USING C TO ACCELERATE COMPUTE-INTENSIVE APPLICATIONS

Joe Hanson (Stretch Inc., Mountain View, CA, USA, hanson@stretchinc.com)
Bruce McNamara (Stretch Inc., Mountain View, CA, USA, bruce@stretchinc.com)

Many electronics applications are very compute-intensive (e.g., software-define radio, wireless communications, radar, and sonar). The challenge in developing these applications is to get the highest compute performance possible, while retaining tremendous flexibility to configure applications for specific functions. This increasing demand for compute capacity has challenged fixed instruction set processors. To overcome this, designers have begun trying to off-load portions of their algorithms onto hardware accelerators using FPGA or ASIC devices. Software-configurable processors provide programmability through the simultaneous reconfiguration of functions in both hardware (custom instructions) and software. With an instruction set that can change dynamically, a single team of hardware / software developers can approach complex and compute-intensive algorithms from a completely different perspective. A software-configurable processor combines the ease of software development associated with GPPs and DSPs, with the parallelism and flexibility of FPGAs. This paper will discuss how software-configurable programming differs from traditional software-only methodologies and mixed-language hardware/software methodologies by describing the implementation of several compute intensive functions using only the "C" programming language.

1. INTRODUCTION

The complexity of embedded systems, especially software define radio, has reached a point where hardware acceleration is often required in order to meet real-time processing requirements and market cost points. Introducing hardware acceleration provides the necessary performance increase but at the cost of increasing complexity and extending the overall design cycle. Given rapidly evolving and emerging standards, a flexible, scalable, and programmable architecture with most of the application software written in C/C++ is desired.

Developers have increased performance by partitioning an application across multiple processors. This approach however, increases device size, power consumption, and application complexity, often exceeding cost and power budgets to meet minimum performance requirements. A more recent approach has been to offload processing to an FPGA-based engine acting as a co-processor to assist in computations done on an application processor. The primary disadvantage of this approach is that the heterogeneous nature of the architecture requires separate development environments. Additionally, having to design additional interfaces between the processor and FPGA—including a hardware interface, data exchange mechanism, and processing overhead—increases design complexity and introduces unnecessary design constraints.

In reality, hardware acceleration of an algorithm cannot begin until the algorithm has been completely designed in software. What this does is merely reverses the traditional "hardware first, then software" design model. What developers really need in order to improve performance without completely undermining time-to-market is concurrent software and hardware development. This is only possible if hardware and software are created at the same time.

Today developers have access to software-configurable architectures which provide the flexibility of a general-purpose processor with the computational capacity of a DSP or ASIC. Development is done entirely in software—both hardware and software functionality are described in C/C++, effectively enabling developers to design "hardware as software", resulting in reduced design complexity and speeding time-to-market.

Software-configurable architectures achieve this by abstracting hardware acceleration. Four key aspects of design that provide an opportunity for substantial performance acceleration include operator fusion, vectorization, data bandwidth, and deep pipelining.

2. OPERATOR FUSION

Operator fusion must be designed from the ground up with these other three factors in mind to achieve optimal acceleration benefits. Operator fusion is the combining of multiple computation operations into a single instruction. This custom instruction in affect transforms a generic instruction set architecture into a highly specialized set of operations specific to the application. As a result, an entire function can be encapsulated as a single extension instruction.

3. VECTORIZATION

Vectorization is one of the traditional first stages of hardware acceleration. The ability to process multiple words of data with a single instruction (single instruction multiple data, SIMD architecture) is critical for improving performance without having to clock processors at higher frequencies that lead to greater manufacturing cost and increased power consumption.

One capability important to overall performance efficiency is the ability to work with different sizes and formats of data. A constraint of using standard instructions is that the choice of data width and format is limited. For example, depending upon the application and task at hand, the ideal data size may be from one to several bytes, aligned or unaligned, sequential or streaming, or perhaps even bit-reversed. The advantage of a software-configurable architecture is that data size and format can be determined on an instruction-by-instruction basis. It is unnecessary to convolute data to fit the size of the instruction; extension instructions are specifically designed to match data to reduce parsing overhead and facilitate maximum performance and optimal use of resources. Additionally, management of data can be simplified by implementing circular-buffer load/stores, rotates, constants, and offsets as a part of an extension instruction to reduce the number of standard instructions required to preparing data for hardware acceleration.

4. DATA BANDWIDTH

Sufficient data bandwidth is one of the most critical aspects of performance acceleration as it determines the degree of vectorization possible. Typical embedded processors operate on fixed register widths, e.g. 32-bit registers. Providing registers the same width as a cache line fill and from which multiple data words can be extracted is required for meeting the data bandwidth requirements for software acceleration and vectorization.

5. DEEP PIPELINING

Deep pipelining is the ability of a processing architecture to execute multiple instructions simultaneously while making optimal use of the overall pipeline. Different instructions require a variable number of pipeline stages to complete execution, especially as extension instructions become more complex and represent entire functions. Managing timing and dependencies between variable-cycle instructions adds great complexity to the software development process. To simplify code development, the compiler must schedule instructions efficiently to contain latency to achieve single-cycle effective throughput for every instruction. In this way, complexity is handled by the compiler and processor, rather than left as a burden to make design more complex for developers.

Deep pipelining is only possible when accelerated, extension instructions share the same pipeline and instruction decode unit as standard instructions. When extension instructions are executed in a separate pipeline, it becomes extremely difficult to manage dependencies between standard and extension instructions. Mechanisms must be put in place to manage these dependencies, increasing instruction latency and undermining deterministic processing performance. Because a co-processor may require a variable number of cycles to complete an operation, developers must assume worst-case latency to simplify development and operation. Additionally, overhead inefficiencies are introduced when extension instruction context must be passed to a co-processor with data to be processed and then passed back when processing is completed.

When extension instructions and standard instructions are logically the same to the main processor, passing context is a matter of passing a pointer to shared memory or, more often, leaving relevant values in state registers. Dependencies can be managed by the pipeline, minimizing latency. Such latency is also deterministic, and can be automatically accounted for by the compiler when generating application code.

An important aspect enabled by deep pipelining is that the implementation details of extension instructions in programmable logic are encapsulated in the same format as software instructions. Architectures that use a separate FPGA device significantly complicate design by requiring developers to introduce a completely new development language, such as HDL or Verilog, and corresponding tool chain to the design process.

This is the basis of the "hardware as software" design model for software-configurable processors. Software instructions and hardware accelerated instructions are described in a high order language such as C/C++ and converted to software and hardware through an optimizing compiler. Such a compiler is able to implement hardware acceleration in an optimal fashion, managing overall latency, the efficiency of pipelining, and maximizing the frequency with which can extension instructions can be issued.

Optimizing compilers can also improve performance and efficient use of resources in ways that are simply too time-consuming for a person to implement. For example, the compiler can recognize shared structures between different extension instructions and implement them using the same programmable resources, preserving these valuable resources for either additional extension instructions or further accelerate those already existing. Additionally, the compiler can readily identify and track dependencies and context sharing instances that may not be readily apparent to developers. As a consequence, register forwarding, multiplexing, and context save/restore functions can be minimized or even eliminated.

6. APPLICATION DEVELOPMENT

Developing applications for software-configurable architectures follows the same process as the traditional software development cycle. An integrated development environment manages the project and acts as a front-end to the development tool chain, including compiler, debugger, and profiler. When it comes time to improve application performance, however, rather than hand-coding assembly language or, for FPGA coprocessor architectures, passing the software algorithm to a second hardware development team to implement the function in hardware, developers instead identify "hot spots" within the program. This enables the compiler to accelerate algorithmic code by creating an extension instruction. Without requiring additional programming from the developer, the compiler creates an optimized configuration to be implemented in a programmable fabric and schedules the instruction as it would any other instruction. Developers can then profile the performance of the extension instruction. If required, the function can be characterized to process multiple data words in parallel.

A key benefit of the "hardware as software" development flow is that it keeps design in a single development environment that is well established and familiar to software developers. FPGA-based architectures require the use of the second development team and any repartitioning of application code requires a re-architecting of hand-optimized logic to match the new partitioning. With a software-configurable processor, the compiler is responsible for re-architecting. This means that even though extension instructions are implemented in hardware / programmable logic, developers design, create, and use them entirely in a software context.

Another important benefit of developing an application entirely in software is that the code can be compiled for targets other than the software-configurable processor. For example, the compiler could create a functionally equivalent code image for an x86 processor. This allows developers to develop, test, and debug application and algorithmic code while hardware is still being developed.

Together, all of these factors have a tremendous impact on the way developers approach application design. Not only can a single development team create an application, development time is significantly reduced by enabling concurrent software and hardware development without time-consuming hand optimization.

7. RGB-TO-YCbCr COLOR SPACE CONVERSION EXAMPLE

Perhaps the most efficient means for quantifying the impact of a software-configurable architecture on performance is providing a straightforward real-world example such as an implementation of a color space conversion algorithm. This example will use the Stretch software-configurable processor to illustrate the impact of "hardware as software" acceleration.

The software-configurable processor combines a RISC processor with a configurable fabric known as the Instruction Set Extension Fabric (ISEF). Extension instructions are implemented in the ISEF using field programmable technologies and provide performance comparable to custom hardware implementations. The primary distinction of the software-configurable processor is that extension instructions are coded in C/C++ and can be tuned to match a specific application. New extension instructions can be introduced at any time during application development if a developer has such a need.

The base processor of the software-configurable processor is a standard five-stage pipeline Tensilica Xtensa V RISC core with 32 KBytes of both instruction and data cache, memory management unit, and 24 DMA channels (see Figure 1). The ISEF is interlocked to the instruction pipeline of the Xtensa core and provides a large array of computational resources (4096 arithmetic unit bits and 8912 multiplier unit bits) that can be used at any bit width, thus conserving resources. The RISC core and ISEF exchange data via a 128-bit wide register (WR).

Figure 2 shows the base mathematical expression for the conversion of red, green, and blue (RGB) pixel data to Luminance and Chrominance (YCbCr). As it stands, this function converts a single RGB pixel during each loop iteration. If implemented as part of an application in this form, this function would consume over 3.5 million cycles to convert a large block of pixel data (see Table 1).

Figure 3 shows the process of accelerating this function using extension instructions. It has been rewritten for the software-configurable processor using Stretch C. The use of "SE_FUNC" informs the compiler that this function should be implemented in the ISEF. The extraction operator in Stretch C maps the variables R, G, and B to specific bits within the WR; concatenation operators map Y, Cb, and Cr results back to the WR.

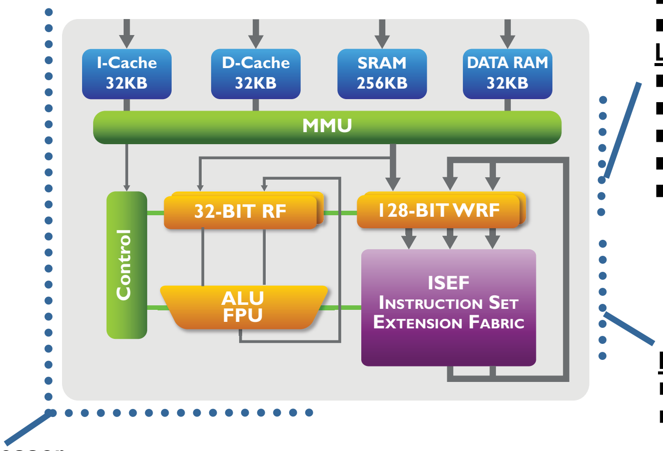
Note that the coding of the algorithm itself has not changed. The compiler, however, has implemented all of the additions and multiplies in a single extension instruction. As a result, all of these mathematical operations are completed within two pipelined clock cycles and an overall 15X performance improvement (see Table 1).

The 128-bit wide-registers between the RISC core and ISEF enable the passing of five pixels simultaneously, enabling the conversion of multiple pixels in parallel (see Figure 4). While the RGB data extraction is coded to resemble a loop, the compiler is able to directly extract values without using any ISEF compute resources. As a result, the computation has not changed other than that now five values are computed in parallel. Results can be stored in a single operation, again requiring no compute resources. Thus, 15 bytes are loaded and stored for each iteration of the program loop, increasing performance further for a final improvement in performance of over 80X compared to the original function (see Table 1).

8. CONCLUSION

Through the use of software-configurable processors, developers can implement hardware acceleration for compute-intensive algorithms through the use of extension instructions coded in C/C++. Extension instructions provide the performance of hardware implementations with the flexibility of software design. Specialized computations on specialized application data sizes and formats increases flexibility and optimize the use of computational resources. By describing software and hardware functionality using a single programming language and development tool chain, a single development team can design hardware and software concurrently, significantly reducing time-to-market.

The flexibility of software-configurable processors also enables developers to further improve performance by developing functions that are able to process multiple data in parallel. Wide registers provided sufficient data bandwidth to keep computational resources fed and maximize parallelism. Finally, because hardware and software are described simultaneously, the software compiler is able to implement and schedule extension instructions to achieve maximum performance (80X in the colorspace example) by keeping the processor pipeline optimally filled.



Wide Register File
- 128-bit wide
- 32 entries

Load/store unit
- 128-bit load/store
- Auto increment/decrement
- Immediate, indirect, circular
- Variable-byte load/store
- Variable-bit load/store

ISEF
- 3 inputs and 2 outputs
- Pipelined, bypassed, interlocked
- 32 16-bit MACs and 256 ALUs
- Bit-sliced for arbitrary bit-width

RISC Processor
- Tensilica – Xtensa V
- 32 KB I & D Cache
- On-Chip Memory, MMU
- 24 Channels of DMA, FPU

Figure 1 Stretch S5 Engine

```
void RGB2YCBCR (
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;
}
Program Loop:
for (…) {
/* Convert 1 RGB Pixel to 1 YCbCr pixel */
RGB2YCBCR (RGB[i], RGB[i+1], RGB[i+2], &YCC[i], &YCC[i+1], &YCC[i+2]);
}}
```

Figure 2 Color Space Conversion - C Code

```
SE_FUNC /* Tells Stretch C-Compiler to reduce this function to an instruction */
void RGB2YCBCR (WR A, WR *B)   /* Data Bandwidth – Move 24 bits in Single Register */
{

se_sint<8> r, g, b, y, cb, cr;
     r = A(7,0); g = A(15,8); b = A(24,16); /* Extract Data; No Compute Cycles */

     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;

   *B = (cr,cb, y);               /* pack YCbCr to B; No Compute Cycles*/
}

Program Loop:
for (…) {
            WRGET0(&A, 3);      /* Load 3 bytes (1 RGB pixels)  */
            RGB2YCBCR(A, &B);   /* Convert 1 pixel             */
            WRPUT0(B, 3);       /* Store 3 bytes (1 YCbCr pixel */
}
```

Figure 3 Application Specific Instruction

```
SE_FUNC /* Extension instruction converting pixels */
void RGB2YCBCR (WR A, WR *B)  {      /* Data Bandwidth – Move 96 bits */
  se_sint<8> r[5], g[5], b[5], y[5], cb[5], cr[5];
  int i, j;
  /* Unpack A to RGB Data, Does Not Use Any Compute Cycles */
  for (i = 0; i < 5; i++, j = i*24) { r[i] = A(j+7, j); g[i] = A(j+15, j+8); b[i] = A(j+23, j+16) }
  /* Convert 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; Does Not Use Any Compute Cycles */
  *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]);
}

Program Loop:
for (…) {
            WRGET0(&A, 15);      /* Load 15 bytes (5 RGB pixels)  */
            RGB2YCBCR(A, &B);    /* Convert 5 pixels              */
            WRPUT0(B, 15);       /* Store 15 bytes (5 YCbCr pixels */
}
```

Figure 4   Application Specific Instruction – with Vectorization

Table 1 Software Acceleration Results

Software	WR	ISEF (Bit-Width)	ISEF (State Reg.)	Instruction Pipeline	Cycle (K Cycles)	Factor
RGB2YCC ANSI – C Only					17092	1
Compiler Optimization				√	3458	5
Operator Specialization	√	√	√		2307	7
Compiler Optimization	√	√	√	√	219	78
Data Parallelism	√	√	√		429	40
Compiler Optimization	√	√	√	√	42	404

Extending The Possibilities™

Accelerating Compute Intensive Functions Using "C" and Software-Configurable Processors

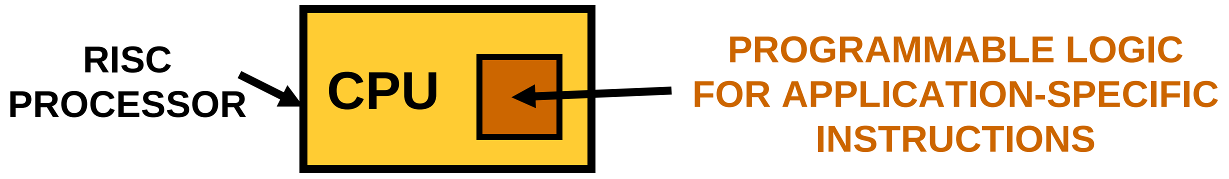
SDR Forum, November, 2005

Joe Hanson, Director of Business Development

Agenda

- » Software-Configurable Processor
- » Development Flow
- » Example
- » Summary

New Approach to Computing Applications

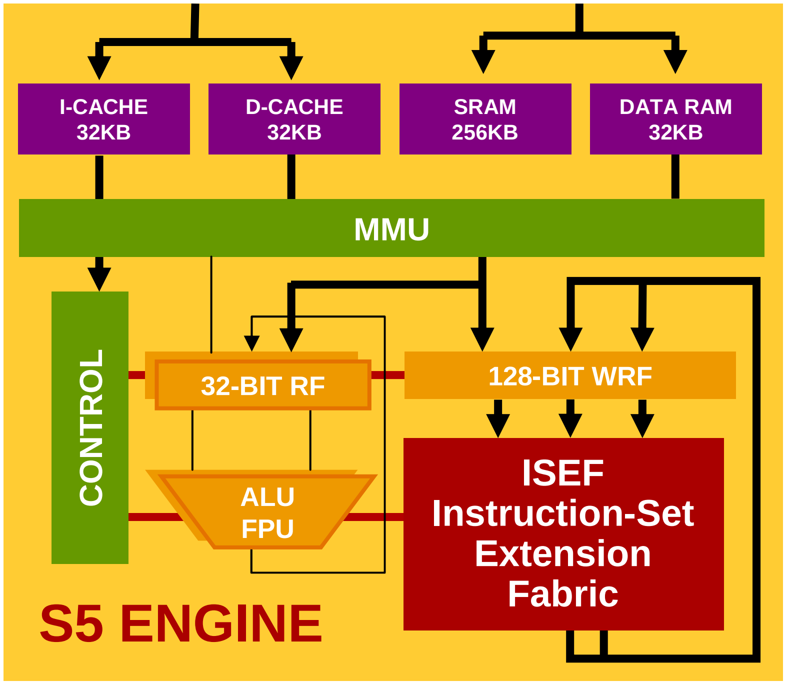


Software-Configurable Processor

- Faster Time-to-Performance
  - H/W Compute Performance within a Software Design Flow
- Greater Algorithm Flexibility and Control
  - Software Design Methodology in C/C++
- Delivers Faster Time–to–Market
  - Integrated Programmable Solution

S5 Engine

RISC Processor
- Tensilica – Xtensa V
- 32 KB I & D Cache
- On-Chip Memory, MMU, FPU
- 24 Channels of DMA

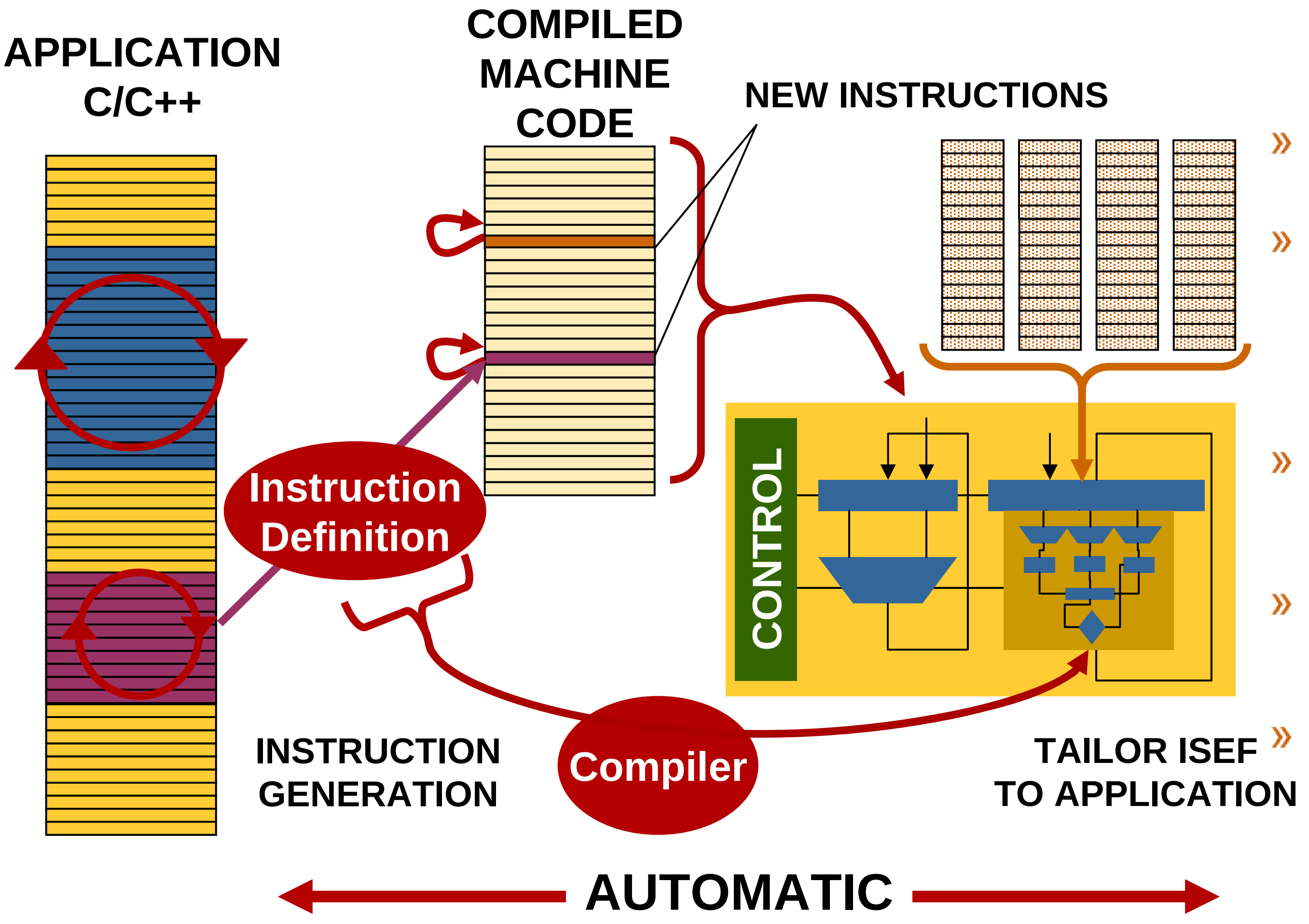


Wide Register File (WRF)
- 32 Wide Registers (WR)
- 128-bit Wide

Load/Store Unit
- 128-bit Load/Store
- Auto Increment/Decrement
- Immediate, Indirect, Circular
- Variable-byte Load/Store
- Variable-bit Load/Store

ISEF
- Instruction-Set Extension Fabric
- Compute Intensive
- Arbitrary Bit-width Operations
- 3 Inputs and 2 Outputs
- Pipelined, Bypassed, Interlocked
- Random Logic Support
- Internal State Registers

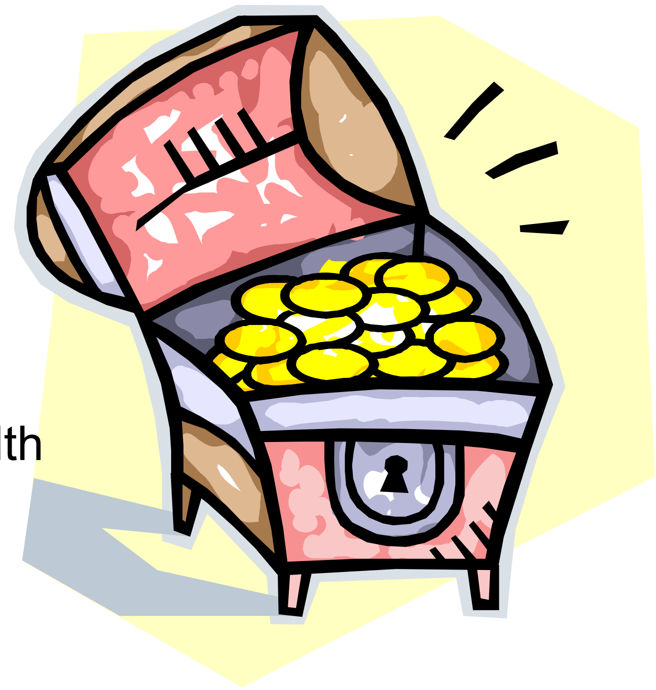
Software Acceleration Development Flow



- » Profile Code
  - › Identify "Hotspots"
- » Specialized Instructions
  - › Implement 'C' Functions in Single Instructions
  - › Bit-Width Optimizations
- » Software Compiler
  - › Instruction Generation
  - › Instruction Scheduling
- » Multiple Data (WR)
  - › Perform Operations in Parallel
- » Efficient Data Movement
  - › Intrinsic Load Store Operations
  - › 20+ DMA Channels

Sources of Software Acceleration

- » Operator Fusion
  - › Resource Sharing, Operator Merging
  - › Constant Propagation, Partial Evaluation
  - › Bit Width Optimization
- » Vectorization
  - › Operate on Multiple Data Objects in Parallel
- » Data Bandwidth
  - › Wide Data Registers – 128-bit Cache Line Width
  - › Rich Load/Store Instructions
- » Deep Pipelining
  - › Instruction Scheduling by Compiler
  - › Hardware Interlocks



Stretch C Extensions

- ANSI – C Plus Limited Set of Extensions
  - Defined in `<stretch.h>`
  - Extensions Imply ISEF Usage
- "SE_FUNC" Identifies a Function to Compile into Extension Instruction
- Data Types – All Standard Integer Data Types Plus
  - `se_uint<n>` Defines an Unsigned Word of N-bits
  - `se_sint<n>` Defines a Signed Word of N-bits
  - `WR` Defines a Wide Register File Variable (se_uint<128>)
- Operators - All Standard Operators Except Divide and Modulus Plus
  - Extraction: R = `A(7,0)` G = `A(15,8)` B = `A(24,16)`
  - Concatenation: *Y = `(B,G,R)`
- Specialized WR Load/Store Instructions
- Functionally Correct for ANY C++ Compiler (GCC Etc.)
  - No Presumed C++ Dynamic Memory Structures

Example C: RGB $\rightarrow$ YC_bC_r Conversion

Color Conversion Function:

```
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;
}
```

Program Loop:

```
for (…) {
    /* Convert 1 RGB Pixel to 1 YCbCr pixel */
  rgb2ycc(RGB[i], RGB[i+1], RGB[i+2], &YCC[i], &YCC[i+1], &YCC[i+2]);
}}
```

RGB → YC_bC_r - Operator Fusion

Color Conversion Function:

```
SE_FUNC /*  Tells Stretch C-Compiler to reduce this function to an instruction  */
void RGB2YCC (WR A, WR *B)   /* Data Bandwidth – Move 24 bits in Single Register */
{
  se_sint<8> r, g, b, y, cb, cr;
  int i;
     r = A(7,0); g = A(15,8); b = A(24,16); /* Extract Data; No Compute Cycles */

     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;

   *B = (cr,cb, y);                /* pack YCbCr to B; No Compute Cycles*/
}
```

Program Loop:

```
for (…) {
     WRGET0(&A, 3);       /* Load 3 bytes (1 RGB pixels)  */
     RGB2YCC(A, &B);    /* Convert 1 pixel                */
     WRPUT0(B, 3);        /* Store 3 bytes (1 YCbCr pixel */
}
```

RGB $\rightarrow$ YC_bC_r - Vectorization

Color Conversion Function:

```
SE_FUNC  /* Extension instruction converting pixels */
void RGB2YCC (WR A, WR *B)  {      /* Data Bandwidth – Move 120 bits */
  se_sint<8> r[5], g[5], b[5], y[5], cb[5], cr[5];
  int i, j;
  /* Unpack A to RGB Data, Does Not Use Any Compute Cycles */
  for (i = 0; i < 5; i++, j = i*24) { r[i] = A(j+7, j); g[i] = A(j+15, j+8); b[i] = A(j+23, j+16) }
  /* Convert 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; Does Not Use Any Compute Cycles */
  *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]);
}
```

Program Loop:

```
for (…) {
    WRGET0(&A, 15);       /* Load 15 bytes (5 RGB pixels)   */
    RGB2YCC(A, &B);       /* Convert 5 pixels               */
    WRPUT0(B, 15);        /* Store 15 bytes (5 YCbCr pixels */
}
```

Software Profiling Results

Software (RGB2YCC)	WR	ISEF (Bit-Width)	ISEF (State Reg.)	Instruction Pipeline	Cycle (K Cycles)	Factor
ANSI – C Only				√	3458	1
Operator Fusion		√	√	√	219	15
Vectorization	√	√	√	√	42	80

Over 80x Performance Improvement From C/C++

CPU Cycles for RGB to YCC Conversion

Extension Instructions for Convolutional Encoder

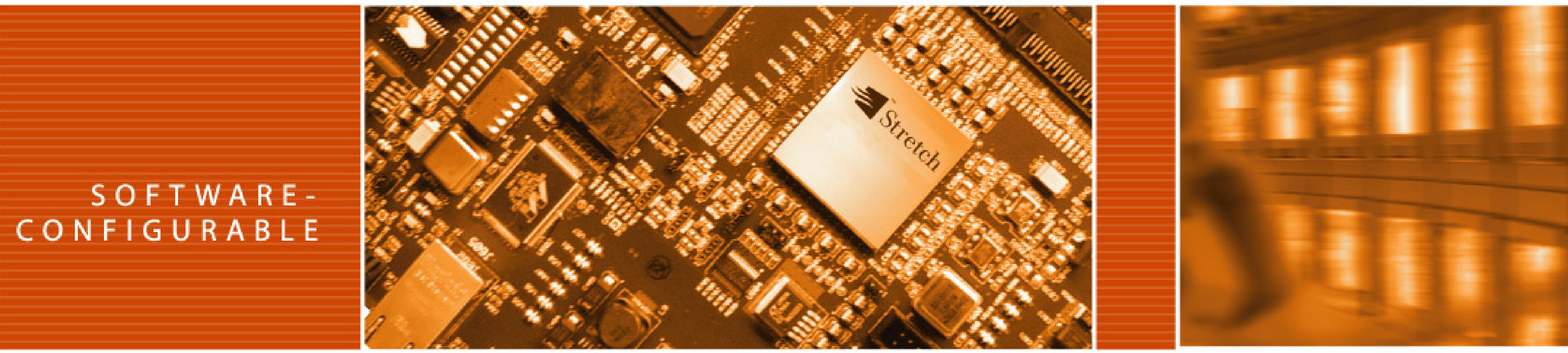
```
#include <stretch.h>
#define K   (7)
#define M   (48)
#define M12 (48)
#define M23 (32)
#define M34 (48)
#define M56 (40)
static se_uint<K> code0, code1;
static se_uint<K-1> hist;

SE_FUNC void CONVEN_INIT(unsigned char c0, unsigned char c1)
{
   code0 = c0;       code1 = c1;
   hist = 0;
}

SE_FUNC void CONVEN(SE_INST CONVEN12,
                     SE_INST CONVEN23,
                     SE_INST CONVEN34,
                     SE_INST CONVEN56,
                     WRA *d0)
{
   int i;
   /* up to M new input bits + K-1 history bits */
   se_uint<M+K-1> dIn = ( (se_uint<M>)(*d0), hist );
   /* 2 convolutions per input bit */
   se_uint<1>    X[M], Y[M];
   /* For each input bit, do two convolutions (length <= K)
    * to produce two output bits. */
   for (i = M-1; i >= 0; i--) {
      X[i] = (code0(0) & dIn(i+0)) ^
            (code0(1) & dIn(i+1)) ^
            (code0(2) & dIn(i+2)) ^
            (code0(3) & dIn(i+3)) ^
            (code0(4) & dIn(i+4)) ^
            (code0(5) & dIn(i+5)) ^
            (code0(6) & dIn(i+6));
      Y[i] = (code1(0) & dIn(i+0)) ^
            (code1(1) & dIn(i+1)) ^
            (code1(2) & dIn(i+2)) ^
            (code1(3) & dIn(i+3)) ^
            (code1(4) & dIn(i+4)) ^
            (code1(5) & dIn(i+5)) ^
            (code1(6) & dIn(i+6));
   }
```

```
    /* 1/2 rate: no puncturing */
    if (CONVEN12) {
       hist = (se_uint<K-1>)((*d0)(M12-1,M12+1-K));
       *d0 = 0;
       for (i = M12/1 - 1; i >= 0; i--) {
          *d0 = (*d0, Y[i], X[i]);
       }
    }

    /* 2/3 rate: puncture using (Y1, Y0, X0) (drop X1) */
    else if (CONVEN23) {
       hist = (se_uint<K-1>)((*d0)(M23-1,M23+1-K));
       *d0 = 0;
       for (i = M23/2 - 1; i >= 0; i--) {
          *d0 = (*d0, Y[2*i+1], Y[2*i], X[2*i]);
       }
    }

    /* 3/4 rate: puncture using (X2, Y1, Y0, X0) (drop Y2 & X1) */
    else if (CONVEN34) {
       hist = (se_uint<K-1>)((*d0)(M34-1,M34+1-K));
       *d0 = 0;
       for (i = M34/3 - 1; i >= 0; i--) {
          *d0 = (*d0, X[3*i+2], Y[3*i+1], Y[3*i], X[3*i]);
       }
    }

    /* 5/6 rate: puncture using (X4, Y3, X2, Y1, Y0, X0)
     * (drop Y4, X3, Y2 & X1) */
    else {   /* CONVEN56 */
       hist = (se_uint<K-1>)((*d0)(M56-1,M56+1-K));
       *d0 = 0;
       for (i = M56/5 - 1; i >= 0; i--) {
          *d0 = (*d0, X[5*i+4], Y[5*i+3], X[5*i+2],
                      Y[5*i+1], Y[5*i],   X[5*i]  );
       }
    }
 }
```

Summary

- Software-Configurable Processors and Stretch “C”
  - New Technology Addressing Compute Intensive Functions
- Software Development Flow
  - Enables Hardware Performance from “C” Software
  - No Hardware Development Required
- Tune the Instruction Set to the Applications
  - Run-time Configurable, Dynamically Loadable
  - Fast and Easy to Develop

Extending The Possibilities™



Accelerating Compute Intensive Functions Using "C" and Software-Configurable Processors

SDR Forum, November, 2005

Joe Hanson, Director of Business Development