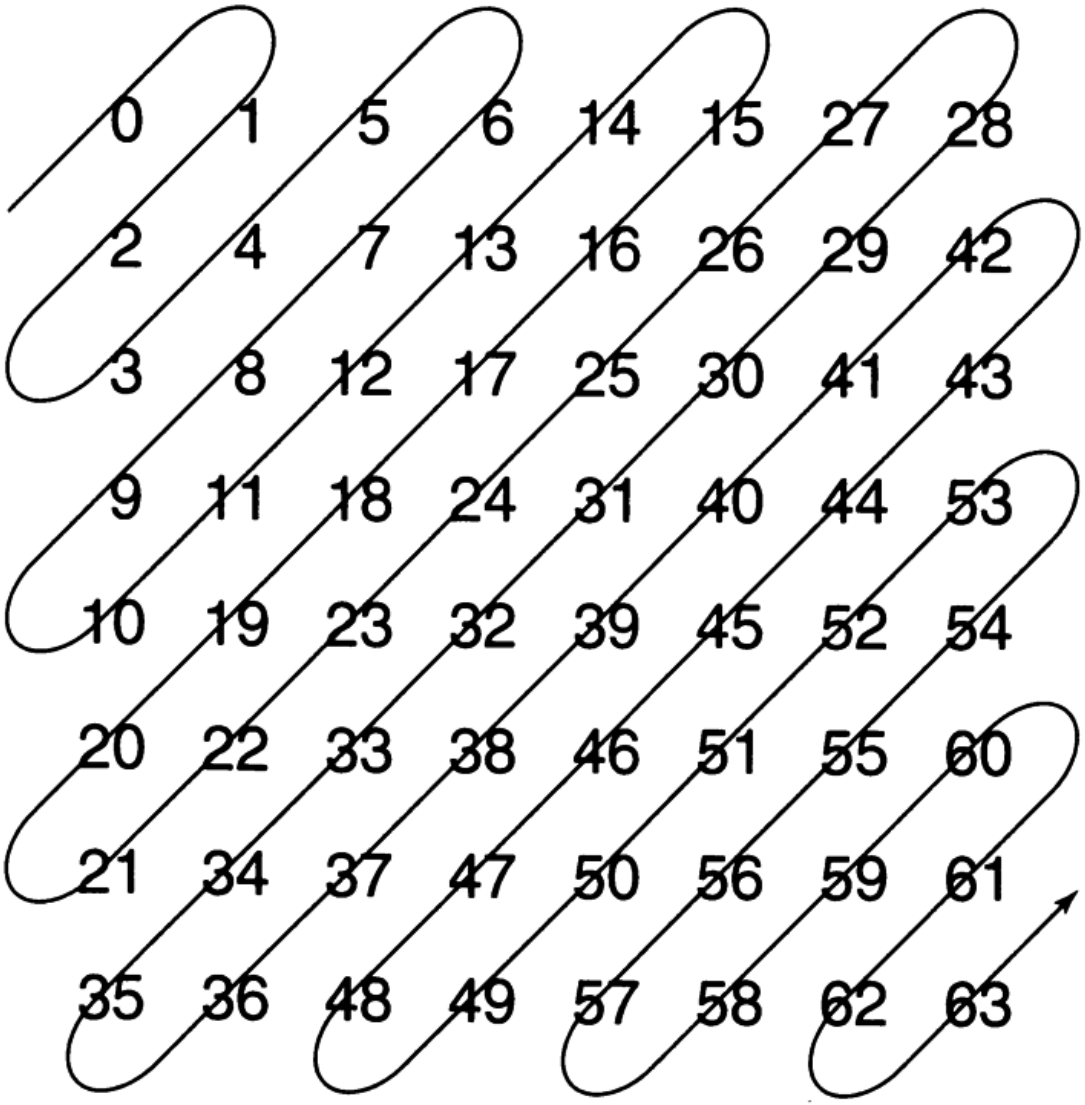
RLE = run length encoding

Run Length encoding

Raw data: 0 0 0 1 1 1 1 0 0 0 0 1 0
Encoded as: 3:0, 4:1, 4:0, 1:1, 1:0
Alternative (if only zeroes are plentiful):
Raw data: 5 0 0 0 7 0 0 9 0 0 0 0 0 0 0
Encoded as: 5:3, 7:2, 9:DONE



Zigzag Pattern



Magnitude Encoding

Encoded value	DC Value Range	
0	0	
1	$[-1]$	$[1]$
2	$[-3, -2]$	$[2, 3]$
3	$[-7, -4]$	$[4, 7]$
4	$[-15, -8]$	$[8, 15]$
5	$[-31, -16]$	$[16, 31]$
6	$[-63, -32]$	$[32, 63]$
7	$[-127, -64]$	$[64, 127]$
8	$[-255, -128]$	$[128, 255]$
9	$[-511, -256]$	$[256, 511]$
10	$[-1023, -512]$	$[512, 1023]$
11	$[-2047, -1024]$	$[1024, 2047]$

An example of RLE

1) After Q

$$\begin{bmatrix} 22 & 12 & 0 & -12 & 0 & 0 & 0 & 0 \\ 0 & 0 & -8 & 0 & 0 & 0 & 0 & 0 \\ 4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

2) After zig-zag

22
12
0 4
0 0 -12
-8
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 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1

3) After RLE

value	raw bits
05	10110
04	1100
13	100
24	0011
04	0111
F0	
F0	
D1	1
00	

Run of 0:s

magnitude

4) Huffman coding

*The values are HC (variable length) using table lookup.
* Raw bits are left untouched

Huffman Encoding/Decoding

- Analogy
  - Morse Code

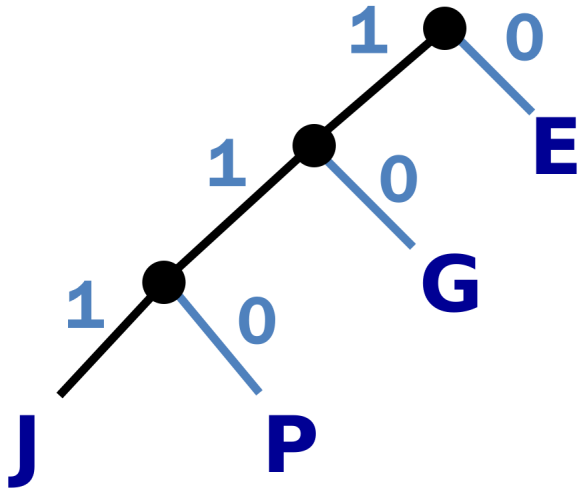
J P E G

.--- .--. . --.

- Binary Codes
  - Mutally Exlusive Codes
  - Binary Tree

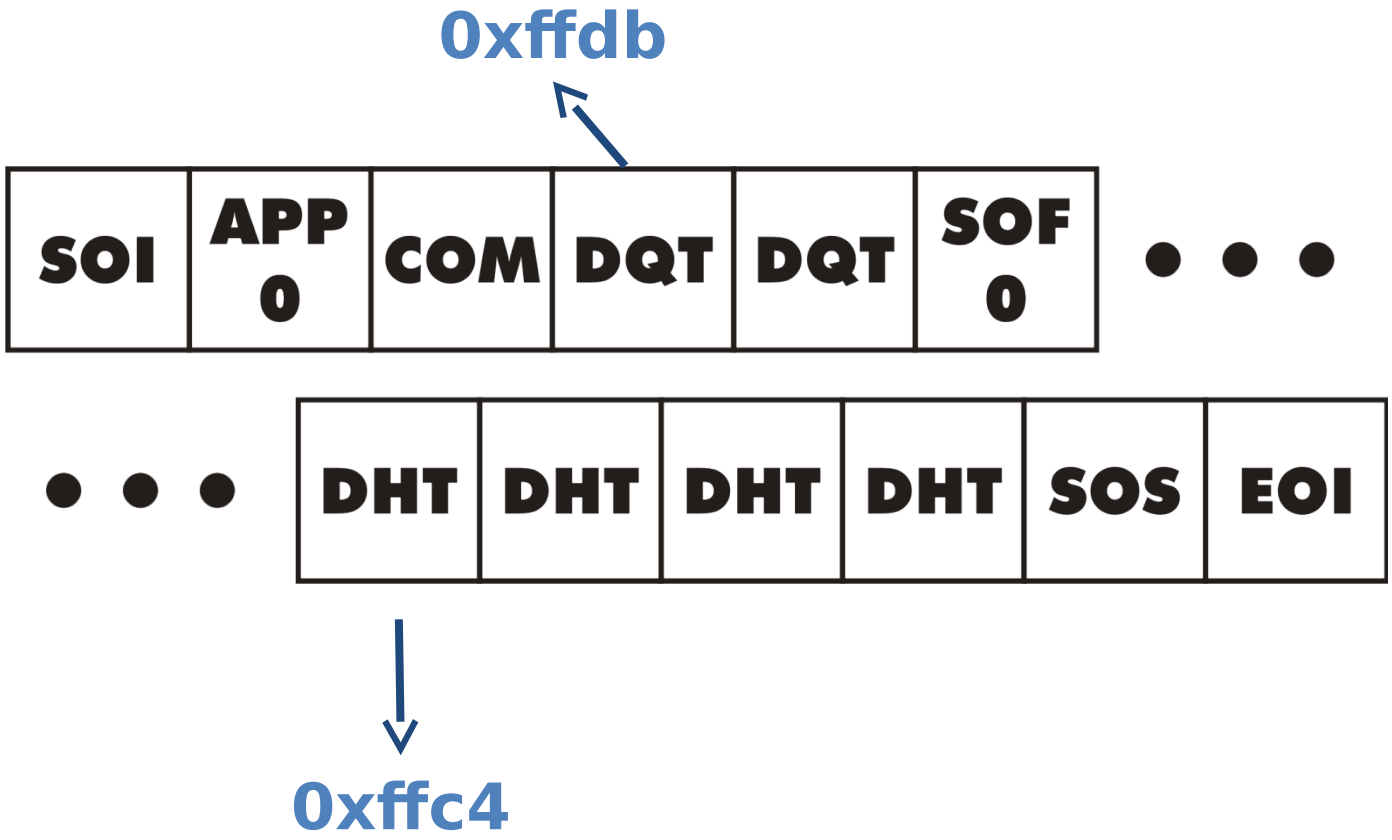
111|110|0|10

J | P | E | G



JFIF Format

- JPEG File Interchange Format
  - Markers
  - Data



Finally

- AC and DC values are treated differently
- Two Huffman LUTs are used
- DC
  - differential, magnitude encoding, Huffman table lookup
- AC
  - as mentioned, raw bits left untouched Huffman table lookup

in	code	length
0x00	1010	4
0x01	00	2
0x02	01	2
0x03	100	3
0x04	1011	4
…	…	

max length=16

04 1100 => …10111100…

Lab 2 – A JPEG acc

1. Design HW
2. Change existing software

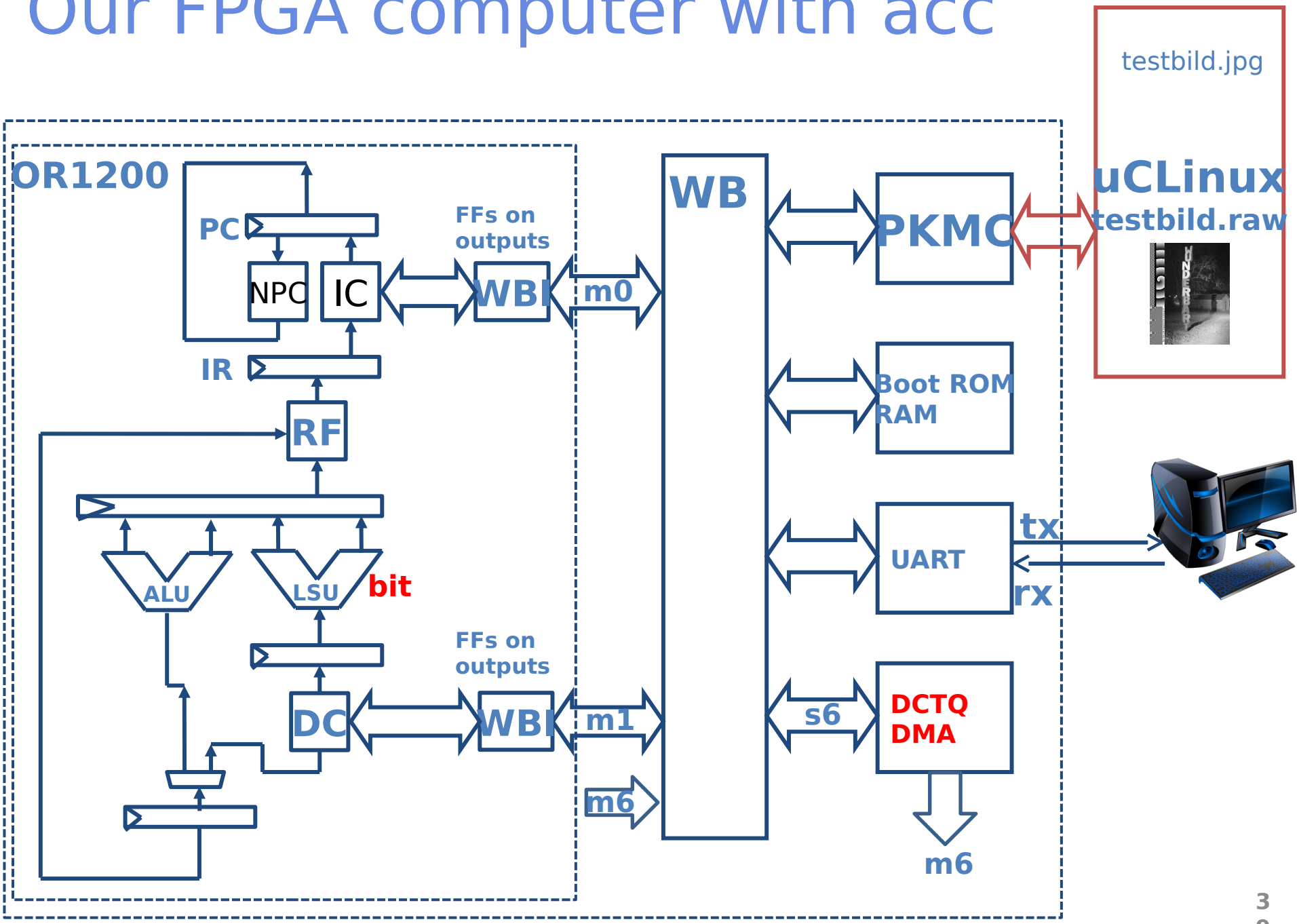
   `jpegfiles` under µCLinux

   a) insert your acc

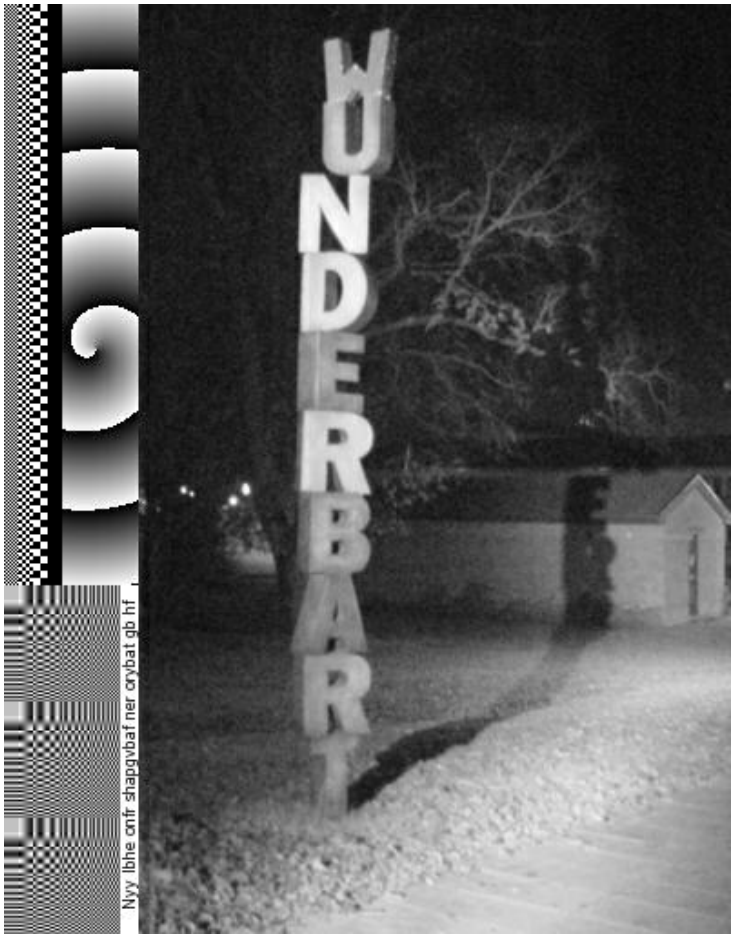
   b) insert your DMA

   c) insert your instruction

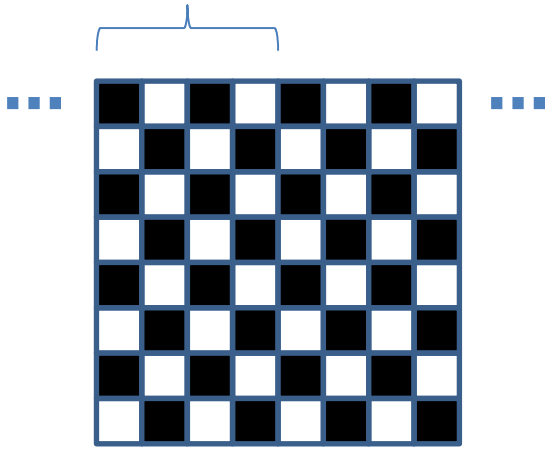
Our FPGA computer with acc



Raw image format in memory



0x00ff00ff

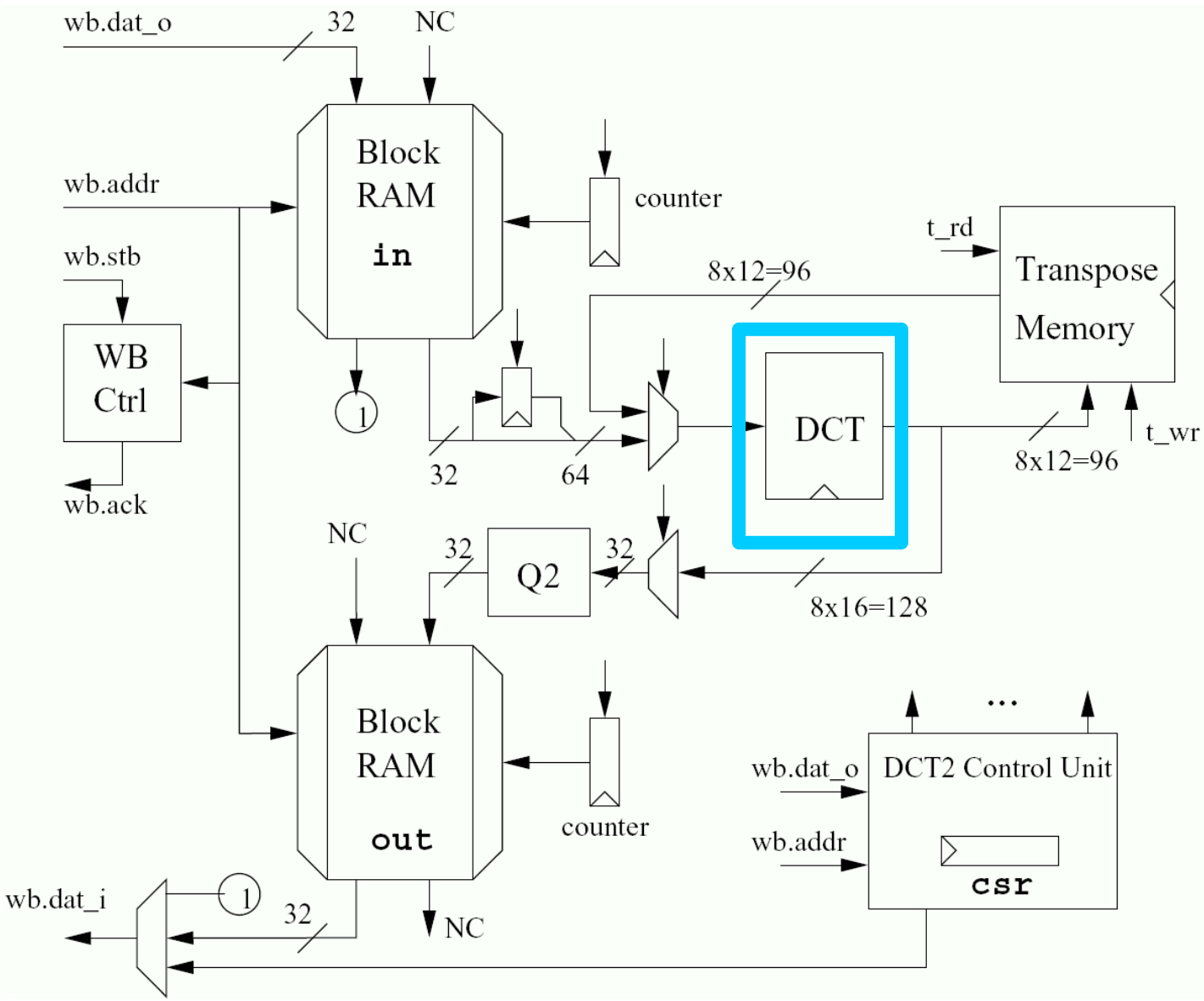


8 bit pixels [0,255]

4 pixels/word

Somewhere 128 must be subtracted from each pixel!

Proposed Architecture



A couple of test cases

Case 1

MATLAB test program

1) Before transform: a=

1	2	3	4	5	6	7	8
9	10	11	12	13	14	15	16
17	18	19	20	21	22	23	24
25	26	27	28	29	30	31	32
33	34	35	36	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	64

2) After sqrt(8)*DCT_1[a-128] (and truncating) on the rows

-988	-19	0	-2	0	-1	0	-1
-924	-19	0	-2	0	-1	0	-1
-860	-19	0	-2	0	-1	0	-1
-796	-19	0	-2	0	-1	0	-1
-732	-19	0	-2	0	-1	0	-1
-668	-19	0	-2	0	-1	0	-1
-604	-19	0	-2	0	-1	0	-1
-540	-19	0	-2	0	-1	0	-1

3) After another sqrt(8)*DCT_1 (and truncating), now on the columns,
we have 8*DCT_2[a-128] =

-6112	-152	0	-16	0	-8	0	-8
-1167	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-122	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-37	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-10	0	0	0	0	0	0	0

4) The quantization matrix is given by Q =

16	11	10	16	24	40	51	61
12	12	14	19	26	58	60	55
14	13	16	24	40	57	69	56
14	17	22	29	51	87	80	62
18	22	37	56	68	109	103	77
24	35	55	64	81	104	113	92
49	64	78	87	103	121	120	101
72	92	95	98	112	100	103	99

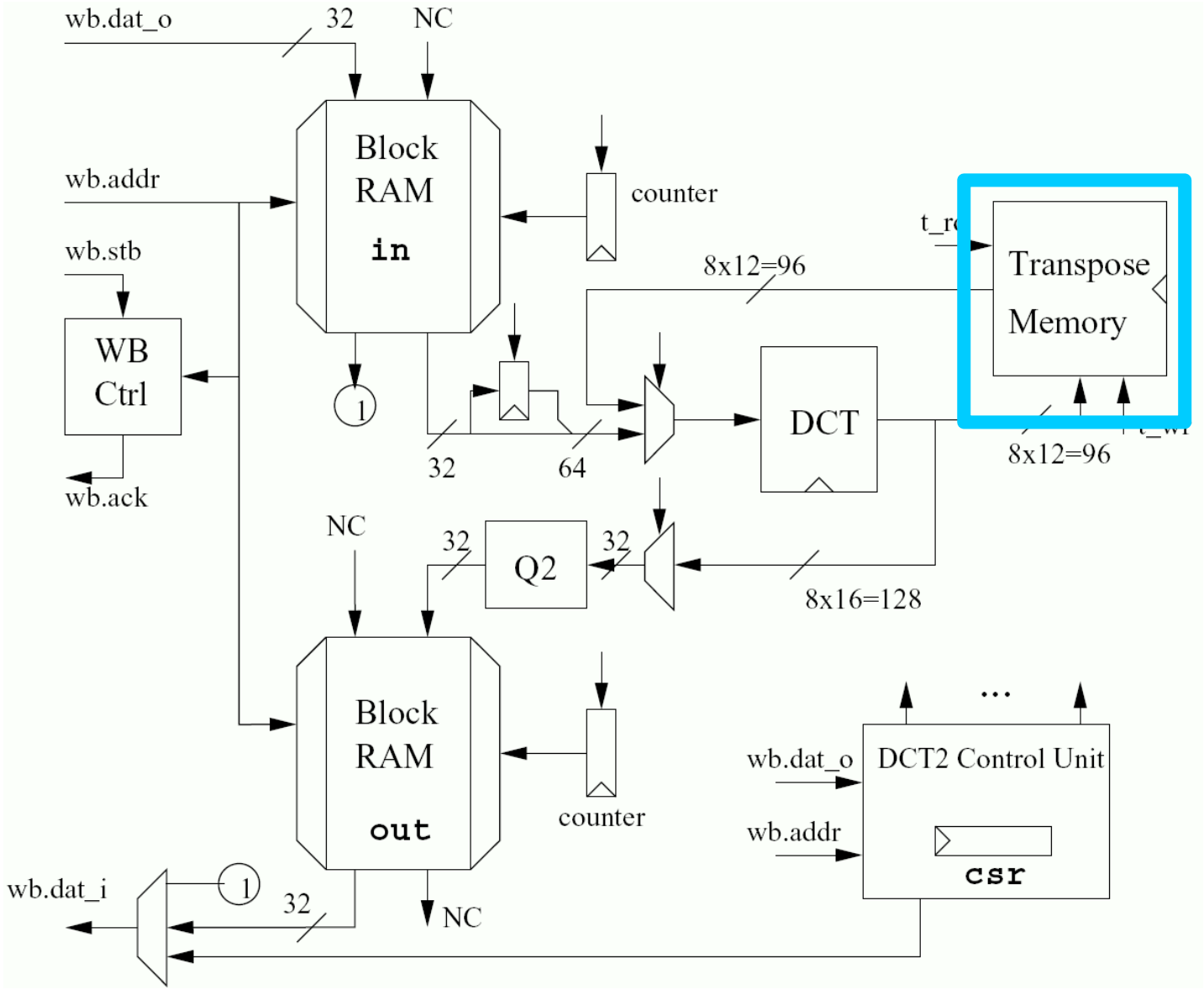
5) After quantization the result should be
Y = round(8*B/(8*Q*1/2))
(1/2 is a quality factor used by jpegfiles)

-96	-3	0	0	0	0	0	0
-24	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
-2	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	0	0	0	0	0	0

DCT Module

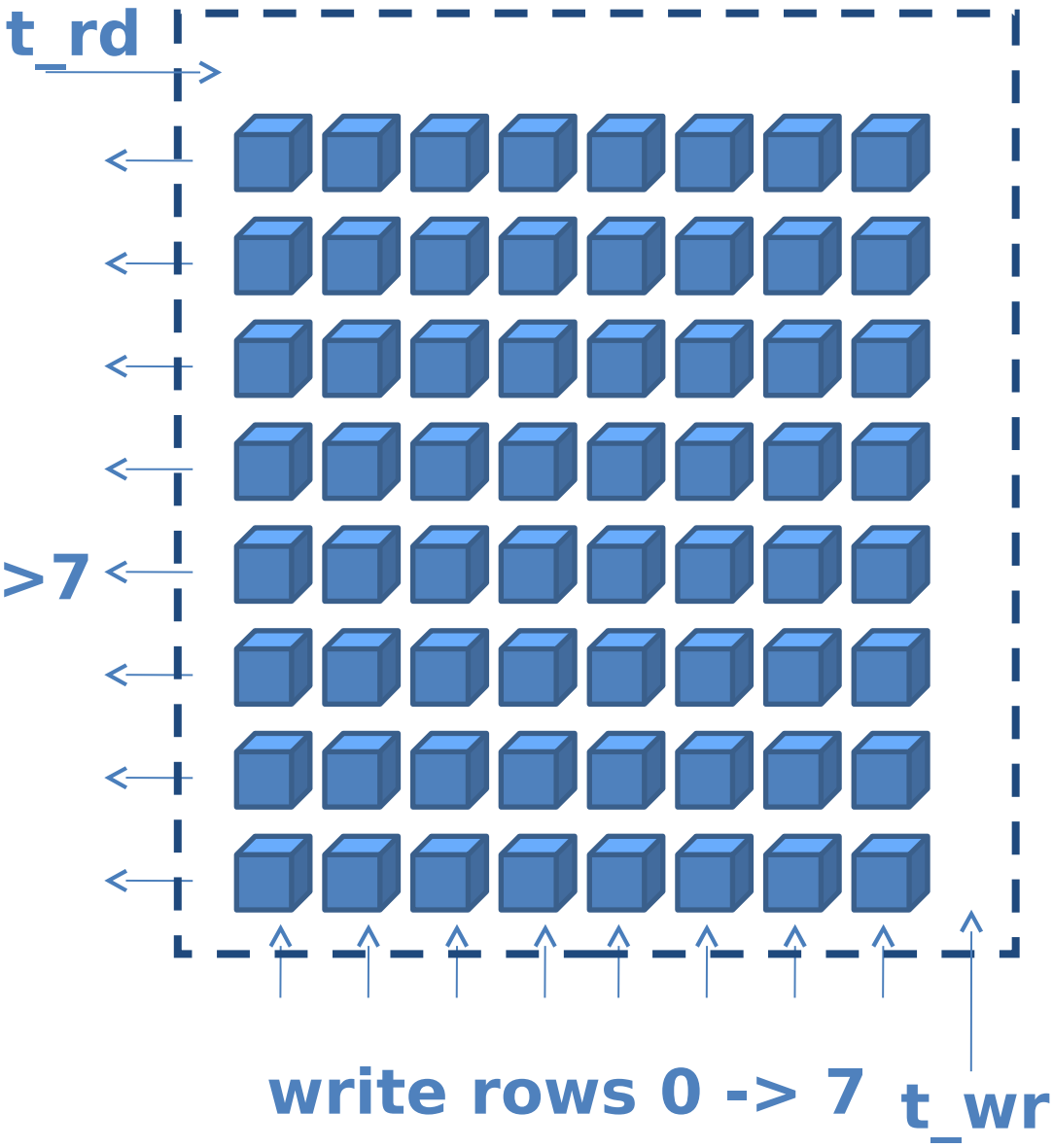
- Given to you
  - 1D DCT
    - 8 in ports (12 bits), 8 out ports (16 bits)
    - Fix point arithmetic
    - Straightforward implementation of Loeffler's algorithm

Proposed Architecture

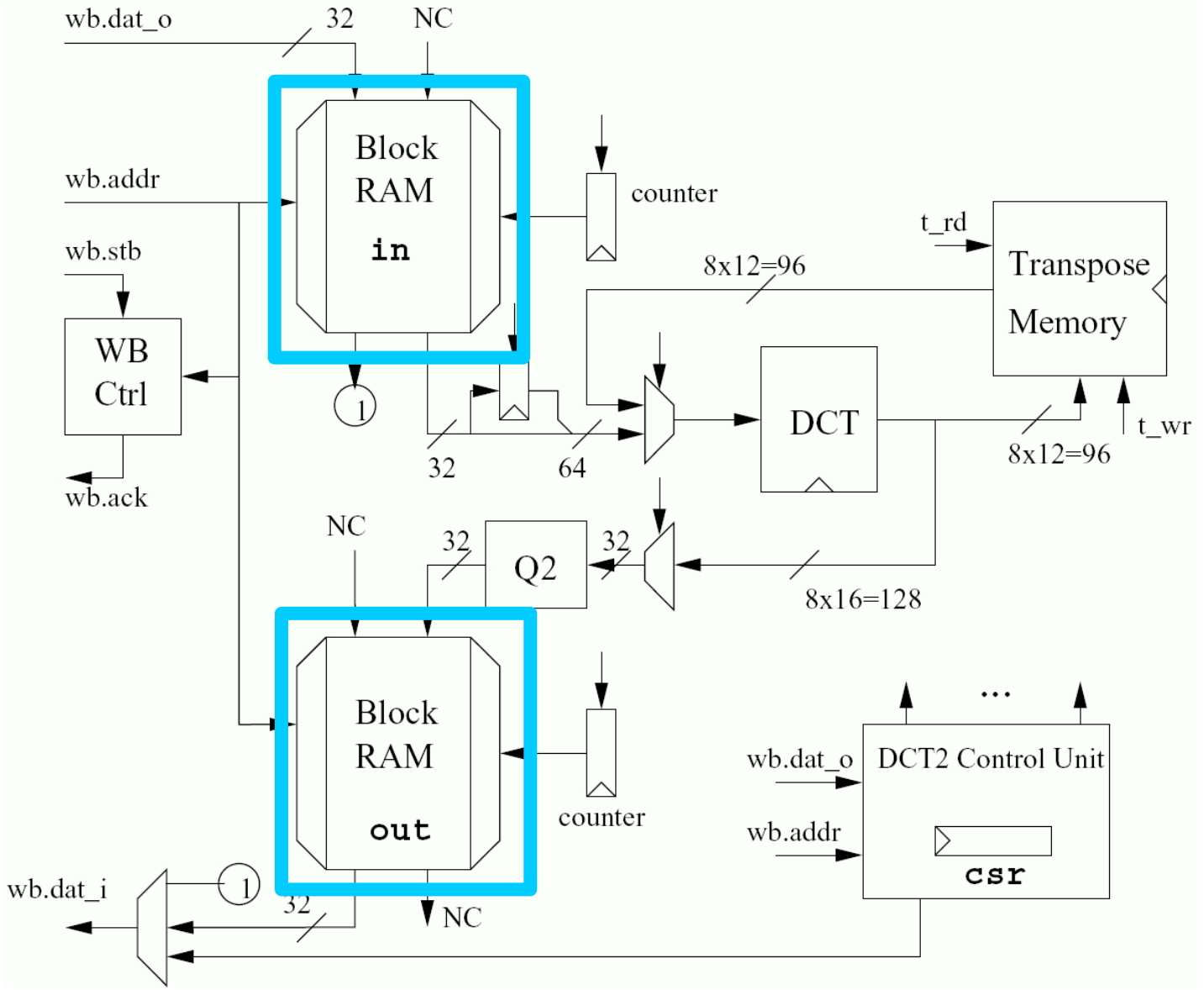


Transpose Memory

- Rearrange rows to columns
  - Use distributed RAM
    -synch write
    -asynch read



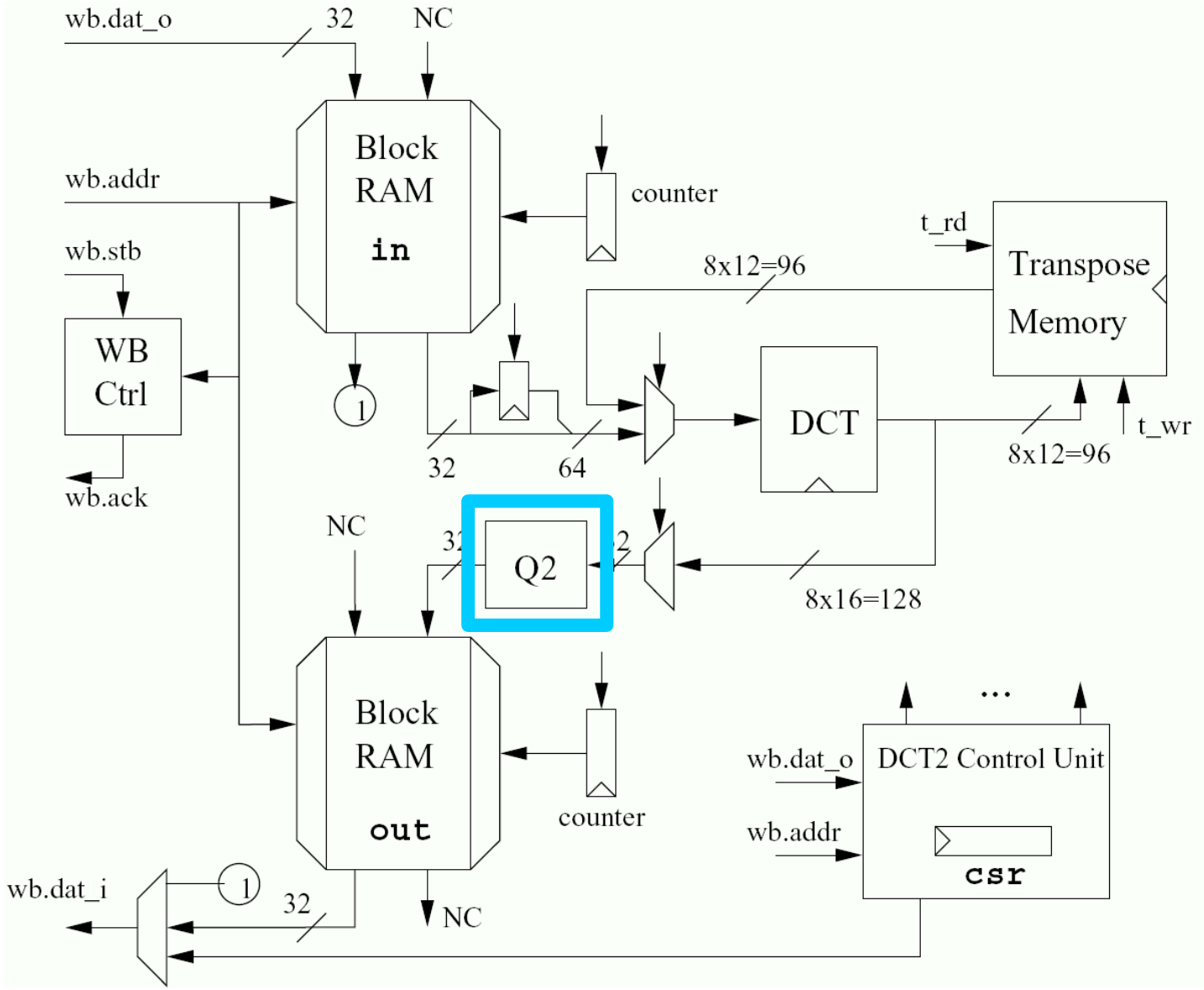
Proposed Architecture



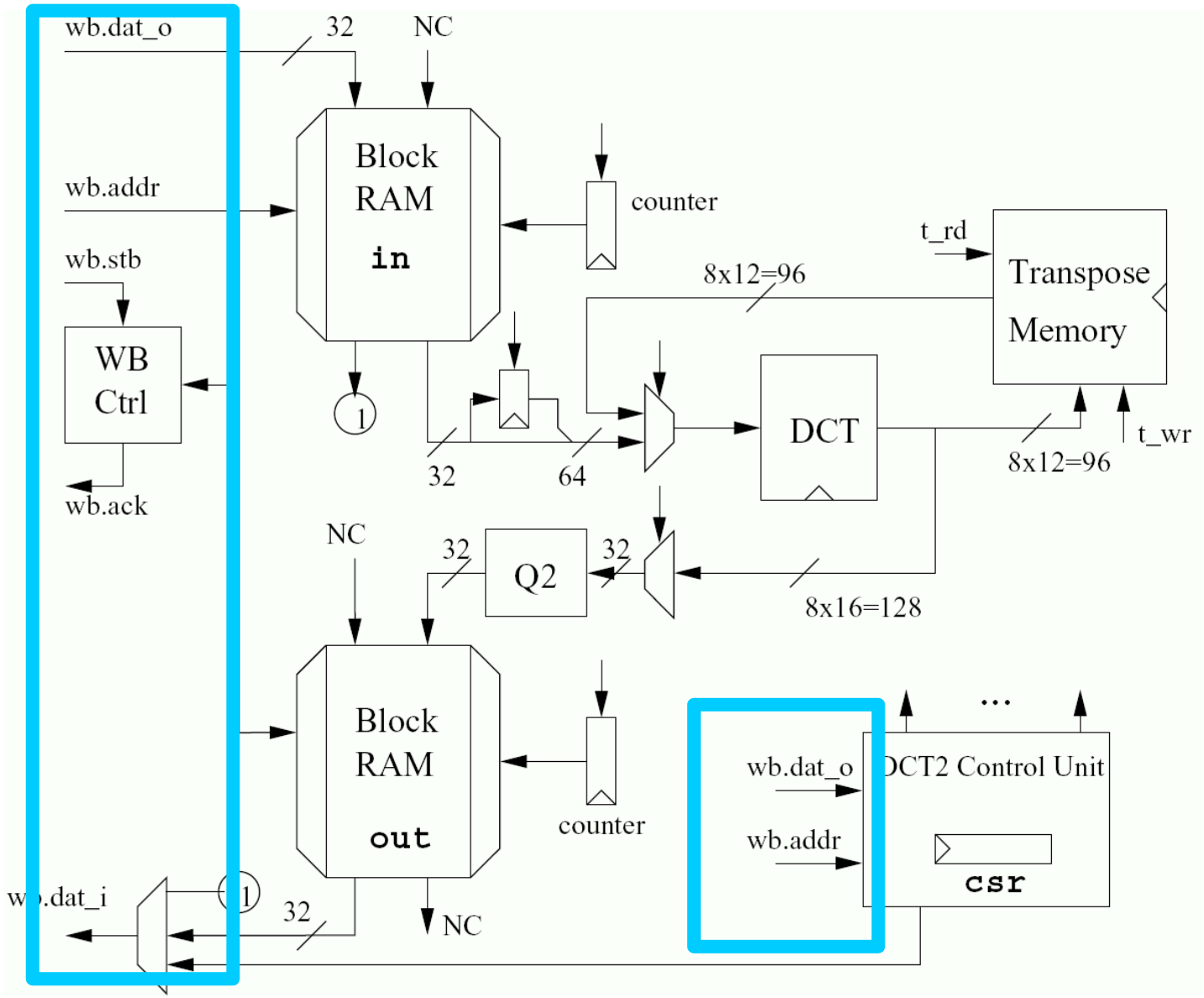
Block RAM

- Different timing
- “Normal” SRAM
  - Asynchronous read
  - Asynchronous write
- Block RAM in Virtex 2
  - Synchronous read
  - Synchronous write

Proposed Architecture

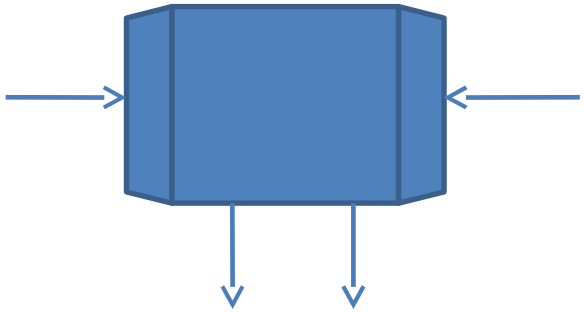


Proposed Architecture

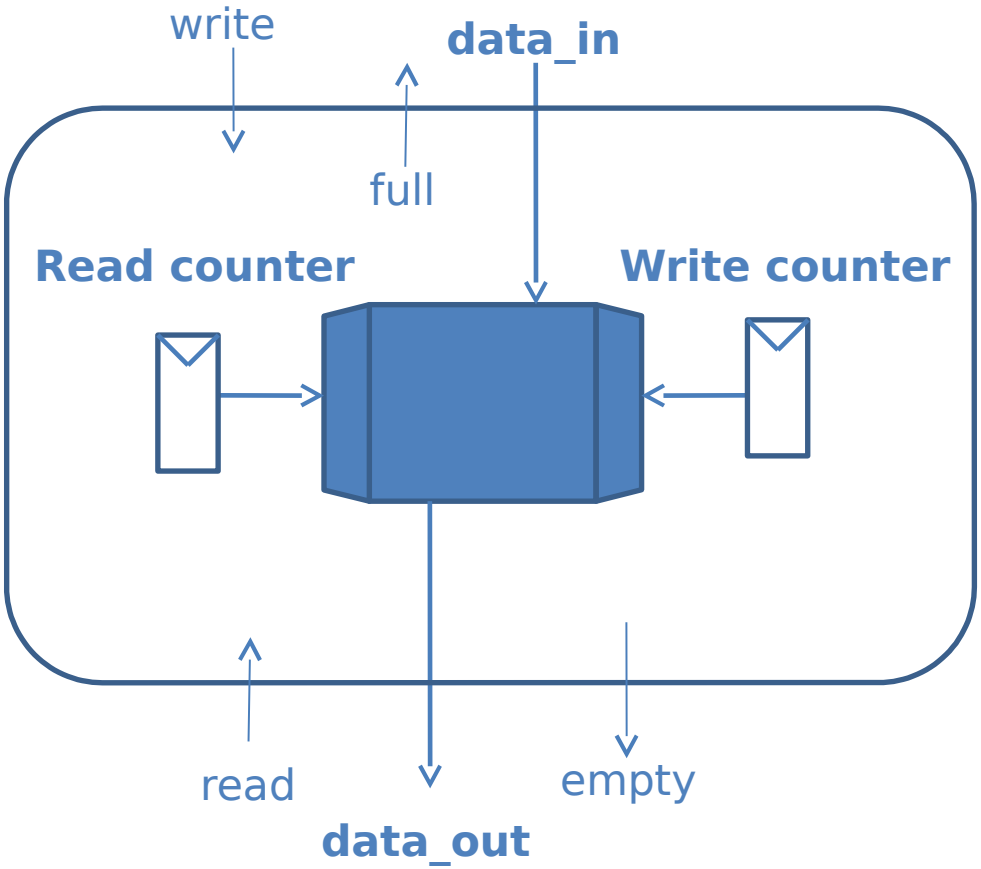


Some ideas

**You can read 8 pixels per clock,
If you use both ports**

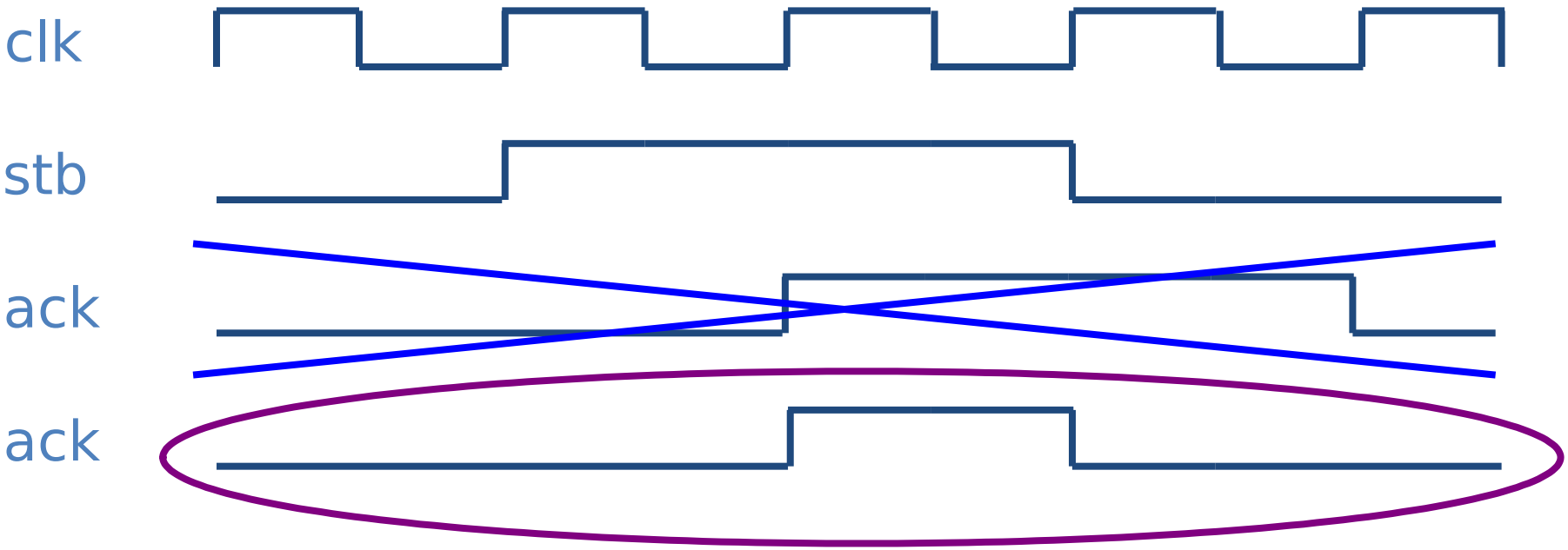


You can rebuild the BRAM to a FIFO



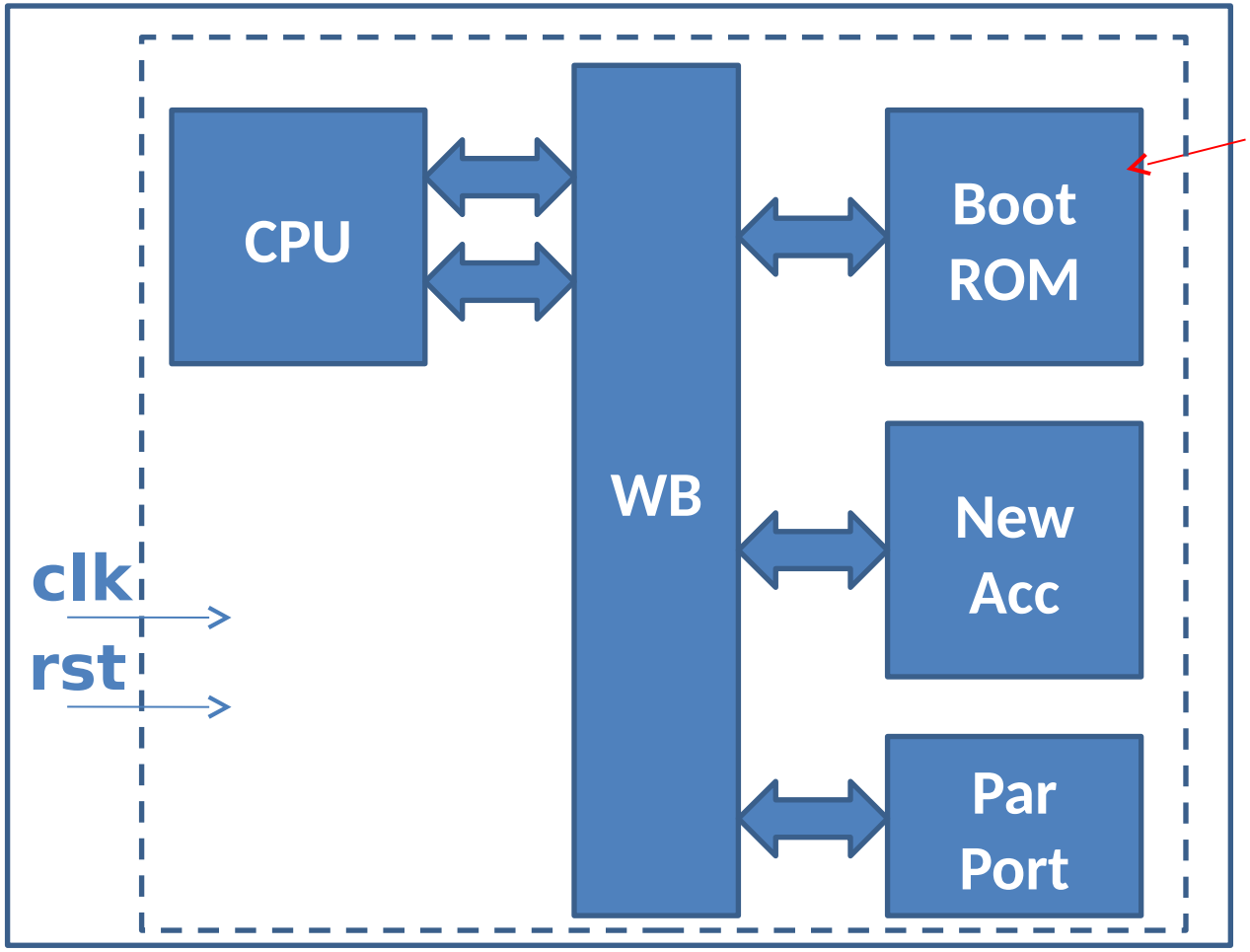
Some Notes on the WB I/F

- Be careful with wb.ack



Test benches – 2 alternatives

1) Simulate the whole computer - `make sim`



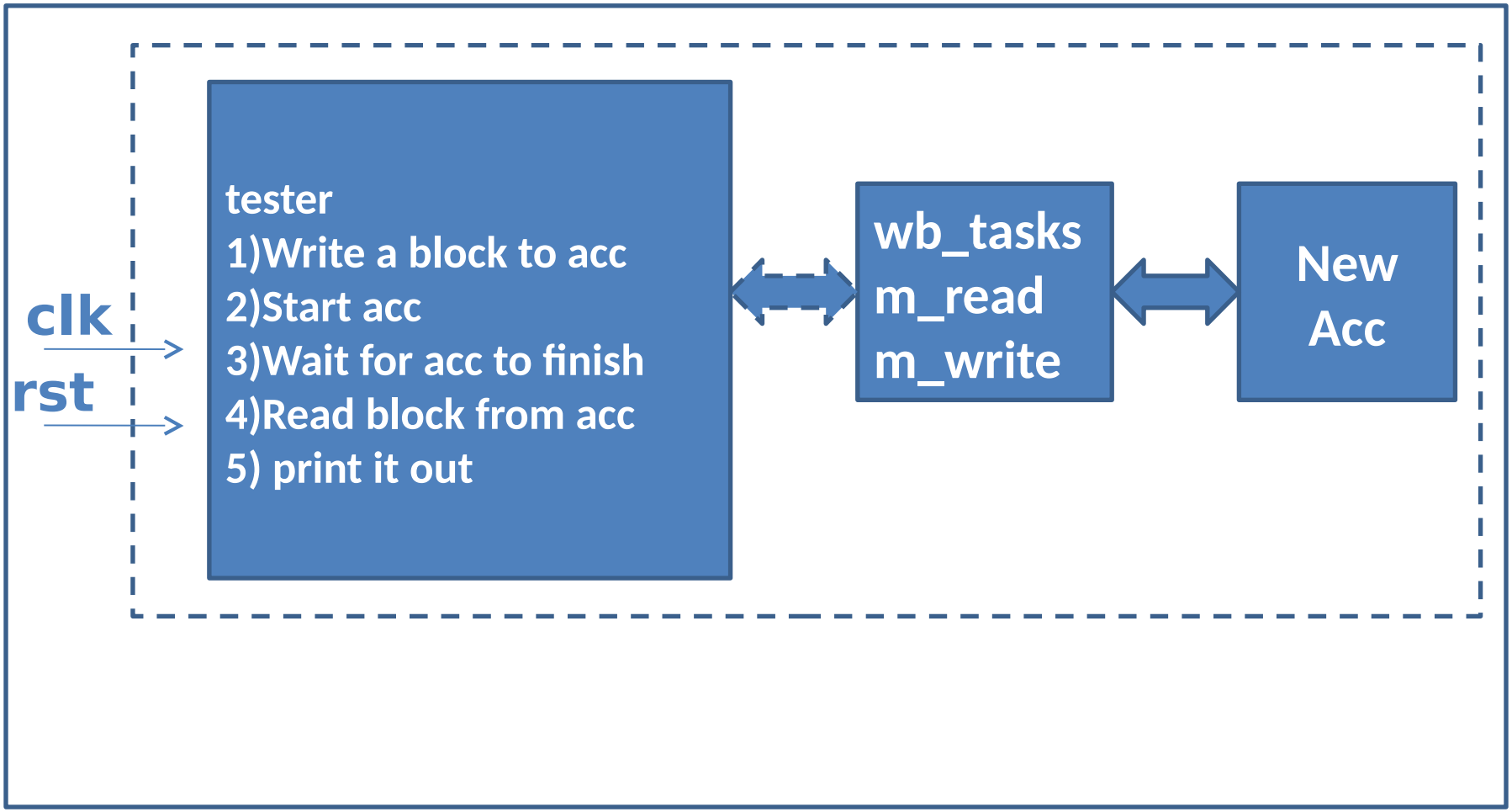
Insert some code In the beginning of the monitor **`mon2.c`**

There are some alternatives to uncomment

Tip: You can write to parport to make it easier to find things in ModelSim

Test benches – 2 alternatives

2) Simulate the acc – `make sim_jpeg`



wb_tasks.sv

```
module wishbone_tasks(wishbone.master wb);
   int result = 0;
   reg oldack;
   reg [31:0] olddat;

   always @(posedge wb.clk) begin
      oldack <= wb.ack;
      olddat <= wb.dat_i;
   end

   task m_read(input [31:0] adr, output logic [31:0] data);
      begin
      @(posedge wb.clk);
      wb.adr <= adr;
      wb.stb <= 1'b1;
      wb.we  <= 1'b0;
      wb.cyc <= 1'b1;
      wb.sel <= 4'hf;

      @(posedge wb.clk);
      #1;
      while (!oldack) begin
         @(posedge wb.clk);
              #1;
      end

      wb.stb <= 1'b0;
      wb.we  <= 1'b0;
      wb.cyc <= 1'b0;
      wb.sel <= 4'h0;

      data = olddat;
      end
   endtask // m_read
   ...
endmodule // wishbone_tasks
```

µClinux

- Operating system
- Flat memory
  - User program can crash OS
- /mnt and /var writable
  - /mnt/htdocs
- TFTP to transfer files
  - jpegtest
  - jcdctmgr
  - jdct
  - jchuff