COST-EFFECTIVELY IMPLEMENTING 802.16 SDR USING SOFTWARE-CONFIGURABLE ARCHITECTURES

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

The 802.16 WiMAX specifications contain a rich set of options addressing increasing bandwidth requirements for "last mile" digital communications applications. This wireless building block, however, is a fast-moving target that cannot be adequately implemented using fixed architectures such as FPGAs and ASSPs. This paper will demonstrate how 802.16 can be cost-effectively implemented using software-configurable processors which merge hardware and software development in a single design methodology based on C and using extension instructions to hardware-accelerate high-speed signal processing tasks, such as FFT and Viterbi decoding. By abstracting hardware as software, software-configurable processors achieve the same throughput as FPGA and high-end DSP-based architectures while extending overall programmability and flexibility to enable developers to support evolving standards in a timely fashion.

1. INTRODUCTION

The need to increase wireless data transfer rates while reducing deployment costs continues to keep the software-defined radio (SDR) market in the state of flux. Continuing innovation of new techniques to identify digital data amongst large amounts of noise results in increased data transfer bandwidth at greater distances but at the cost of increasing overall computational complexity. Not only do new algorithms give rise to new standards, they stress the capacity of traditional processors and hybrid-based architectures.

The emerging WiMAX standard, also known as IEEE 802.16, has garnered widespread support because of the efficiency and additional revenue it promises to bring to wireless applications. Figure 1 shows the basic receiver and transmitter block diagram for an 802.16 implementation. Many of these operations are computationally intense. For example, on the transmitter the physical layer (PHY) encodes the raw data stream and prepares it for upconversion to an analog radio signal. On the receiver side, the PHY extracts and then decodes the data stream from an analog radio signal. Both of these blocks require operations such as:

- Fast Fourier Transforms (FFT)
- Forward Error Correction (FEC)
- Block coding operations such as Reed-Solomon codecs
- Bit-level coding such as convolution encoding
- Viterbi decoding
- Quadrature Amplitude Modulation (QAM)
- Interleaving
- Scrambling

The media access control (MAC) layer is more control oriented and provides the interface between the PHY and network layers by scheduling transport of packets from the network layer according to quality of service (QoS) requirements on the transmit side and reassembling data for handing back to the network on the receiver side. The MAC layer is also responsible for automatically requesting the retransmission of any bad packets and maintaining communications between base and subscriber stations. Typically, a TCP/IP stack serves as the networking stack. Of course, all of these layers must be interconnected to form a complete system.

2. HARDWARE ACCELERATION WITHOUT A FIXED IMPLEMENTATION

The continuing evolution of WiMAX makes basing a design upon a fixed implementation such as an ASIC a risky proposition. Certainly an ASIC can provide the required performance, but it lacks the capacity to easily adapt to changing requirements. To adapt in a timely and cost-effective manner, a programmable development environment is required.
Traditional processors such as those based on RISC or DSP architectures provide the programmable foundation necessary to keep implementations current but they lack the necessary processing capacity to process WiMAX in real-time. RISC processors are limited by a narrow data bandwidth (only 4 bytes per clock for a 32-bit processor). While DSPs support more efficient dataflow, like RISC processors, they are tied to a general purpose and fixed instruction set. DSPs and RISC processors can perform only limited processing per cycle, and as a result processing complex algorithms can take thousands to tens of thousands of cycles.

Software-configurable processors combine the flexibility of a programmable processor with the high performance of hardware acceleration by integrating programmable logic into the processor pipeline. In this way, developers can implement extension instructions, which accelerate compute-intensive processing in hardware but are accessible through software, enabling a single extension instruction to perform the equivalent of hundreds to thousands of cycles on a traditional processor. Extension instructions are defined in program code using C/C++ and are compiled into a bit stream by an optimizing compiler to configure programmable logic resources tailored to meet application-specific requirements. Through extension instructions, performance can be increased from 10X to 100X.

The software-configurable processors from Stretch, for example, use an instruction set extension fabric (ISEF) that serves as a programmable fabric interlocked to the instruction pipeline (see Figure 2). Backed by a substantial set of computationally rich resources—4096 arithmetic units and 8912 multiplier units—computations can be accelerated for any bit width. Additionally, 128-bit wide registers and 24 powerful DMA channels enable the ISEF to process multiple data concurrently through a deep pipeline, enabling developers to exploit inherent parallelism in algorithms to significantly accelerate processing performance. Since the ISEF is reconfigurable, multiple instructions can reuse the same resources.

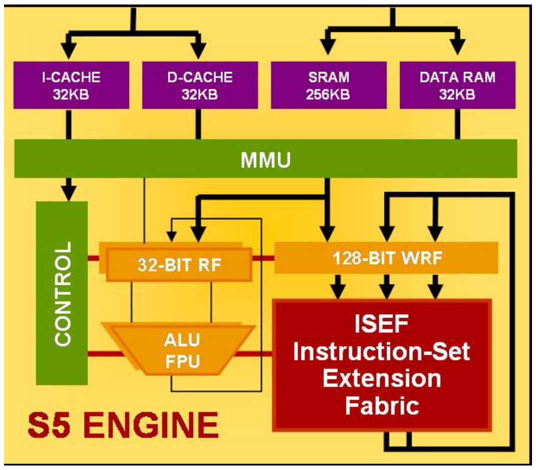


Figure 2 Software-Configurable Processor Architecture

3. ACCELERATING 802.16

The 802.16 WiMAX PHY makes use of a 256-point FFT and Orthogonal Frequency Division Multiplexing (OFDM). Depending upon the application, OFDM channel width can vary, as can the particular modulation scheme. For applications in noisy environments, FEC is mandatory and the 802.16 standard offers a variety of choices here as well.

Given the computationally intense nature of these calculations and the high signal frequencies involved, conventional RISC and DSP processors simply do not have enough processing capacity to support both the high demand of WiMAX baseband processing and control task management at the same time.

In fixed implementations, such as ASICs, hardware sources are locked, meaning that if multiple modulation schemes, for example, are to be supported, then multiple hardware implementations are required. Not only does this drive up device cost, the additional development resources necessary to create multiple hardware implementations make such an approach impractical.

With a software-configurable architecture, the same hardware resources can be reconfigured to accelerate completely different functionality. Because of the dynamic, reconfigurable nature of software-configurable processors, supporting multiple variations and schemes is a matter of reconfiguring the programmable processing resources. Thus it is possible for a single software-configurable processor to operate across multiple channel widths such as 3.5 MHz, 7 MHz, and 10 MHz, as well as across modulation schemes such as BPSK, QPSK, 16 QAM, or 64 QAM simply by reconfiguring the integrated programmable logic resources.

The run-time reconfigurability of software-configurable architectures provides an extended level of flexibility to developers. As the need for different instructions changes, so can the configuration. From a development perspective, since extension instructions are created by the compiler based on C code, adapting and introducing new innovations to existing algorithms is a straightforward process that eliminates the time-consuming stages of hand-coding in assembly/HDL and profiling performance. Note that dynamic reconfiguration overhead can be reduced to zero for software-configurable architectures. Unlike FPGAs, which introduce unacceptable delay to the point of interrupting dataflow processing when reconfiguring, software-configurable processors can "ping-pong" between configurations so that new extension instructions are available immediately.

This flexibility enables developers to scale applications based on software-configurable architectures easily. In this way, developers can create a base design that serves as the foundation across a wide range of applications and product variations, reducing design complexity and overall development investment while speeding time-to-market.

One of the most important efficiencies gained using software-configurable architectures is the ability of developers to design both hardware and software in a single integrated development environment. Instead of requiring two development teams, one for hardware and one for software, developers write code in C/C++ and map computationally intensive hotspots for implementation in the ISEF through the use of extension instructions. Not only does the compiler generate the appropriate extension instructions, it schedules them to achieve optimal pipeline parallelism to maximize single-cycle throughput. Put another way, hardware is abstracted as software, greatly simplifying the development process and avoiding time-consuming manual optimization.

Being able to perform operations in hardware and in parallel introduces substantial processing efficiencies. For example, consider the following implementation of an FFT using Radix-2 on a Stretch software-configurable processor. A single extension instruction is able to perform sixteen 16 x 16 multiplies, eight 32-bit adds, and sixteen 16-bit adds with rounding and rescaling.

One critical bottleneck of RISC and DSP architectures employing hardware acceleration through coprocessors implemented as ASICs or discrete FPGAs is dataflow. While the coprocessor is able to process large amounts of data, the RISC or DSP is limited in how quickly it can pass data to the coprocessor. Software-configurable architectures overcome this limitation through the use of wide registers. For example, the Stretch S5000 family of software-configurable processors have 32 128-bit Wide Registers (WR) that facilitate efficient transfer of data and eliminate dataflow bottlenecks.

In the case of FFT processing, an extension instruction is able to pass three sets of 4 complex values to the ISEF through wide registers for concurrent processing. In terms of real-world performance, this translates to the ability to perform a 256-point FFT in 4 µs. Developers can achieve, by implementing a Radix-4 FFT, an additional 28% performance improvement.

4. EFFICIENT CONVOLUTION

Forward error correction (FEC) compensates for the presence of channel noise by transmitting data with enough redundancy so that errors can be corrected on the receive side. Because the data rate and distance over which data can be transmitted are directly tied to the transmission error rate, significant research investment continues to be made in FEC technology. As a consequence, communication systems must be able to support new innovations in FEC technology if they are to take advantage of these to increase system throughput and reliability while reducing cost.

FEC schemes typically employ bit-level encoding techniques such as convolution where each encoded bit is generated by convoluting the current input bit with previous input bits. WiMAX uses convolution encoding with a constraint length (i.e., the number of bits used in the convolution) of K=7 and a rate (i.e., the number of input bits per output bit) of ½ (see Figure 3). A ½-rate encoder can be followed by a puncture to produce other rates such as 2/3, 3/4, and 5/6.

RISC architectures are notoriously inefficient at bit-level operations. Programmable logic, on the other hand, is particularly well suited to convolutions; for example, an extension instruction can be implemented to take 64 bits of input data and generate 128 outputs (see Code Listing 1).

Another feature of software-configurable architectures, which accelerates bit-level processing, is the capacity to store state information. In this way, a large number of intermediate results can be efficiently transferred between different extension instructions without incurring excessive latency from load/store operations to preserve these results as required by coprocessor implementations. In this example, six input bits of history can be preserved for use by the next convolution operation.

Convolution can be traced from one stage to the next for a ½-rate convolutional encoder, as illustrated by a Trellis diagram (see Figure 4). As each new input is fed into the encoder, state changes are propagated through K-1 shift registers while producing 2 output bits. As each new bit comes in, each shift register transitions from one state to another for a total of 2^6 possible states per input bit. A Trellis diagram shows these transitions from one input to the next. For example, let Sn = {S0, S1, …, SK-2} represent the state bits at the nth stage. Assume the states of the K-1 shift registers are set to 0, i.e, S = {0, …, 0} in the initial stage (0th stage). If the new bit is 0, the state S will continue to be at S={0, 0, …, 0} at the 1st stage. If the new bit is 1, the state S will be S = {1, 0, …, 0}. The horizontal axis can be considered symbol time 0, 1, …, etc, since each input bit will be transmitted as a symbol in the communication channel.

The Trellis diagram is helpful in finding the most likely sequence of codes when decoding a convoluted bitstream. Viterbi coders are a particularly efficient way to decode a bitstream because they record the most likely path for each state at each Trellis stage, thereby limiting the number of sequences needing to be examined. This efficiency does not come without cost, as Viterbi decoding is quite computationally intensive through the use of add-compare-selection (ACS) for each state at each stage while simultaneously tracking the history of the selected path.

Viterbi decoding has three primary steps: branch metric computation, ACS, and traceback. The branch metric (BM) measures the distance between the received signal Rn = {Xn, Yn} and the appropriate output branch level L={L1, L2}. Therefore possible output labels for a ½-rate encoder are {0,0}, {0,1}, {1,0}, and {1,1}. The branch metric is computed as follows:

$$BMn(L1, L2) = Xn * (-1)L1 + Yn * (-1)L2$$

yielding 4 possible branch metric values:

$$\begin{aligned} BMn(0, 0) &= Xn + Yn \\ BMn(0, 1) &= Xn - Yn \\ BMn(1, 0) &= - Xn + Yn \\ BMn(1, 1) &= - Xn - Yn \end{aligned}$$

Each state at each Trellis stage also has another metric associated with it called the path metric (PM) that needs to be updated each trellis stage and is computed as follows:

PMj+1[i] = max(PMj[2i] + BM[i], PMj[2i+1] - BM[i])
PMj+1[i+32] = max(PMj[2i] – BM[i], PMj[2i+1] + BM[i]);
for i =0 to 31
where i is the index to the state and j the index to the Trellis stage.

The branch metric computation for 64 bits can be performed by a single extension instruction (EI_ACS64 in Code Listing 2). This extension instruction also adds the branch metric with the path metric of the previous stage, compares the path metrics of the two incoming pads, updates the maximum path metric, and finally selects the appropriate path. The extension instruction also performs all ACS operations for all the states at one Trellis stage concurrently (i.e., 32 butterfly operations in parallel). Path metrics are stored as internal states and as processing progresses through each Trellis stage, the function updates output registers with one bit for each state indicating the selected path (for a total of four bits for each state for an accumulated 4*64 = 256 bits for all states). 128-bit store instructions (WRAS128IU) move these bits to memory in just two cycles.

The final stage required to decode received symbols into data bits traces backwards through the Trellis along the most likely path. Typically, the length of traceback is 4 or 5 times the constraint length of the convolutional encoder. Depending upon the application, traceback may not begin until the entire data frame is received.

Traceback begins from a known last state, typically state 0. A final state of 0 is easily forced by sending an extra K-1 bit to transition all states to 0. The particular bit stored for each state determines which branch to traverse to move from stage j to stage j-1. By traversing the Trellis in reverse, the original input bit stream can be decoded.

An optimized Viterbi decoder can be implemented using a single extension instruction that performs traceback for four Trellis stages (VITERBI_TB in Code Listing 2). The function stores internal states to support the next round of traceback while outputting four bits of the decoded bitstream. Every second time VITERBI_TB is called, an 8-bit segment of the bitstream is stored back to memory.

This example illustrates the effectiveness of implementing complex and evolving algorithms with a software-configurable architecture. Note that similar performance acceleration can be achieved when implementing turbo codes and Read Solomon decoders. By accelerating the compute-intensive functions, it is possible to cost-effectively implement WiMAX, including PHY and MAC processing as well as TCP/IP stack software, all on a single 300 MHz device (see Table 1).

Current software defined radio applications rely heavily upon hardware to cost effectively implement many compute intensive functions in real-time. With the introduction of software-configurable processors, developers now have the ability to achieve exacting performance specifications through optimized hardware acceleration on a purely software programmable platform.

Code Listing 1
This program, written in C, makes use of user-defined extensive instruction EI_CONVEN for convolution encoding. WRAGET01 and WRAPUTI are stream load/store instructions.

```
for (i = 0; i < N/128; i++) {  // N is the total number of output bits in convolutional coding
  WRAGET0I(&wd, 8);     // load 64 bits of input data into wide register wd
  EI_CONVEN(&wd);              // extension instruction for convolutional coding
  WRAPUTI(wd, 16);       // store output bits (128 bits) to memory
   }
```

Code Listing 2
This C program uses extension instructions EI_ACS64 and VITERBI_TB to implement an optimized Viterbi decoder. WRAL16IU and WRAL128IU are 2-byte and 16-byte load instructions and WRAS16IU and WRAS128IU are 2-byte and 16-byte store instructions.

```
 for (i = 0; i < n/8; i++)  {  // n is the number of input bits at convolutional encoder
   for (j = 0; j < 4; j++) {   // for 4 Trellis stages
      WRAL16IU(&win, &indata_ptr, 2); // load (X, Y) input
      EI_ACS64(win,&lower_state,&upper_state); // perform ACS for all the states at 1 Trellis stage

 }
   WRAS128IU(lower_state, &ls_ptr, 32);   // store the input bits (for the lower 32 states) to memory
   WRAS128IU(upper_state, &us_ptr, 32):  // store the input bits (for the upper 32 states) to memory
 }

  /* perform traceback */
  for (i=0; i<n/8; i++) {
 WRAL128IU(&wa, &ls_ptr, -32);  // load the lower state bits
 WRBL128IU(&wb, &us_ptr, -32); // load the upper state bits
 VITERBI_TB(&wa, wb, mask);        // perform trace back for 4 stages
 WRAL128IU(&wa, &ls_ptr, -32);
 WRBL128IU(&wb, &us_ptr, -32);
 VITERBI_TB(&wa, wb, mask);    // perform trace back for another 4 stages
 WRAS8IU(wa, &decode_output, -1); // store decoded 8 bits to memory
}
```

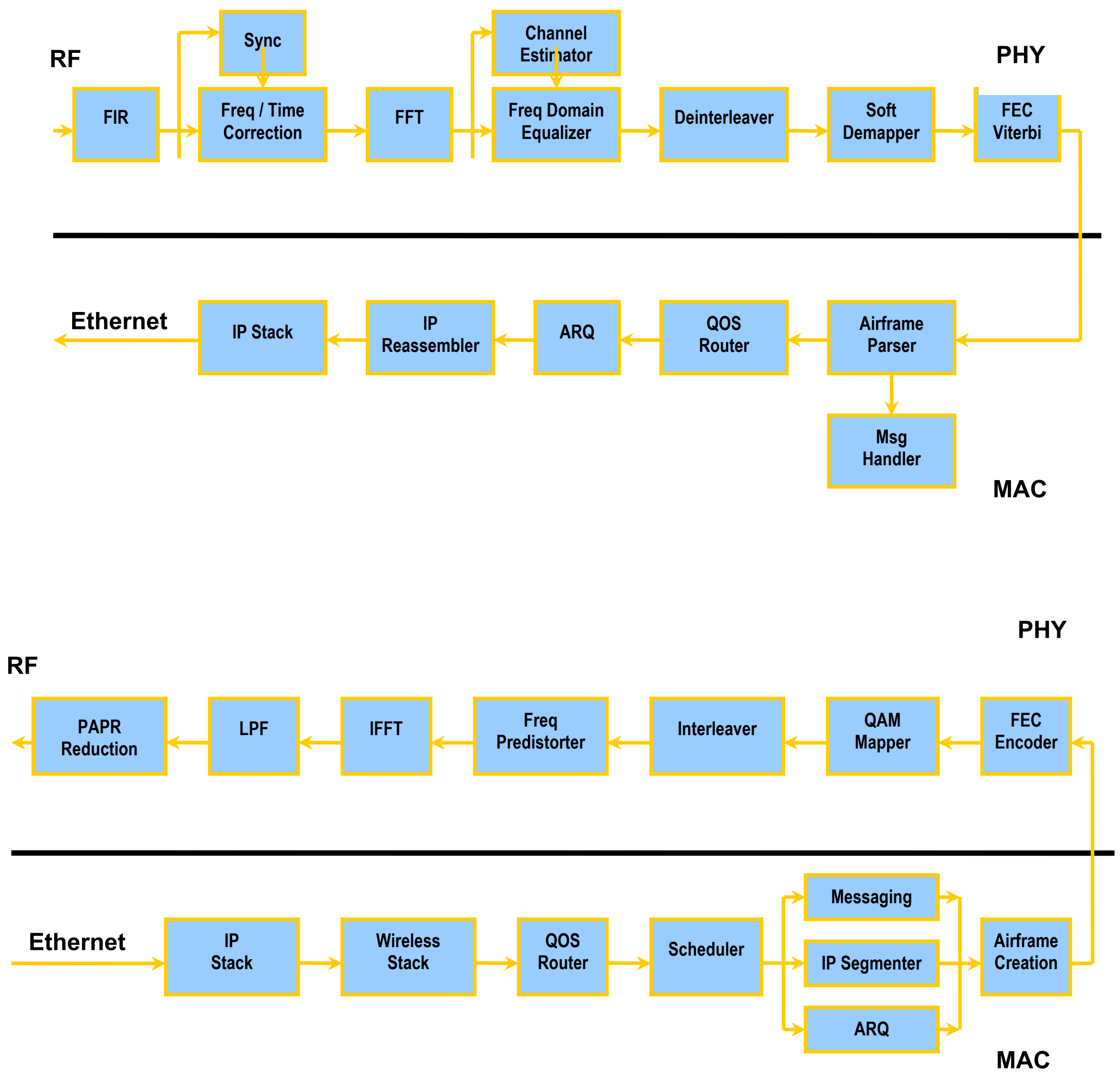
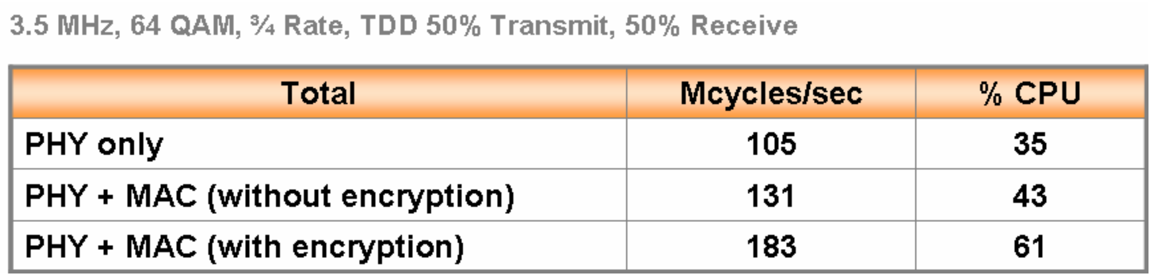


Figure 1 802.16 Receiver and Transmitter

Table 1 Compute Cycles on Stretch Software-Configurable Processor



3.5 MHz, 64 QAM, ¾ Rate, TDD 50% Transmit, 50% Receive

Total	Mcycles/sec	% CPU
PHY only	105	35
PHY + MAC (without encryption)	131	43
PHY + MAC (with encryption)	183	61

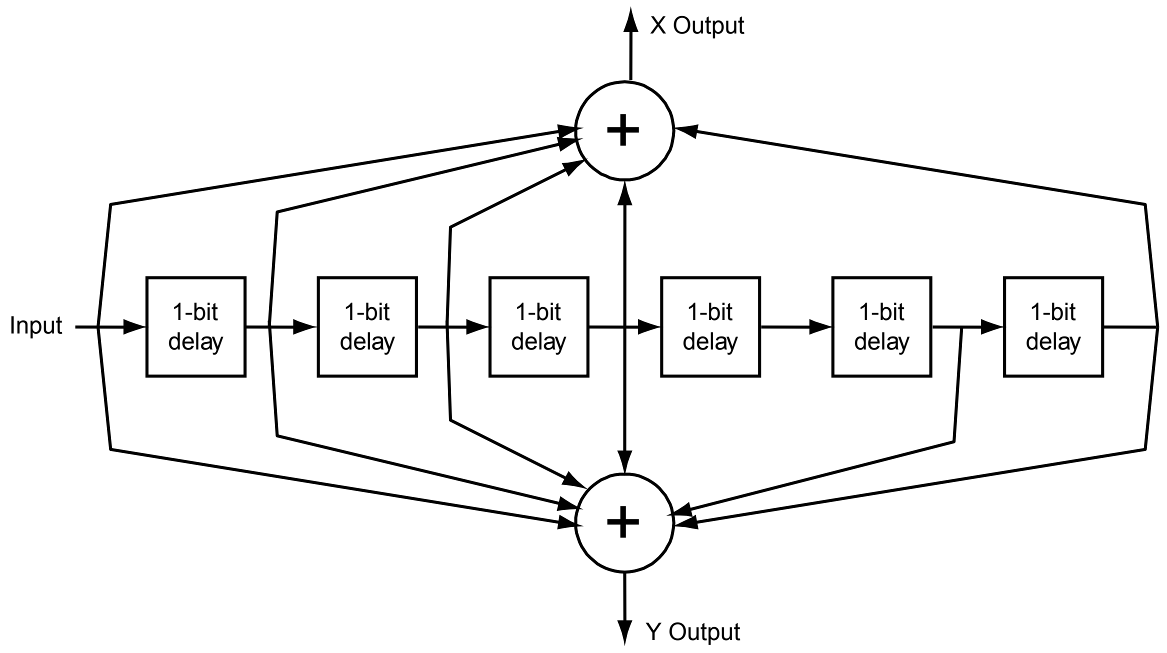


Figure 3 Convolutional Encoding with Constraint Length of 7 and ½ Rate

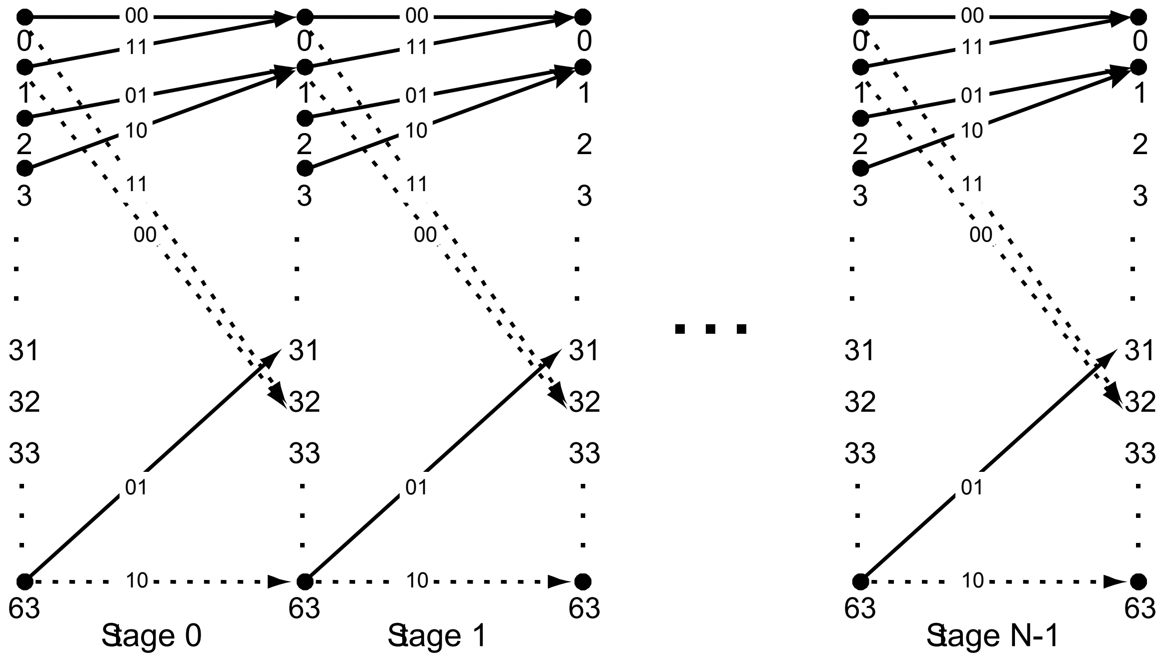
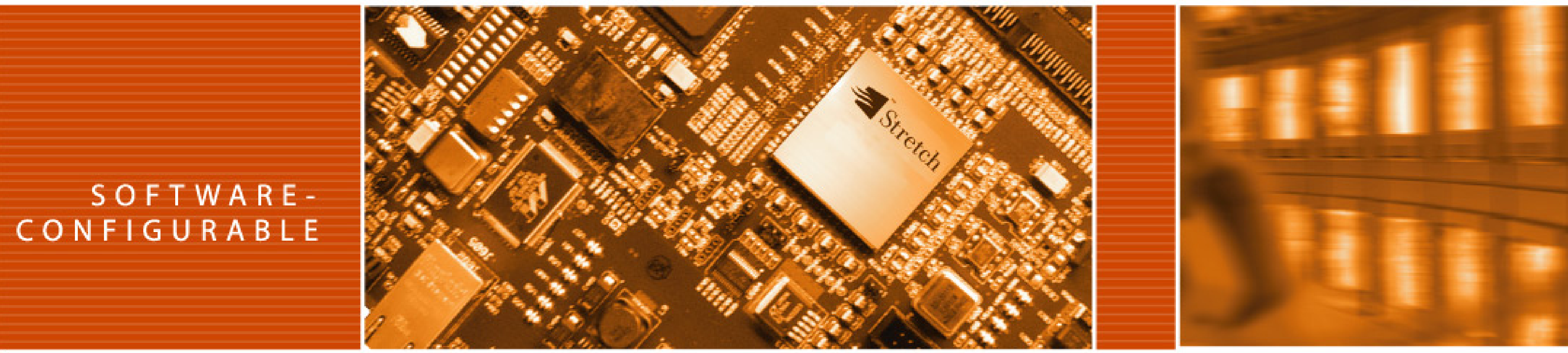


Figure 4 Trellis Diagram for ½ Rate and Constraint Length of 7

© 2005 Stretch Inc. All rights reserved.

Extending The Possibilities™



Implementing an 802.16 SDR Using a Software-Configurable Processor

SDR Forum , 2005

Bruce McNamara, Director of Applications Engineering at Stretch

with

Joe Hanson, Director of Business Development at Stretch

Michael Ji, Member Technical Staff at Stretch

Michael Leabman, Chief Technical Officer at Purewave Networks

© 2005 Stretch Inc. All Rights Reserved. Stretch, the Stretch logo, and Extending the Possibilities are trademarks of Stretch Inc.

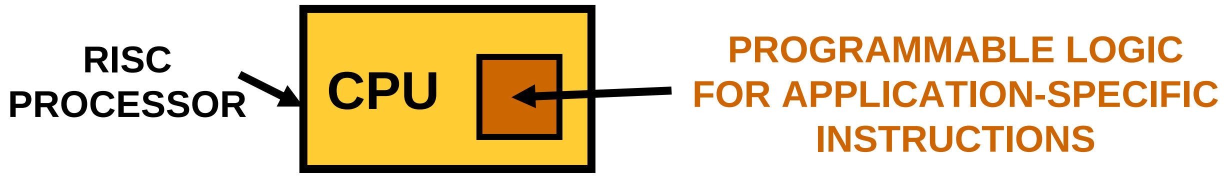
Agenda

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

Stretch™

© 2005 Stretch Inc. All Rights Reserved.

New Approach to Computing Applications

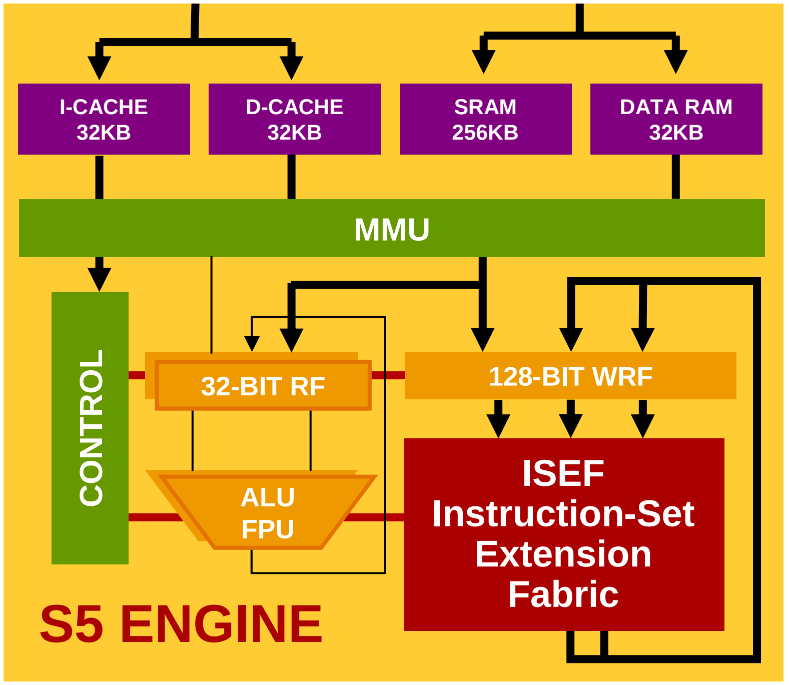


Software-Configurable Processor

- 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



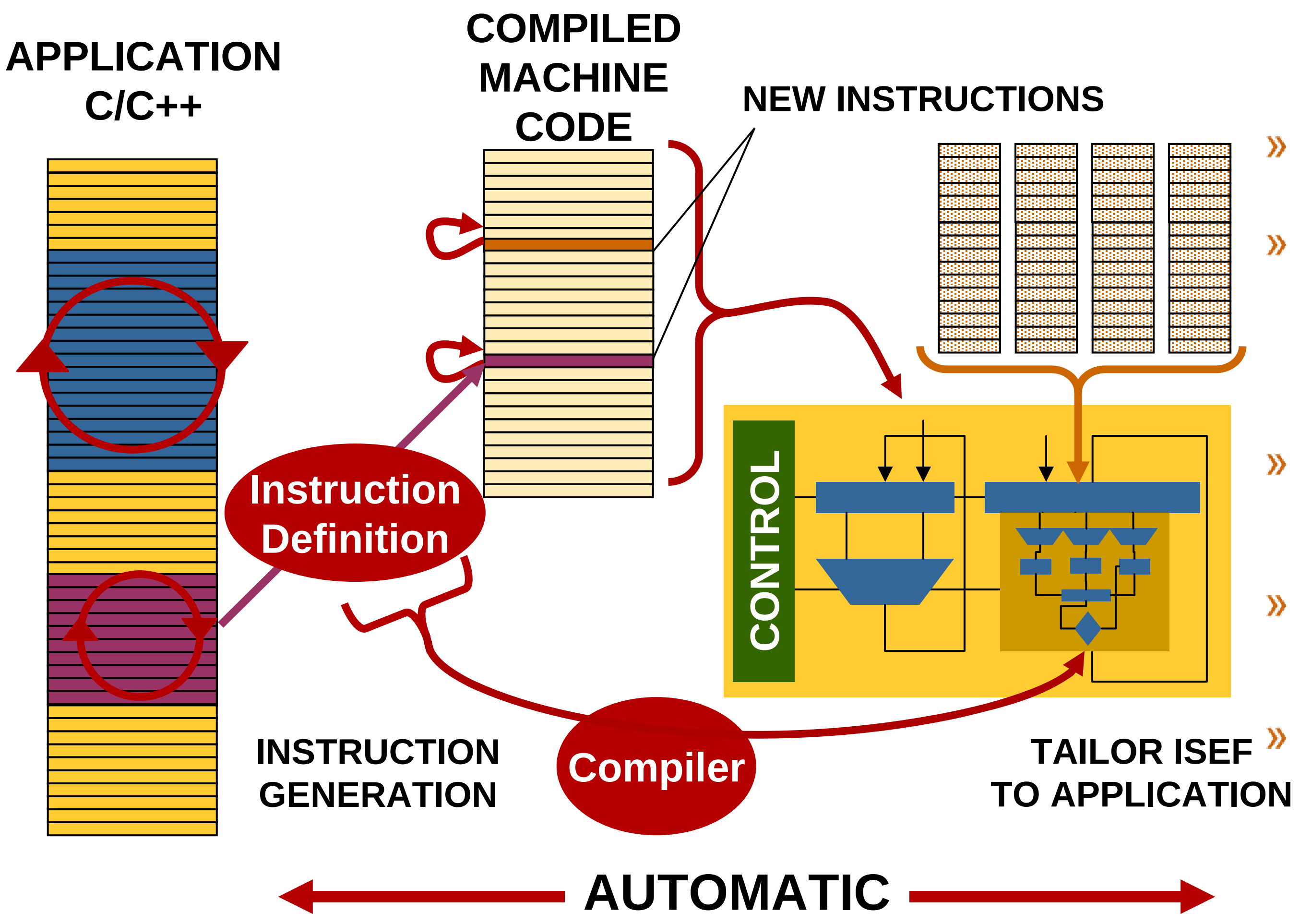
Wide Register File (WRF)
- 32 Wide Registers (WR)
- 128-bit Wide

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

© 2005 Stretch Inc. All Rights Reserved.

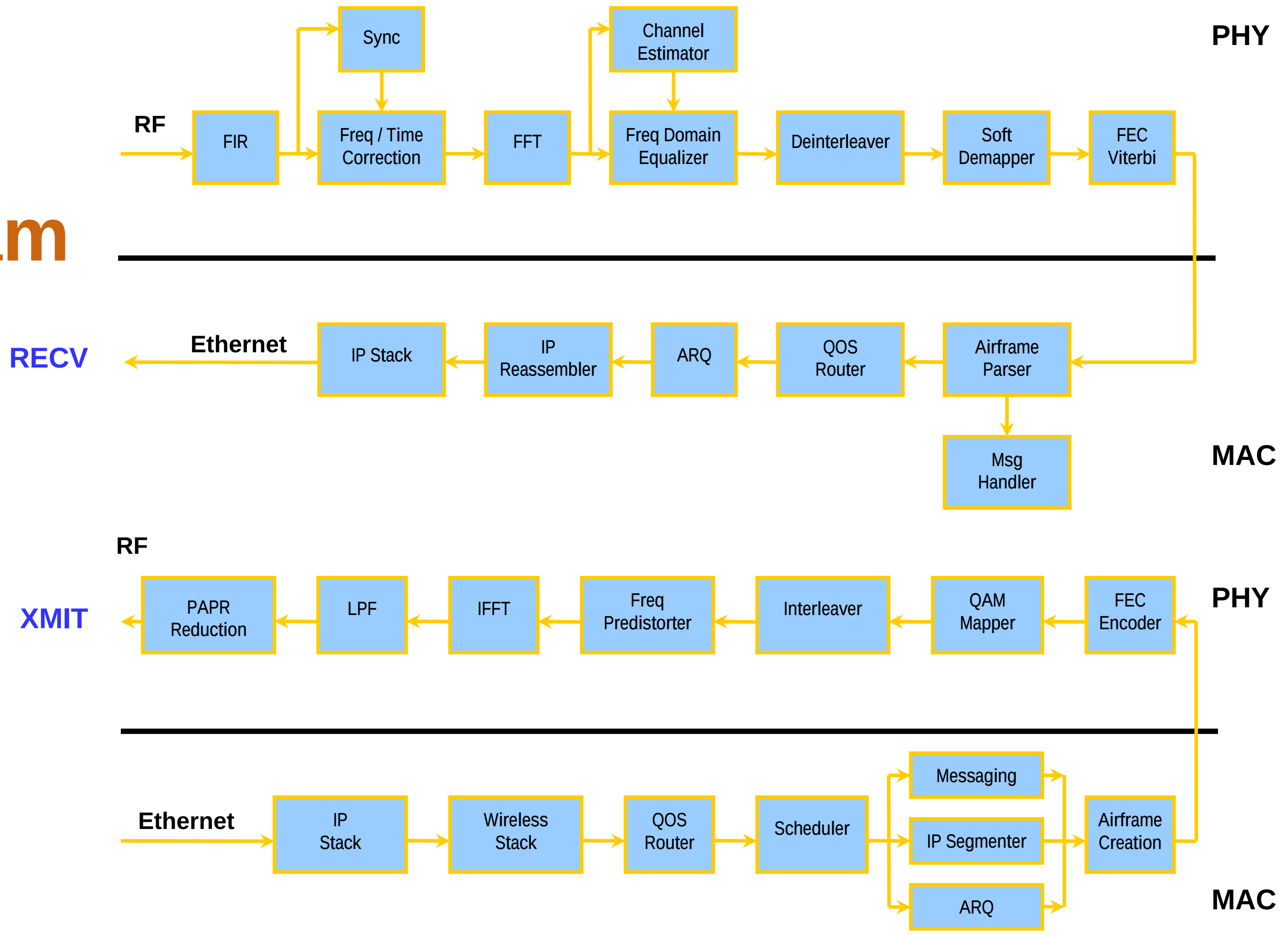
Software Acceleration Development Flow



- 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

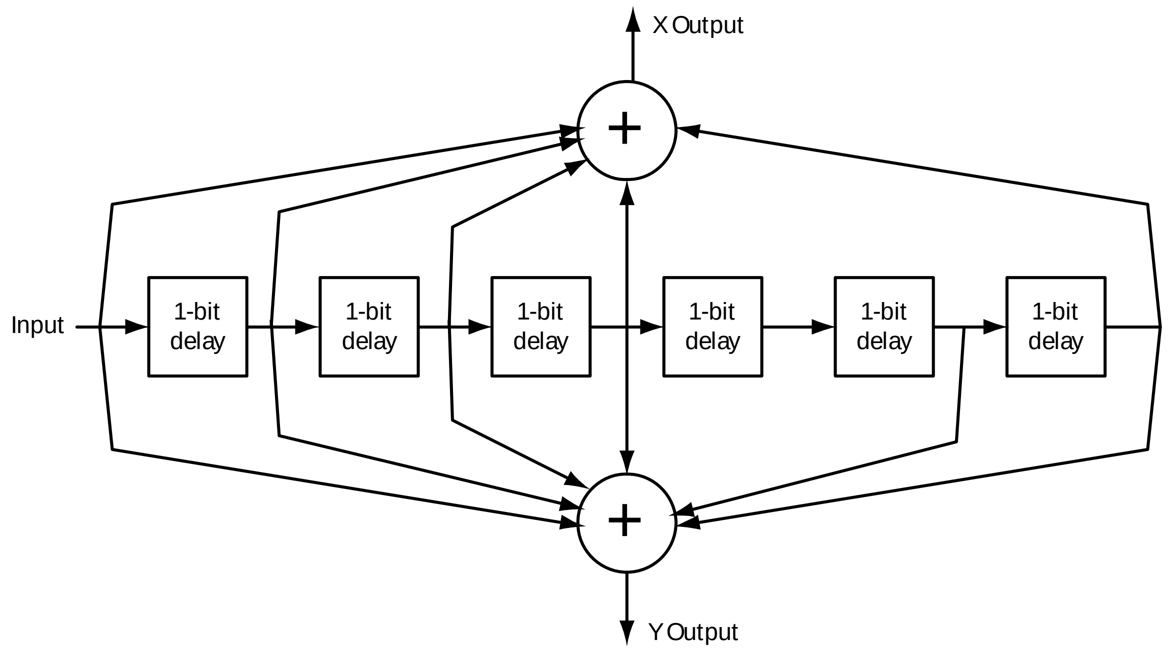
© 2005 Stretch Inc. All Rights Reserved.

OFDM Block Diagram



© 2005 Stretch Inc. All Rights Reserved.

Convolutional Encoder



Stretch™
© 2005 Stretch Inc. All Rights Reserved.

Optimizing the Convolutional Encoder

» Seven 1-bit ANDs and XORs per output bit

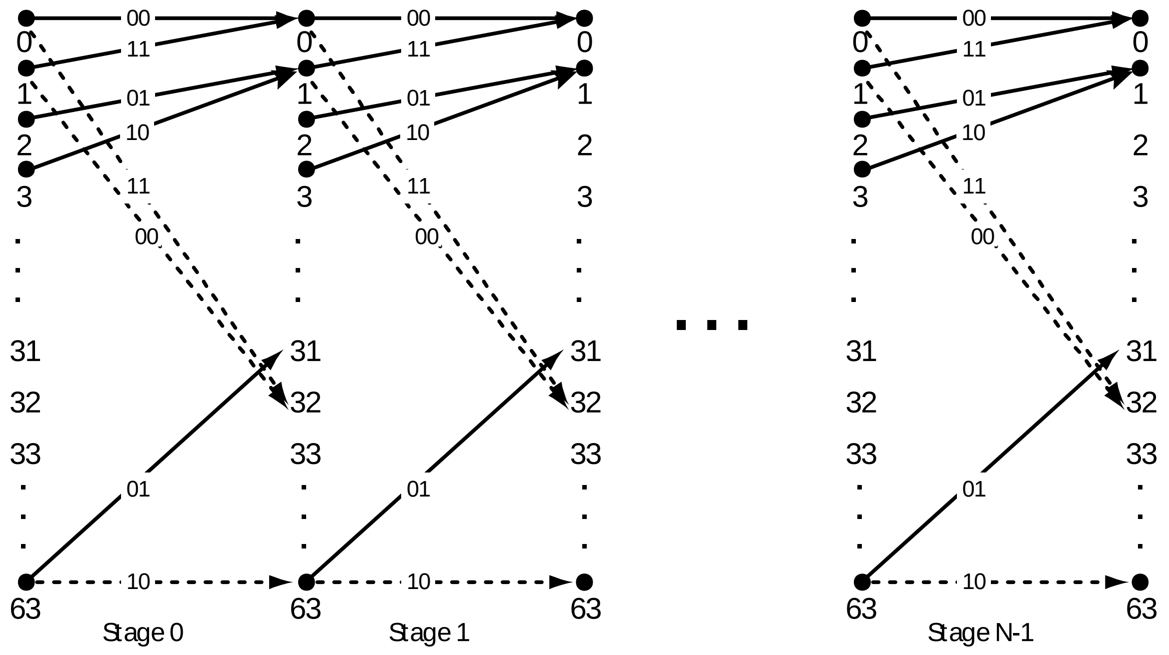
» Need puncturing to support 2/3, 3/4 and 5/6 rates
  › “Puncturing” means removing some bits from the bitstream
  › Pattern of removed bits depends on the rate

» The ISEF efficiently implements bitwise operations
  › AND, XOR, lookup table, mux (if-else), etc.
  › Uses only the resources needed, no more

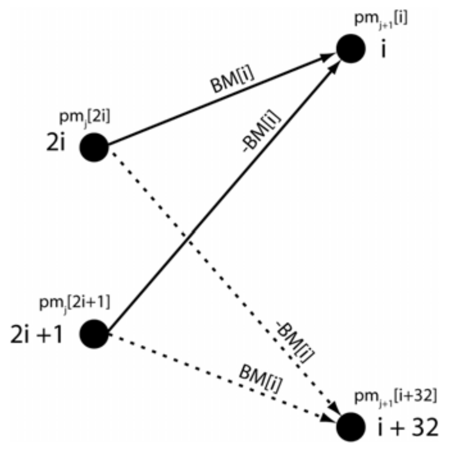
» Extension Instructions are easily defined in **C** or **C++**

Stretch™ © 2005 Stretch Inc. All Rights Reserved.

Viterbi Trellis Diagram



Add-Compare-Select "Butterfly"



© 2005 Stretch Inc. All Rights Reserved.

Optimizing the Viterbi Decoder

» Add-compare-select Part: For each of 64 states, must

1. Use soft decision inputs to compute branch metrics
2. Add/subtract branch metrics to state metrics (watch out for rollover)
3. Compare pairs of state metrics
4. Choose the larger metric and save a corresponding state bit

» A single Extension Instruction can do *all* of this for *all 64 states*

» Traceback Part: For each output bit, must determine the preceding state and save the corresponding bit

» A single Extension Instruction can do this for *4 states at a time*

© 2005 Stretch Inc. All Rights Reserved.

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));
   }
```

© 2005 Stretch Inc. All Rights Reserved.

```
    /* 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]  );
       }
    }
 }
```

© 2005 Stretch Inc. All Rights Reserved.

Extension Instructions for Viterbi ACS

```
#include <stretch.h>
#include "poly_para.h"
static const se_uint<1> parity[32] = {
   0, 1, 1, 0, 1, 0, 0, 1, 1, 0, 0, 1, 0, 1, 1, 0,
   1, 0, 0, 1, 0, 1, 1, 0, 0, 1, 1, 0, 1, 0, 0, 1
};
#define NS (1<<(K-1))                   /* the number of states */
#define NBF (NS/2)                      /* the number of butterflies */
#define SOFTBITS (6)                    /* the width of input soft bits */
#define PMBITS (11)                     /* path metric width in bits */
#define SIGNBIT (1 << (PMBITS - 1))
/* Internal state:
 *    PM:    Keep path metric for each state at each Trellis stage
 *    STATE: For each state, keep the 4 least significant bits in the
 *    state of conv encoding states. This covers 4 Trellis stages.
 */
static se_uint<PMBITS> PM[NS];
static se_uint<4>  STATE[NS];
SE_FUNC void _viterbi64_metric(SE_INST ACS64_INIT,
                               SE_INST ACS64,
                               WRA input, WRA *lowStates, WRB *highStates)
{
   int i, j, k, aIn, aOut, bIn, bOut;
   se_sint<SOFTBITS> x, y;
   se_sint<SOFTBITS+2> xy[2][2];
   se_sint<SOFTBITS+2> BM[NBF];
   se_uint<PMBITS> pmsubbm[NS], pmaddbm[NS];
   se_uint<4> newState[NS];
   x = input(SOFTBITS-1,0);                                             (1)
   y = input(SOFTBITS+7,8);
   /* compute the 4 branch metrics of one butterfly */
   xy[0][0] =  x + y;
   xy[0][1] =  x - y;
   xy[1][0] = -x + y;                                                   (2)
   xy[1][1] = -x - y;
   /* Assign all the Branch Metrics
    * The assignment depends on the polynomial generator
    */
   for (i = 0; i < NBF; i++) {
      j = integer(parity[integer((poly7[0]>>1) & i)]);
      k = integer(parity[integer((poly7[1]>>1) & i)]);                  (3)
      BM[i] = xy[j][k];
   }
```

© 2005 Stretch Inc. All Rights Reserved.

```
    /* Compute the path metrics associated with both outgoing branches
     * from state 2*i and 2*i+1.  ACS64_INIT initializes all metrics to 0,
     * except state 0 which is set to a big number
     */
    pmaddbm[2*0]   = ACS64_INIT ? 100 : (PM[2*0]   + BM[0]);                (4)
    pmsubbm[2*0]   = ACS64_INIT ?   0 : (PM[2*0]   - BM[0]);
    pmaddbm[2*0+1] = ACS64_INIT ?   0 : (PM[2*0+1] + BM[0]);
    pmsubbm[2*0+1] = ACS64_INIT ?   0 : (PM[2*0+1] - BM[0]);
    for (i = 1; i < NBF; i++) {
       pmaddbm[2*i]   = ACS64_INIT ? 0 : (PM[2*i]   + BM[i]);               (5)
       pmsubbm[2*i]   = ACS64_INIT ? 0 : (PM[2*i]   - BM[i]);
       pmaddbm[2*i+1] = ACS64_INIT ? 0 : (PM[2*i+1] + BM[i]);
       pmsubbm[2*i+1] = ACS64_INIT ? 0 : (PM[2*i+1] - BM[i]);
    }

    /* Compare & select butterflies
     * "Out-of-place" form: (2*i, 2*i+1) -> (i, i+32)
     * E.g.: (0,1)->(0,32)  (2,3)->(1,33) ... (62,63)->(31,63)
     * Because "out-of-place", need temp variables "newState"
     * until all "state"s are used
     */
    for (i = 0; i < NBF; i++) {
       se_uint<1> mux[2];
       aIn = 2*i;             aOut = i;                                     (6)
       bIn = 2*i + 1;         bOut = i + NBF;
       mux[0] = ((pmaddbm[aIn] - pmsubbm[bIn]) & SIGNBIT) ? 1 : 0;          (7)
       mux[1] = ((pmsubbm[aIn] - pmaddbm[bIn]) & SIGNBIT) ? 1 : 0;
       PM[aOut]   = mux[0] ? pmsubbm[bIn] : pmaddbm[aIn];                   (8)
       PM[bOut]   = mux[1] ? pmaddbm[bIn] : pmsubbm[aIn];
       newState[aOut] = mux[0] ? (se_uint<4>)ident(0x08 | (STATE[bIn]>>1))
                               : (se_uint<4>)(0x07 & (STATE[aIn]>>1));      (9)
       newState[bOut] = mux[1] ? (se_uint<4>)ident(0x08 | (STATE[bIn]>>1))
                               : (se_uint<4>)(0x07 & (STATE[aIn]>>1));
    }

    /* update the states since all old states have been read */
    for (i = 0; i < NBF; i++) {
       STATE[i] = newState[i];                                             (10)
       STATE[i+NBF] = newState[i+NBF];
    }
    /* Output all the states with 4 bits per state */
    *lowStates = *highStates = 0;
    for (i = 0; i < NBF; i++) {
       *lowStates  |= ((se_uint<128>)(STATE[i]    & 0xf) << (4 * i));      (11)
       *highStates |= ((se_uint<128>)(STATE[i+32] & 0xf) << (4 * i));
    }
}
```

© 2005 Stretch Inc. All Rights Reserved.

Extension Instructions for Viterbi Traceback

```
#include <stretch.h>
static se_uint<6> PRE_STATE;
/* For 64 state, use both *sa and sb for input bits
 * (state0 = sa(3,0) ... state63 = sb(127,124))
 */
SE_FUNC void viterbi_tb_func(
               SE_INST VITERBI_TB_INIT,
               SE_INST VITERBI_TB,
               WRA *sa, WRB sb)
{
   int i;
   se_uint<4> s[64];
   se_uint<4> cur_s;
   se_uint<6> ind, pre_s;
   se_uint<8> sout;

   for (i = 0; i < 32; i++) {
      s[i]    = (se_uint<4>) (*sa)(i*4 + 3, i*4);                     (12)
      s[i+32] = (se_uint<4>)    sb(i*4 + 3, i*4);
   }
   ind   = VITERBI_TB_INIT ? 0 : PRE_STATE;
   cur_s = s[integer(ind)];
   if (VITERBI_TB_INIT)
      PRE_STATE = 0;
   pre_s = (se_uint<6>)((se_uint<2>)PRE_STATE(1, 0), cur_s);            (13)

   /* Note that the 4 MSbits of the traversed state are the input bits.
    * Concatenating 2 traversed states of 4 stages apart produces 1 byte
    * of output.  These correspond to the bits originally input to the
    * convolutional encoder.
    */
   sout = ((se_uint<4>)PRE_STATE(5, 2), (se_uint<4>)pre_s(5, 2));       (14)
   *sa = ((se_sint<120>)0, sout);
   PRE_STATE = pre_s;                                                   (15)
}
```

Stretch™
© 2005 Stretch Inc. All Rights Reserved.

Viterbi Decoder Using Extension Instructions

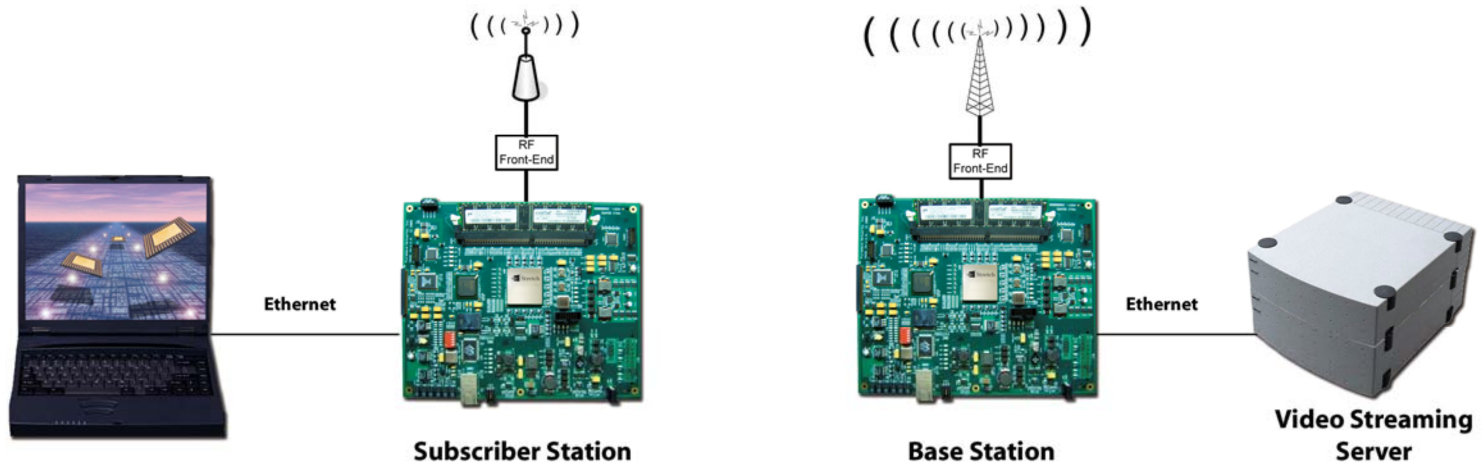
```
#include "ei_isef.h"
#include "poly_para.h"
#define NSTAGES (N/2)
#define BYTES_PER_STAGE (32/4)
u8 align_alloc(16, ".dram.data") pathHistory[NSTAGES * BYTES_PER_STAGE];
void opt_viterbi_decoder_64_12 (s8 *Input, u8 *Output, int nBits)
{
   int i, j;
   int nOutBits, nOutBytes;
   WRA in1, in2;
   WRA out1a, out2a;
   WRB out1b, out2b;
   WRA a0, a1, a2, a3, a4, a5, a6, a7;
   WRB b0, b1, b2, b3, b4, b5, b6, b7;
   u8 *pOut;
   s16 *ldx = (s16 *)Input;
   WRA *sta = (WRA *)&pathHistory[0];
   WRA *stb = (WRA *)&pathHistory[16];
   /* Assumes nBits is a multiple of 16 */
   /* Assumes tailbits (zeros) were added at the sending end */
   nOutBits = nBits >> 1;
   nOutBytes = nOutBits/8;                                            (1)
   /* setup the initial state inside ISEF */
   ACS64_INIT(in1, &out1a, &out1b);                                   (2)
   /* Prolog */
   for (j = 0; j < 4; j++) {                                          (3)
      WRAL16IU(&in1, &ldx, 2);
      ACS64(in1, &out1a, &out1b);
   }
   /* Main body */
   for (i = 0; i < nOutBytes - 1; i++) {                              (4)
      for (j = 0; j < 4; j++) {
         WRAL16IU(&in2, &ldx, 2);
         ACS64(in2, &out2a, &out2b);
      }
      WRAS128IU(out1a, &sta, 32);
      WRBS128IU(out1b, &stb, 32);
      for (j = 0; j < 4; j++) {
         WRAL16IU(&in1, &ldx, 2);
         ACS64(in1, &out1a, &out1b);
      }
      WRAS128IU(out2a, &sta, 32);
      WRBS128IU(out2b, &stb, 32);
   }
```

© 2005 Stretch Inc. All Rights Reserved.

```
   /* Epilog */
   for (j = 0; j < 4; j++) {                                                  (5)
      WRAL16IU(&in2, &ldx, 2);
      ACS64(in2, &out2a, &out2b);
   }
   WRAS128IU(out1a, &sta, 32);
   WRBS128IU(out1b, &stb, 32);
   WRAS128IU(out2a, &sta, 32);
   WRBS128IU(out2b, &stb, 32);

   /* Perform Traceback */
   sta -= 2;                                                                  (6)
   stb -= 2;
   pOut = &Output[nOutBytes - 1];                                             (7)

   WRAL128IU(&a0, &sta, -32);                                                 (8)
   WRBL128IU(&b0, &stb, -32);
   VITERBI_TB_INIT(&a0, b0);
   WRAS8IU(a0, &pOut, -1);                                                    (9)

   for (i = 1; i < nOutBytes; i++) {                                         (10)
      WRAL128IU(&a2, &sta, -32);
      WRBL128IU(&b2, &stb, -32);
      VITERBI_TB(&a2, b2);

      WRAL128IU(&a3, &sta, -32);
      WRBL128IU(&b3, &stb, -32);
      VITERBI_TB(&a3, b3);

      WRAS8IU(a3, &pOut, -1);
   }
}
```

Stretch™
© 2005 Stretch Inc. All Rights Reserved.

Demonstration System

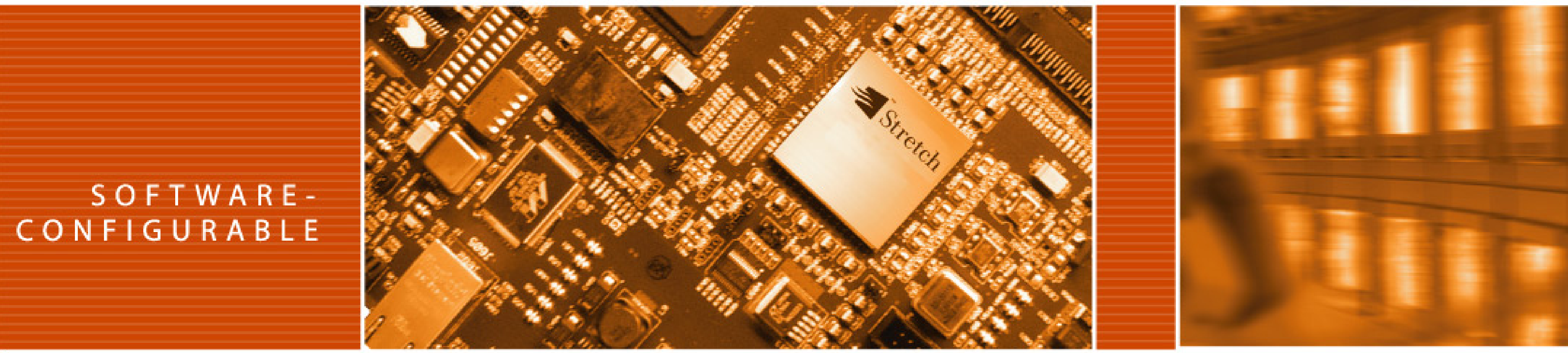


Stretch™
© 2005 Stretch Inc. All Rights Reserved.

Example Performance

7 MHz, 256-OFDM, single antenna TDD (50% UL, 50% DL)	Mcycles/sec	% CPU
Phy receive (16QAM, ¾ rate)	76	25
Phy transmit (64QAM, ¾ rate)	16	6
Total	92	31

Stretch™ © 2005 Stretch Inc. All Rights Reserved.



Implementing an 802.16 SDR Using a Software-Configurable Processor

SDR Forum , 2005

Bruce McNamara, Director of Applications Engineering at Stretch

with

Joe Hanson, Director of Business Development at Stretch

Michael Ji, Member Technical Staff at Stretch

Michael Leabman, Chief Technical Officer at Purewave Networks

© 2005 Stretch Inc. All Rights Reserved. Stretch, the Stretch logo, and Extending the Possibilities are trademarks of Stretch Inc.