What can TIE do for you?

Marcus Binning, Senior AE Manager, Europe (marcusb@cadence.com)
43276169 User Day
Hanover
Feb 9th 2016
Agenda

• Overview
• Building Hardware and architectures
• Integrating into the target toolchain – SW aspects of TIE
• Tools to help you
• Summary
Overview
TIE does not stand alone
Part of the Xtensa Configurable Processor Concept

Base Processor
Example controller templates

Pre-verified Options
Off-the-shelf DSPs, Interfaces, Peripherals, Debug, etc.

Application Based
Choose a processor template

Optional Customization
With pre-verified options and/or create your own

Iterate in minutes!

Complete Hardware Design
Pre-verified RTL EDA scripts Test suite

Every Tensilica processor shares the same base Xtensa instruction set

Advanced Software Tools
IDE C/C++ compiler Debuggers Simulators RTOSes

© 2016 Cadence Design Systems, Inc. All rights reserved.
Tensilica Scalable Product Portfolio

Providing the Most Efficient Processors in Multiple Markets
IoT, Mobile Phones, Storage/SSD, Networking, Video, Security, Cameras, Watches, Printers...

Control Processor
Energy & area efficient – the foundation of all Tensilica processors

Wide range of DSPs

- **Fusion**
  - IoT/Multi Purpose
  - Scalable DSP
  - Always-alert
  - Sensor processing
  - Audio/Video/Speech
  - Comm’s/Security

- **HiFi**
  - Audio/Voice/Speech
  - Encode & Decode
  - Voice trigger
  - Noise Reduction
  - Post-Processing
  - 150+ Codecs

- **ConnX**
  - Communications
  - Narrow to wide band Wireless
  - LTE/LTE-A, WiFi, Smart Grid
  - Infrastructure & Terminals

- **Vision**
  - Computer Vision/Imaging
  - Image processing and analytics
  - Video Pre- Post Processing

Custom

- **Custom ISA**
  - Application Specific
  - High Performance
  - Energy efficient
  - Application specific data types

Thousands of designs
Common Development Tools and 3rd Party Ecosystem

2B+ Cores per Year

© 2016 Cadence Design Systems, Inc. All rights reserved.
What affects the efficiency of Software?

- **Cycle counts** – the higher the longer it takes to complete
  - Targeting the right instructions crucially important

- **Sufficient local storage**
  - Access to registers very low energy cost. Insufficient registers can be a major efficiency loss

- **Efficient Memory System**
  - This varies greatly from system to system
  - With large software sets, a sophisticated cache based system may be appropriate
    - Support for Prefetch of Instructions and/or Data may be beneficial
  - With self-contained very small, low power systems the complete opposite may apply.
    - Local tightly coupled memory may be appropriate

- **Efficient Architecture must span all these**
  - (and more, of course – debug, trace ..)
Important Concepts about Xtensa + TIE
Adapting the Core to the need, not the need to the core ..

• Abstract
  – Don’t need detailed knowledge of how processors work
  – Many choices can be made using a mouse (pick from menus)
  – High level language (called TIE) used to express processor functionality
    – Available to end users

• Automated
  – Processor build is completely automated
  – Hardware (RTL) and software tools/models all created at the same time
    – Ensures consistency, eliminates need for user to verify e.g. ISS vs RTL
  – Extremely rich set of tools, libraries etc

• Powerful
  – Very wide range of architectural features can be added/modified
What can be customised?

- **The Instruction set .. Huge range of options**
  - Use a pre-defined one, or write your own. Could be 100’s of new instructions

- **The Register File infrastructure**
  - Many DSP cores have different data types – Audio samples, vectors of complex numbers – each needs different local storage.
  - Choices are (almost) unlimited in regfile width and depth

- **Memory System**
  - Access width to memory (up to 512 bits per load/store unit)
  - Caches, scratchpad memory, Prefetch, Multiple LoadStore units

- **Interfaces – Create new “HW” interfaces**
  - Different styles (FIFO, “lookup”, simple registered Wires)
  - Can be WIDE (up to 1024 bits each)
  - Can be PLENTIFUL (up to 1024 interfaces)
Many Different Architectures Possible
Engineers come up with different solutions to similar problems

- Only Xtensa offers this range of options and flexibility in choices
- Customers can further extend, customize to their specific cost, performance requirements
- One single unified tool set for all
Xtensa LX6 Block Diagram
Xtensa LX6 Block Diagram

**Customization**
- Multi-issue FLIX (automatically used by the C compiler)
- SIMD Instructions
- Compound and Fusion Instructions
- Multi-Cycle Execution Units
- Registers / Register Files with automatic C data-type support
- GPIO and Queue Interfaces
- Lookup Interfaces
- Wide (512-bit) load/store instructions
Building Hardware and Architectures
TIE is a very abstract language

• Don’t need to worry about too many low level details
  – How does my new instruction get encoded?
  – How does my new instruction get interfaced to the pipeline?
  – Will my new multi-cycle instruction work if it is interrupted half way through calculation?

• Answer to all these is … “Don’t worry, the tools handle it for you”!
  – (Well, technically it might be “It’s done by a combination of the Xtensa architecture and the TIE compiler” but that is not so snappy 😊)

• In TIE you describe things at a pretty high level
  – You have the option of specifying things more explicitly
    – E.g. encoding, specifying fields in the instruction word, you can describe your clever 11x17 multiplier at a very low level..
  – Normally handled much better by the TIE compiler.
    – There are a number of “pre-built” parameterisable datapath modules available
Useful constructs for building Hardware

• “Functions” – create hardware templates that increase readability, and re-use

```hierarchical
/*
* Function to zero pad 5 to 16 bits
*
* ***********************************************/

function [15:0] fn_zpad_5_16 ([4:0] a)
{
    assign fn_zpad_5_16 = {11'b0, a[4:0]} ;
}
```

• Semantics – create hardware shared between different operations
  – E.g. “expensive” hardware like multipliers and large adders/shifters
  – Common in complex DSPs to have many 10’s of flavours of multiplies
Hardware Construction in TIE

Suggested flow

• Create atomic operations first
• Decide if we need additional local storage (register files, state)
  – May be able to re-use existing register files
• Decide which operations should share hardware
• Decide whether we want a single issue machine or multi-issue
  – Increase software performance by issuing multiple instructions at the same time (“VLIW”)
• Create the multi-issue machine
  – This uses FLIX –”Flexible Length Instruction eXtensions”
• Create new interfaces if you need them

• Some examples follow
  – As an example we add some “population count” instructions to the Fusion core
Operations

• Note use of functions to aid readability
  – In this case the input operands come from an existing register file “AD_DR” that is defined in the Fusion ISA

```cpp
/***********************************************
 *
 * This operation performs popcounts across 32bit fields
 *
***********************************************

operation POP_COUNT64_32 {out AE_DR res, in AE_DR src}
{

    wire[31:0] w32_0, w32_1, res32_0, res32_1;
    assign {w32_1, w32_0} = src;

    assign res32_1 = fn_zpad_6_32(fn_popcount32(w32_1));
    assign res32_0 = fn_zpad_6_32(fn_popcount32(w32_0));

    assign res = {res32_1, res32_0};
}
```
Semantic

- Simple way to share hardware required for several instructions
  - Can also be multi-cycle

```cpp
.fromString("*
* Semantic to share HW across the POP_COUNT64 instructions
*

Semantic popc_sem { POP_COUNT64_8, POP_COUNT64_16, POP_COUNT64_32}
{
    wire[7:0] w8_0, w8_1, w8_2, w8_3, w8_4, w8_5, w8_6, w8_7;
    assign {w8_7, w8_6, w8_5, w8_4, w8_3, w8_2, w8_1, w8_0} = src;

    wire[3:0] p8_0 = fn_popcount8(w8_0);
    wire[3:0] p8_1 = fn_popcount8(w8_1);
    wire[3:0] p8_2 = fn_popcount8(w8_2);
    wire[3:0] p8_3 = fn_popcount8(w8_3);
    wire[3:0] p8_4 = fn_popcount8(w8_4);
    wire[3:0] p8_5 = fn_popcount8(w8_5);
    wire[3:0] p8_6 = fn_popcount8(w8_6);
    wire[3:0] p8_7 = fn_popcount8(w8_7);

    wire[4:0] p16_0 = TIEadd(p8_0, p8_1, 1'b0);
    wire[4:0] p16_1 = TIEadd(p8_2, p8_3, 1'b0);
    wire[4:0] p16_2 = TIEadd(p8_4, p8_5, 1'b0);
    wire[4:0] p16_3 = TIEadd(p8_6, p8_7, 1'b0);

    wire[5:0] p32_0 = TIEadd(p16_0, p16_1, 1'b0);
    wire[5:0] p32_1 = TIEadd(p16_2, p16_3, 1'b0);

    ...

    ...
}

... /* */
```

© 2016 Cadence Design Systems, Inc. All rights reserved.
Creating multi-issue machines

- Example of simple “VLIW” machine creation
- In reality things are much more complex
  - Can have multiple “format” declarations → very flexible machines
- TIE Compiler “wires up” all the hardware for you

```plaintext
format myflix1 64 {slot_a, slot_b, slot_c}

slot_opcodes slot_a {L32I, S32I}
slot_opcodes slot_b {ADDI}
slot_opcodes slot_c {ADD, SRAI}
```

64-bit FLIX instruction format with three slots
Each slot is given a unique name
List of operations that can execute in "slot_c"
Xtensa Processors

…as RTL or Finite State Machine (FSM) Replacement

• Use optimized Xtensa processors instead of RTL for new blocks
  – Reduce the verification effort and time
  – Replace the hardware control FSM with software on the processor
  – Get automatic RTL generation with fine-grained clock gating
  – Reprogram processor to adapt to upgrades and bugs in algorithms

• Create data paths similar to hardwired RTL data paths – really!
  – Multi-cycle, complex functional units
  – Custom, high bandwidth data/control connections to other blocks with predictable latencies
  – Automatic generation of pre-verified RTL
  – Verify only the input-output relationship on the interfaces

In RTL designs more than 90% of the bugs that cause re-spins are in the 10% of logic found in hard-wired control FSMs. Use programmable Xtensa processors to reduce re-spins!
SW Aspects of TIE
The domain of the “proto”

- Create the programming model for the user
- We recommend using ‘C’ (or C++ if you prefer) for target code (no asm!)
- New types defined to represent “real world” data
  - Vectors of complex numbers, vectors of audio data
- Compiler can be “taught” how to handle these new types
  - How to load/store/exchange them in the regfile
  - How to do pointer casting, and pointer arithmetic
  - How to do type conversions
  - How to do operator overloading * + - << >> > < >= etc
- Sometimes you need to use “intrinsics”
  - Direct calling of individual operations or sequences – to simplify things
- Example follows – using BBE32EP as the processor
Source Code
#ifdefs are folded by Xplorer for clarity

```c
#if XCHAL_HAVE_BBEN_EP_DUAL_PEAK_SEARCH
#endif

void bbe_dupeekmag_32b(int16_t *data_in, uint32_t *__restrict peaks,
                       int16_t *__restrict indices, int32_t size, int *ratioNorm)
{
    xc_vecn16 *__restrict data_ptr = (xc_vecn16 *)data_in;
    int32_t j, size16 = (size/SIMD_N);
    BBE_SETDUALMAX(0);
    xc_vecn40 normAccumW=0;
    xc_vecn40 normAccum=0;
    xc_vecn16 normAccumN=0;

    // Add loop_count pragma - this means that the number of peaks to be found
    // should be a multiple of 2 - a reasonable assumption - this is anyway vmax2
    xc_vecn16 seqInd=BBE_SEQN16();

    #pragma loop_count min=2, factor=2
    for(j=0; j<size16; j++)
    {
        xc_vecn16 vec0, vec1, vec2;
        xc_vecn40 wvec0;

        /* process two complex vectors */
        vec0 = data_ptr[2*j]; vec1 = data_ptr[2*j+1];

        #if PREC_40 // 40 bit
        wvec0 = BBE_MAGN16C(vec1,vec0); // find norm
        // Accumulate norm and weighted norm
        normAccum = normAccum + wvec0;
        normAccumW += seqInd * BBE_PACKN40(wvec0);
        seqInd = seqInd + (xc_vecn16)SIMD_N;
        #else // 16 bit[]
        
        //=else // 16 bit[]
    }

    // Dual max state updates
    BBE_DUALMAXWMNX32(wvec0, 0);
}
```

```c
// Weighted index average
int numNorm=BBE_RADDNX40(normAccumW);
#if PREC_40
    int denomNorm=BBE_RADDNX40(normAccum);
#else
    *ratioNorm=numNorm/denomNorm;
#endif

    // Peaks and indexes
    uint32_t maxpeak, secondpeak;
    bool selector, selector2;
    BBE_RDUALMAXUR(maxpeak,selector); // get the maximum and lane flag for max
    int32_t maxindex = BBE_SELMAXIDX(selector,1); // get corresponding index
    BBEMOVUDUALMAXT(selector); // replace max with max2 for that lane
    BBE_RDUALMAXUR(secondpeak,selector2); // get the maximum and lane flag for max2
    int32_t secondindex = BBE_SELMAXIDX(selector2,1); // get corresponding index

    peaks[0] = maxpeak;
    peaks[1] = secondpeak;
    indices[0] = maxindex;
    indices[1] = secondindex;
    x_bvecn16 normAccumW=0;
    x_bvecn16 normAccum=0;
    x_bvecn16 normAccumN=0;

    // Add loop_count pragma - this means that the number of peaks to be found
    // should be a multiple of 2 - a reasonable assumption - this is anyway vmax2
    x_bvecn16 seqInd=BBE_SEQN16();

    #pragma loop_count min=2, factor=2
    for(j=0; j<size16; j++)
    {
        x_bvecn16 vec0, vec1, vec2;
        x_bvecn40 wvec0;

        /* process two complex vectors */
        vec0 = data_ptr[2*j]; vec1 = data_ptr[2*j+1];

        #if PREC_40 // 40 bit
        wvec0 = BBE_MAGN16C(vec1,vec0); // find norm
        // Accumulate norm and weighted norm
        normAccum = normAccum + wvec0;
        normAccumW += seqInd * BBE_PACKN40(wvec0);
        seqInd = seqInd + (x_bvecn16)SIMD_N;
        #else // 16 bit[]
        
        //=else // 16 bit[]
    }

    // Dual max state updates
    BBE_DUALMAXWMNX32(wvec0, 0);
```
Compile time constant indicates certain features are available
- In this case there are specific peak search instructions which speed up the algorithm

Declaration uses standard 'C' types
- Makes it easy to test with pure 'C' implementations - any cast to machine specific types done inside function
- Use of __restrict same as normal 'C' (== no pointer aliasing)

New vector types specific to BBE32EP
- Type xb_vecNx16 is a vector of 16bit quantities
- In this machine 'N' == 16 so data_ptr is a pointer to 32B items
- Compiler is 'taught' through TIE how to do pointer casting, arithmetic etc

Declare some local variables
- Scope rules exactly the same as 'C'
- Type xb_vecNx40 is a 16-way vector of 40bit "accumulator" values

Variable initialised to zero across all lanes
- What is not explicitly shown is the type conversion from "int" (a constant, in this case '0' is always a 32-bit value in the Xtensa architecture) to xb_vecNx16. This has been defined in TIE to replicate the value across all lanes.
- Clearer would be "xb_vecNx40 normAccum = (xb_vecNx40) 0 ;"
Use of an intrinsic
- BBE_SEQNX16() is an atomic operation that initialises a vector to 0, 1, 2, 3 ... 15 across its lanes - very handy sometimes

This pragma helps the compiler with code generation
- If we know something about the loop iterations, then we can let the compiler know, in this case we don’t bother with odd numbers or 0, 1 so the loop preamble/cleanup code will be smaller

Start of a loop
- The compiler will decide whether it can infer a xero overhead loop (in this and most cases, "yes"). No need for user to consider whether it can or not.

We don’t explicitly use loads
- Compiler knows how to compute array offsets for data_ptr which is a pointer to xb_vecNx16[]. It will automatically schedule loads according to use in the loop, number of unrolls, register pressure etc

This is a specific intrinsic which maps to one operation that calculates magnitude² of a vector of N complex
- N complex requires 2N real values
- Compiler will actually decide which registers are used

The '+' operator is overloaded (through TIE) to be a vector add of xb_vecNx40 elements in this case
- Could have been written as "normAccum += wvec0;"
- Same thing to the compiler
Two atomic operations here
- The multiply '*' is on \( \text{xb\_vecNx16} \) types
- BBE\_PACK\* does a shift-round of 40-bit values to produce 16-bit
- The add '+' is on \( \text{xb\_vecNx40} \) types (multiply using '*' generates full-precision 40-bit results)
- The compiler 'knows' that a multiply followed by and add can be represented by an appropriate "MAC" instruction
- The compiler is taught through TIE how to handle ALL of the above scenario and infers BBE\_MULA\* (MAC) instructions

Each lane of seqInd is incremented by SIMD\_N
- As before, specific type conversion from int \( \rightarrow \text{xb\_vecNx16} \) is done with a cast
- SIMD\_N is constant 16 in this case. seqInd is therefore a running index of the current 16 complex vectors being processed in the loop
- normAccumW is a weighted normalised value

This is a specific operation that computes the highest two peaks, lane-by-lane
- It has it's own private storage not visible to the compiler
- Reduction processing is done after the loop
- Shown in next slide
This is a "reduction" operator
- i.e. it works across the vector - adding up all 16 40bit values and placing the result in the LS Lane

This is simple integer division

This is a Boolean type
- Type vboolN is a vector of 16 Boolean values
- Used for all sorts of comparison, and predicated operations

Computes the final max value and index

This special intrinsic moves data between the "private" DUAL PEAK SEARCH registers
- The "T" suffix means "Move the 'true' lanes" where the Boolean variable provides the predicate values
- There are also predicated versions of MOST basic operations - ALU, MAC, MOV etc

BBE_SEL* operations are "select" operations
- Wide range of data shuffling available on both special registers and the general purpose registers. This is on the private registers
Notes

• Source code is quite "boring" 😊
  – Few hints of the complexity of the underlying machine
  – No need to annotate the source code to indicate what the programmer thinks may be opportunities for Instruction Level Parallelism (ILP), loop unrolling etc
    – Yes, we can control the compiler with switches, but we don't need to put a lot of annotations in the source code

• No need to explicitly use instructions / intrinsics everywhere
  – In actual fact, most of the time you can let the compiler do all loads/stores, and many "simple" operations like +-* >> << > < & | ^ ...
    – They are all "overloaded" for many types through the TIE language
  – The source code is still pretty readable, and if we compile for Debug (no optimisation) we can easily step through and do source level debugging

• Use of "N-way" types and intrinsics means code is basically portable to other members of the BBE*EP family with different "N" (e.g. BBE64EP has N==32)

• When compiling with higher optimisation, the compiler will effectively software pipeline according to the architecture (next couple of slides)
Body of the Loop after compilation - Notes (i)

- In the next slides you will see the actual assembly code generated by the compiler.
- Note that there are many lines of the form:
  - `{ op1 ; op2 ; op3 ; op4 }`
- These are called "FLIX bundles" (or "FLIX packets") and represent a single cycle issue of multiple atomic operations (op1 .. op4 in this case).
- The operations can have different total latencies.
- They all enter the various pipelines together.
- They do not interact with each other except through architectural state.
  - This means it's possible to trace loop iterations as no "hidden data paths" exist from intra-stage flops.
  - This is a central concept of Xtensa.
  - Provides "safe" operation:
    - The hardware may insert a bubble if there is a data hazard - but the compiler will try to avoid these with loop unrolling and careful software pipelining.
• Code before the loop is not shown for brevity - basically part of the first iteration is outside to "prime the pipeline"

• 3 Instructions before the loop - 2 scalar, 1 "FLIX" which contains 2 loads (to each loadstore unit)
  ➢ Low order address interleaving means there will be no contention and no stall

• Zero over head loop instruction - sets up some specific hardware registers to control instruction fetch
  ➢ No branch penalties when executing the code between 0x4000a600 and 0x4000a636 inclusive

• We can see by the two uses of "BBE_MAGINX16C" that the loop has been unrolled by factor 2

• We can also see that the loop is resource bound in the "MAC" slot
  ➢ There are limits to the number of copies of large datapaths in the machine (user choice == CDNS choice here!)
  ➢ Compiler cannot do any better
Tools to help you
Several specific tools to help develop TIE code

• **TIE compiler**
  – Does all the “heavy lifting” transforming your abstract TIE code into real Hardware and extensions to the basic software toolchain
  – Fast, efficient, reliable.
  – Generates reports which reveal a lot of detail about your design

• **‘C’/C++ Compiler**
  – Can utilise complex machines very well, and allow you to program in a high level language

• **Xplorer**
  – GUI environment which can be used to front-end the tools
  – Debug the “hardware” of your TIE in a “software” environment
    – TIE wires view in the debugger can annotate the wires of your instructions/semantics as you step through your source code
  – Utilise “TIEprint” statements in your TIE code – “embedded printf in TIE”. Does not create hardware (just a simulation artefact) – very handy for tracking corner case bugs
  – Profiling tools – easily spot “non-efficient” pipelining of TIE instructions using the pipeline view
Debug View - TIE Wires
Profile View – Pipeline Window
To Summarise
Summary of Xtensa + TIE development platform

• It’s a basic ISA and set of tools that allows:
  – Fast creation of novel architectures
  – Complete toolchains and models “correct by construction” in an hour
  – Powerful development tools
  – Wide range of architectural features that can be included in the design

• Create wide range of cores from “Almost HW” to “Sophisticated General Purpose DSPs” ….
  – Calling at all intermediate stops on the way 😊

• It’s Fast and Easy to understand and use –
  – No difficult languages to learn, everything looks “pretty familiar”
Thank you for listening
cadence®