# Hardware Acceleration

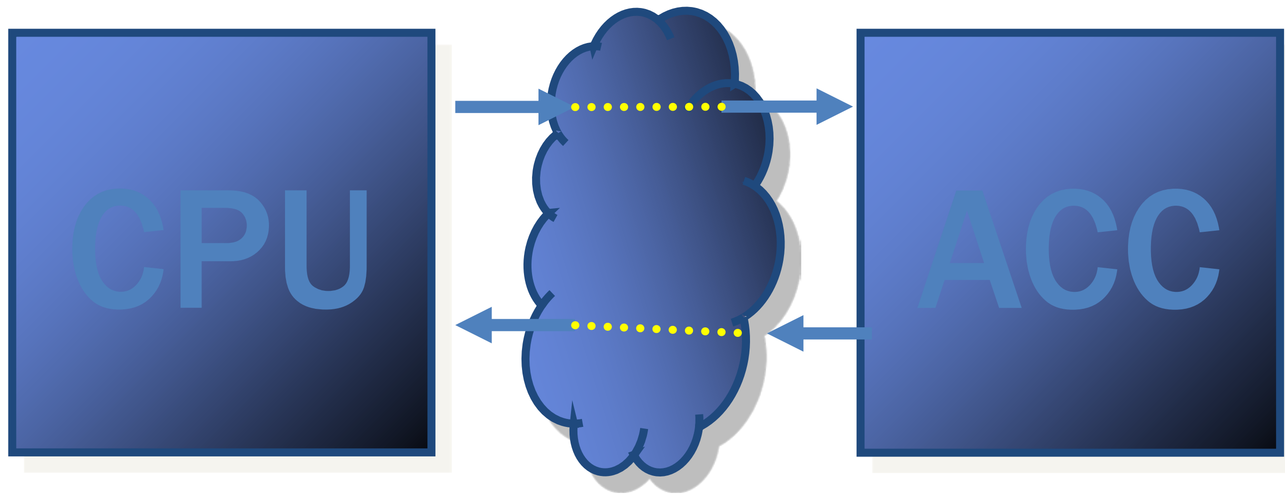
The Need for Speed

or Power

or Cost

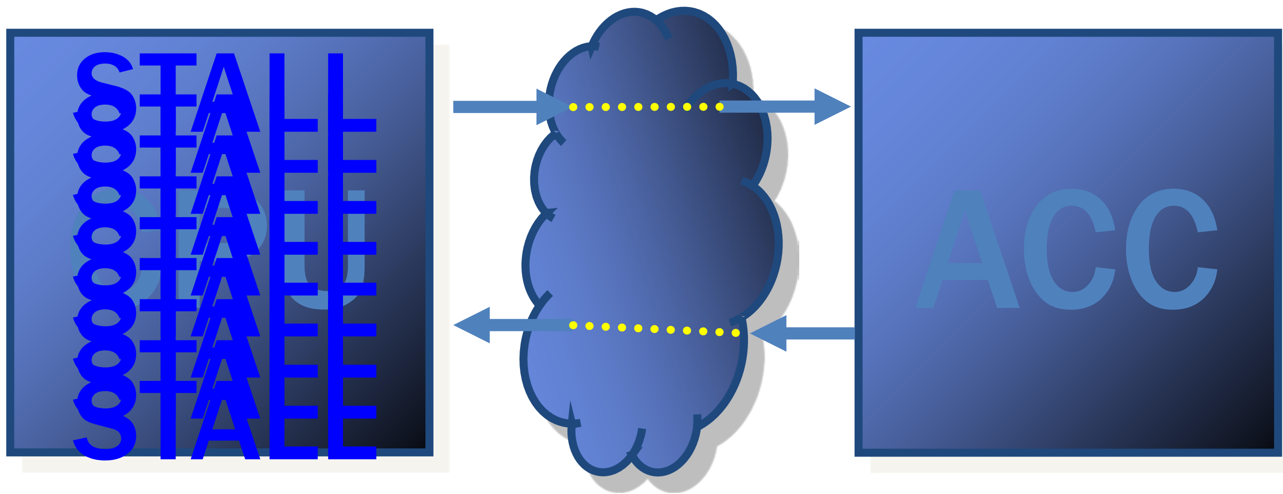
**Per Karlström**
**Olle Seger**

# Accelerators, single cycle



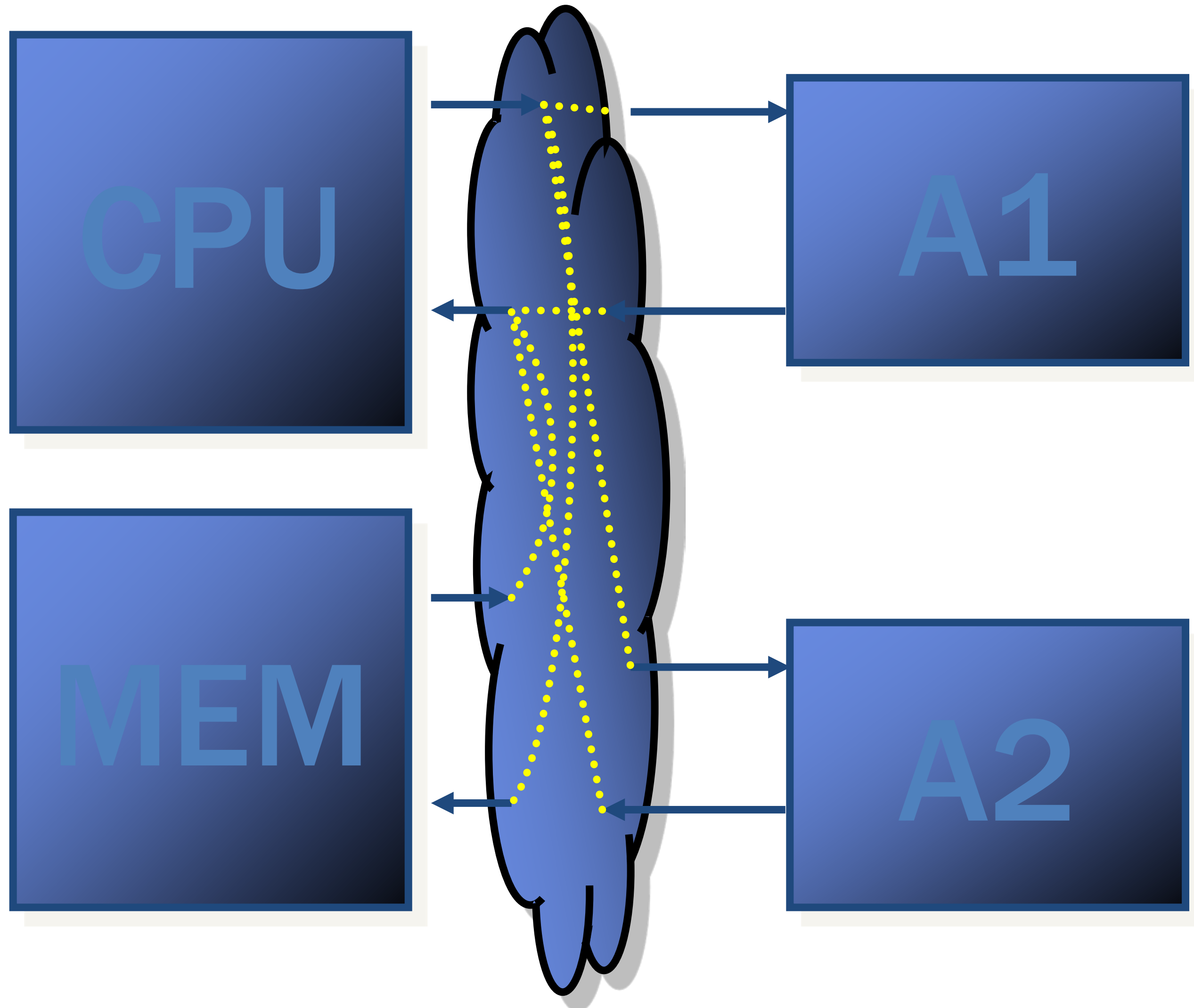

**1 cycle**

# Accelerators, multi cycle

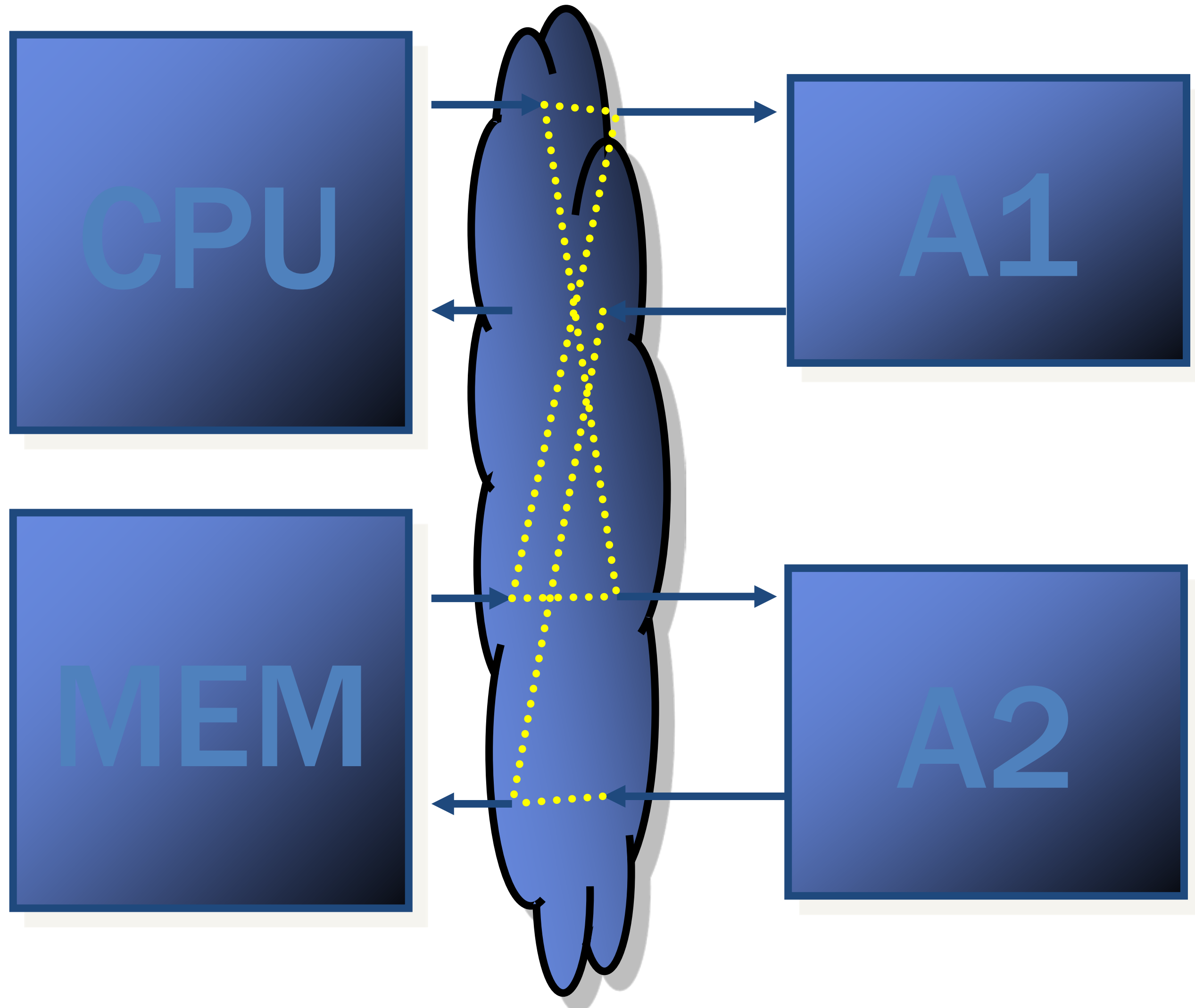



**n cycles**

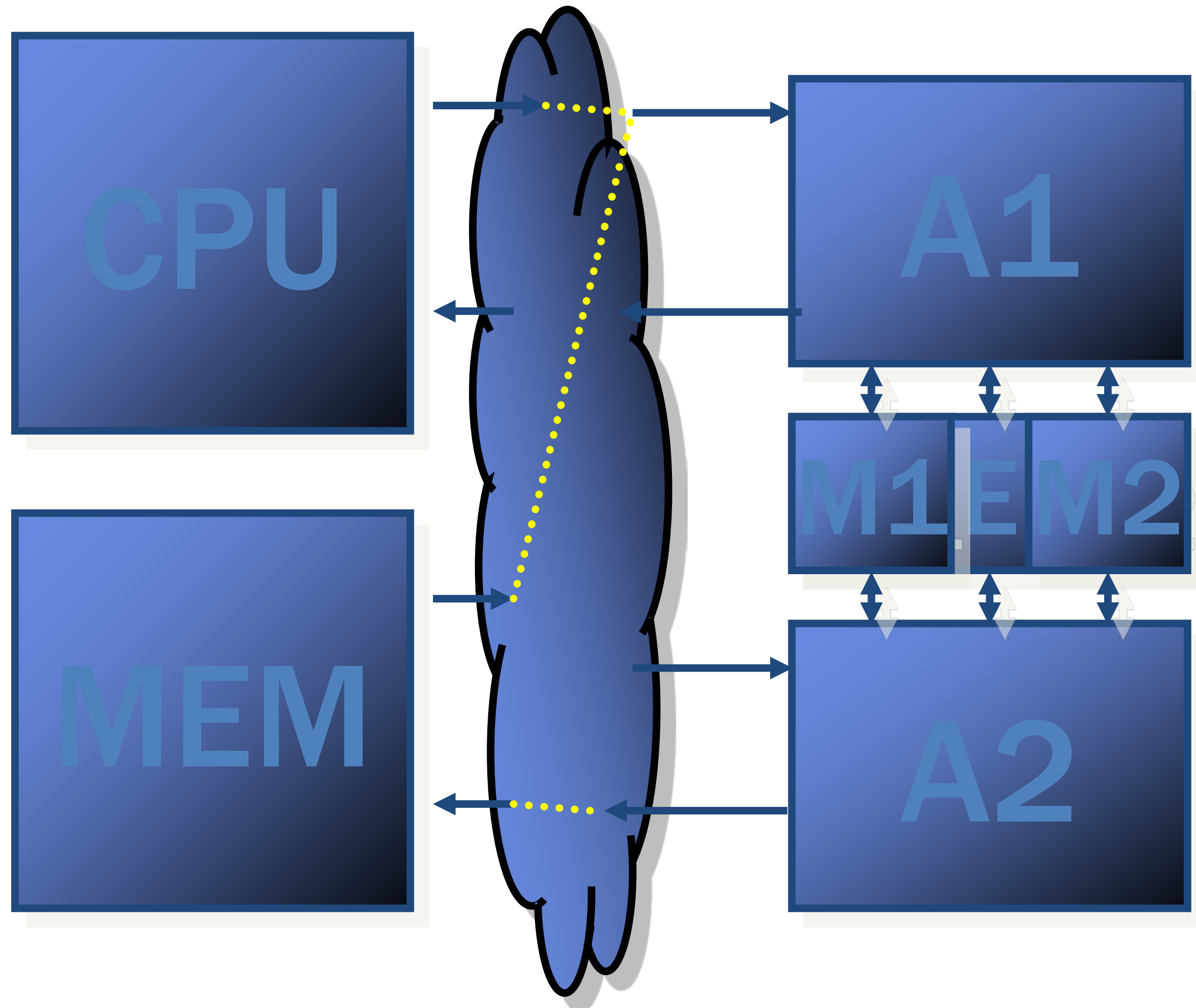
# Accelerators - Through CPU



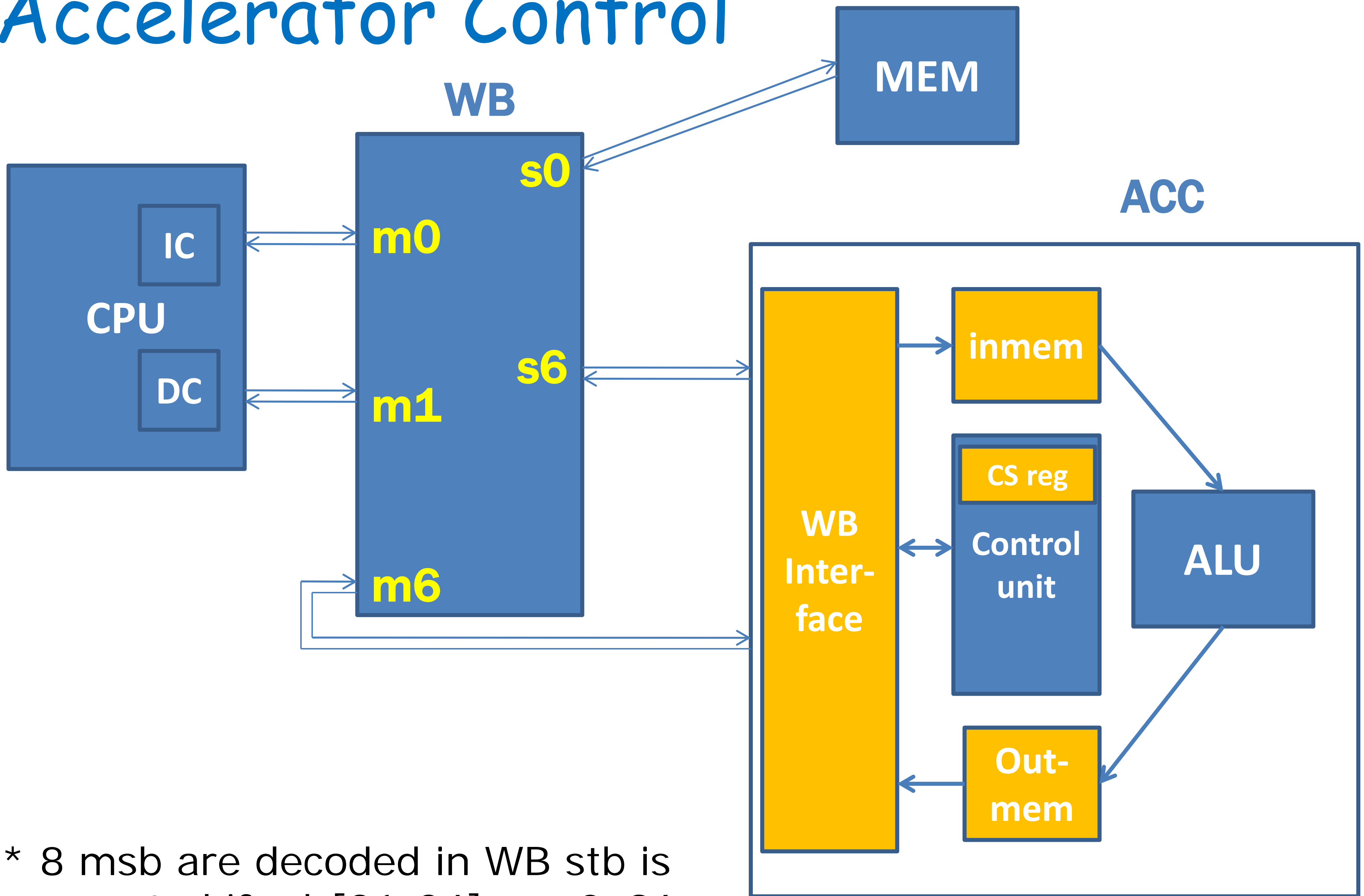
# Accelerators - DMA



# 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

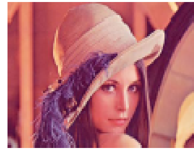
Joint Photographers Expert Group

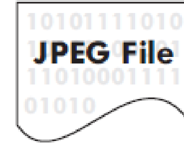
Remove things we cannot see

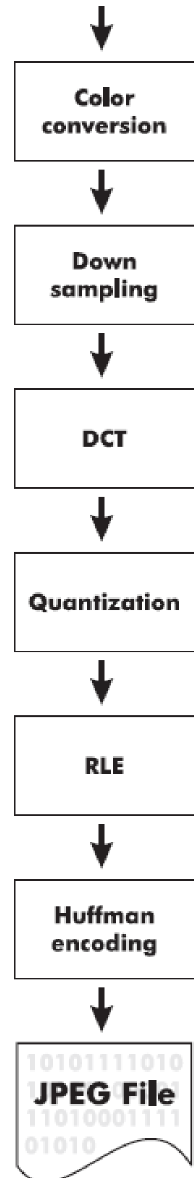
## encode

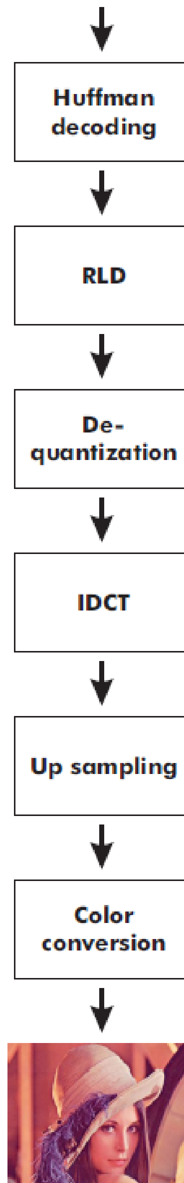
## decode

**Raw image**
**Camera**






**Raw image**
**Display**

# Problem

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

Build DMA    Build accelerator    Build new instruction

CPU - SW

ACC - HW

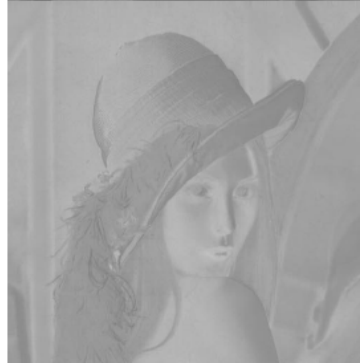
Competition for bus, caches important

# 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

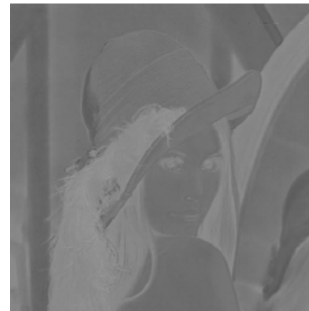
B

**Y = luminance,**
**Cb/Cr = chrominance**

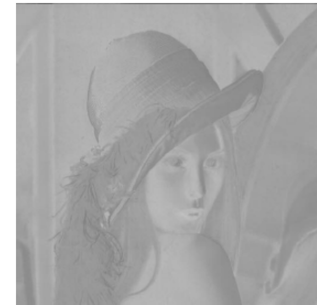
# Resampling

Y

Cb



Cr



**Data Reduction**
**50%**

# 8-point 1-D DCT/IDCT

$$\begin{cases} T(k) = c(k)\sum_{x=0}^{7} v(x)\cos\left(\frac{(2x+1)k\pi}{16}\right), & k = 0\cdots 7 \\ v(x) = \sum_{k=0}^{7} c(k)T(k)\cos\left(\frac{(2x+1)k\pi}{16}\right), & x = 0\cdots 7 \end{cases}$$

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

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

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

coord (x) freq (k)

# 8x8-point 2-D DCT/IDCT

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

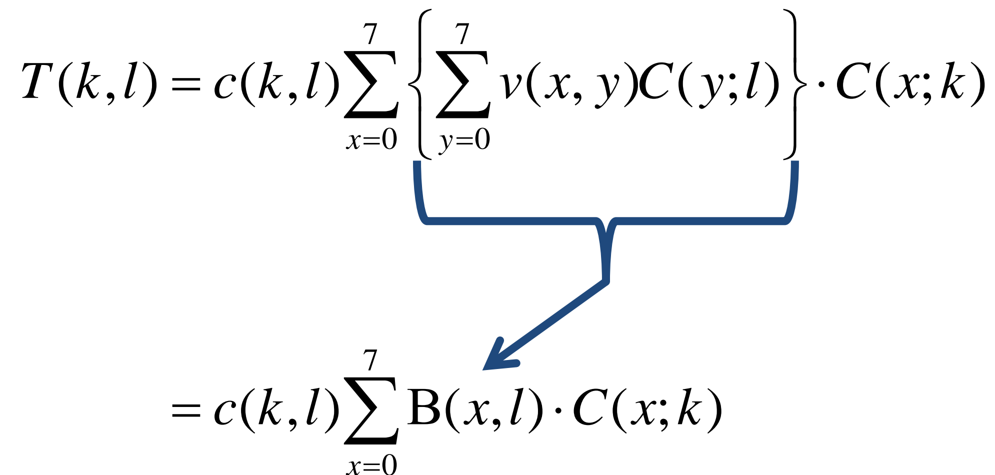
$$v(x,y) = \sum_{k=0}^{7}\sum_{l=0}^{7} c(k,l)T(k,l)\cdot C(y;l)C(x;k), \quad x,y = 0\cdots 7$$

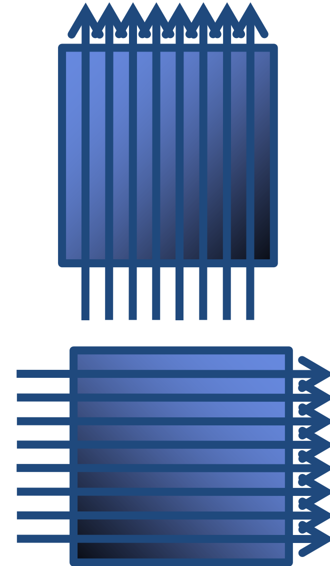
$$c(0,0) = \frac{1}{8} \quad k = l = 0$$

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

# Simplifications

## 1) Separation in x and y



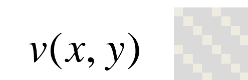
## 2) 1-D DCT can be simplified for N=8

# Meaning of the transform

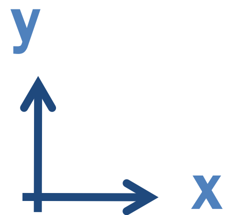
$$v(x,y) = \sum_{k=0}^{7}\sum_{l=0}^{7} T(k,l)\cdot c(k,l)C(y;l)C(x;k)$$

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

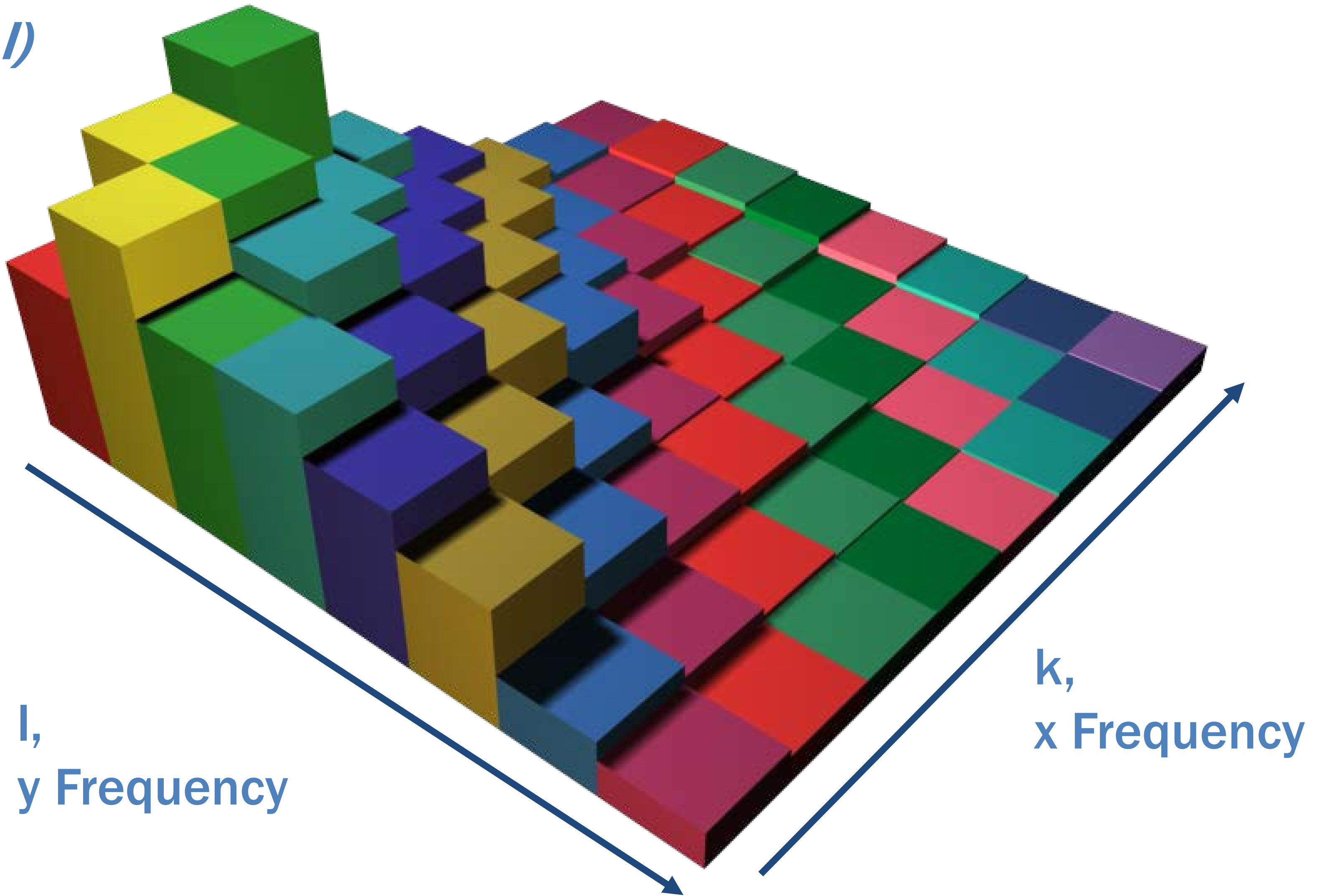
$C(x,y;0,0)$ $C(x,y;7,0)$




$C(x,y;0,7)$ $C(x,y;7,7)$

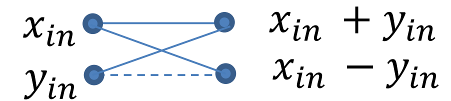


# Quantization

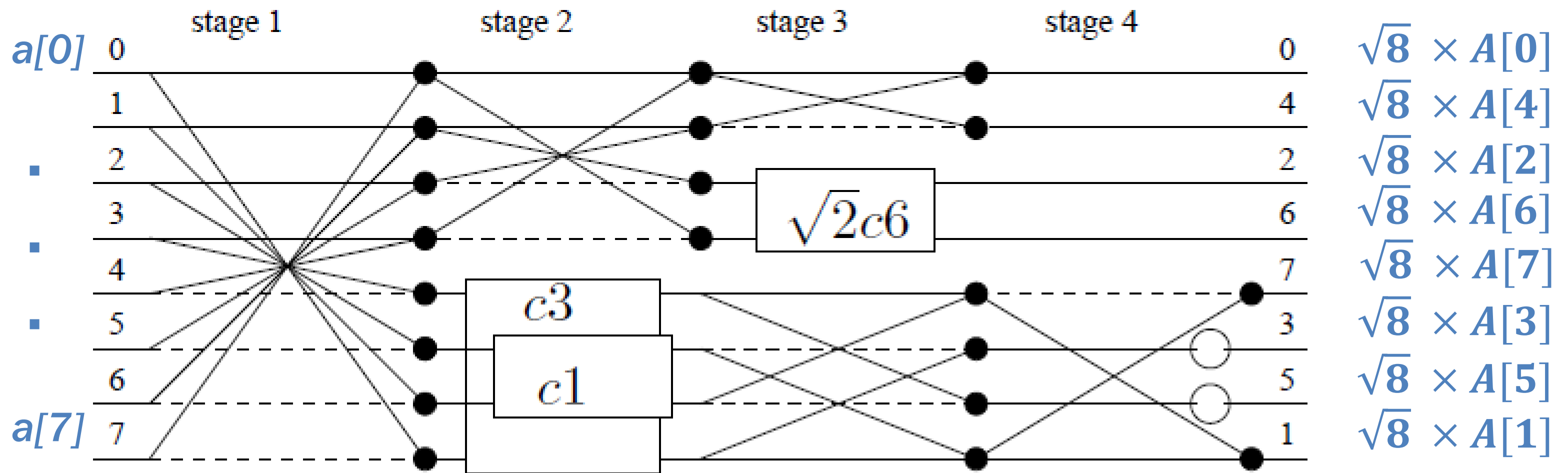


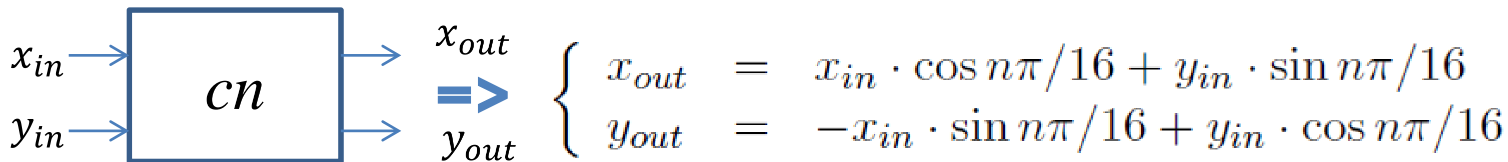
# Loefflers algorithm

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




## 1-D 8-point DCT can be simplified



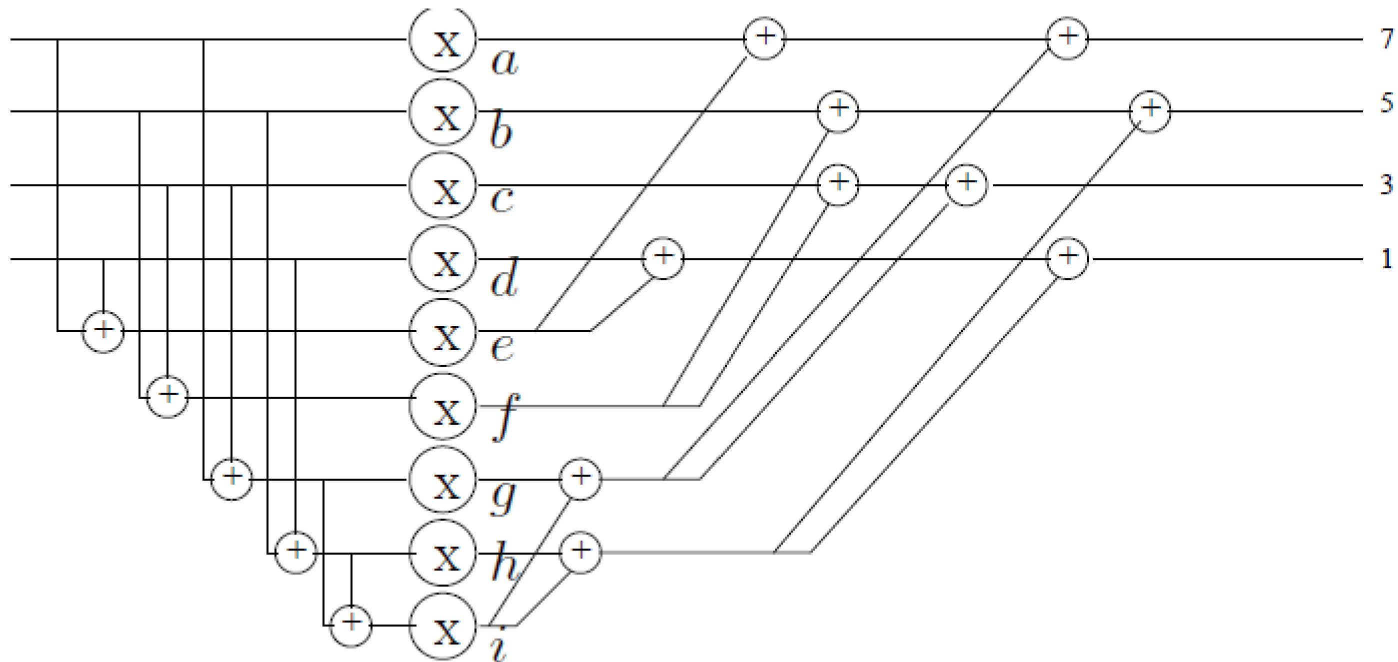

$$\begin{cases} x_{out} = x_{in} \cdot \cos n\pi/16 + y_{in} \cdot \sin n\pi/16 \\ y_{out} = -x_{in} \cdot \sin n\pi/16 + y_{in} \cdot \cos n\pi/16 \end{cases}$$

=> multiplication with $\sqrt{2}$

# Final modification

precompute

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



# Complexity 2D 8x8 DCT

| | MUL | ADD |
|---|---|---|
| **2D** | **4096** | **4032** |
| **Separated** | **1024** | **896** |
| **Loeffler orig.** | **224** | **416** |
| **Loeffler mod.** | **208** | **496** |

# Complexity 1D 8 DCT

| | MUL | ADD |
|---|---|---|
| **Definition** | **64** | **56** |
| **Loeffler orig.** | **14** | **26** |
| **Loeffler mod.** | **13** | **32** |

# Example computed with Loeffler DCT

## original

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 |

## subtract 128, DCT on rows

Ax=

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -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 |

## DCT on columns

A=

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -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 |

## after decoding

$\hat{a}$=

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 2 | 3 | 3 | 4 | 6 | 7 | 7 | 8 |
| 8 | 9 | 10 | 11 | 12 | 13 | 14 | 14 |
| 17 | 18 | 19 | 20 | 21 | 22 | 23 | 23 |
| 26 | 26 | 27 | 28 | 29 | 30 | 31 | 31 |
| 33 | 33 | 34 | 35 | 36 | 37 | 38 | 38 |
| 41 | 41 | 42 | 43 | 44 | 45 | 46 | 47 |
| 50 | 50 | 51 | 52 | 53 | 54 | 55 | 56 |
| 56 | 57 | 57 | 58 | 60 | 61 | 61 | 62 |

## after quantization

*Y=A./(8*Q*1/2)=*

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -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 |

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 |

# RLE = run length encoding

Run Length encoding

000 1111 0000 1 0

3:0 4:1 4:0 1:1 1:0

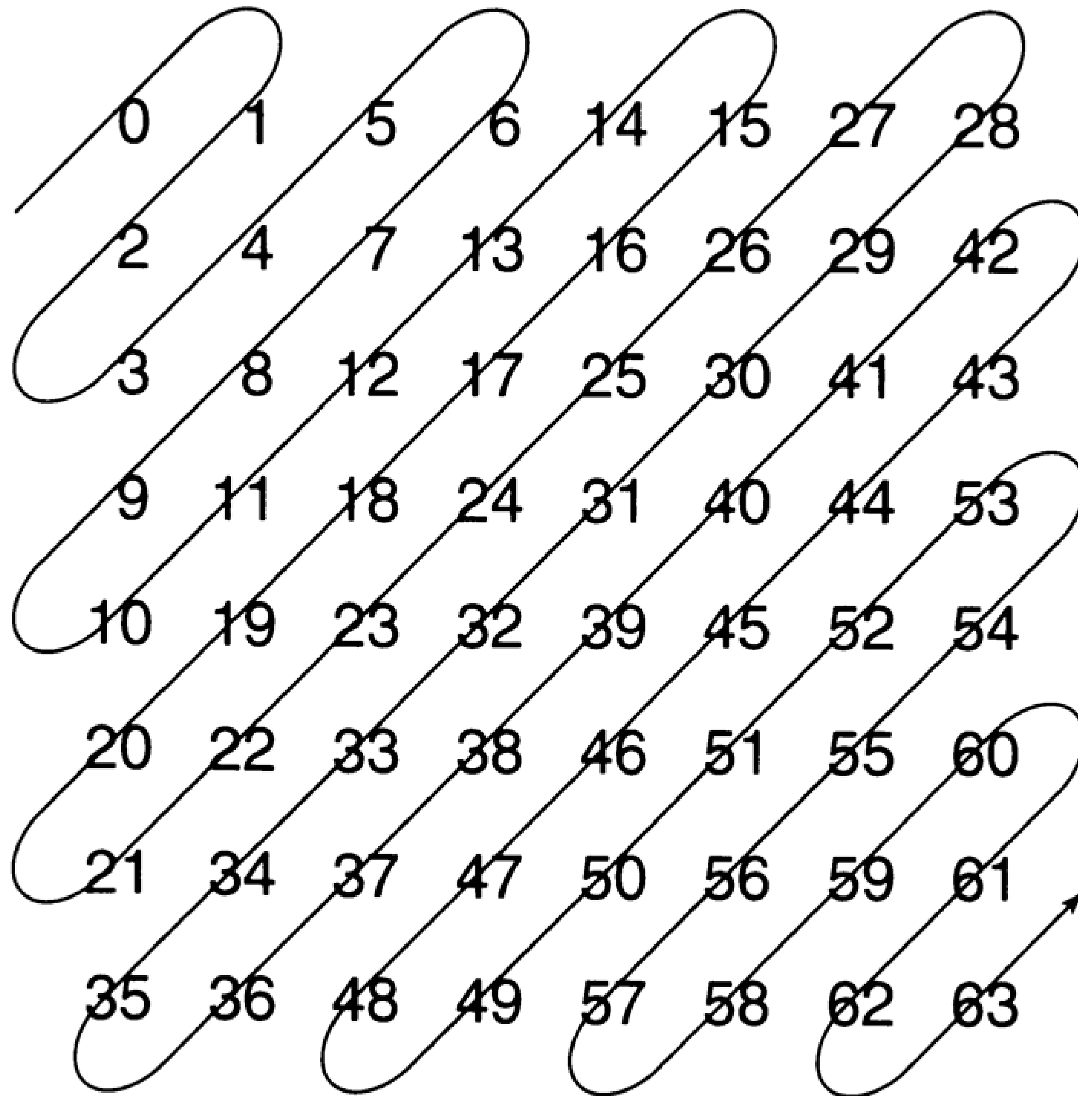
5000 7000 0...0...0

5,3 7,2 16,0 0,0

**Special codes**

**Rest is 0**

# Zigzag Pattern



# Magnitude Encoding

| Nr of bits | Range | |
|---|---|---|
| 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

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| -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 |

## 2) After zig-zag

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

## 3) After RLE

| Value(hex) | raw bits |
|---|---|
| 07 | 0011111 |
| 02 | 00 |
| 05 | 001111 |
| 62 | 10 |
| F0 | |
| F0 | |
| F0 | |
| 00 | |

Run of 0:s → (first hex digit) ; Magnitude ← (second hex digit)

## 4) Huffman coding

* The values are HC (variable length) using table lookup. Different tables for AC/DC and DC differentially encoded
* Raw bits are left untouched

## 5) Bitstream, written to mem bytewise (or wordwise)

11110_0011111 01_00 11010_001111 ...

11111111001 11111111001 11111111001 1010

# Huffman Encoding/Decoding

- Analogy
  - Morse Code

| J | P | E | G |
|---|---|---|---|
| .--- | .--. | . | --. |

- Binary Codes
  - Mutally Exlusive Codes
  - Binary Tree

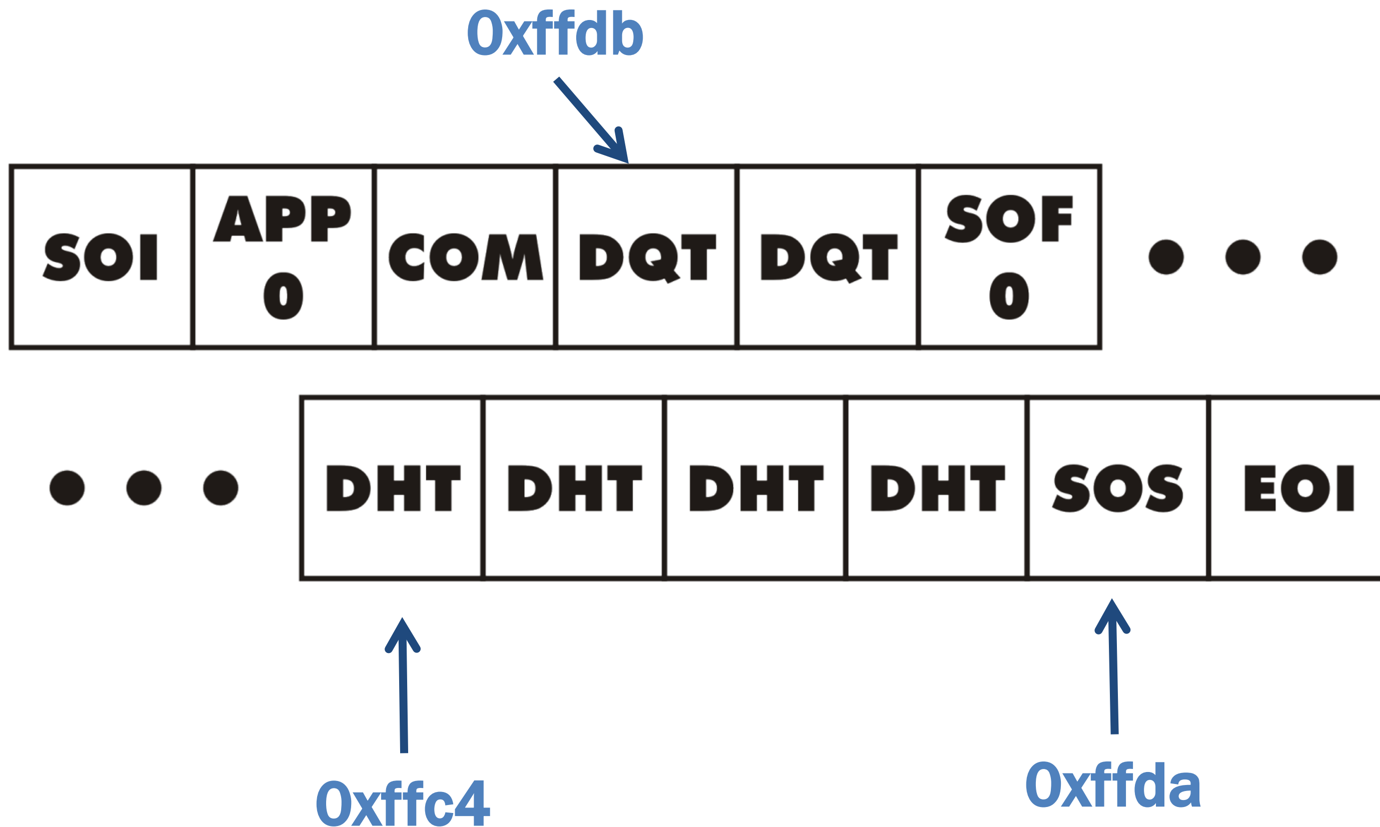
111 | 1100 | 0 | 10

J | P | E | G

1 0 E
1 0 G
1 0
J P

# JFIF Format

- JPEG File Interchange Format
  - Markers
  - Data



```
0xff,0xd8,  // SOI
0xff,0xe0,0x00,0x10,  // APP0
0x4a,0x46,0x49,0x46,0x00, // JFIF
0x01,0x01,0x00,0x00,0x01,
0x00,0x01,0x00,0x00,
0xff,0xdb,0x00,0x43,0x00, // DQT
0x08,0x06,0x06,0x07,0x06,0x05,0x08,0x07, // Quantization table in ZZ order
0x07,0x07,0x09,0x09,0x08,0x0a,0x0c,0x14, // div by 2
...
0x3d,0x38,0x32,0x3c,0x2e,0x33,0x34,0x32,
0xff,0xc0,0x00,0x0b,0x08, // SOF0
0x02,0x00, //height
0x01,0x90, //width
0x01,0x01,0x11,0x00,
0xff,0xc4, //DHT
0x00,0x1f,0x00,0x00,0x01,0x05,0x01,0x01,
0x01,0x01,0x01,0x01,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x01,0x02,0x03,0x04,
0x05,0x06,0x07,0x08,0x09,0x0a,0x0b,
0xff,0xc4, //DHT
0x00,0xb5,0x10,0x00,0x02,0x01,0x03,0x03,0x02,0x04,0x03,0x05,0x05,0x04,0x04,0x00,
...
0xda,0xe1,0xe2,0xe3,0xe4,0xe5,0xe6,0xe7,0xe8,0xe9,0xea,0xf1,0xf2,0xf3,0xf4,0xf5,
0xf6,0xf7,0xf8,0xf9,0xfa,
0xff,0xda, //SOS
0x00,0x08,0x01,0x01,0x00,0x00,0x3f,0x00
// image
0xffd9 // EOI
```

# 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 |
| 0x05 | 11010 | 5 |
| ... | ... | |

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

OR1200

PC

NPC

IC

FFs on outputs

WBI

m0

IR

RF

ALU

LSU

bitfield

FFs on outputs

DC

WBI

m1

m6

WB

PKMC

Boot ROM RAM

UART

tx

rx

s6

DCTQ DMA

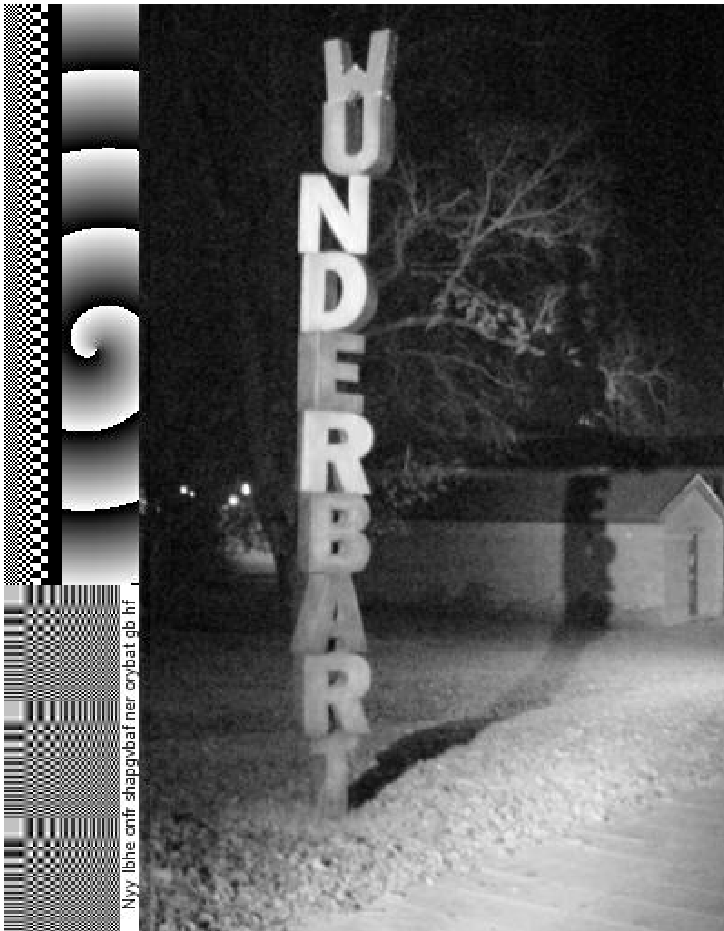
m6

testbild.jpg

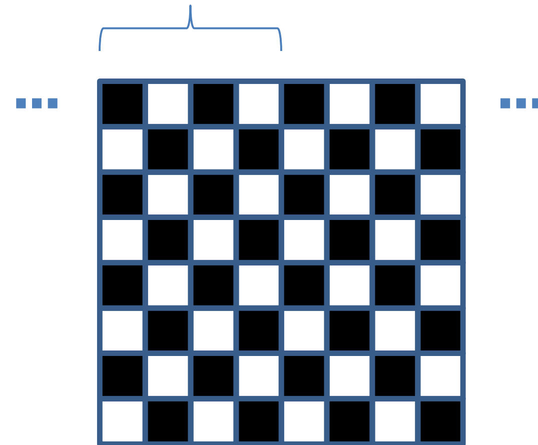
uCLinux

testbild.raw

# Raw image format in memory



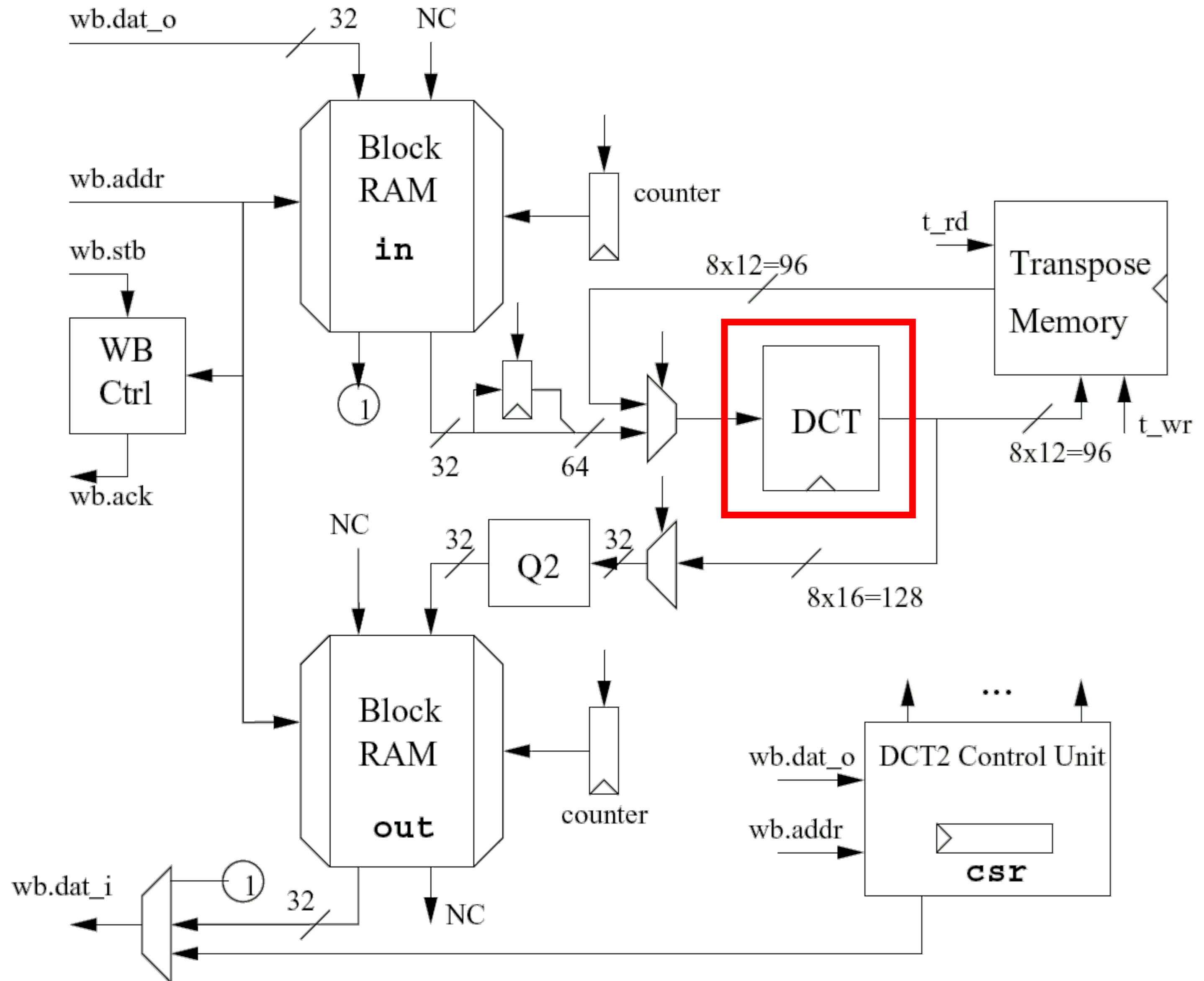

0x00ff00ff



8 bit pixels [0,255]
4 pixels/word

Somewhere 128 must be subtracted from each pixel!

# Proposed Architecture

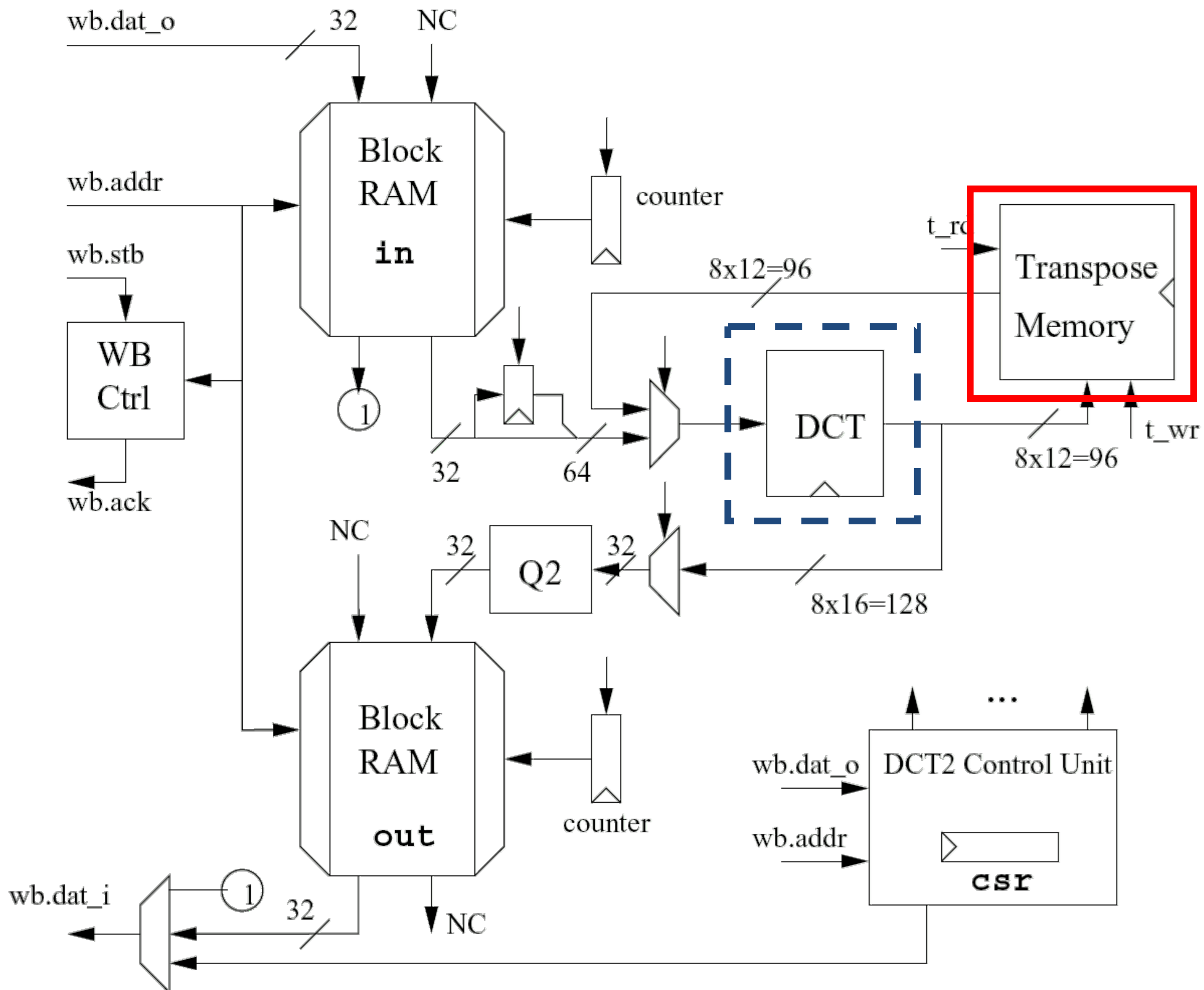


# DCT Module

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

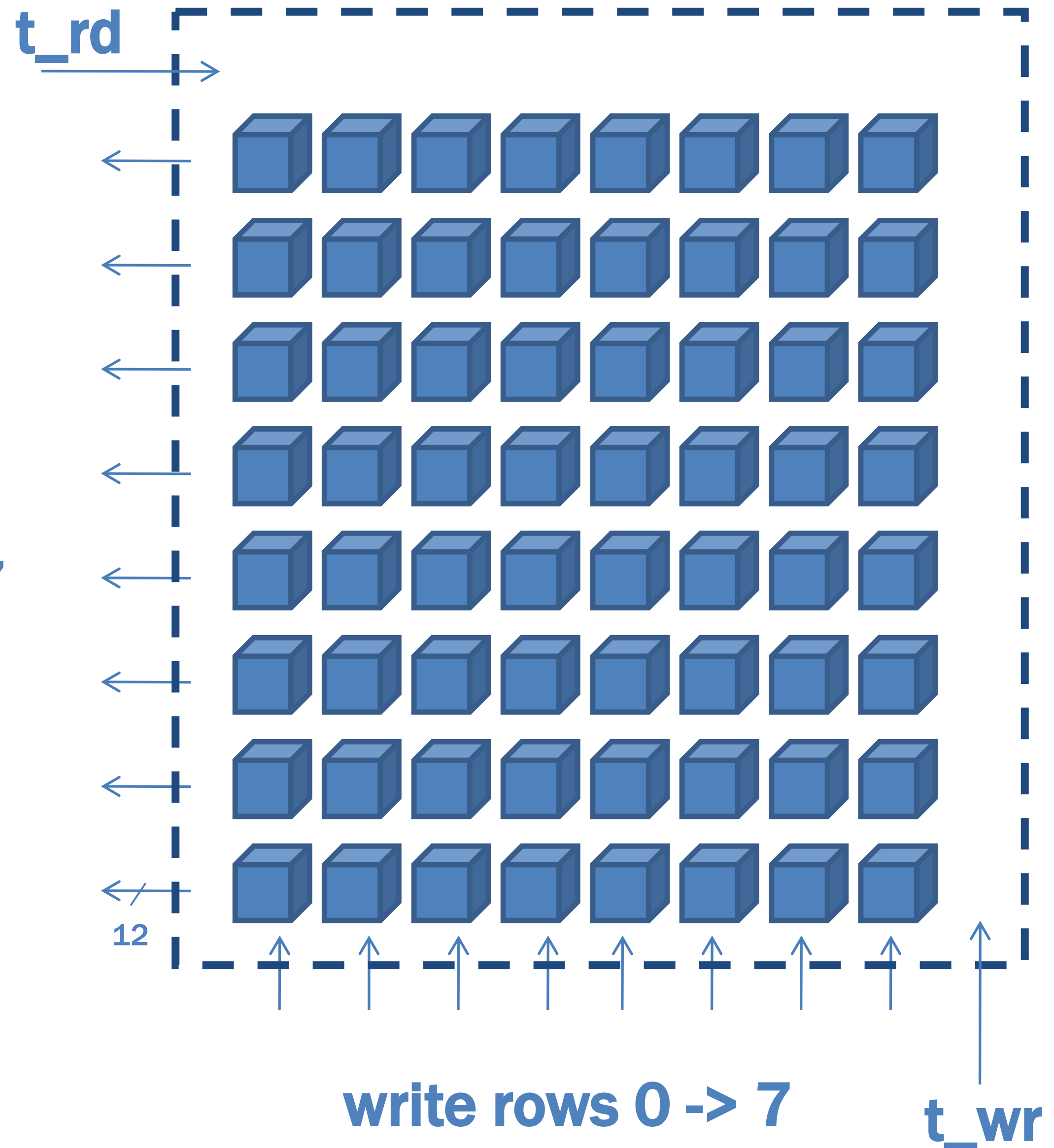
```
y = floor(sqrt(8)*dct(x))
```

# Proposed Architecture

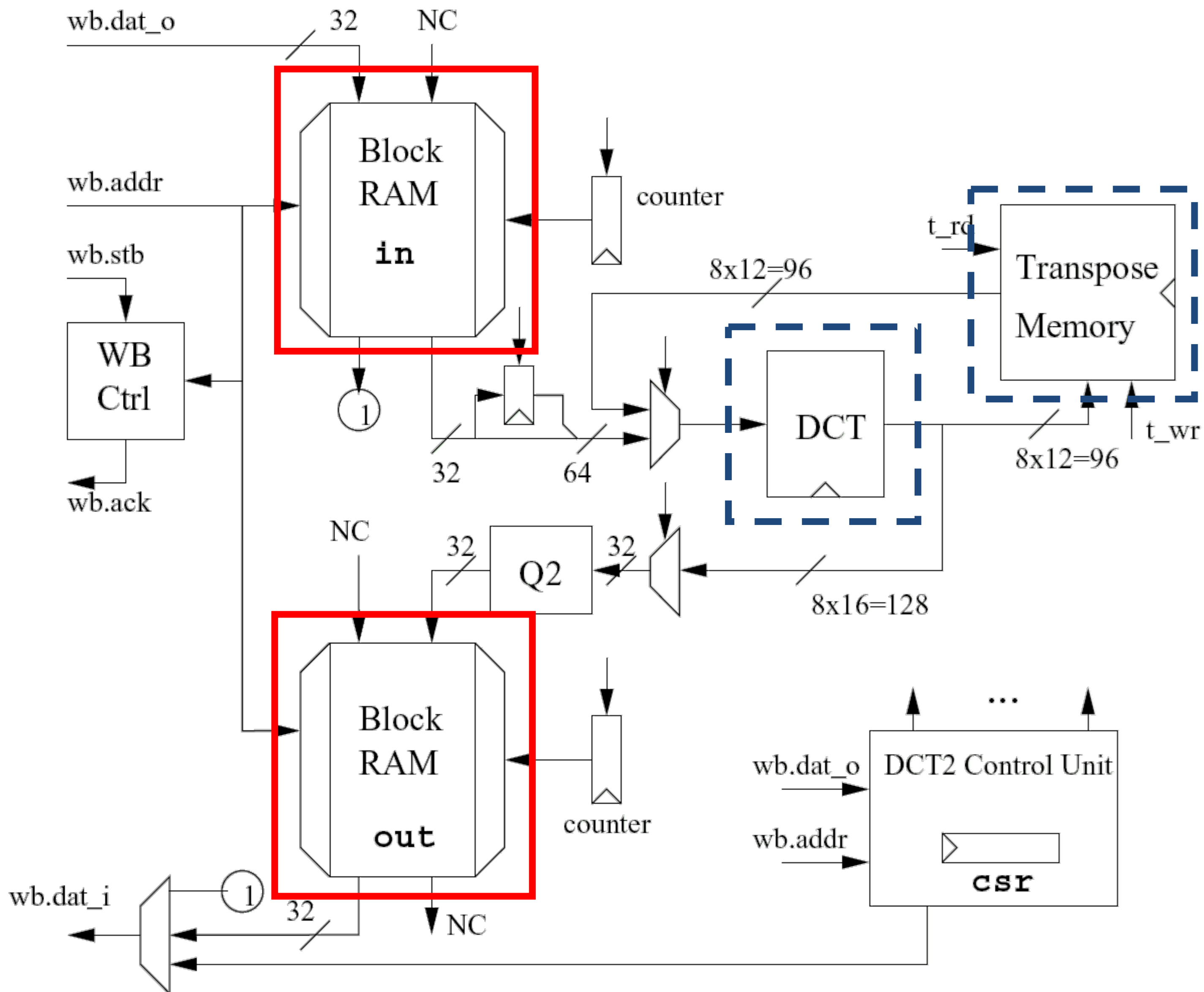


# Transpose Memory

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

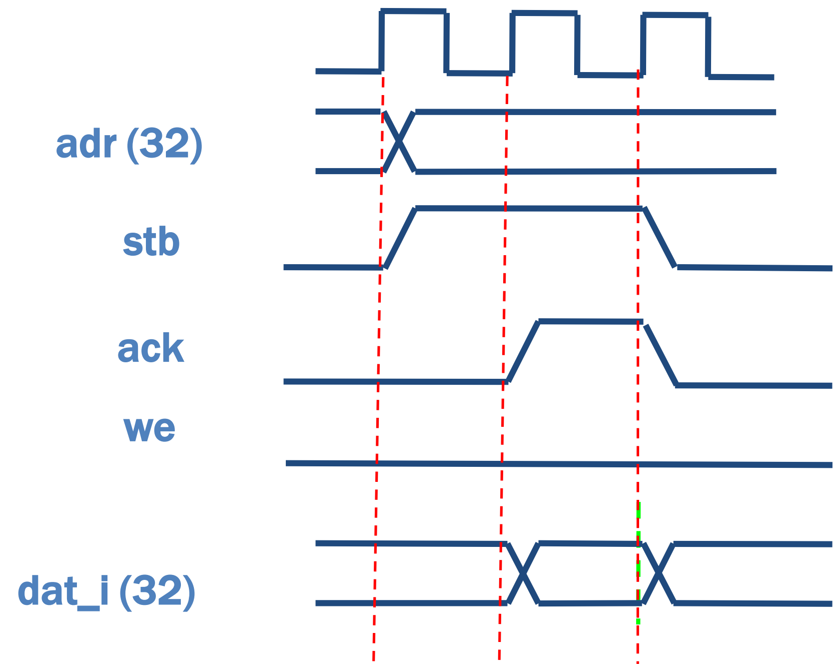


# Proposed Architecture

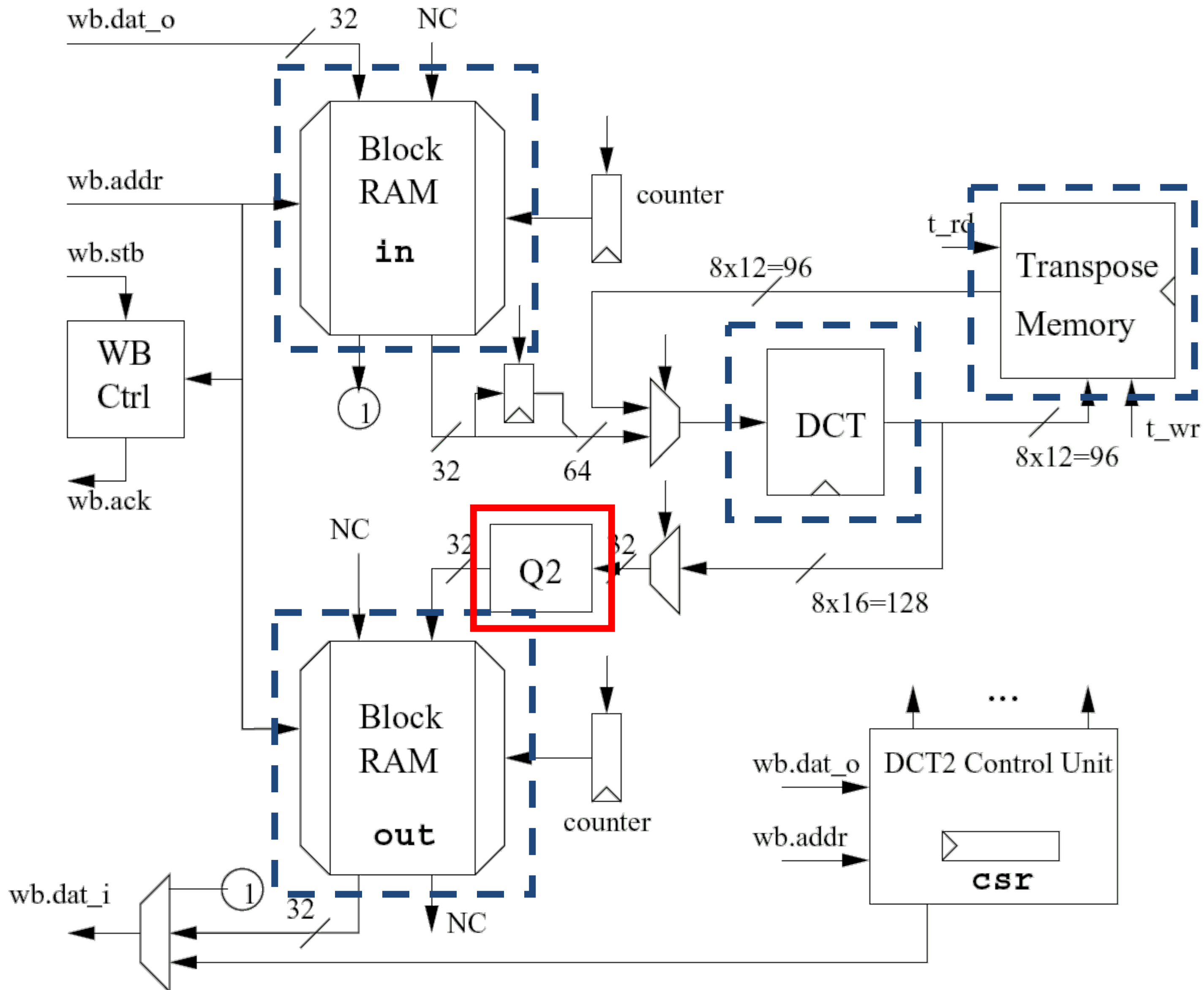


# Block RAM

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



# Proposed Architecture



# About Q2

```
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];
```

```
R = round(2^15 ./ Q);
```

**Precomputed reciprocals**

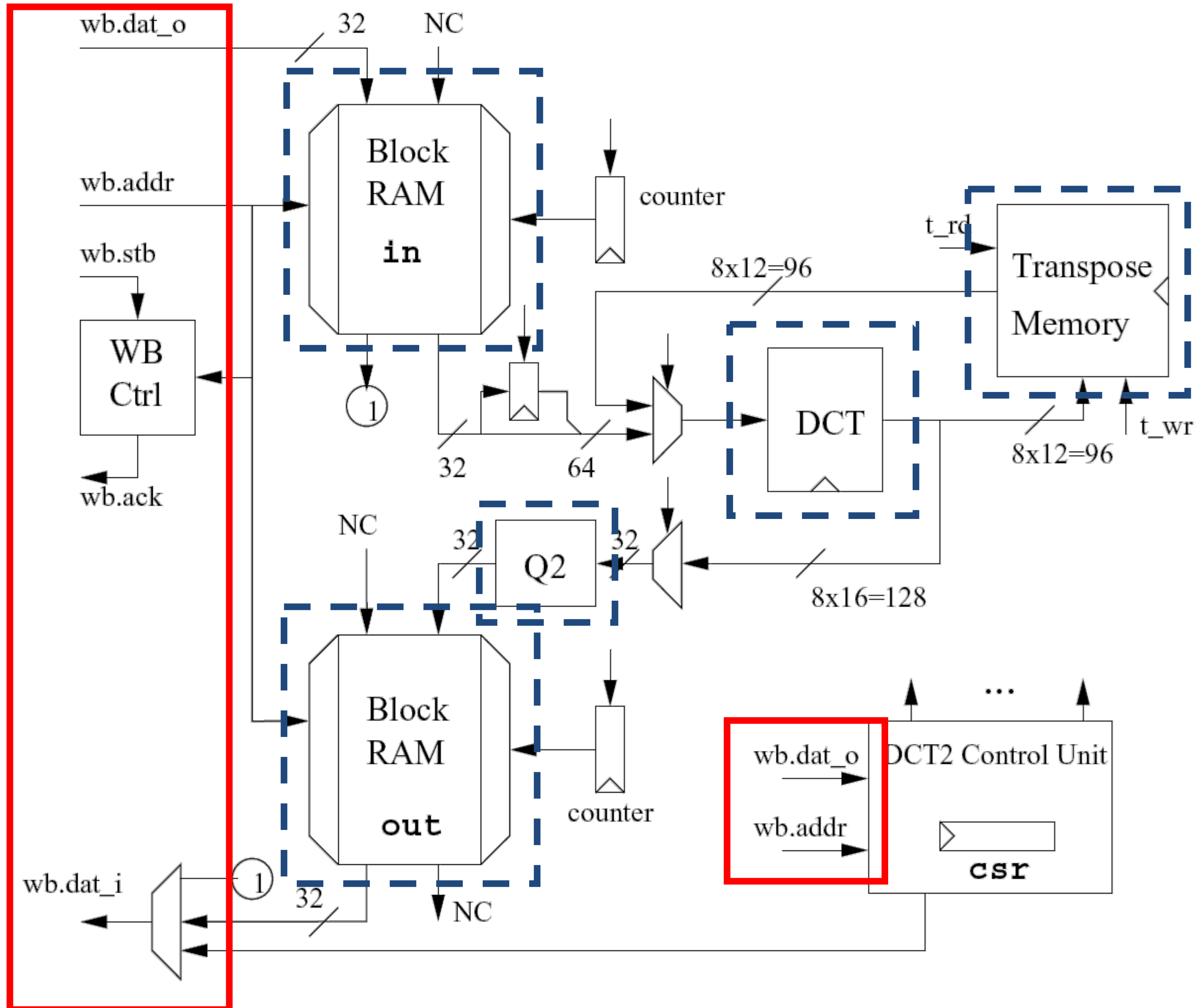
```
R =
    2048   2979   3277   2048   1365    819    643    537
    2731   2731   2341   1725   1260    565    482    596
    2341   2521   2048   1365    819    575    475    585
    2341   1928   1489   1130    643    377    410    529
    1820   1489    886    585    482    301    318    426
    1365    936    596    512    405    315    290    356
     669    512    420    377    318    271    273    324
     455    356    345    334    293    328    318    331
```

**YQ = round((Y .* R).*2^(-17));**

Y: 8 times too big

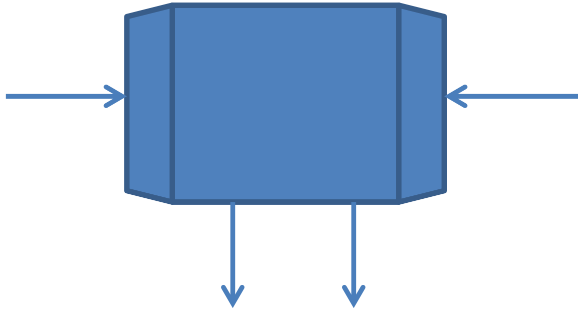
R: 2 times too small, since libjpeg uses 1/2*Q

# Proposed Architecture

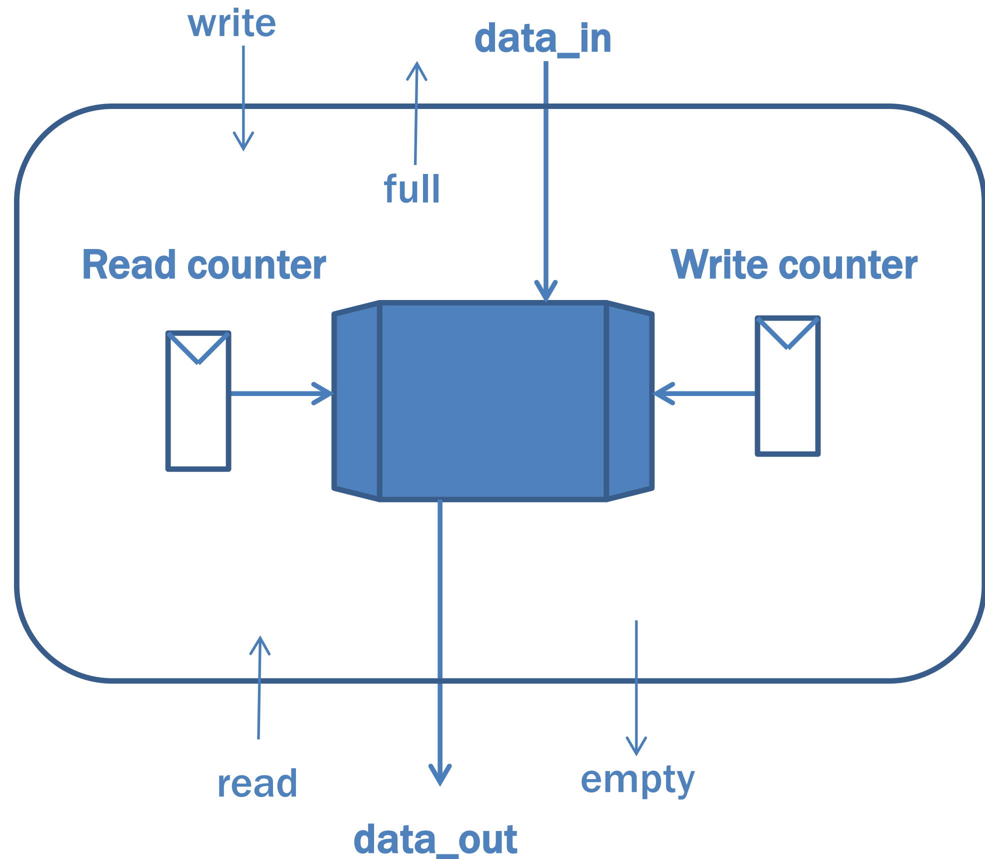


# Some ideas

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

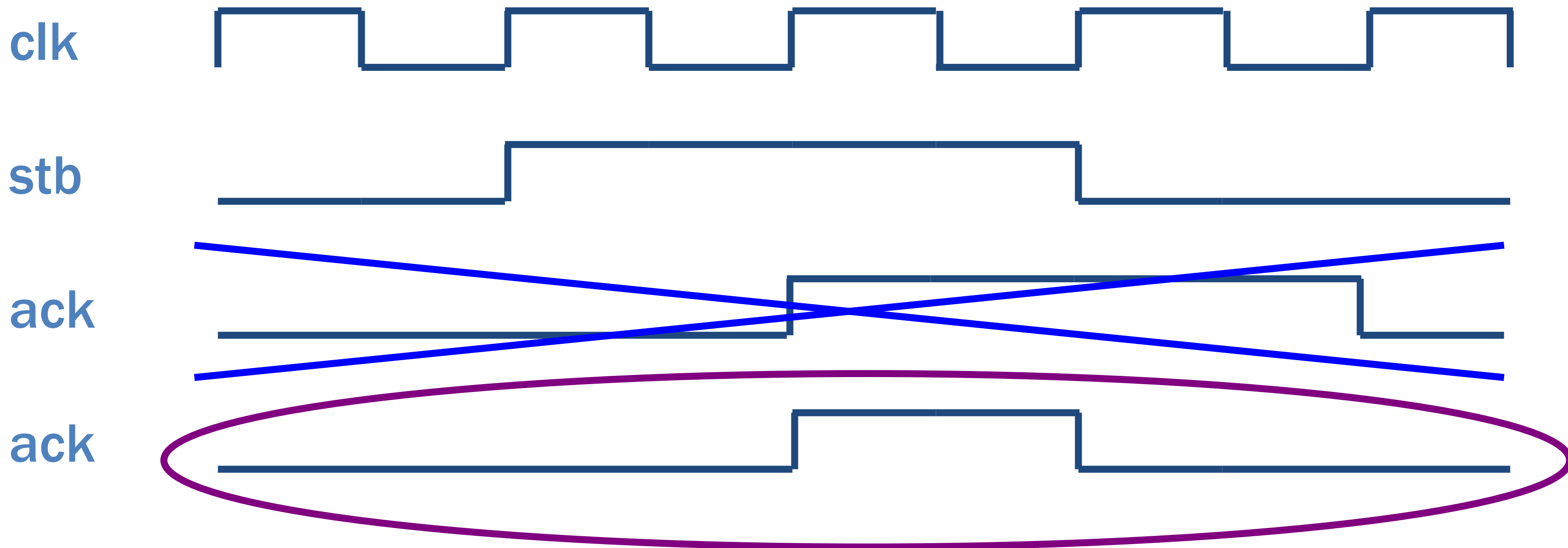


You can rebuild the BRAM
to a FIFO



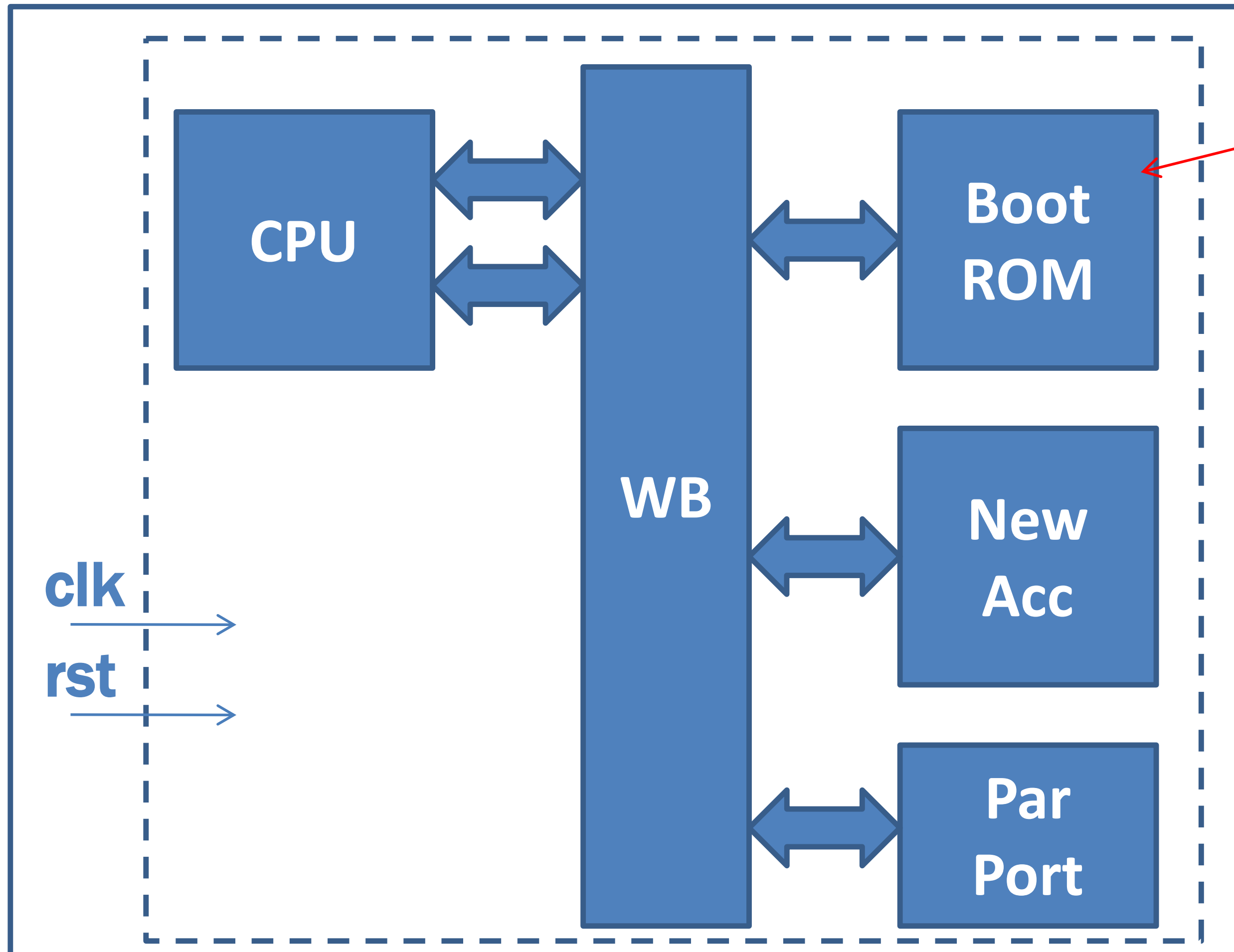
# Some Notes on the WB I/F

- Be careful with wb.ack



# Test benches - 2 alternatives

## 1) Simulate the whole computer – `make sim`



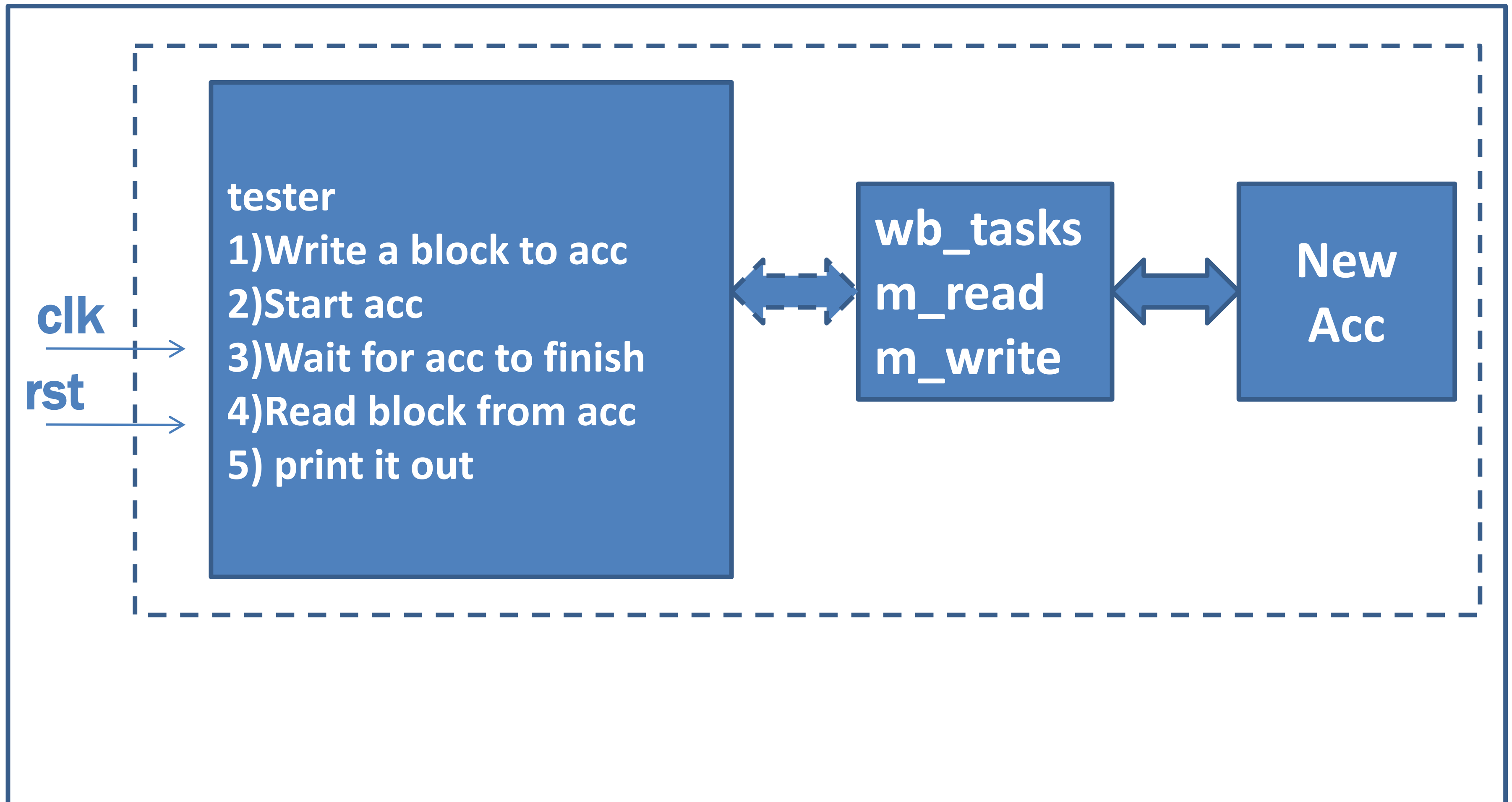

Insert some code
In the beginning of
the monitor `mon2.c`

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

**jpegtest**