



# 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.

# COURSE ABSTRACT

Software techniques that are crucial to extracting performance from DM642 DSP for video applications.

- Techniques That Will be Reviewed
- Code Optimization Techniques.
  - System Optimization Techniques.
  - 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.

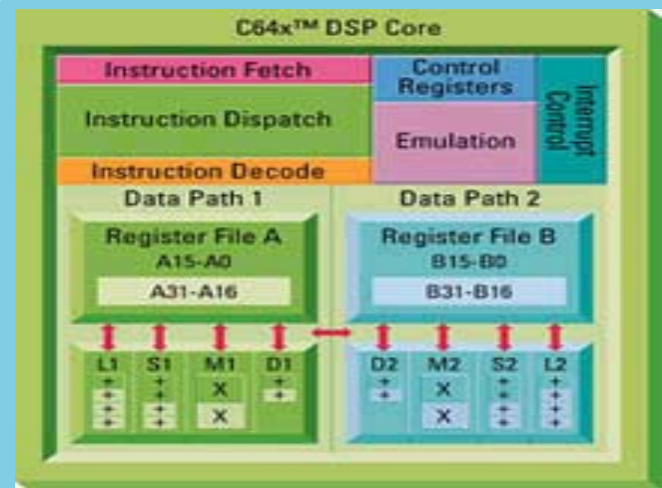
# KEY FEATURES OF TMS320C6x VLIW DSP

01010 001010000110100010010010010000110101101010010010010

## TI Developer Conference

February 18-20, 2004 • Houston, TX • Westin Galleria Hotel

Connecting Real People with Real Solutions



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

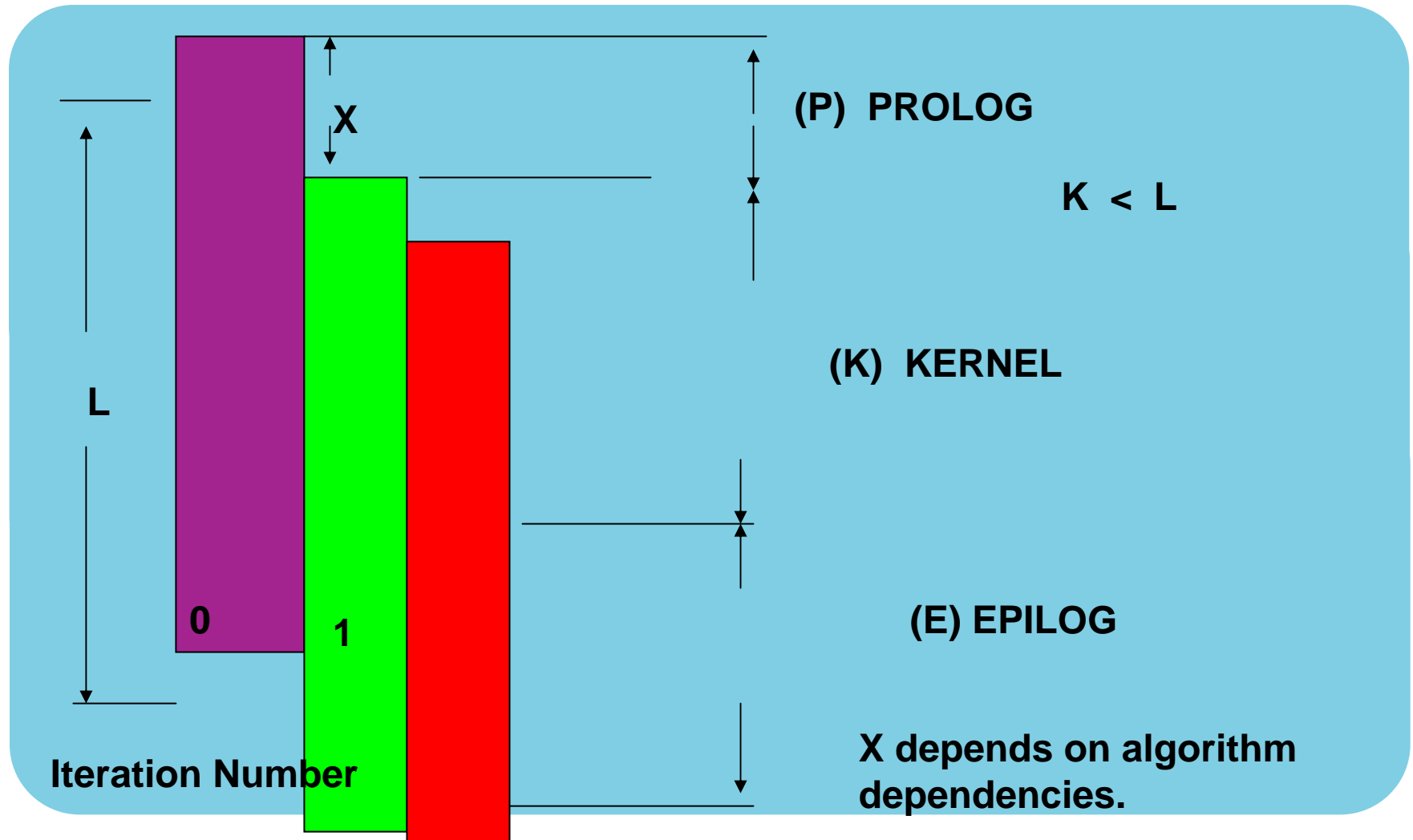
1 Execute Cycle:

```

        BDEC    .S1    LOOP,      A_i                ;[24,1]
||[!A_pd]ADD  .L2X    B_t0,      A_t0,      B_temp1    ;[24,1]
||[!A_pd]ADD  .L1    A_prodC,   A_prodD,   A_v0         ;[24,1]
||          DOTP2   .M2    B_x5x4,  B_y4y3,   B_prod6    ;[16,2]
||          DOTP2   .M1    A_x3x2,  A_y2y1,   A_prod5    ;[16,2]
||          PACKLH2 .S2    B_y7y6,  B_y5y4,   B_y6y5    ;[16,2]
||          LDDW    .D2T2  *B_xptr++[2],    B_x7x6:B_x5x4 ;[ 8,3]
||          LDDW    .D1T1  *A_xptr++[2],    A_x3x2:A_x1x0 ;[ 8,3]
    
```

# SOFTWARE PIPELINING

Connecting Real People with Real Solutions





## 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.

## Code Optimization Methodology

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.



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:

- Improvements to the compiler and code generation tools.
- Can be re-scheduled to avoid new architecture restrictions. eg. C64x cross path stall can be avoided in C62x code by re-compiling.
- 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.

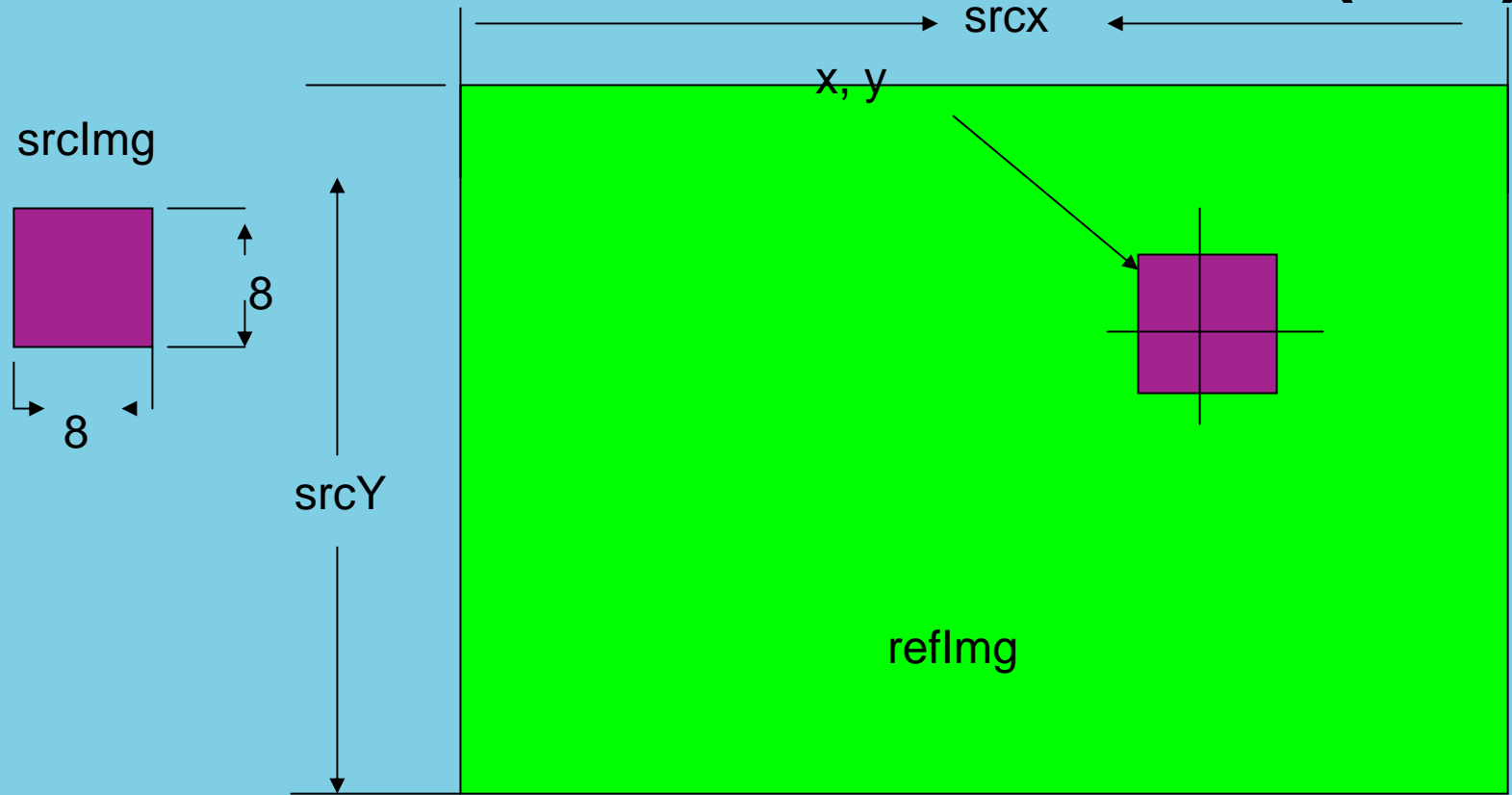
## Image/Video Processing Lab Examples on DM642

- 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.

# Minimum Absolute Difference (8x8)



## Natural C Code for mad\_8x8

Connecting Real People with Real Solutions



```

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

Connecting Real People with Real Solutions

Number of Byte loads required:  $128 * s_x * s_y$ .  
Number of Absolute and differences required:  $64 * s_x * s_y$ .  
Number of additions required:  $64 * s_x * s_y$ .  
Number of compares:  $s_x * s_y$ .

## Observations

- 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.
- 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 ???

## Analysis Of Optimal Implementation of mad\_8x8 for C64x DSP

Connecting Real People with Real Solutions



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.

```
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; }
}
```

# TI Developer Conference

February 18-20, 2004 • Houston, TX • Westin Galleria Hotel

## Connecting Real People with Real Solutions



```

***** 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_hv1, 1, B_hv1
[B_bst] MV B_mad, B_best
[B_bst] SUB B_hv1, 1, B_bhv1
[A_i] BDEC loop, A_i

```



```
mad8x8_loop:
```

```

      ADD     .D2      B_hv1,      1,      B_hv1      ;[25,1]
      DOTPU4  .M1      A_err01,    A_k_one,  A_row_0     ;[17,2]
      ADD     .S2      B_row_2,    B_mad_3,   B_mad_2     ;[17,2]
      DOTPU4  .M2      B_err3h,    B_k_one,   B_row_3     ;[ 9,3]
      SUBABS4 .L1      A_src51,    A_ref51,   A_err51     ;[ 9,3]
      SUBABS4 .L2X    B_src5h,    A_ref5h,   B_err5h     ;[ 9,3]
      LDNDW   .D       *A_ref_d(A_p7),  A_ref7h:A_ref71 ;[ 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     .S2      B_row_1,    B_mad_2,   B_mad_1     ;[18,2]
      ADD     .S1      A_row_3,    A_mad_4,   A_mad_3     ;[18,2]
      SUBABS4 .L1X    A_src41,    B_ref41,   A_err41     ;[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_err61,    A_k_one,   A_row_6     ;[10,3]
      LDNDW   .D       *A_ref_d(A_p6),  B_ref6h:B_ref61 ;[ 2,4]

```

```
loop_2:
```

```

[A_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_src31,    A_ref31,   A_err31     ;[11,3]
      DOTPU4  .M2      B_err4h,    B_k_one,   B_row_4     ;[11,3]
      DOTPU4  .M1      A_err51,    A_k_one,   A_row_5     ;[11,3]
      LDNDW   .D       *A_ref_d(A_p3),  A_ref3h:A_ref31 ;[ 3,4]
      SUB     .D2      B_v1,      1,      B_v1       ;[ 3,4]

```



loop\_3:

```

    CMPGT2 .S2      B_best,      B_mad,      B_bst      ;[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,     A_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_hv1,      1,          B_bhv1     ;[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     B_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]
    SUBABS4 .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]

```

## Connecting Real People with Real Solutions



```

ADD      .D2X      A_row_1,      B_mad_1,      B_mad_1      ; [23,2]
DOTPU4   .M1       A_err11,      A_k_one,      A_row_1      ; [15,3]
ADD      .S2       B_row_4,      B_mad_5,      B_mad_4      ; [15,3]
ADD      .S1       A_row_5,      A_mad_6,      A_mad_5      ; [15,3]
[A_i]LDNDW .D      *A_ref_d(A_p1),      A_ref1h:A_ref1l ; [ 7,4]
[A_i]SUBABS4 .L2    B_src6h,      B_ref6h,      B_err6h      ; [ 7,4]
[A_i]DOTPU4 .M2    B_err7h,      B_k_one,      B_row_7      ; [ 7,4]
[A_i]SUBABS4 .L1    A_src7l,      A_ref7l,      A_err7l      ; [ 7,4]

ADD      .S2       B_row_0,      B_mad_1,      B_mad_0      ; [24,2]
ADD      .D2      B_row_3,      B_mad_4,      B_mad_3      ; [16,3]
ADD      .S1       A_row_4,      A_mad_5,      A_mad_4      ; [16,3]
[A_i]LDNDW .D      *A_ref_d++(A_f),      B_ref0h:B_ref0l ; [ 8,4]
[A_i]SUBABS4 .L2X   B_src3h,      A_ref3h,      B_err3h      ; [ 8,4]
[A_i]DOTPU4 .M2    B_err6h,      B_k_one,      B_row_6      ; [ 8,4]
[A_i]SUBABS4 .L1X   A_src6l,      B_ref6l,      A_err6l      ; [ 8,4]
[A_i]DOTPU4 .M1    A_err7l,      A_k_one,      A_row_7      ; [ 8,4]
    
```

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.

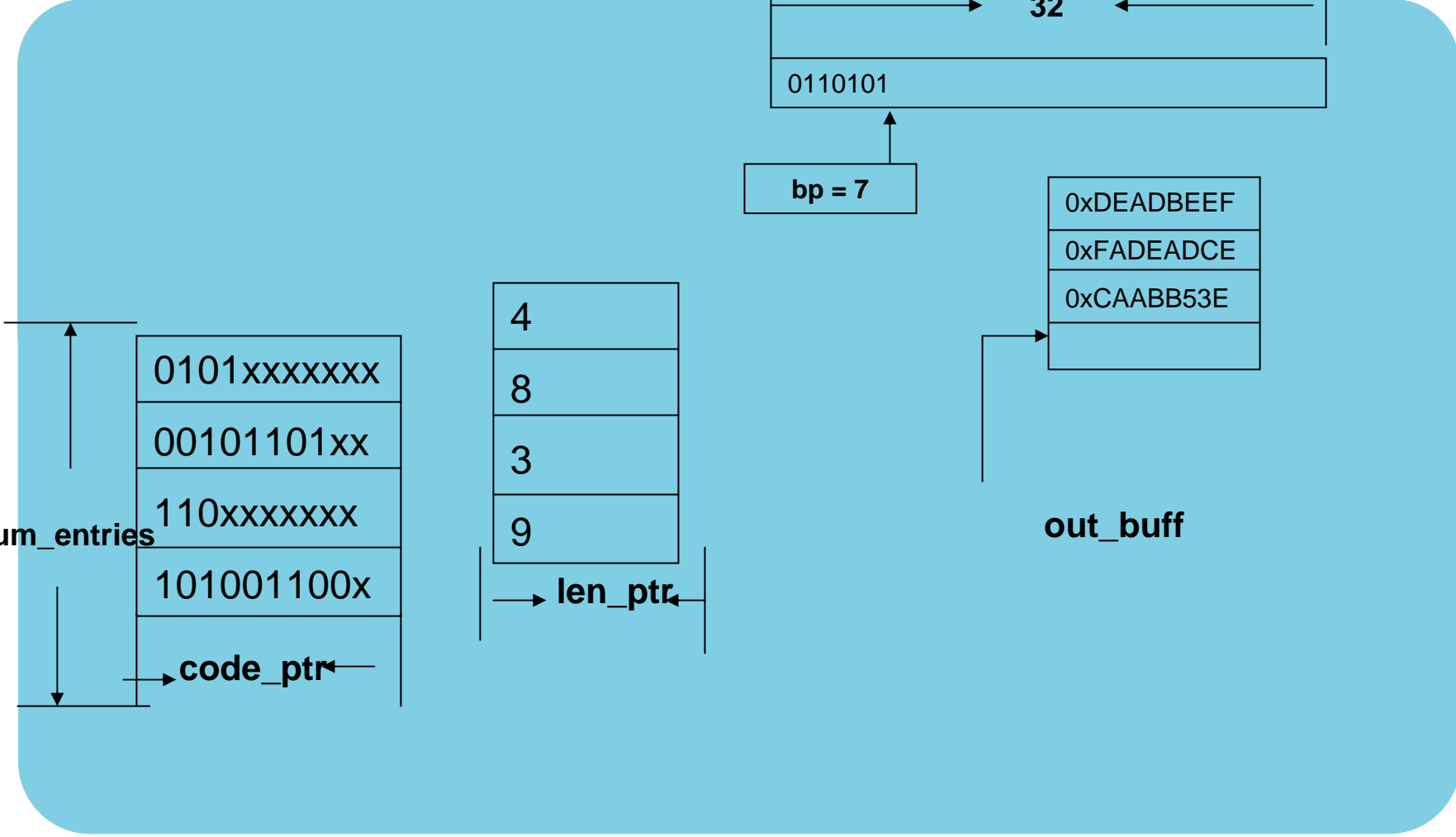


## MAD\_8x8 Performance Summary for (64x32)

CN	CO	Intrin.	Serial Assem.	Part. Serial Assem.	Hand Assem.
22254 cycles	22254 cycles	22254 cycles	20561 cycles	16454 cycles	16458 cycles
1280 bytes	1280 bytes	1248 bytes	1056 bytes	1280 bytes	800 bytes
					7.96 MAD/cycle

# Put Bits, Variable Length Encoding

Connecting Real People with Real Solutions





# Put Bits, Variable Length Encoding

Connecting Real People with Real Solutions

```

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

```

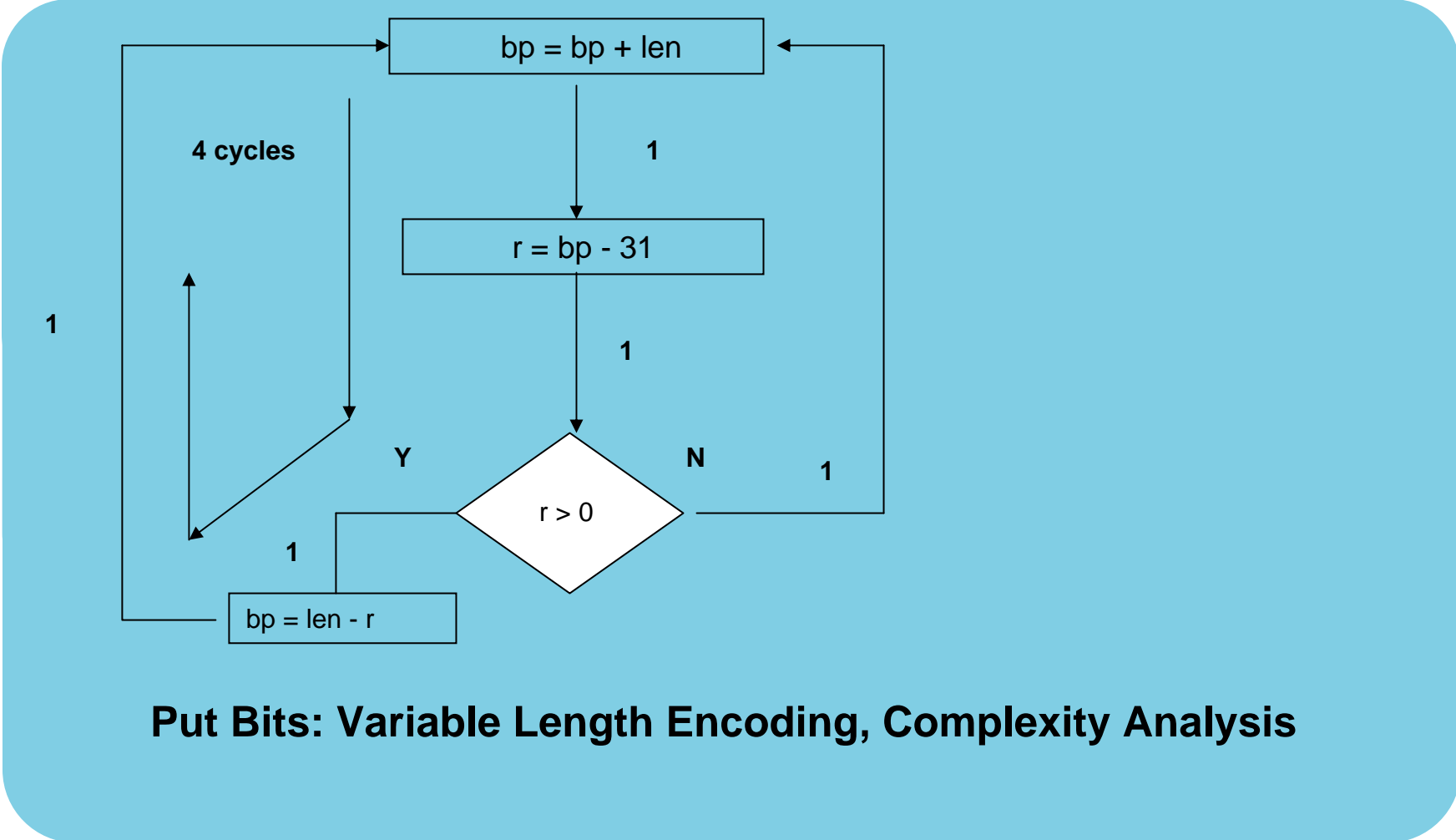
|| [ A0] OR   .S2   B8,B6,B6      ; |30| <0,9>
|| [ A1] MV   .L2X  A5,B7        ; |38| <0,9> Define a twin register
|| [ A0] OR   .D1X  B8,A3,A3     ; |30| <0,9> Define a twin register
||   ROTL   .M1   A8,0,A0       ; |27| <1,5> Split a long life
||   ADD    .S1   A8,A4,A4       ; |31| <1,5> ^
||   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> ^

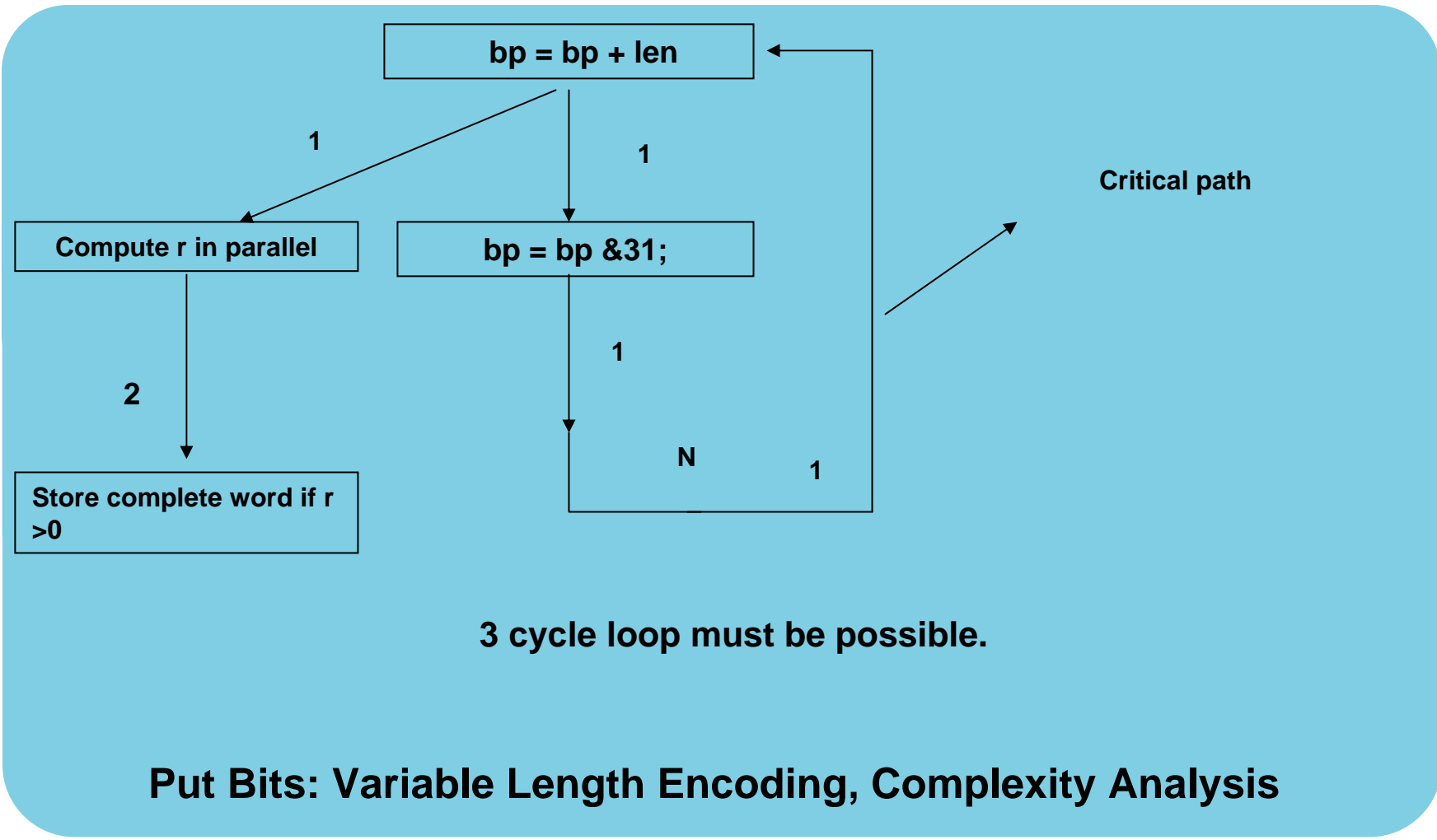
||   ROTL   .M1   A2,0,A1       ; <1,7> Split a long life
|| [ B0] BDEC  .S2   L2,B0       ; |40| <1,7>
||   SUB    .D1   A0,A2,A5      ; |37| <1,7> ^

|| [ A1] MV   .D2X  A3,B6        ; |37| <0,12> Define a twin register
|| [ A2] MV   .S1   A5,A4        ; |38| <1,8> ^
||   LDW    .D1T1 *A7++,A8      ; |27| <3,0>

```



### Put Bits: Variable Length Encoding, Complexity Analysis



# Optimized Put Bits Routine

```

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, cwl, r;

    for( i = 0; i < num_entries; i++)
    {
        code = code_ptr[i];
        len  = len_ptr[i];

        cws  = (code >> bp);
        cwl  = (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    = cwl;
        }
    }
}

```

## 3 cycle loop for put bits.



```

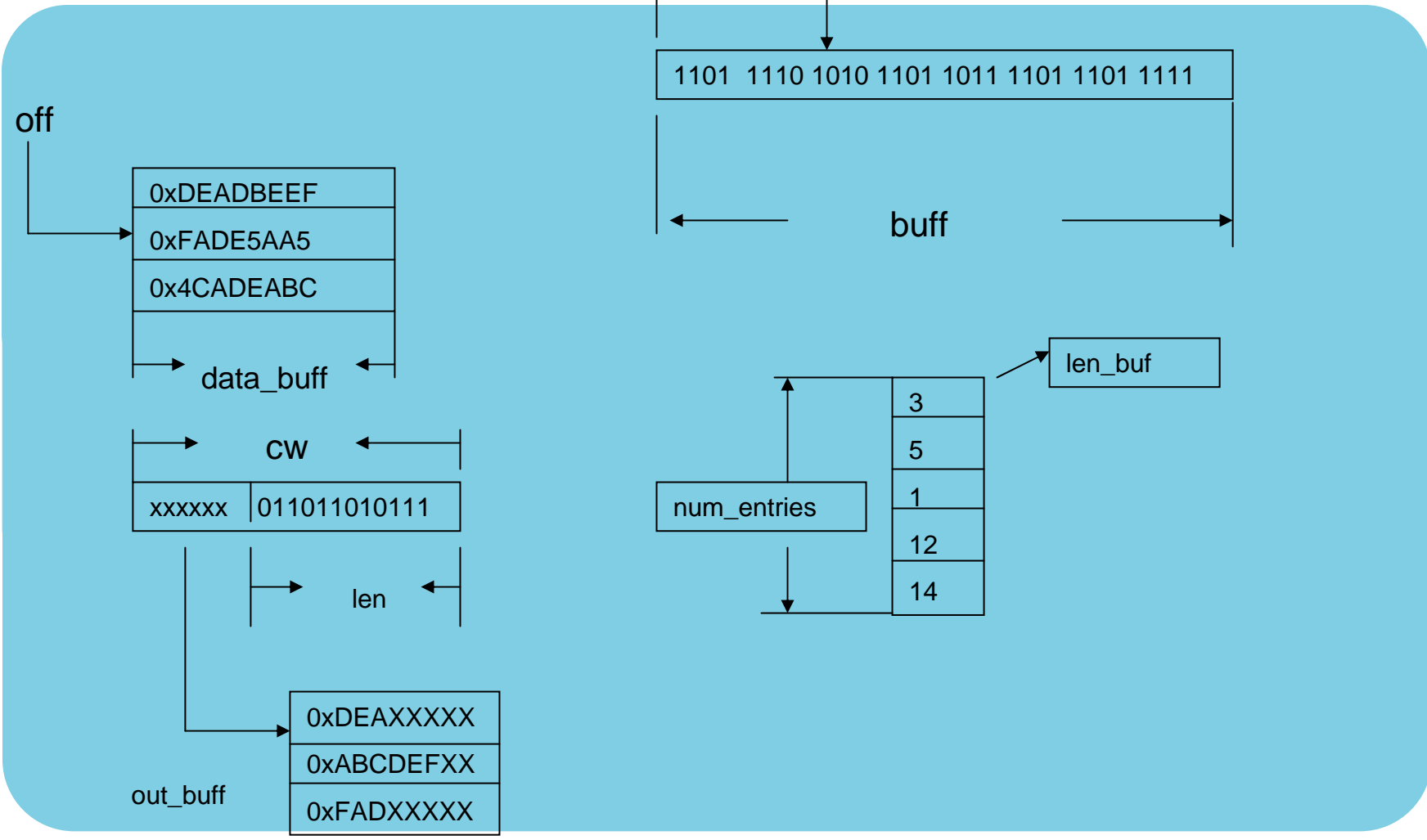
L2:      ; PIPED LOOP KERNEL

      SHL      .S1X      B16, A6, A6      ; 30  <0,9>
      MVD      .M1       A0, A1           ; 27  <1,6> Split a long life
[ B0]  BDEC     .S2       L2, B0          ; 42  <1,6>
      LDW      .D2T2     *B7++, B8       ; 30  <3,0>

      ADD      .D2       B4, B9, B1      ; 34  <0,10>
[ A1]  OR       .S1X     B5, A5, A5      ; 32  <0,10> ^
      ADD      .L2X     A0, B6, B9      ; 33  <1,7> ^
[ A0]  SHRU    .S2       B16, B6, B5     ; 32  <1,7>
      LDW      .D1T1     *A7++, A0      ; 27  <3,1>

[ B1]  MV       .L1      A6, A5          ; 40  <0,11> ^
[ B1]  STB      .D1T1    A5, *A4++      ; 39  <0,11> ^
      EXTU     .S2       B9, 27, 27, B6 ; 35  <1,8> ^
      SUB      .S1X     A3, B6, A6      ; 30  <1,8> ^
      ROTL     .M2      B8, 0, B16     ; 30  <2,5> Split a long life
    
```

# Get Bits, Inverse of Put Bits



## Get Bits: Function to Mimic Variable Length Decoding

```
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;
```



## Get Bits: Function to Mimic Variable Length Decoding

```
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;
}
```

```

; *-----*
; * 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> ^

[!B1] 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>

```

# Optimized C Code for get bits/VLD

Connecting Real People with Real Solutions

```
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;
}
```

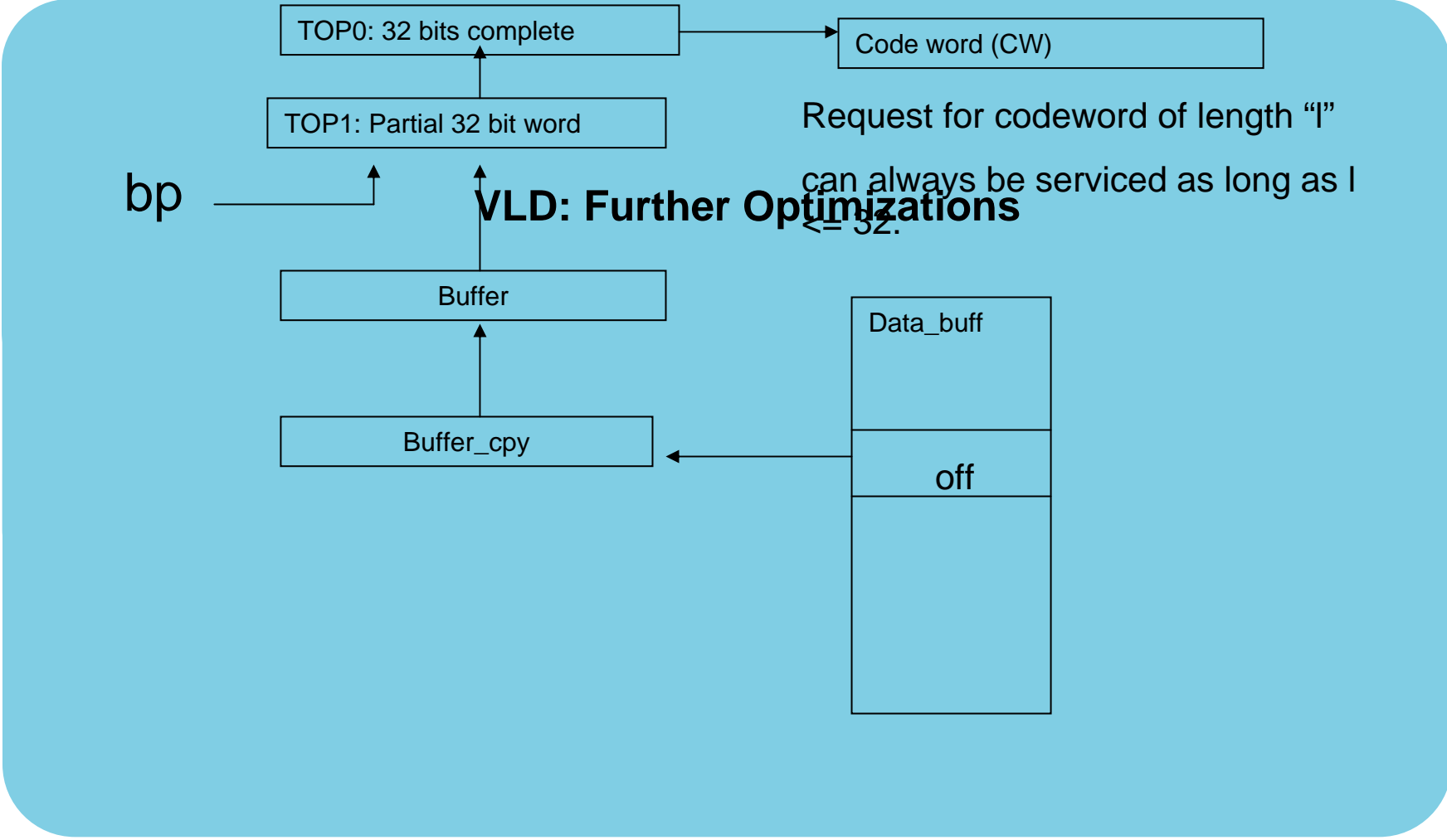


```

; *-----*
; *   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:
; *
; *           A-side   B-side
; *   .L units           0         2
; *   .S units           3         2
; *   .D units           2         1
; *   .M units           0         0
; *   .X cross paths     3         4
; *   .T address paths   2         1
; *   Long read paths    1         0
; *   Long write paths   0         0
; *   Logical ops (.LS)  2         2      (.L or .S unit)
; *   Addition ops (.LSD) 6         3      (.L or .S or .D unit)
; *   Bound(.L .S .LS)   3         3
; *   Bound(.L .S .D .LS .LSD) 5*      4
; *
; *   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.





LOOP:

```

LDW  *A_len++,    A_lcw           ; length of code word needed
MV   A_lcw,      B_lcw           ; copy of length

SUB   A_32,      A_lcw,          A_rs       ; rs = 32 - lcw
SUB   B_32,      A_lcw,          B_rs       ; rs = 32 - lcw
SHRU A_top0,     A_rs,           A_top0_rs  ; top0_rs = top0 >> rs
STW  A_top0_rs,  *A_optr++       ; Store out

                                           ; Keep top0 at 32 bits
SHL  A_top0,     A_lcw,          A_top0_ls  ; top0_ls = top0 << lcw
SHRU B_top1,     B_rs,           B_top1_rs  ; top1_rs = top1 >> rs
OR   A_top0_ls,  B_top1_rs,     A_top0      ; top0      = top0_ls | top1_rs
                                           ; New almost 32 bit top0

SHL  B_top1,     B_lcw,          B_top1     ; top1  = top1 << lcw
ADD  B_p,        B_lcw,          B_p        ; p = p + lcw
SUB  B_p,        B_32,           B_r        ; r = p - 32
AND  B_p,        B_31,           B_p        ; p = p % 32

CMPGT B_r,       -1,             A_rw       ; Check for overflow
SUB   B_32,      B_r,            B_ru       ; Left shift for
SHRU B_buff,     B_ru,           B_top1_ru  ; Missing bits

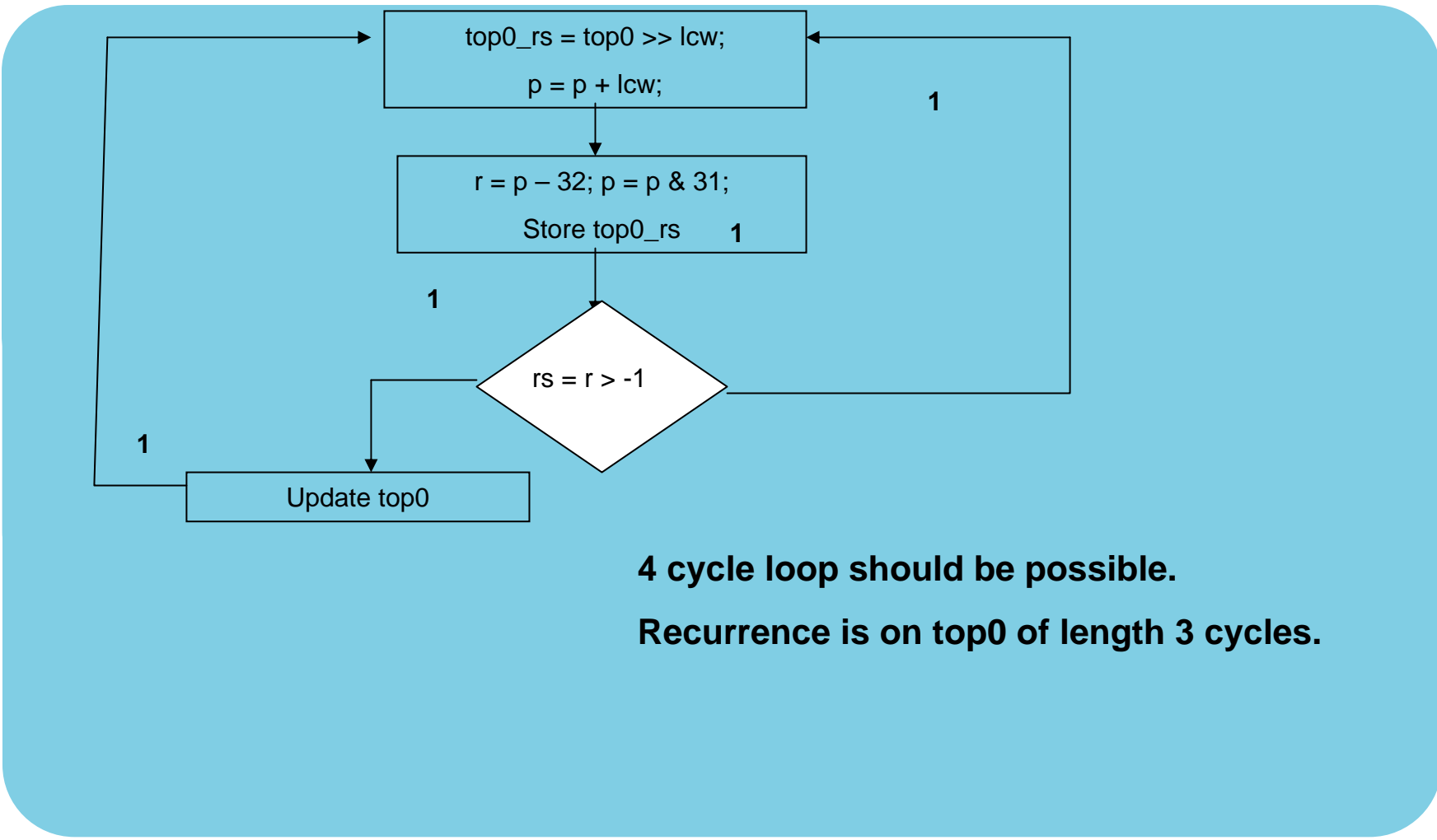
[A_rw]OR  A_top0,    B_top1_ru,    A_top0    ; Complete 32 bits
[A_rw]SHL B_buff,    B_r,          B_top1    ; Update top1
[A_rw]MV  B_buff_cpy, B_buff      ; Update buff
[A_rw]LDW *B_ptr++,  B_buff_cpy    ; Load  buff_cpy

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

```

## GET BITS COMPLEXITY ANALYSIS

Connecting Real People with Real Solutions







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          ;[ 6,3] copy of length

OR       .L1X     A_top0_ls,    B_top1_rs,     A_top0        ;[11,2] top0_ls | top1_rs
[ A_rw]LDW .D2T2  *B_ptr++,    B_buff_cpy     ;[11,2] Load buff_cpy
[ A_it]B   .S1      LOOP                    ;[11,2] Branch
SHRU     .S2      B_buff,      B_ru,          B_top1_ru     ;[11,2] Missing bits
SUB      .L2X     B_32,          A_lcw,          B_rs          ;[ 7,3] rs = 32 - lcw
[ A_it]SUB .D1      A_it,        1,          A_it          ;[ 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
    
```

### Scheduled Code for Get Bits;

22 operations in 4 cycles for an average IPC of 5.5 instructions/cycle



# CONVOLUTION: 3x3 LAB - 1

## CONVOLUTION 3x3

Connecting Real People with Real Solutions

```

/* ===== */
/* conv_3x3_cn  -- Natural C version of conv_3x3().  */
/* ===== */

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. Alos set the output poin- */
    /* ter. */
    /*-----*/

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

```

## CONVOLUTION 3x3, Natural C Code

Connecting Real People with Real Solutions

```

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

Connecting Real People with Real Solutions

# 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.

b00	b01	b02
b10	b11	b12
b20	b21	b22

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

# Analysis of Convolution Algorithm

Connecting Real People with Real Solutions



# 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.





# Optimized C Code

```

/*-----*/
/* 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                  =   inptr;
IN2                  =   IN1+ x_dim;
IN3                  =   IN2+ x_dim;
OUT                  =   outptr;

/*-----*/
/* In order to minimize data loads, dat 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++;

```

# Optimized C Code

Connecting Real People with Real Solutions



```
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;
```

```
}
```

```
}
```

# COMPILER REPORT FOR OPTIMIZED C CODE

Connecting Real People with Real Solutions



```

; *-----*
; * SOFTWARE PIPELINE INFORMATION
; *
; * Loop source line : 200
; * Loop opening brace source line : 201
; * Loop closing brace source line : 263
; * Known Minimum Trip Count : 1
; * Known Max Trip Count Factor : 1
; * Loop Carried Dependency Bound(^) : 3
; * Unpartitioned Resource Bound : 10
; * Partitioned Resource Bound(*) : 12
; * Resource Partition:
; *
; *           A-side   B-side
; * .L units           3       1
; * .S units           3       0
; * .D units           4       4
; * .M units          10      10
; * .X cross paths     10     12*
; * .T address paths   3       5
; * Long read paths    0       0
; * Long write paths   0       0
; * Logical ops (.LS)   0       0       (.L or .S unit)
; * Addition ops (.LSD) 11      21      (.L or .S or .D unit)
; * Bound(.L .S .LS)   3       1
; * Bound(.L .S .D .LS .LSD) 7       9
; *
; * 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.

## INTRINSIC C CODE

Connecting Real People with Real Solutions

```

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;

```

# INTRINSIC C CODE

Connecting Real People with Real Solutions

```
line00 = _lo(10_dword0);
line01 = _hi(10_dword0);
line10 = _lo(11_dword0);
line11 = _hi(11_dword0);
line20 = _lo(12_dword0);
line21 = _hi(12_dword0);

line02 = _lo(10_dword1);
line03 = _hi(10_dword1);

#if 0
line12 = _lo(11_dword1);
line13 = _hi(11_dword1);
#endif

line22 = _lo(12_dword1);
line23 = _hi(12_dword1);

line04 = _lo(10_dword2);
line05 = _hi(10_dword2);
line14 = _lo(11_dword2);
line15 = _hi(11_dword2);
line24 = _lo(12_dword2);
line25 = _hi(12_dword2);

#if 1
line12 = _shrmb(line11, line10);
line13 = _shlmb(line14, line15);
#endif
```

# INTRINSIC C CODE

Connecting Real People with Real Solutions

```
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

```
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

Connecting Real People with Real Solutions



```

; *-----*
; * 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.

## Convolution 3x3 for a row of 480 pixels.

Connecting Real People with Real Solutions




Natural C	Optimized C	Intrinsic C	Serial Assembly	Hand Assembly
2934	2934	655	581	581
6.1 cycles/output	6.1 cycles/output	1.36 cycles/output	1.21 cycles/output	1.21 cycles/output

# Lab 2: Threshold

Connecting Real People with Real Solutions



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];
```



# C64x SIMD Instructions

Connecting Real People with Real Solutions



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.

