Stretch™

Color Space Conversion

Application Note

Version 1.1

Confidential & Proprietary

Last modified: 01/20/2005

This preliminary publication is provided "AS IS." Stretch, Inc. (hereafter "Stretch") DOES NOT MAKE ANY WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF TITLE, NONINFRINGEMENT, MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Information in this document is provided solely to enable system and software developers to use Stretch S5000 processors. Unless specifically set forth herein, there are no express or implied patent, copyright or any other intellectual property rights or licenses granted hereunder. Stretch does not warrant that the contents of this publication, whether individually or as one or more groups, meets your requirements or that the publication is error-free. This publication could include technical inaccuracies or typographical errors. Changes may be made to the information herein, and these changes may be incorporated in new editions of this publication.

Part #: AN-0000-0001-001

Contents

This page intentionally left mostly blank.

Chapter 1 Color Space Conversion

The six basic development steps for implementing any application on the S5 Engine are:

1. Define the application.
2. Write the application in C/C++.
3. Compile and link the application source files using the Stretch C Compiler (scc).
4. Run and profile the application.
5. Determine which parts of the application need to be accelerated based on the profile data.
6. Rewrite the computationally intensive code (hot spots) for the S5 Engine's Instruction Set Extension Fabric (ISEF), and repeat steps 2 through 5 until you are satisfied with the application's computation performance.

This application note shows you how to use steps 2 through 6 to implement a Color Space Conversion application on the S5 Engine.

1.1 Color Space Conversion Application

The color space conversion algorithm is used for converting video data from one color space (RGB) to another color space (YCbCr) or vice-versa.

The basic equations to convert between 8-bit digital RGB data and 8-bit YCbCr are

$$Y = 0.299R + 0.587G + 0.114B$$
$$Cb = -0.169R - 0.331G + 0.500B + 128$$
$$Cr = 0.500R - 0.419G - 0.082B + 128$$

These equation can be implemented in fixed-point arithmetic as follows:

$$Y = (77R + 150G + 29B) \gg 8$$
$$Cb = (-43R - 85G + 128B + 32768) \gg 8$$
$$Cr = (128R - 107G - 21B + 32768) \gg 8$$

As seen in the preceding equations, the computations required for implementation of such an algorithm are composed of multiplication, addition, and shift operations, all of which are well-suited for the S5 Engine.

1.2 Color Space Conversion Implementation in C

The preceding fixed-point equations can be implemented in C as follows:

```
void rgb2ycc ( char r, char g, char b, char *y, char *cb,
               char *cr )
{
   *y  = ( 77*r + 150*g + 29*b         ) >> 8;
   *cb = (-43*r -  85*g + 128*b + 32768) >> 8;
   *cr = (128*r - 107*g - 21*b  + 32768) >> 8;
}
```

As observed in the preceding implementation, the `rgb2ycc` module requires one set of RGB samples (3 bytes) to generate one set of YCbCr samples (3 bytes). In the preceding code, the arguments `r`, `g`, and `b` represent the input data, and the three pointers, `*y`, `*cb`, and `*cr` are the addresses of the respective `y`, `cb`, and `cr` locations. This module can be executed in a loop for a long set of RGB data as follows:

```
void rgb2ycc_wrapper(int NP, char *RGB, char *YCC)
{
   int i;
   for (i = 0; i < 3 * NP; i += 3) {
   rgb2ycc(RGB[i], RGB[i+1], RGB[i+2], &YCC[i],
          &YCC[i+1], &YCC[i+2]);
   }
}
```

The preceding wrapper function, loops 3 * NP times, which is the total number of RGB inputs, and generate 3 * NP YCbCr outputs. NP represents the total number of 3-byte sets of RGB and YCbCr components. For each set of RGB data, we call the `rgb2ycc` function. Thus, we will make a total of NP calls to the `rgb2ycc` function.

For the complete source listing of the application, refer to Appendix A. In addition, the file referenced here and the Makefile are available in the `\Examples\kernels` distribution directory. You can use these to compile, run, and profile the Color Space Conversion application.

1.3 Analyze Profile Output

After compiling, linking, running, and profiling the C implementation you can analyze the output from the profiler to find the hot spots in your code. The output from the profiler produces a summary of performance statistics collected by the simulator during execution. Table 1-1 is an excerpt of these performance statistics for the C implementation of the Color Space Conversion application showing those functions that use the highest percentage of cycles.

Table 1-1 Excerpt of performance statistics for C implementation

%	Cumulative cycles	Self cycles	Number of calls	Self cycles/call	Total cycles/call	Function Name
77.34	63348.00	63348.00				ResetH
7.25	69286.00	5938.00	160	37.11	37.11	rgb2ycc
5.00	73384.00	4098.00	1	4098.00	17257.63	main
4.10	76739.00	3355.00	1	3355.00	9293.00	rgb2ycc_wrapper
1.34	77834.00	1095.00	1	1095.00	3308.63	_vfprintf_r
0.73	78429.00	595.00	1	595.00	778.50	_malloc_r
0.43	78783.00	354.00	2	177.00	355.06	__sfvwrite

As you can see in Table 1-1, the `rgb2ycc` function is called 160 times and takes 7.25% of the total cycles. Also, observe that the `rgb2ycc_wrapper` function that calls the `rgb2ycc` module takes 9293 cycles. Of these cycles, the `rgb2ycc` function contributes 5938 cycles. Thus, we see that the `rgb2ycc` function is an excellent candidate for execution on the ISEF. By optimizing the execution of the `rgb2ycc` function, we inherently reduce the cycles required by the `rgb2ycc_wrapper` function.

1.4 Rewrite the Hot Spot for the ISEF

On the S5 Engine, there is a lot of flexibility that allows us to exploit the pattern of computations for a given algorithm and accordingly map them to Extension Instructions to achieve maximum performance gain. But before we jump into this analysis, following are a few important features of the S5 Engine to remember when writing programs for it:

- All data access to and from the ISEFs are through register files located in the Extension Unit. There are two banks of register files (A and B). Each bank

contains 16 registers, and each register is 128 bits wide. Thus, in total we have 32 128-bit wide registers (WRs).

- Each ISEF has three read and two write ports. At any given time, each can perform a maximum of three 128-bit reads (384-bits) and two 128-bit writes (256-bits).
- The Stretch processor and the ISEF run off the same clock source. The timing between them is skewed by the issue rate (see the *SCP Architecture Reference* for details on issue rate). What this means is that we can perform 128-bit registers loads and stores in between consecutive invocations of the Extension Instruction without under-utilizing the ISEF. In addition, because Extension Instruction execution is pipelined, there is no need to wait for an Extension Instruction to finish before the next Extension Instruction is issued. Thus, if an Extension Instruction does not require the output generated by the previous instruction, Extension Instructions can be issued at the issue rate. Eventually, however, some processor cycles will be spent waiting for the final results from the ISEF execution, which depends on the ISEF latency times the issue rate.
- The Stretch processor can read or write to or from the register files for memory-to-ISEF data transfers. Stretch processor instructions include support for 8-, 16-, 32-, 64-, and 128-bit loads and stores for the WRs with immediate, indexed, circular, or bit-reversed addressing. There is also support for bit and byte puts and gets to or from the WRs, and a 128-bit move instruction between WR registers.

  For anything less than 128-bit loads or stores, the upper bits are zero-padded.
- As noted in the Section 1.4.8, "The SCP Instruction Set Extension Fabric Capability and Capacity" of the *SCP Architecture Manual*, an Extension Instruction using many AU or MU elements leaves fewer ISEF resources for other Extension Instructions in the same ISEF configuration.

We can now show how to leverage the flexibility provided by the ISEF in creating instruction that maximize the performance of the Color Space Conversion application.

1.5 Creating Extension Instruction

As we saw in the profile information, the `rgb2ycc` function is called 160 times and takes about 50% of the total cycles. Thus, it is an appropriate candidate for us to represent it as an Extension Instruction.

The ISEF can have three 128-bit Wide Registers as input operands and two 128-bit Wide Registers as output operands, so typically we would try to pack as many input data into the Wide Register as can be handled by the ISEF. Thus, the instruction would typically operate on several inputs and generate several outputs per invocation. In defining the instruction we either push the ISEF's input–output limit or the compute limit.

Also, the Extension Instruction is represented in C; thus creating an Extension Instruction for the `rgb2ycc` function is straightforward. We include the C implementation of the `rgb2ycc` function as the body of the instruction. Wrapped around it is the prolog and the epilog. The prolog typically consists of extracting data from the Wide Registers, which act as input operands for the instruction. The epilog consists of packing the outputs generated by the instruction into the Wide Register, which acts as an output operand. The instruction is defined using a unique `typedef` that tells the Stretch compiler to compile the function as an Extension Instruction. The arguments to this function are representative of the input–output operands for the instruction. In addition, the Extension Instruction needs to reside in a separate file with a `.xc` extension.

Figure 1-1 shows the ISEF function (`rgb2ycc()`) that defines the Extension Instruction for the inner loop RGB to YCC conversion computations. `rgb2ycc()` is defined in the Stretch C file `rgb2ycc.xc`. Table 1-2 explains what each part of the code does.

Figure 1-1 ISEF function rgb2ycc()

```
A ——— #include <stretch.h>

      /* Extension instruction converting 5 pixels */
B ——— SE_FUNC void rgb2ycc(WR A, WR *B)
      {                                              D
C ———   se_sint<8> r[5], g[5], b[5];
        se_sint<8> y[5], cb[5], cr[5];
        int i, j;

        /* unpack A to RGB data, does not use any ISEF logic */
        for (i = 0; i < 5; i++) {
          j = i * 3 * 8;
          r[i] = A(j+7, j);
          g[i] = A(j+15, j+8);
          b[i] = A(j+23, j+16);
        }

        /* converting 5 pixels */
E ——— for (i = 0; i < 5; i++) {
          y[i]  = ( 77*r[i] + 150*g[i] +  29*b[i]        ) >> 8;
          cb[i] = (-43*r[i] -  85*g[i] + 128*b[i] + 32768) >> 8;
          cr[i] = (128*r[i] - 107*g[i] -  21*b[i] + 32768) >> 8;
        }

        /* pack YCbCr to B */
F ——— *B = (cr[4],cb[4],y[4],
            cr[3],cb[3],y[3],
            cr[2],cb[2],y[2],
            cr[1],cb[1],y[1],
            cr[0],cb[0],y[0]);
      }
```

Table 1-2 Components of the rgb2ycc() function

A This header file defines arbitrary-sized integer types and the usual arithmetic operations on them. It must be included in all source files that use Stretch-defined declarations and data types.

B The ISEF function is declared as type `SE_FUNC void` and named `rgb2ycc`. All ISEF functions must be declared as type `SE_FUNC void`.

C Stretch C lets you define arbitrary-width data types using the `se_sint<n>` declaration. The term `se_sint<>` is used for signed quantities, the term `se_uint<n>` is used for unsigned quantities, and `<n>` defines the data type's width. Because the width of these data types is arbitrary, `<n>` can be any value required for your application, within the physical limitation of the ISEF, of course.

D The Extension Instruction `rgb2ycc` is defined to have two arguments: A and B. They are declared as type `WR`, which means that they are wide registers. In this example, A is input and B is an output. The asterisk(*) before B means that B is an output. In some cases, we can also use B as an input (that is, in–out). In such cases, we would still declare B in this fashion if the result were to be written out using B.

E Computation loop. Because the inputs to the conversion of the five pixels are independent, the Stretch C compiler (scc) recognizes this and executes the five iterations in parallel.

F The values are packed together and written to the output wide register.

The wrapper function that invokes this instruction is modified as follows:

Figure 1-2 Using the Extension Instruction

```
A ———— #include "rgb2ycc.h"

       void
       rgb2ycc_wrapper(signed char *RGB, signed char *ycc)
       {
B ———— WR A, B;
       int i;

       WRGET0INIT(0, RGB);     /* initialize input stream from RGB */
C ———— WRPUTINIT(0, ycc);      /* initialize output stream to ycc */

       /* loop over RGB data, converting 5 pixels at a time */
       for (i = 0; i < NP/5; i++) {

D ———————— WRGET0I(&A, 15);      /* load 5 RGB pixels to A */

E ———————— rgb2ycc(A, &B);       /* convert 5 pixels */

F ———————— WRPUTI(B, 15);        /* store 5 YCbCr pixels from B */
       }

G ———— WRPUTFLUSH();           /* flush output stream */
       }
```

Table 1-3 Code components

A This is the generated header file from the compilation of `rgb2ycc.xc`

B Wide Register allocations for the Input and Output samples.

Table 1-3 Code components

C The `WRGET0INIT()` intrinsic is defined to be the `WRGET0INIT` instruction. We initialize the input byte stream mechanism with the memory address `RGB`, with automatic incrementing adresses.
The `WRPUTINIT()` intrinsic is defined to be the `WRPUTINIT` instruction. We initialize the output byte stream mechanism with the memory address `ycc` where the data is to be written, with automatic incrementing addresses.
In addition, the fact that the data pointers are provided during their initialization, means that the hardware maintains the increment of the pointers internally for every `GET` or `PUT` invocation. Thus, the application need not worry about pointer management when executing these instructions.

NOTE: `WRGET0I()` and `WRPUTI()` are intrinsic functions that have one-to-one machine instruction counterparts. For example, `WRGET0I(&A, 15)` is shown as `WRAGET0I <wra>,15` in the disassembly file.

D The `WRGET0I` instruction provides mechanisms to load 1–16 bytes from an unaligned memory location into a wide register. Here, we fetch 15 bytes of input data for every call.

E Invoke the Extension Instruction using function call notation. The calling module supplies all the required I/O arguments when invoking the instruction. This instruction performs five pixels of RGB to YCbCr conversion for every call.

NOTE: The SE function `rgb2ycc()` call is equivalent to the user-created Extension Instruction `se_rgb2ycc <wr>, <wra>` in the disassembly file.

F The `WRPUTI` instruction provides mechanisms to store 1–16 bytes to an unaligned memory location from a wide register. Here, we store 15 bytes of output for every call.

G The `WRPUTFLUSH` instructions flush the output byte stream. They are required when using `WRPUT` instructions to write data to memory.

1.6 Analyze Profile Output with Extension Instruction

After compiling, linking, running, and profiling the Color Space Conversion application with the Extension Instruction, we get the performance statistics shown in Table 2.2.

Table 1-4 Excerpt of performance statistics for Extension Instruction code *(Sheet 1 of 2)*

%	Cumulative cycles	Self Cycles	Number of calls	Self cycles /call	Total cycles /call	Function name
86.46	63348.00	63348.00				ResetH
5.58	67434.00	4086.00	1	4086.00	8714.63	main
1.50	68531.00	1097.00	1	1097.00	3305.63	_vfprintf_r
1.04	69296.00	765.00	1	765.00	765.00	rgb2ycc_wrapper

Table 1-4 Excerpt of performance statistics for Extension Instruction code *(Sheet 2 of 2)*

%	Cumulative cycles	Self Cycles	Number of calls	Self cycles /call	Total cycles /call	Function name
0.81	69891.00	595.00	1	595.00	778.5	_malloc_r
0.48	70245.00	354.00	2	177.00	352.56	__sfvwrite

As you can see in Table 2.2, the `rgb2ycc` function no longer exists, and all the compute cycles fall under the `rgb2ycc_wrapper` function. Comparing the cycles for the wrapper function with the C implementation, we see that the wrapper function with the Extension Instruction takes a total of 765 cycles per call versus 9293 cycles per call for the C implementation. This results in a 12x performance gain.

Appendix A Source Listings

This appendix contains the C source listing for the Color Space Conversion application.

A.1 rgb2ycc_a.c

```
#include "data.h"

void
rgb2ycc(
    signed char r, signed char g, signed char b,
    signed char *y, signed char *cb, signed char *cr)
{
    *y  = ( 77*r + 150*g +  29*b        ) >> 8;
    *cb = (-43*r -  85*g + 128*b + 32768) >> 8;
    *cr = (128*r - 107*g -  21*b + 32768) >> 8;
}
void
rgb2ycc_wrapper(int np, signed char *RGB,
                signed char *YCC)
{
    int i;

    for (i = 0; i < 3 * np; i += 3) {
        rgb2ycc(RGB[i], RGB[i+1], RGB[i+2], &YCC[i],
                &YCC[i+1], &YCC[i+2]);
    }
}

int main()
{
    signed char ycc[3 * NP];
    int i, err=0;
    rgb2ycc_wrapper(NP, RGB, ycc);

    for (i = 0; i < 3 * NP; i++) {
        err |= YCC[i] != ycc[i];
    }

    printf("%s\n", err ? "Error" : "Pass");
    return err;
}
```

A.2 rgb2ycc_b.c

```
#include "data.h"
#include "rgb2ycc.h"

#if (!defined(__STRETCH_S5_ISS__) && !defined(__STRETCH_NATIVE__))
#include <s5000/sx-isef.h>
#endif

void
rgb2ycc_wrapper(signed char *RGB, signed char *ycc)
{
  WR A, B;
  int i;

  WRGET0INIT(0, RGB);/* initialize input stream from RGB */
  WRPUTINIT(0, ycc);/* initialize output stream to ycc */

  /* loop over RGB data, converting 5 pixels at a time */
  for (i = 0; i < NP/5; i++) {
    WRGET0I(&A, 15);/* load 5 RGB pixels to A */
    rgb2ycc(A, &B);/* convert 5 pixels */
    WRPUTI(B, 15);/* store 5 YCbCr pixels from B */
  }

  WRPUTFLUSH();  /* flush output stream */
}

int main()
{
  signed char ycc[3 * NP];
  int i, err=0;

#if (!defined(__STRETCH_S5_ISS__) && !defined(__STRETCH_NATIVE__))
   /* Load the ISEF */
   printf("Loading ISEF with bitstream rgb2ycc....\n");
   err = sx_isef_load_by_name_async(sx_isef_a, "rgb2ycc");
   if (err) {
      while (1); // Spin in an endless loop so we can break here and debug
    }
   printf("Done. \n");
#endif

  rgb2ycc_wrapper(RGB, ycc);

  for (i = 0; i < 3 * NP; i++) {
    err |= YCC[i] != ycc[i];
  }
  printf("%s\n", err ? "Error" : "Pass");

  return err;
}
```

A.3 rgb2ycc.xc

```
#include <stretch.h>

/* Extension instruction converting 5 pixels */
SE_FUNC void rgb2ycc(WR A, WR *B)
{
  se_sint<8> r[5], g[5], b[5];
  se_sint<8> y[5], cb[5], cr[5];
  int i, j;

  /* unpack A to RGB data, does not use any ISEF logic */
  for (i = 0; i < 5; i++) {
    j = i * 3 * 8;
    r[i] = A(j+7, j);
    g[i] = A(j+15, j+8);
    b[i] = A(j+23, j+16);
  }

  /* converting 5 pixels */
  for (i = 0; i < 5; i++) {
    y[i]  = ( 77*r[i] + 150*g[i] +  29*b[i]        ) >> 8;
    cb[i] = (-43*r[i] -  85*g[i] + 128*b[i] + 32768) >> 8;
    cr[i] = (128*r[i] - 107*g[i] -  21*b[i] + 32768) >> 8;
  }

  /* pack YCbCr to B */
  *B = (cr[4],cb[4],y[4],cr[3],cb[3],y[3],cr[2],cb[2],y[2],cr[1],cb[1],y[1],
        cr[0],cb[0],y[0]);
}
```

A.4 Makefile

```
#
#*********************************************************************\
#*                                                                   *
#* Copyright 2003-2004 Stretch, Inc. All rights reserved.            *
#*                                                                   *
#* THIS SOFTWARE CONTAINS CONFIDENTIAL INFORMATION AND TRADE SECRETS OF *
#* STRETCH, INC. USE, DISCLOSURE, OR REPRODUCTION IS PROHIBITED WITHOUT *
#* THE PRIOR EXPRESS WRITTEN PERMISSION OF STRETCH, INC.             *
#*                                                                   *
#*********************************************************************/
#

OUT_DIR = .

ifndef VER
VER = b
endif

ifdef OPT
O = -O3
else
O = -O0
endif

ifndef TARGET
X  = -ms5-native
E  = -NATIVE

else#ifndef TARGET

ifeq ($(TARGET),NATIVE)
X       = -ms5-native
E       = -NATIVE
endif

ifeq ($(TARGET),ISS)
X       = -ms5-iss
E       = -S5ISS
RUN     = st-run
PROF    = st-gprof
endif

ifeq ($(TARGET),S5610)
X       = -ms5610
E       = -S5610
LDFLAGS = -mlsp=s56db-ddr
endif

endif  #ifndef TARGET

CFLAGS = -g $(O) $(X)

default:
```

```
   @echo "usage: make [VER=a|b] [OPT] [TARGET=NATIVE|ISS|S5610]
[build|run|profile]"

build: rgb2ycc_$(VER)$(O)$(E).exe

ifneq ($(TARGET),S5610)
run: rgb2ycc_$(VER)$(O)$(E).exe
   $(RUN) $(OUT_DIR)/$^

ifeq ($(TARGET),ISS)
profile: rgb2ycc_$(VER)$(O)$(E).exe
   $(RUN) --mem_model --profile=gmon.out $(OUT_DIR)/$^
   $(PROF) $(OUT_DIR)/$^ > $^.prof

else
profile:
   @echo "Profiling not supported for this target"
endif

else
run:
   @echo "Please use IDE or st-debug for remote debugging target S5610"

profile:
   @echo "Profiling not supported for target S5610"
endif

rgb2ycc_a$(O)$(E).exe: rgb2ycc_a.c data.h
   scc $(LDFLAGS) $(CFLAGS) -o $(OUT_DIR)/rgb2ycc_a$(O)$(E).exe rgb2ycc_a.c


rgb2ycc_b$(O)$(E).exe: rgb2ycc_b$(O)$(E).o
   scc $(LDFLAGS) $(CFLAGS) -o $(OUT_DIR)/rgb2ycc_b$(O)$(E).exe -Irgb2ycc_b$(E)
rgb2ycc_b.c rgb2ycc.a

rgb2ycc_b$(O)$(E).o: rgb2ycc_b$(E)/rgb2ycc.h rgb2ycc_b.c data.h
   scc $(LDFLAGS) $(CFLAGS) -c -o rgb2ycc_b$(O)$(E).o -I./rgb2ycc_b$(E) rgb2ycc_b.c

rgb2ycc_b$(E)/rgb2ycc.h: rgb2ycc.xc rgb2ycc_b$(E)
   scc $(CFLAGS) -stretch-h rgb2ycc_b$(E)/rgb2ycc.h -o rgb2ycc.a rgb2ycc.xc

rgb2ycc_b$(E):
   mkdir rgb2ycc_b$(E)

clean_native:
ifeq ($(OS),Windows_NT)
   if EXIST rgb2ycc_b-NATIVE. (rmdir /s /q rgb2ycc_b-NATIVE.)
else
   if [ -d "rgb2ycc_b-NATIVE" ]; then\
   rm -Rf rgb2ycc_b-NATIVE;\
   fi;
endif

clean_iss:
ifeq ($(OS),Windows_NT)
   if EXIST rgb2ycc_b-S5ISS. (rmdir /s /q rgb2ycc_b-S5ISS.)
else
   if [ -d "rgb2ycc_b-S5ISS" ]; then\
   rm -Rf rgb2ycc_b-S5ISS;\
```

```
   fi;
endif

clean_s5610:
ifeq ($(OS),Windows_NT)
   if EXIST rgb2ycc_b-S5610. (rmdir /s /q rgb2ycc_b-S5610.)
else
   if [ -d "rgb2ycc_b-S5610" ]; then\
   rm -Rf rgb2ycc_b-S5610;\
   fi;
endif

clean_other:
   $(RM) *.exe *.xr *.o *.a rgb2ycc.h
ifeq ($(OS),Windows_NT)
   if EXIST stretch-tdk. (rmdir /s /q stretch-tdk.)
else
   if [ -d "stretch-tdk" ]; then\
   rm -Rf stretch-tdk;\
   fi;
endif

clean: clean_native clean_iss clean_s5610 clean_other

distclean: clean
   $(RM) profiles rgb2ycc.a rgb2ycc.h *.xo *.xr stretch-tdk
```

This page intentionally left mostly blank.

Part #: AN-0000-0001-001

Stretch™

777 E. Middlefield Road
Mountain View, CA 94043

650-864-2700 Tel
650-623-0150 Fax

www.stretchinc.com