Software Optimization Workshop for Real Time Video Applications

By

Dr. Jagadeesh Sankaran,
Member of the Group of Technical Staff,
DSP Software Applications Engineer,
Streaming Media Team.
SOFTWARE TECHNIQUES THAT ARE CRUCIAL TO EXTRACTING PERFORMANCE FROM DM642 DSP FOR VIDEO APPLICATIONS.

Techniques That Will be Reviewed
a) Code Optimization Techniques.
b) System Optimization Techniques.
c) Showcase DM642 performance for video applications.
Why do we need to optimize?

My algorithm is in C, TI says it has an optimizing compiler that can get up to 80% of the performance in assembly coding from C, yet my performance is terrible.

I have used the correct instructions in C using intrinsics yet the compiler is showing a high dependency bound, and hence my performance is messed up.

I do not believe in the compiler, I always write my own hand coded assembly.

This one algorithm consumes my whole chip (or nearly) and hence it cannot be performed on a DSP. TI C6000 DSP’s are hard to optimize.

Does not matter whether the DSP runs at 600MHz, 720MHz if the code that is written does not take advantage of the available units and the instructions of the architecture.

VLIW is a good architectural style in the hands of developers who know how to expose the available parallelism for the compiler to take advantage of.
What does Optimum Mean

An act, process, or methodology of making something fully perfect, functional, or effective as possible.

Continuous process of refinement in which code being optimized executes faster and takes fewer cycles, until a specific objective is achieved (real-time execution).

How do we know when to stop ??

Optimum:
Greatest degree attained or attainable under specified or implied conditions.

How do we figure out how fast a given algorithm can run on a given architecture ?
How do we determine the Optimum

Analysis is specific to the algorithm.

Raw performance for a given computation loop depends on:

a) Number of loads and stores needed.
b) Number of multiply operations needed.
c) Number of logical operations needed.
d) Size of the data that is being worked on shorts, bytes.

Given this many operations and the capabilities of the architecture how long should it take to perform this algorithm?

Are there any data related dependencies between the computations that will prevent performing “K” computations in parallel?

If there are no data dependencies then we should take only as many cycles as the most intensive arithmetic operation divided by how many such arithmetic operations can be performed on the architecture in a given cycle, which is the raw throughput of the architecture.

Because of the data dependency how much does the performance degrade from the raw throughput of the architecture.
KEY FEATURES OF TMS320C6x VLIW DSP

32-bit Registers: 64
2 Data paths: A and B

Units:
L: Logical units, AND, OR, XOR
S: Shifter unit, ADD, SHR, SHL, Extract, Branch
D: Load/Store unit
M: Multiply unit

Predication Registers:
A0, A1, A2, B0, B1, B2

Fetch packet: 256 bits

1 Execute Cycle:

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Register Usage</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDEC</td>
<td>S1</td>
<td>LOOP, A_i</td>
</tr>
<tr>
<td>!A_pd]ADD</td>
<td>L2X</td>
<td>B_t0, A_t0, B_temp1</td>
</tr>
<tr>
<td>!A_pd]ADD</td>
<td>L1</td>
<td>A_prodC, A_prodD, A_v0</td>
</tr>
<tr>
<td>DOTP2</td>
<td>M2</td>
<td>B_x5x4, B_y4y3, B_prod6</td>
</tr>
<tr>
<td>DOTP2</td>
<td>M1</td>
<td>A_x3x2, A_y2y1, A_prod5</td>
</tr>
<tr>
<td>PACKLH2</td>
<td>S2</td>
<td>B_y7y6, B_y5y4, B_y6y5</td>
</tr>
<tr>
<td>LDDW</td>
<td>D2T2</td>
<td>*B_xptr++[2], B_x7x6:B_x5x4</td>
</tr>
<tr>
<td>LDDW</td>
<td>D1T1</td>
<td>*A_xptr++[2], A_x3x2:A_x1x0</td>
</tr>
</tbody>
</table>
SOFTWARE PIPELINING

(P) PROLOG

(K) KERNEL

(E) EPILOG

X depends on algorithm dependencies.
Code Optimization Goals

Code should be written only after analyzing the optimum to serve as a proof of concept that the analysis performed can indeed translate to actual performance.

Several issues with code and code generation that may limit the performance obtained from compilers.

Is there any preferred/systematic methodology to perform code development for developers to share their results.

This systematic methodology can be provided back to TI (if there are no IP issues) for them to improve the compiler or provide suggestions on ways and means to improve the performance.
Code Optimization Methodology

Six different flavors of the same function being optimized.

1. Natural C code or Committee code. Text Book implementation of the algorithm to be optimized. Used to compare other flavors for speedup and for bit-exactness. This can also be viewed as the golden C code.

2. Optimized C code can use manual loop unrolling of inner and outer loops. It can also use compiler pragma’s and _nasserts to the compiler to inform it about the alignment of various input and output arrays. This allows the compiler to perform automatic SIMD transformations, which works in some cases but not all.

3. Intrinsic C code allows for the use of all the instructions on the given architecture and can be used to express any assembly language code in a high level environment. The only limitation is circular addressing support. The compiler may not be able to perform memory alias disambiguation or partition instructions correctly between the two data paths of the architecture.
4. Serial assembly is a mapping of the intrinsic C code into assembly by directly using the instructions in an assembly language format. Assembly optimizer is invoked to act on these instructions and perform register allocation and scheduling.

5. Partitioned serial assembly performs partitioning of the instructions by appending a .1 or .2 in front of the unit. Load/Store operations can be partitioned as .DxTx to indicate which side the pointer comes from and to which side the loaded value lands. The use of .1x shows that the second operand comes from the opposite data path.

Both partitioned and linear assembly do not have any latencies that the programmer must take care of. The assembly optimizer figures out latencies and dependencies and then performs instruction scheduling.

6. Of course, there is always hand code where the user does instruction set selection, register allocation and the latencies of the instructions.
Code Optimization Methodology

Optimized C code: Smallest effort. Relies totally on compiler to perform automatic code transformations to use new instructions.

Intrinsic C code: More effort. However very flexible method to map out all the instructions and the algorithmic transformations that are needed for optimizing the algorithm. Can be very useful for mapping algorithm to serial assembly.

Serial assembly and Partition serial assembly allow further control and mapping to units and remove scheduling issues associated with C code.

All flavors except hand code can benefit as follows:

a. Improvements to the compiler and code generation tools.
b. Can be re-scheduled to avoid new architecture restrictions. eg. C64x cross path stall can be avoided in C62x code by re-compiling.
c. Memory dependencies and latencies are automatically taken care of by tools, whereas hand code assumes a fixed latency. Hence serial and partitioned serial assembly code are pipeline independent but yet high performance.

CHOICE IS IN YOUR HANDS
EXAMPLES IN THIS CLASS

Video Processing Examples (illustration)

1) Motion Estimation (8x8) search for minimum SAD value. Put Bits, Variable length encoding.

Image/Video Processing Lab Examples on DM642

3) Convolution, Quantization conv_3x3
4) Threshold operation (Rate Control).

Each example will be optimized using the code optimization methodology outlines earlier in six flavors as discussed earlier.
Find \((x, y)\) that is the best match for the 8x8 region in blue that we are searching in a green region of size \(X,Y\). The \((x, y)\) may not be word or double word aligned. Each pixel is differenced and the sum of absolute values is added. The value of \((x,y)\) is the location which yields the minimum such value.
void mad_8x8_cn
{
    const unsigned char *restrict refImg,
    const unsigned char *restrict srcImg,
    int pitch, int sx, int sy,
    unsigned int *restrict motvec
}
{
    int i, j, x, y, matx, maty;
    unsigned matpos, matval;
    matval = ~0U;   matx = maty = 0;
    pitch = sx;

    for (x = 0; x < sx; x++)
        for (y = 0; y < sy; y++)
        {
            unsigned acc = 0;
            for (i = 0; i < 8; i++)
                for (j = 0; j < 8; j++)
                    acc += abs(srcImg[i*8 + j] - refImg[(i+y)*pitch + x + j]);

            if (acc < matval)
                {
                    matval = acc; matx = x; maty = y;
                }
        }

    matpos = (0xffff0000 & (matx << 16)) | (0x0000ffff & maty);
    motvec[0] = matpos;
    motvec[1] = matval; }
Analysis Of Optimal Implementation of mad_8x8 for C64x DSP

Number of Byte loads required: 128 * sx * sy.
Number of Absolute and differences required: 64 * sx * sy.
Number of additions required: 64 * sx * sy.
Number of compares: sx * sy.

Observations

a. C64x DSP can perform 8 sum of absolute differences per cycle. Since 64 SAD computations need to be performed for a 8x8 block, this can at best be performed in 8 cycles provided we can load the required data within 8 cycles.

b. The C64x has a load/store bandwidth of 16 bytes/cycle if the loads are performed at an aligned double address. It has a load/store bandwidth of 8 bytes/cycle if the loads are performed at a non-aligned address.

Since the data needs to be loaded from any byte location, the addresses are not aligned to a word or a double word boundary and hence the C64x does not have the required bandwidth to load both source and reference data in 8 cycles. How do we overcome this???
Since we do not have the required load bandwidth, we should look for opportunities to re-use the data. This can be done by loading the 8x8 region into register file as it never changes during the loop. It is essentially loop invariant and hence can be kept in registers in 16 registers. Hence this data is read outside the loop into registers.

Given that we have solved the load bandwidth problem, there seems to be nothing limiting us from achieving an 8 cycle loop for computing a 8x8 minimum absolute difference.

Before we start it is interesting to observe the compiled output for natural C code to see our current performance.

For a search region of 32x64 the natural C code takes 22254 cycles to perform 32 * 64 * 64 = 131072 achieving 5.88 SAD/cycle.
Assembly code for Natural C Code
The performance obtained from natural C code is more impressive than usual as the compiler unrolls the two inner loops and uses the subabs4 instructions on its own performing 16 SUBABS4 instructions in 10 cycles in the inner loop to achieve a 6.4 SAD/cycle.

```c
for (x = 0; x < sx; x++)
for (y = 0; y < sy; y++)      Collapse these two loops together.
{
    #pragma UNROLL(8);
    for (i = 0; i < 8; i++) acc += abs(srcImg[i*8 + 0] -
        refImg[(i+y)*pitch + x + 0]) + abs(srcImg[i*8 + 1] -
        refImg[(i+y)*pitch + x + 1]) + abs(srcImg[i*8 + 2] -
        refImg[(i+y)*pitch + x + 2]) + abs(srcImg[i*8 + 3] -
        refImg[(i+y)*pitch + x + 3]) + abs(srcImg[i*8 + 4] -
        refImg[(i+y)*pitch + x + 4]) + abs(srcImg[i*8 + 5] -
        refImg[(i+y)*pitch + x + 5]) + abs(srcImg[i*8 + 6] -
        refImg[(i+y)*pitch + x + 6]) + abs(srcImg[i*8 + 7] -
        refImg[(i+y)*pitch + x + 7]);
    if (acc < matval) { matval = acc; matx = x; maty = y; }
}
```
*********** Loop Control*************************

SUB B_v1, 1, B_v1
MV A_p1, A_f

[!B_v1] MV A_hfix, A_f
[!B_v1] MV B_v_dim, B_v1

*********** Row 7 ***********************************

LDNDW *A_ref_d(A_p7), A_ref7h:A_ref7l
SUBABS4 B_src7h, A_ref7h, B_err7h
SUBABS4 A_src7l, A_ref7l, A_err7l
DOTPU4 B_err7h, B_k_one, B_mad_7
DOTPU4 A_err7l, A_k_one, A_mad_7

*********** Row 6 ***********************************

LDNDW *A_ref_d(A_p6), B_ref6h:B_ref6l
SUBABS4 B_src6h, B_ref6h, B_err6h
SUBABS4 A_src6l, B_ref6l, A_err6l
DOTPU4 B_err6h, B_k_one, B_row_6
DOTPU4 A_err6l, A_k_one, A_row_6
ADD B_row_6, B_mad_7, B_mad_6
ADD A_row_6, A_mad_7, A_mad_6

.. *********** Check for best match ***********

ADD B_mad_0, A_mad_2, B_mad
CMPGT2 B_best, B_mad, B_bst
ADD B_hvl, 1, B_hvl
[B_bst] MV B_mad, B_best
[B_bst] SUB B_hvl, 1, B_bhvl
[A_i] BDEC loop, A_i
mad8x8_loop:

ADD .D2  B_hvl,  1,  B_hvl ; [25,1]
DOTPU4 .M1  A_err0l,  A_k_one,  A_row_0 ; [17,2]
ADD  A_err3h,  B_k_one,  B_row_3 ; [ 9,3]
SUBABS4 .L1  A_src5l,  A_ref5l,  A_err5l ; [ 9,3]
SUBABS4 .L2X  B_src5h,  A_ref5h,  B_err5h ; [ 9,3]
LDNDW .D  *A_ref_d(A_p7),  A_ref7h:A_ref7l ; [ 1,4]
MV .S1  A_p1,  A_f ; [ 1,4]

loop_1:
ADD .D2X  A_row_0,  B_mad_0,  B_mad_0 ; [26,1]
ADD  B_row_1,  B_mad_2,  B_mad_1 ; [18,2]
ADD  A_row_3,  A_mad_4,  A_mad_3 ; [18,2]
SUBABS4 .L1X  A_src4l,  B_ref4l,  A_err4l ; [10,3]
SUBABS4 .L2  B_src4h,  B_ref4h,  B_err4h ; [10,3]
DOTPU4 .M2  B_err5h,  B_k_one,  B_row_5 ; [10,3]
DOTPU4 .M1  A_err6l,  A_k_one,  A_row_6 ; [10,3]
LDNDW .D  *A_ref_d(A_p6),  B_ref6h:B_ref6l ; [ 2,4]

loop_2:
[B_i]  BDEC .S1  mad8x8_loop,  A_i ; [27,1]
ADD .S2X  B_mad_0,  A_mad_2,  B_mad ; [27,1]
SUBABS4 .L2  B_src2h,  B_ref2h,  B_err2h ; [11,3]
SUBABS4 .L1  A_src3l,  A_ref3l,  A_err3l ; [11,3]
DOTPU4 .M2  B_err4h,  B_k_one,  B_row_4 ; [11,3]
DOTPU4 .M1  A_err5l,  A_k_one,  A_row_5 ; [11,3]
LDNDW .D  *A_ref_d(A_p3),  A_ref3h:A_ref3l ; [ 3,4]
SUB .D2  B_v1,  1,  B_v1 ; [ 3,4]
loop_3:
  CMPGT2 .S2  B_best, B_mad, B_best ;[28,1]
  SUBABS4 .L2X B_src1h, A_ref1h, B_err1h ;[12,3]
  DOTPU4 .M2 B_err2h, B_k_one, B_row_2 ;[12,3]
  SUBABS4 .L1X A_src2l, B_ref2l, B_err2l ;[12,3]
  DOTPU4 .M1 A_err4l, A_k_one, A_row_4 ;[12,3]
  ADD .D2 B_row_6, B_row_7, B_mad_6 ;[12,3]
  LDNDW .D *A_ref_d(A_p5), A_ref5h:A_ref5l ;[ 4,4]
  ![B_v1]MV .S1 A_h_fix, A_f ;[ 4,4]

loop_4:
  ![B_bst]SUB .D2 B_hvl, 1, B_bhvl ;[29,1]
  ![B_bst]MV .S2 B_mad, B_best ;[29,1]
  ADD .S1 A_row_2, A_mad_3, A_mad_2 ;[21,2]
  SUBABS4 .L1X A_src0l, B_ref0l, A_err0l ;[13,3]
  SUBABS4 .L2 A_src0h, B_ref0h, B_err0h ;[13,3]
  DOTPU4 .M2 B_err1h, B_k_one, B_row_1 ;[13,3]
  DOTPU4 .M1 A_err3l, A_k_one, A_row_3 ;[13,3]
  ![A_i]LDNDW .D *A_ref_d(A_p4), B_ref4h:B_ref4l ;[ 5,4]

  DOTPU4 .M2 B_err0h, B_k_one, B_row_0 ;[14,3]
  SUBPU4 .L1 A_src1l, A_ref1l, A_err1l ;[14,3]
  DOTPU4 .M1 A_err2l, A_k_one, A_row_2 ;[14,3]
  ADD .D2 B_row_5, B_mad_6, B_mad_5 ;[14,3]
  ADD .S1 A_row_6, A_row_7, A_mad_6 ;[14,3]
  ![A_i]LDNDW .D *A_ref_d(A_p2), B_ref2h:B_ref2l ;[ 6,4]
  ![A_i]SUBABS4 .L2X B_src7h, A_ref7h, B_err7h ;[ 6,4]
  ![B_v1]MV .S2 B_v_dim, B_v1 ;[ 6,4]
64 sum of absolute differences is computed in 8 cycles at the peak throughput rate of 8/cycle. Also as suspected the full non-aligned load bandwidth of 64 bits/cycle is used.

The performance of the different flavors of code is now summarized in the next slide.
## MAD_8x8 Performance Summary for (64x32)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>22254 cycles</td>
<td>22254 cycles</td>
<td>22254 cycles</td>
<td>20561 cycles</td>
<td>16454 cycles</td>
<td>16458 cycles</td>
</tr>
<tr>
<td>1280 bytes</td>
<td>1280 bytes</td>
<td>1248 bytes</td>
<td>1056 bytes</td>
<td>1280 bytes</td>
<td>800 bytes</td>
<td>7.96 MAD/cycle</td>
</tr>
</tbody>
</table>
Put Bits, Variable Length Encoding

- `code_ptr`
- `len_ptr`
- `num_entries`
- `out_reg`
- `out_buff`
- `bp = 7`
- `code_ptr`
- `len_ptr`
Put Bits, Variable Length Encoding

```c
void put_bits
(
    unsigned int *code_ptr,
    unsigned int *len_ptr,
    int bp,
    unsigned char *out_buf,
    int num_entries,
    unsigned int out_reg
)
{
    int i;
    unsigned int code, len, cws, r;
    for( i = 0; i < num_entries; i++ )
    {
        code = code_ptr[i];  len = len_ptr[i];
        cws = (code >> bp);
        if (len) out_reg = out_reg | cws;
        bp = (bp + len);
        r     = (bp - 31);
        if (r)
        {
            *out_buf++ = out_reg;
            out_reg = (code << (len - r));
            bp -= len - r;
        }
    }
}
```

4 cycle loop from C Code
L2: ; PIPED LOOP KERNEL

[ A0]  OR .S2  B8,B6,B6 ; [30] <0,9>
[ A1]  MV .L2X  A5,B7 ; [38] <0,9> Define a twin register
[ A0]  OR .D1X  B8,A3,A3 ; [30] <0,9> Define a twin register
   ROTL .M1  A8,0,A0 ; [27] <1,5> Split a long life
   ADD .S1  A8,A4,A4 ; [31] <1,5> ^
   LDW .D2T2  *B4++,B8 ; [26] <2,1>
[ A1]  STB .D2T2  B6, *B5++ ; [36] <0,10>
[ A1]  SHL .S1X  B9,A5,A3 ; [37] <0,10>
   ROTL .M2  B8,0,B9 ; [26] <1,6> Split a long life
   ADD .L2X  A8,B7,B7 ; [31] <1,6> Define a twin register
   SHRU .S2  B8,B7,B8 ; [30] <1,6>
   ADD .D1  A6,A4,A2 ; <1,6> ^
   ROTL .M1  A2,0,A1 ; <1,7> Split a long life
[ B0]  BDEC .S2  L2,B0 ; [40] <1,7>
[ A1]  MV .D2X  A3,B6 ; [37] <0,12> Define a twin register
[ A2]  MV .S1  A5,A4 ; [38] <1,8> ^
[ A2]  LDW .D1T1  *A7++,A8 ; [27] <3,0>
Put Bits: Variable Length Encoding, Complexity Analysis

bp = bp + len

r = bp - 31

r > 0

bp = len - r

Y

N

4 cycles
bp = bp + len

Compute r in parallel

bp = bp & 31;

Store complete word if r > 0

3 cycle loop must be possible.

Put Bits: Variable Length Encoding, Complexity Analysis
void put_bits(
    unsigned int *code_ptr,
    unsigned int *len_ptr,
    int bp,
    unsigned char *out_buf,
    int num_entries,
    unsigned int out_reg
) {
    int i;
    unsigned int code, len, cws, cw, r;

    for (i = 0; i < num_entries; i++) {
        code = code_ptr[i];
        len = len_ptr[i];
        cws = (code >> bp);
        cw = (code << (32 - bp));

        if (len) out_reg = out_reg | cws;
        bp = (bp + len);
        r = (bp - 31);
        bp = bp & 31;

        if (r) {
            *out_buf++ = out_reg;
            out_reg = cw;
        }
    }
}
3 cycle loop for put bits.

L2:    ; PIPED LOOP KERNEL

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SHL .S1X</td>
<td>B16,A6,A6</td>
<td>30</td>
<td>&lt;0,9&gt;</td>
</tr>
<tr>
<td>MVD .M1</td>
<td>A0,A1</td>
<td>27</td>
<td>&lt;1,6&gt; Split a long life</td>
</tr>
<tr>
<td>BDEC .S2</td>
<td>L2,B0</td>
<td>42</td>
<td>&lt;1,6&gt;</td>
</tr>
<tr>
<td>LDW .D2T2</td>
<td>*B7++,B8</td>
<td>30</td>
<td>&lt;3,0&gt;</td>
</tr>
</tbody>
</table>

[B0]
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD .D2</td>
<td>B4,B9,B1</td>
<td>34</td>
<td>&lt;0,10&gt;</td>
</tr>
<tr>
<td>OR .S1X</td>
<td>B5,A5,A5</td>
<td>32</td>
<td>&lt;0,10&gt; ^</td>
</tr>
<tr>
<td>ADD .L2X</td>
<td>A0,B6,B9</td>
<td>33</td>
<td>&lt;1,7&gt; ^</td>
</tr>
<tr>
<td>SHRU .S2</td>
<td>B16,B6,B5</td>
<td>32</td>
<td>&lt;1,7&gt;</td>
</tr>
<tr>
<td>LDW .D1T1</td>
<td>*A7++,A0</td>
<td>27</td>
<td>&lt;3,1&gt;</td>
</tr>
</tbody>
</table>

[A0]
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>MV .L1</td>
<td>A6,A5</td>
<td>40</td>
<td>&lt;0,11&gt; ^</td>
</tr>
<tr>
<td>STB .D1T1</td>
<td>A5,*A4++</td>
<td>39</td>
<td>&lt;0,11&gt; ^</td>
</tr>
<tr>
<td>EXTU .S2</td>
<td>B9,27,27,B6</td>
<td>35</td>
<td>&lt;1,8&gt; ^</td>
</tr>
<tr>
<td>SUB .S1X</td>
<td>A3,B6,A6</td>
<td>30</td>
<td>&lt;1,8&gt; ^</td>
</tr>
<tr>
<td>ROTL .M2</td>
<td>B8,0,B16</td>
<td>30</td>
<td>&lt;2,5&gt; Split a long life</td>
</tr>
</tbody>
</table>

[B1]
<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
</table>
Get Bits, Inverse of Put Bits

- off
- 0xDEADBEEF
- 0xFADE5AA5
- 0x4CADEABC
- data_buff
- 0x01
- 0x10
- 0x11
- CW
- len
- 0xDEAXXXXX
- 0xABCDEFXX
- 0xFADXXXXX
- out_buff
- bp
- 1101 1110 1010 1101 1101 1101 1111
- buff
- len_buf
- 3
- 5
- 12
- 14
- num_entries
void get_bits
(
    unsigned char   *len_buf,
    unsigned int    *data_buf,
    unsigned int    *bptr,
    unsigned int    *buffer,
    unsigned int    *offset,
    unsigned int    num_entries,
    unsigned int    *output
)
{
    int i;
    unsigned int cw;
    unsigned int bp  = *bptr;
    unsigned int buff = *buffer;
    unsigned int off  = *offset;
    unsigned int length;
    unsigned int rem;
for (i = 0; i < num_entries; i++)
{
    length = len_buf[i];
    if (length <= bp)
    {
        cw = (buff >> (32 - length));
        buff = (buff << length);
        bp = bp - length;
    }
    else
    {
        cw = (buff >> (32 - bp));
        buff = data_buf[off];
        off++;
        rem = (length - bp);
        cw = (cw << rem) | (buff >> (32 - rem));
        buff = (buff << rem);
        bp = 32 - rem;
    }
    output[i] = cw;
}
*buffer = buff;
*offset = off;
*bptr = bp;
Compiler Analysis for Natural C code

;* ----------------------------------------------- ;*
;* SOFTWARE PIPELINE INFORMATION                ;*
;* ----------------------------------------------- ;*
;* Loop source line : 23                         ;*
;* Loop opening brace source line : 24           ;*
;* Loop closing brace source line : 44           ;*
;* Known Minimum Trip Count : 1                 ;*
;* Known Max Trip Count Factor : 1              ;*
;* Loop Carried Dependency Bound(^) : 4          ;*
;* Unpartitioned Resource Bound : 4             ;*
;* Partitioned Resource Bound(*) : 5            ;*
;* Resource Partition:
;*                                A-side   B-side
;* .L units                     0        1
;* .S units                     3        4
;* .D units                     1        2
;* .M units                     0        0
;* .X cross paths               2        3
;* .T address paths             1        2
;* Long read paths              0        0
;* Long write paths             0        0
;* Logical ops (.LS)            1        0    (.L or .S unit)
;* Addition ops (.LSD)          2        8    (.L or .S or .D unit)
;* Bound(.L .S .LS)             2        3
;* Bound(.L .S .D .LS .LSD)     3        5*
;* Searching for software pipeline schedule at ...  
;*      ii = 5  Schedule found with 3 iterations in parallel
Assembly Code From C for Get Bits/ VLD 5 cycle loop

L4: ; PIPED LOOP KERNEL

[!B1] SUB .D2 B21, B22, B21 ; 30 <0,9>  
[ B1] SUB .S1X A4, B21, A5 ; 37 <0,9> Define a twin register  
[ B1] SHL .S2 B20, B4, B6 ; 38 <0,9>  
[ B1] MV .L2 B19, B21 ; 39 <0,9>  
[ B1] ADD .D1 1, A3, A3 ; 36 <0,10>  
[!B1] SHRU .S2X A6, B5, B20 ; 28 <0,10>  
[!B1] SHL .S1 A6, A4, A6 ; 29 <0,10>  
  CMPGTU .L2 B22, B21, B0 ; <1,5>  
  LDBU .D2T2 *++B16, B22 ; 25 <2,0>  
[ B1] SHL .S1 A6, A5, A6 ; 40 <0,11>  
[ B1] SHRU .S2X A6, B19, B7 ; 38 <0,11>  
  ROTL .M2 B0, 0, B1 ; <1,6> Split a long life  
[ B0] SUB .D2 B22, B21, B4 ; 37 <1,6>  
[ B0] SUB .L2 B17, B22, B8 ; 38 <1,6>  
[ B0] LDW .D1T1 *+A7 [A3], A6 ; 29 <1,6>  
[ B1] OR .L2 B7, B6, B20 ; 38 <0,12>  
[ B0] SUB .D2 B17, B21, B9 ; 28 <1,7>  
[ B0] ADD .S2 B21, B8, B19 ; 38 <1,7>  
  STW .D2T2 B20, *B18++ ; 43 <0,13>  
[!B1] SUB .L2 B17, B22, B5 ; 28 <1,8>  
  MV .D1X B22, A4 ; 25 <1,8> Define a twin register  
[ B1] SHRU .S2X A6, B9, B20 ; 28 <1,8>  
[ A0] BDEC .S1 L4, A0 ; 44 <1,8>
for (i = 0; i < num_entries; i++)
{
    length = len_buf[i];
    cw    = (buff >> (32 - length));
    buff  = (buff << length);

    bp_c  = bp - length;
    rem   = (length - bp);
    flag  = (length <= bp);

    buff_t = data_buf[off];
    fld_t  = (buff_t >> (32 - rem));
    buff_n = (buff_t << rem);

    if (flag)  bp = bp_c;
    if (!flag) bp = 32 - rem;
    if (!flag) off++;
    if (!flag) cw = cw | fld_t;
    if (!flag) buff = buff_n;
    output[i] = cw;
}
*buffer = buff;
*offset = off;
*bptr = bp;
SOFTWARE PIPELINE INFORMATION

Loop source line : 32
Loop opening brace source line : 33
Loop closing brace source line : 54
Known Minimum Trip Count : 1
Known Max Trip Count Factor : 1
Loop Carried Dependency Bound(^) : 3
Unpartitioned Resource Bound : 4
Partitioned Resource Bound(*) : 5
Resource Partition:

<table>
<thead>
<tr>
<th></th>
<th>A-side</th>
<th>B-side</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L units</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>.S units</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>.D units</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>.M units</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>.X cross paths</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>.T address paths</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>Long read paths</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>Long write paths</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Logical ops (.LS)</td>
<td>2</td>
<td>2 (/.L or .S unit)</td>
</tr>
<tr>
<td>Addition ops (.LSD)</td>
<td>6</td>
<td>3 (/.L or .S or .D unit)</td>
</tr>
<tr>
<td>Bound(.L .S .LS)</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Bound(.L .S .D .LS .LSD)</td>
<td>5*</td>
<td>4</td>
</tr>
</tbody>
</table>

Searching for software pipeline schedule at ...
ii = 5 Schedule found with 3 iterations in parallel

Loop carried dependency bound has gone down, which is good. We are on the right track.
VLD: Further Optimizations

**TOP0:** 32 bits complete

**TOP1:** Partial 32 bit word

Request for codeword of length “l” can always be serviced as long as \( l \leq 32 \).

**bp**

**VLD: Further Optimizations**

**Code word (CW)**

**Buffer**

**Data_buff**

**Buffer_cpy**

**off**
LOOP:

LDW *A_len++, A_lenw
MV A_lenw, B_lenw
SUB A_32, A_lenw, A_rs
SUB B_32, A_lenw, B_rs
SHRU A_top0, A_rs, A_top0_rs
STW A_top0_rs, *A_optr++

; length of code word needed
; copy of length
; rs = 32 - lcw
; rs = 32 - lcw
; top0_rs = top0 >> rs
; Store out

SHL A_top0, A_lenw, A_top0_ls
SHRU B_top1, B_rs, B_top1_rs
OR A_top0_ls, B_top1_rs, A_top0

; Keep top0 at 32 bits
; top0_ls = top0 << lcw
; top1_rs = top1 >> rs
; top0 = top0_ls | top1_rs
; New almost 32 bit top0

SHL B_top1, B_lenw, B_top1
ADD B_p, B_lenw, B_p
SUB B_p, B_32, B_r
AND B_p, B_31, B_p
CMPGT B_r, -1, A_rw
SUB B_32, B_r, B_ru
SHRU B_buff, B_ru, B_top1_ru

; Check for overflow
; Left shift for
; Missing bits

[A_rw] OR A_top0, B_top1_ru, A_top0
[A_rw] SHL B_buff, B_r, B_top1
[A_rw] MV B_buff_cpy, B_buff
[A_rw] LDW *B_ptr++, B_buff_cpy

; Complete 32 bits
; Update top1
; Update buff
; Load buff_cpy

[A_it] SUB.1 A_it, 1, A_it
[A_it] B.1 LOOP

; Branch
**GET BITS**
**COMPLEXITY ANALYSIS**

```c
top0_rs = top0 >> lcw;
p = p + lcw;

r = p - 32; p = p & 31;
Store top0_rs

rs = r > -1

Update top0
```

4 cycle loop should be possible.
Recurrence is on top0 of length 3 cycles.
Scheduled Code for Get Bits

```
LOOP:
  SUB .D2  B_p,      B_32,    B_r      ;[ 9,2] r = p - 32
  || SHL .S1  A_top0,    A_lcw,    A_top0_ls ;[ 9,2] top0_ls = top0 << lcw
  || AND .L2  B_p,      B_31,    B_p       ;[ 9,2] p = p % 32
  || SHRU .S2  B_top1,   B_rs,     B_top1_rs ;[ 9,2] top1_rs = top1 >> rs
  || LDW .D1T1  *A_len++, A_lcw ;[ 1,4] length of code word needed
  ||| STW .D1T1  A_top0_rs,  *A_optr++ ;[14,1] Store out
  ||| SUB .D2  B_32,    B_r,     B_ru      ;[10,2] Left shift for
  ||| CMPGT .L1X B_r,     -1,      A_rw ;[10,2] Check for overflow
  ||| SHRU .S1  A_top0,    A_rs,    A_top0_rs ;[10,2] top0_rs = top0 >> rs
  ||| SHL .S2  B_top1,   B_lcw,   B_top1   ;[10,2] top1 = top1 << lcw
  ||| MV .L2X  A_lcw,    B_lcw,    B_top1 ;[ 6,3] copy of length
  ||| OR .L1X  A_top0_ls,  B_top1_rs,  A_top0 ;[11,2] top0_ls | top1_rs
  || OR .L1X  A_top0,    B_top1_ru,  A_top0 ;[11,2] Load | buff_cpy
  ||[ A_rw] LDW .D2T2  *B_ptr++,  B_buff_cpy ;[11,2] Load buff_cpy
  ||[ A_it] SUB .L1X  A_it,     1,        A_it ;[ 7,3] r = 32 - lcw
  ||[ A_it] SUB .L2X  A_32,    A_lcw,    B_rs ;[ 7,3]
  || [ A_rw] SHL .S2  B_buff,    B_r,     B_top1 ;[12,2] Update top1
  || [ A_rw] MV .D2  B_buff_cpy,  B_buff ;[12,2] Update buff
  || [ A_rw] OR .L1X  A_top0,    B_top1_ru, A_top0 ;[12,2] Complete 32 bits
  || ADD .L2  B_p,     B_lcw,    B_p       ;[ 8,3] p = p + lcw
  || SUB .S1  A_32,    A_lcw,    A_rs      ;[ 8,3] rs = 32 - lcw
  ADD .L2  B_p,     B_lcw,    B_p
  SUB .S1  A_32,    A_lcw,    A_rs
```

22 operations in 4 cycles for an average IPC of 5.5 instructions/cycle
CONVOLUTION: 3x3
LAB - 1
void conv_3x3_cn(const unsigned char *restrict inptr, unsigned char       *restrict outptr, int x_dim, const char          *restrict mask, int shift)
{
    const unsigned char     *IN1,*IN2,*IN3;
    unsigned char           *OUT;
    short    pix10,  pix20,  pix30;
    short    mask10, mask20, mask30;
    int       sum,    sum00,  sum11;
    int       i;
    int       sum22,  j;

    /* Set imgcols to the width of the image and set three pointers for reading data from the three input rows. Also set the output pointer. */
    IN1      =   inptr;
    IN2      =   IN1 + x_dim;
    IN3      =   IN2 + x_dim;
    OUT      =   outptr;
for (j = 0; j < x_dim ; j++)
{
    sum = 0;

    for (i = 0; i < 3; i++)
    {
        pix10  =   IN1[i];
        pix20  =   IN2[i];
        pix30  =   IN3[i];
        mask10 =   mask[i];
        mask20 =   mask[i + 3];
        mask30 =   mask[i + 6];

        sum00  =   pix10 * mask10;
        sum11  =   pix20 * mask20;
        sum22  =   pix30 * mask30;
        sum   +=   sum00 + sum11 + sum22;
    }

    IN1++;  IN2++;   IN3++;
    sum = (sum >> shift);

    if ( sum < 0 )       sum = 0;
    if ( sum > 255 )       sum = 255;
    *OUT++   =       sum;
}
SOFTWARE PIPELINE INFORMATION

Loop source line : 83
Loop opening brace source line : 84
Loop closing brace source line : 128
Known Minimum Trip Count : 1
Known Max Trip Count Factor : 1
Loop Carried Dependency Bound(^) : 3
Unpartitioned Resource Bound : 5
Partitioned Resource Bound(*) : 5

Resource Partition:

A-side B-side
.L units 1 1
.S units 2 0
.D units 5* 5*
.M units 5* 5*
.X cross paths 3 5*
.T address paths 5* 5*
Long read paths 0 0
Long write paths 0 0
Logical ops (.LS) 0 0 (.L or .S unit)
Addition ops (.LSD) 4 7 (.L or .S or .D unit)
Bound(.L .S .LS) 2 1
Bound(.L .S .D .LS .LSD) 4 5*

Searching for software pipeline schedule at ...
ii = 5 Did not find schedule
ii = 6 Schedule found with 6 iterations in parallel

COMPILER ANALYSIS OF NATURAL C CODE
L4: ; PIPED LOOP KERNEL

[ A0]  BDEC  .S1     L4,A0             ; |128| <0,28>  
      CMPLT  .L1     A8,0,A4           ; |120| <1,22>  ^
      ADD    .S2X    A4,B4,B5          ; |106| <2,16>
      ADD    .D1X    B23,A17,A17       ; |106| <3,10>
      LDBU   .D2T2   *++B19,B22        ; |106| <4,4>

      ADD    .S2X    A25,B5,B24         ; |106| <2,17>
      MPY    .M1     A18,A16,A9         ; |106| <3,11>
      LDBU   .D1T1   *A6++,A5           ; |106| <4,5>
      LDBU   .D2T2   *-B19(1),B20       ; |106| <4,5>
      MPY    .M2     B4,B9,B23          ; |106| <4,5>

      MV      .L2     B4,B24            ; |120| <0,30>
      ADD    .S2     B21,B24,B21        ; |106| <2,18>
      ADD    .S1X    B23,A17,A25        ; |106| <3,12>
      MPY    .M2     B22,B6,B4          ; |106| <3,12>
      MPY    .M1     A9,A19,A4          ; |106| <3,12>
      LDBU   .D1T1   *++A22,A18         ; |106| <4,6>
      LDBU   .D2T2   *+B19(2),B4       ; |106| <5,0>
Analysis of Convolution Algorithm

# of byte loads/output pixel: 18
# of multiplies/pixel: 9 (16 x 16)
# of additions: 9
# of stores/pixel: 1

Based on these numbers, and given that we can load 2 bytes/cycle, this algorithm appears to be load bottlenecked at 9 cycles/output pixel.

To overcome this, we can preload values of the 3x3 mask, which is loop invariant into register file and reduce byte loads/output pixel to 9.

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>b00</td>
<td>b01</td>
<td>b02</td>
</tr>
<tr>
<td>b10</td>
<td>b11</td>
<td>b12</td>
</tr>
<tr>
<td>b20</td>
<td>b21</td>
<td>b22</td>
</tr>
</tbody>
</table>

a) Load data for b02, b12, b22 only.
b) For other bytes pre-load and issue moves, to move the other bytes by moving the data in register file.
c) b00 = b01; b01 = b02; b10 = b11;
   b11 = b12; b20 = b21; b21 = b22;
   Load b02, Load b12, Load b22.
Analysis of Convolution Algorithm

# of byte loads/output pixel = 9

Therefore we can use double word wide loads to load eight pixels along every row. Therefore loads are not a bottleneck.

C64x can perform 8 (8x8) multiplies, and we need to perform 9 (8x8) multiplies/output pixel, therefore an optimal implementation should achieve 9/8 = 1.125 cycles/output pixel.

We can compute multiple output pixels in parallel. If we compute eight output pixels in parallel we would need 72 (8x8) multiplies which can be performed in 72/8 = 9 cycles.

Optimal implementation must compute 8 output pixels in 9 cycles to achieve a compute rate of 1.125 cycles/output pixel.
/* Set loop counter for output pixels and three input pointers x_dim apart from the user passed input pointer. Copy output pointer */

count = x_dim >> 1 ;

IN1 = intptr;
IN2 = IN1+ x_dim;
IN3 = IN2+ x_dim;
OUT = outptr;

/* In order to minimize data loads, data re-use is achieved by moves. */
/* The data to be used for pix10, pix11 are pre-loaded into pix12 and pix13 and moved within the loop. The process is repeated for rows 2 and 3 for pix20, pix21 and pix30 and pix31 respectively. */

pix12 = *IN1++;
pix13 = *IN1++;
pix22 = *IN2++;
pix23 = *IN2++;
pix32 = *IN3++;
pix33 = *IN3++;
for (j = count; j > 0; j--)
{
    pix10 = pix12; pix11 = pix13; pix12 = *IN1++; pix13 = *IN1++;
    pix20 = pix22; pix21 = pix23; pix22 = *IN2++; pix23 = *IN2++;
    pix30 = pix32; pix31 = pix33; pix32 = *IN3++; pix33 = *IN3++;

    sum00 = ((pix10*mask10) + (pix11*mask11) + (pix12*mask12));
    sum11 = ((pix20*mask20) + (pix21*mask21) + (pix22*mask22));
    sum22 = ((pix30*mask30) + (pix31*mask31) + (pix32*mask32));

    sum0 = (sum00 + sum11 + sum22) >> shift;
    if (sum0 < 0) sum0 = 0;
    if (sum0 > constant) sum0 = constant;

    *OUT++ = sum0;

    sum00 = ((pix11*mask10) + (pix12*mask11) + (pix13*mask12));
    sum11 = ((pix21*mask20) + (pix22*mask21) + (pix23*mask22));
    sum22 = ((pix31*mask30) + (pix32*mask31) + (pix33*mask32));

    sum1 = (sum00 + sum11 + sum22) >> shift;
    if (sum1 < 0) sum1 = 0;
    if (sum1 > constant) sum1 = constant;

    *OUT++ = sum1;
}

<table>
<thead>
<tr>
<th>Resource Type</th>
<th>A-side</th>
<th>B-side</th>
</tr>
</thead>
<tbody>
<tr>
<td>.L units</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>.S units</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>.D units</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>.M units</td>
<td>10</td>
<td>10</td>
</tr>
<tr>
<td>.X cross paths</td>
<td>10</td>
<td>12*</td>
</tr>
<tr>
<td>.T address paths</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>Logical ops (.LS)</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Addition ops (.LSD)</td>
<td>11</td>
<td>21</td>
</tr>
<tr>
<td>Bound(.L .S .LS)</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>Bound(.L .S .D .LS .LSD)</td>
<td>7</td>
<td>9</td>
</tr>
</tbody>
</table>

Searching for software pipeline schedule at ...
ii = 12 Schedule found with 4 iterations in parallel
18 multiplies are performed in 12 cycles to achieve a performance of 1.5 cycles/output pixel.

Remember optimal means 1.125 cycles/output pixel.

This requires the use of wider loads using double words and the 8 bit multiply instructions of the C64x.

This requires intrinsics or serial assembly.
mask00  = mask[0];  mask01  = mask[1];  mask02  = mask[2];
mask10  = mask[3];  mask11  = mask[4];  mask12  = mask[5];
mask20  = mask[6];  mask21  = mask[7];  mask22  = mask[8];

h00word  = _pack2(mask00, mask00);
h01word  = _pack2(mask01, mask01);
h02word  = _pack2(mask02, mask02);
h00word  = _pack14(h00word, h00word);
h01word  = _pack14(h01word, h01word);
h02word  = _pack14(h02word, h02word);

10_dword0  = _memd8_const(IN1);
11_dword0  = _memd8_const(IN2);
12_dword0  = _memd8_const(IN3);

IN1  += 1;
IN2  += 1;
IN3  += 1;

10_dword1  = _memd8_const(IN1);
11_dword1  = _memd8_const(IN2);
12_dword1  = _memd8_const(IN3);

IN1  += 1;
IN2  += 1;
IN3  += 1;

10_dword2  = _memd8_const(IN1);
11_dword2  = _memd8_const(IN2);
12_dword2  = _memd8_const(IN3);

IN1  += 6;
IN2  += 6;
IN3  += 6;


line00  = _lo(l0_dword0);
line01  = _hi(l0_dword0);
line10  = _lo(l1_dword0);
line11  = _hi(l1_dword0);
line20  = _lo(l2_dword0);
line21  = _hi(l2_dword0);
line02  = _lo(l0_dword1);
line03  = _hi(l0_dword1);

#if 0
line12  = _lo(l1_dword1);
line13  = _hi(l1_dword1);
#endif

line22  = _lo(l2_dword1);
line23  = _hi(l2_dword1);
line04  = _lo(l0_dword2);
line05  = _hi(l0_dword2);
line14  = _lo(l1_dword2);
line15  = _hi(l1_dword2);
line24  = _lo(l2_dword2);
line25  = _hi(l2_dword2);

#if 1
line12  = _shrmb(line11, line10);
line13  = _shlmb(line14, line15);
#endif
prodA_d0  = _mpysu4(h00word, line00);
prodA_d1  = _mpysu4(h00word, line01);
prodA_d2  = _mpysu4(h01word, line02);
prodA_d3  = _mpysu4(h01word, line03);
prodA_d4  = _mpysu4(h02word, line04);
prodA_d5  = _mpysu4(h02word, line05);

prodA0  = _lo(prodA_d0);
prodA1  = _hi(prodA_d0);
prodA2  = _lo(prodA_d1);
prodA3  = _hi(prodA_d1);
prodA4  = _lo(prodA_d2);
prodA5  = _hi(prodA_d2);
prodA6  = _lo(prodA_d3);
prodA7  = _hi(prodA_d3);
prodA8  = _lo(prodA_d4);
prodA9  = _hi(prodA_d4);
prodAA  = _lo(prodA_d5);
prodAB  = _hi(prodA_d5);
Intrinsic C Code

```c
ta_01    = _add2(prodA0,  prodA4);
tb_01    = _add2(prodB0,  prodB4);
tc_01    = _add2(prodC0,  prodC4);
sum_a01  = _add2(ta_01,   prodA8);
sum_b01  = _add2(tb_01,   prodB8);
sum_c01  = _add2(tc_01,   prodC8);
sum_t01  = _add2(sum_a01, sum_b01);
sum_o01  = _add2(sum_c01, sum_t01);

pix_01   = _shr2(sum_o01, shift);
pix_23   = _shr2(sum_o23, shift);
pix_45   = _shr2(sum_o45, shift);
pix_67   = _shr2(sum_o67, shift);

out_word0 = _spacku4(pix_23, pix_01);
out_word1 = _spacku4(pix_67, pix_45);
out_d0    = _itod(out_word1, out_word0);

_memd8(OUT) = out_d0;
OUT        += 8;
```
COMPILER ANALYSIS FOR INTRINSIC C

;*---------------------------------------------------------------*;
;* SOFTWARE PIPELINE INFORMATION                               *
;*---------------------------------------------------------------*
;*      Loop source line                                        : 284
;*      Loop opening brace source line                          : 285
;*      Loop closing brace source line                          : 514
;*      Known Minimum Trip Count                                : 1
;*      Known Max Trip Count Factor                             : 1
;*      Loop Carried Dependency Bound(^)                        : 2
;*      Unpartitioned Resource Bound                             : 9
;*      Partitioned Resource Bound(*)                           : 10
;*      Resource Partition:
;*                                A-side   B-side
;*      .L units                        0        0
;*      .S units                        5        2
;*      .D units                        3        6
;*      .M units                        9        9
;*      .X cross paths                  9        7
;*      .T address paths                9        9
;*      Long read paths                 0        0
;*      Long write paths                0        0
;*      Logical  ops (.LS)               1       1 (.L or .S unit)
;*      Addition ops (.LSD)              18      20 (.L or .S or .D unit)
;*      Bound(.L .S .LS)                 3       2
;*      Bound(.L .S .D .LS .LSD)         9      10*
;* Searching for software pipeline schedule at ...             *
;*      ii = 10 Schedule found with 4 iterations in parallelh   *
72 multiplies in 10 cycles for 8 output pixels to achieve 10/8 = 1.25 cycles/output pixel.
Convolution3x3 for a row of 480 pixels.

<table>
<thead>
<tr>
<th></th>
<th>Natural C</th>
<th>Optimized C</th>
<th>Intrinsic C</th>
<th>Serial Assembly</th>
<th>Hand Assembly</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles/Output</td>
<td>2934</td>
<td>2934</td>
<td>655</td>
<td>581</td>
<td>581</td>
</tr>
<tr>
<td>6.1 cycles/Output</td>
<td>6.1 cycles/Output</td>
<td>1.36 cycles/Output</td>
<td>1.21 cycles/Output</td>
<td>1.21 cycles/Output</td>
<td>1.21 cycles/Output</td>
</tr>
</tbody>
</table>
Lab 2: Threshold

Thrgt2max:
for (i = 0; i < pixels; i++)
out_data[i] = in_data[i] > threshold ? 255 : in_data[i];

Thrgt2thr:
for (i = 0; i < pixels; i++)
out_data[i] = in_data[i] > threshold? threshold:in_data[i];

Thrle2min:
for (i = 0; i < pixels; i++)
out_data[i] = in_data[i] <= threshold ? 0 : in_data[i];

Thrle2thr:
for (i = 0; i < pixels; i++)
out_data[i] = in_data[i] <= threshold ? threshold : in_data[i];
THRGT2MAX
for (i = 0; i < pixels; i += 4)
{
    p3p2p1p0 = _amem4_const(&in_data[i]);
    x3x2x1x0 = _xpnd4(_cmpgtu4(p3p2p1p0, thththth));
    _amem4(&out_data[i]) = p3p2p1p0 | x3x2x1x0;
}

THRGT2THR
for (i = 0; i < pixels; i += 4)
    _amem4(&out_data[i]) = _minu4(_amem4_const(&in_data[i]), thththth);

Can you guess the sequence of instructions for thrle2min and thrle2thr?

Can you analyze the complexity and throughput of these algorithms?
SYSTEM OPTIMIZATION

Balance processing, transfer times and interrupt thresholds.

C64x CPU

L1 Data 16K

Cache controller

L1 Program 16K

DMA controller

L2 256K-1M Working Set

External Memory

Image Data