

What can TIE do for you ?

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

Agenda

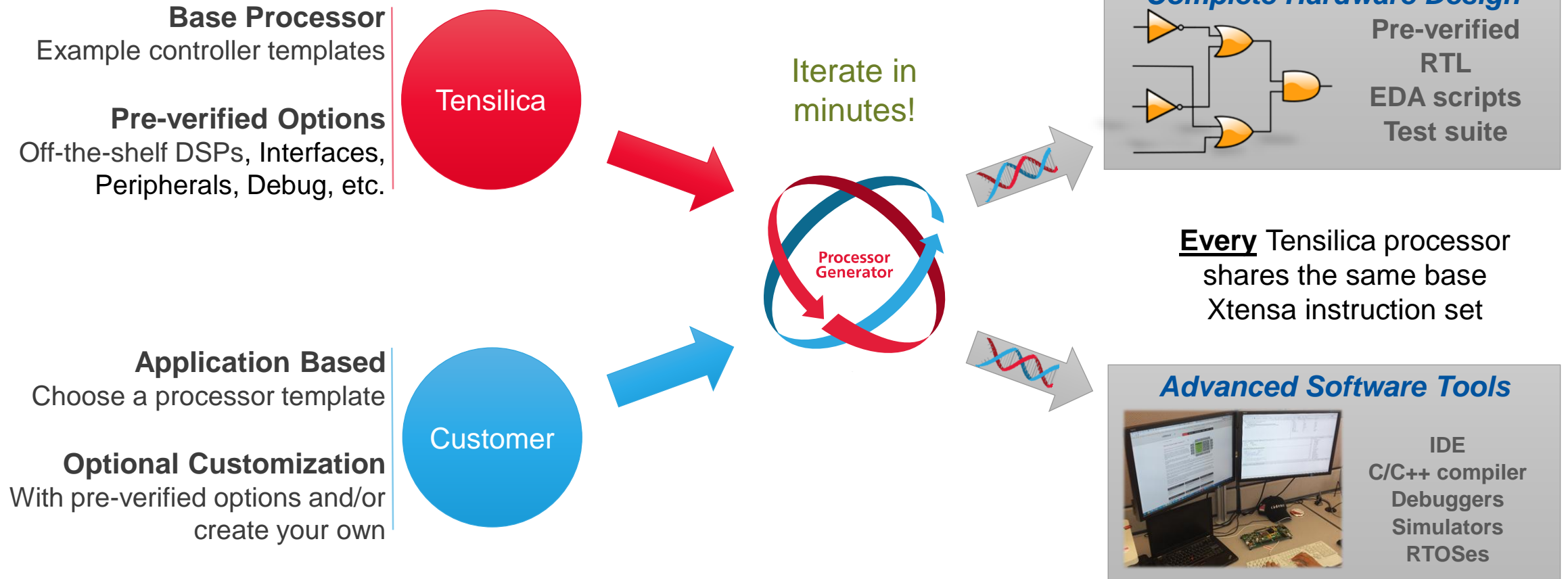
- Overview
- Building Hardware and architectures
- Integrating into the target toolchain – SW aspects of TIE
- Tools to help you
- Summary



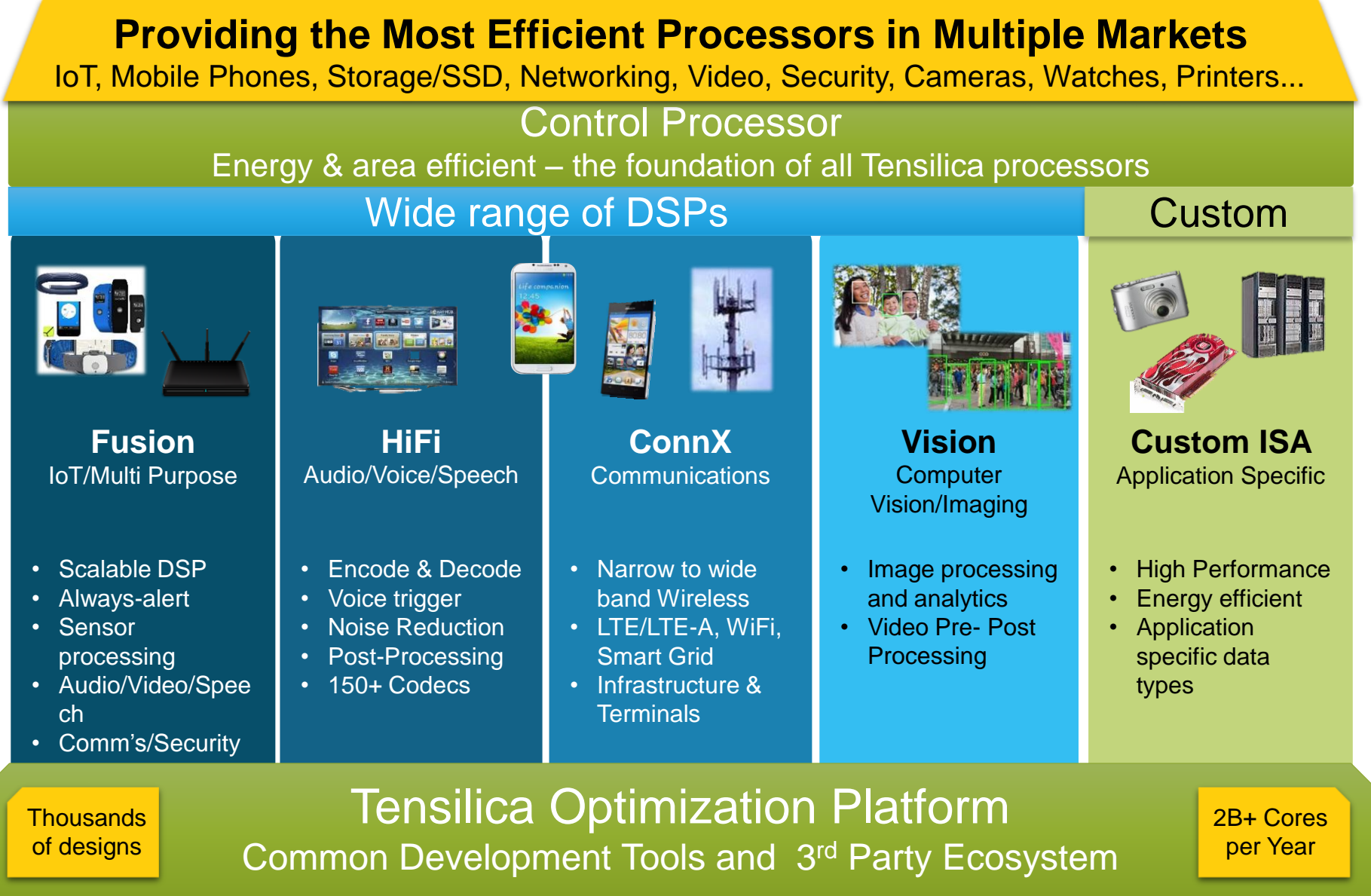
Overview

TIE does not stand alone

Part of the Xtensa Configurable Processor Concept



Tensilica Scalable Product Portfolio



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 ?

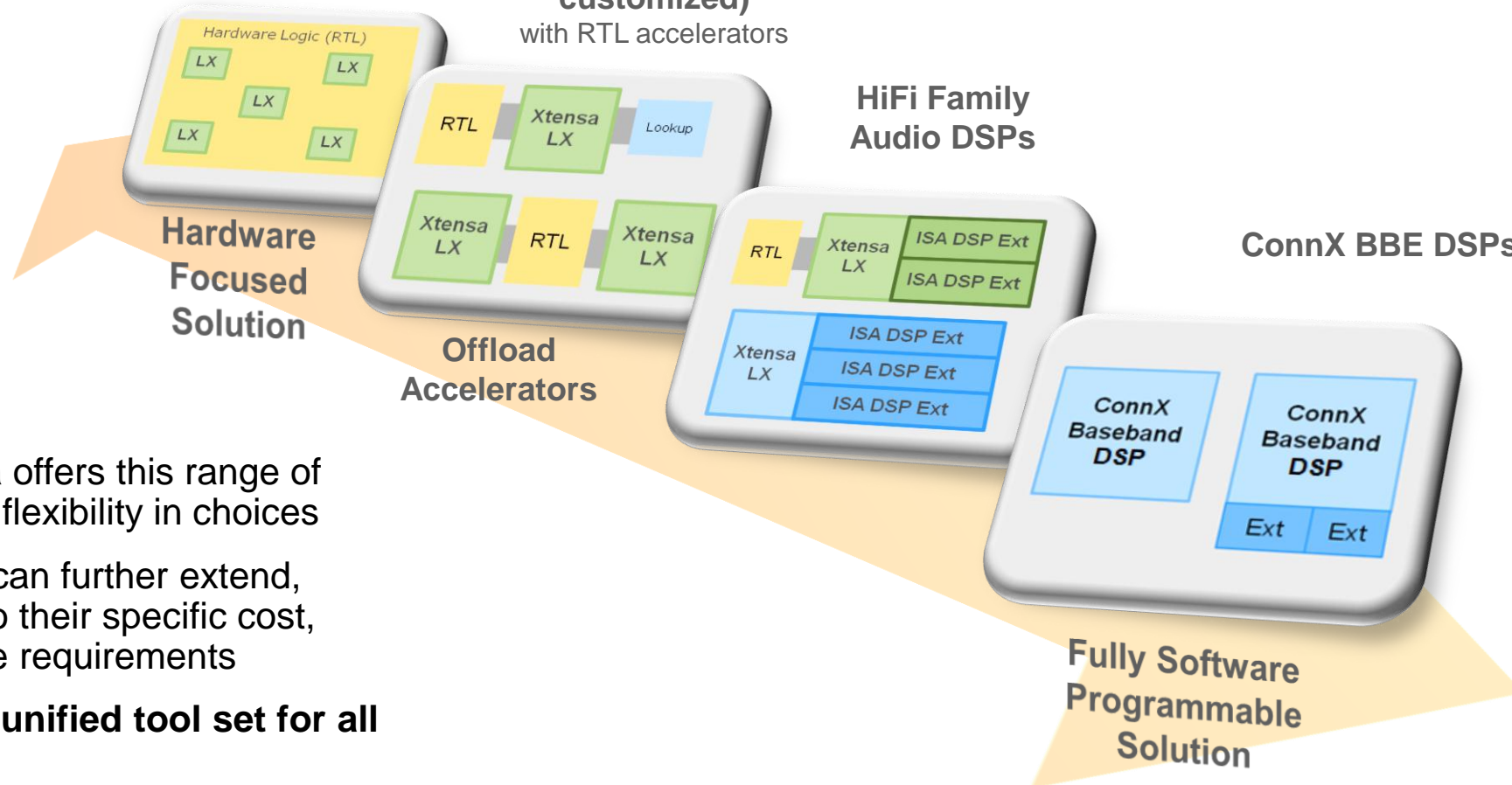
- **The Instruction set .. Huge range of options**
 - Use a pre-defined one, or write your own. Could be 100's of new instructions
- **The Register File infrastructure**
 - Many DSP cores have different data types – Audio samples, vectors of complex numbers – each needs different local storage.
 - Choices 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

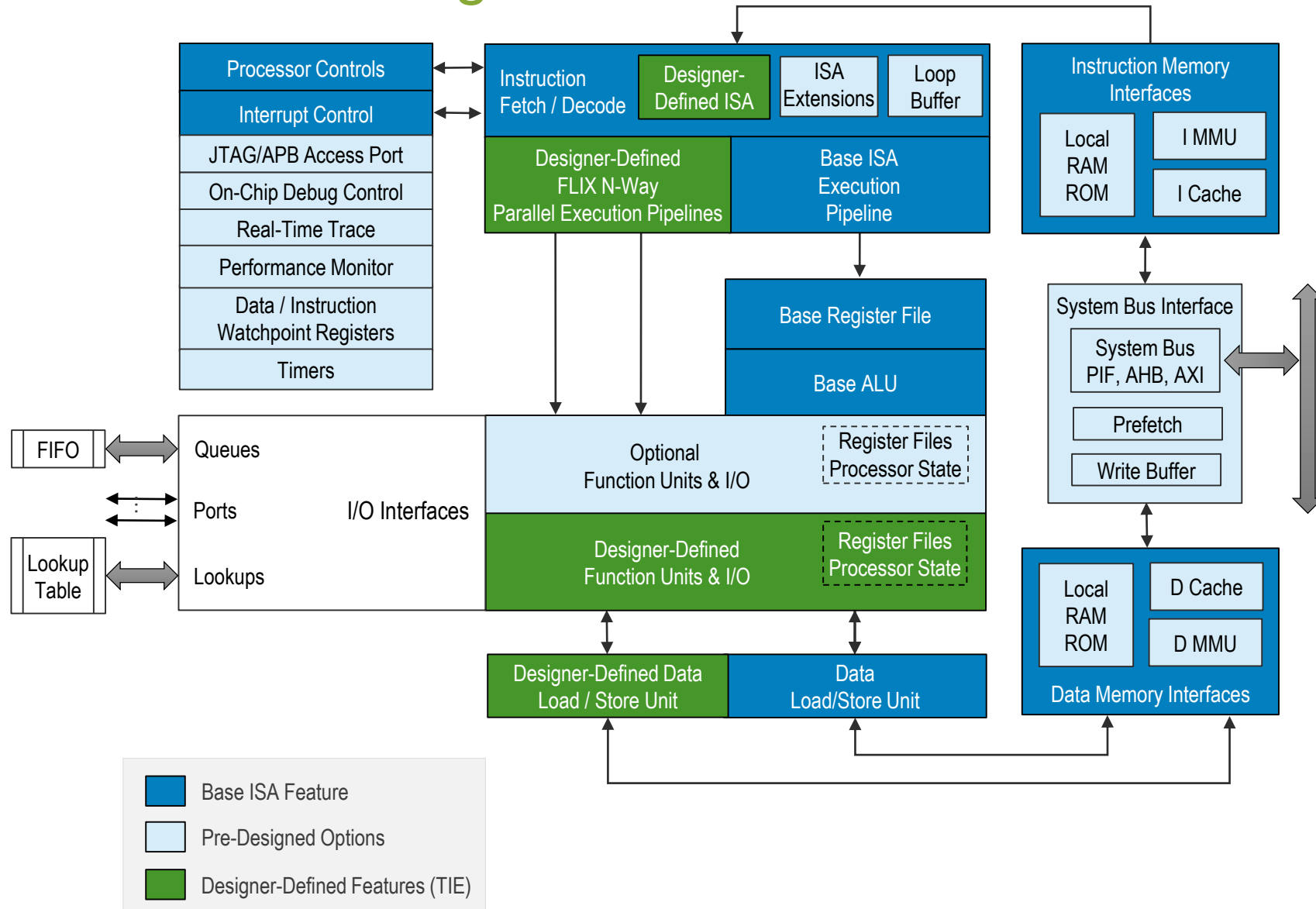
Deeply integrated Task Engines

(Xtensa performs specific task, HW developer creates FW to operate)

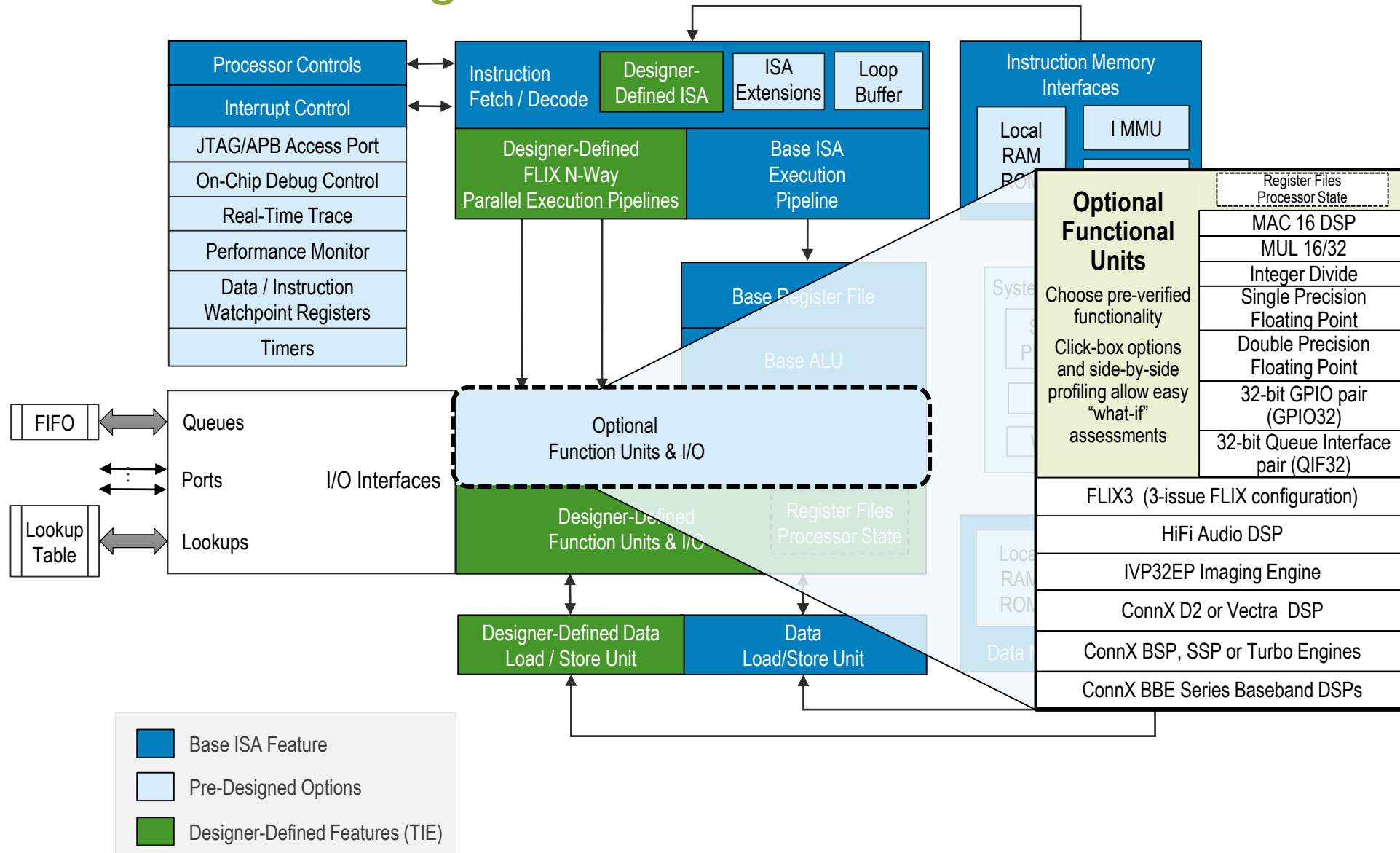


- Only Xtensa offers this range of options and flexibility in choices
- Customers can further extend, customize to their specific cost, performance requirements
- **One single unified tool set for all**

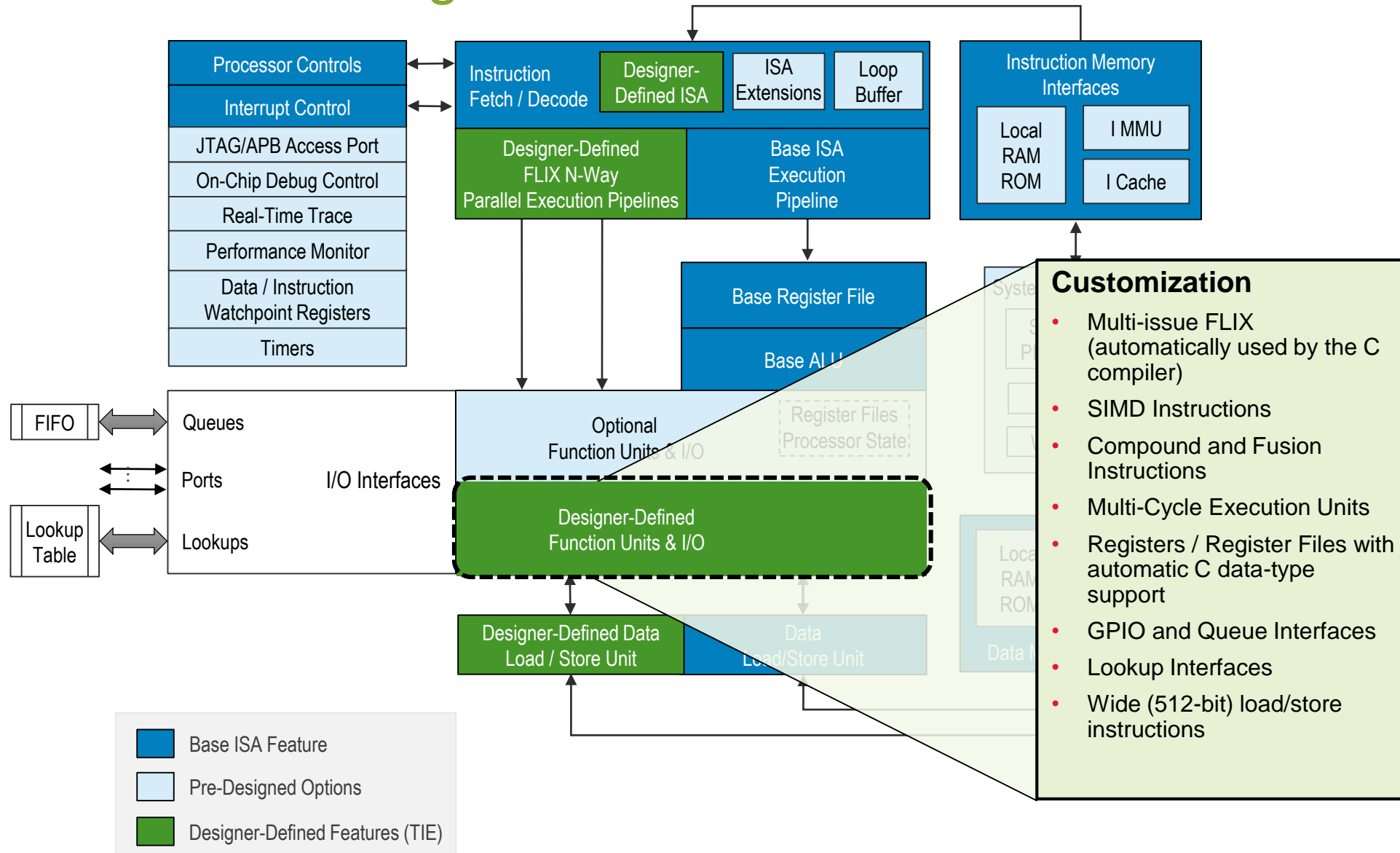
Xtensa LX6 Block Diagram



Xtensa LX6 Block Diagram



Xtensa LX6 Block Diagram





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

```
/******  
*  
* 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

```
/* *****  
 *  
 * 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

```
/*
 *
 * 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 ;

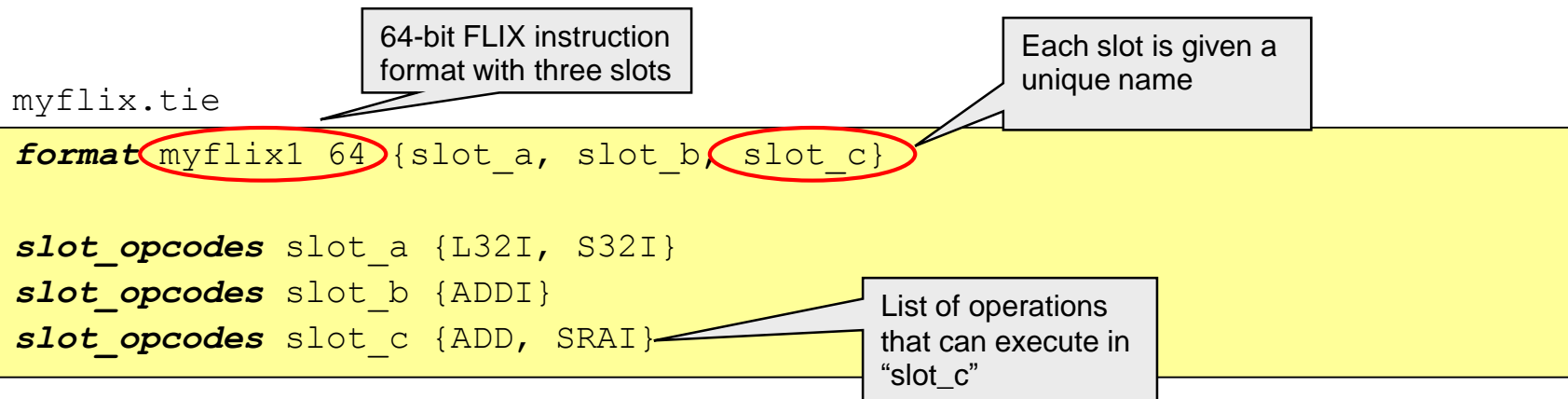
  wire[3:0] p8_0 = fn_popcount8(w8_0) ; wire[3:0] p8_1 = fn_popcount8(w8_1) ;
  wire[3:0] p8_2 = fn_popcount8(w8_2) ; wire[3:0] p8_3 = fn_popcount8(w8_3) ;
  wire[3:0] p8_4 = fn_popcount8(w8_4) ; wire[3:0] p8_5 = fn_popcount8(w8_5) ;
  wire[3:0] p8_6 = fn_popcount8(w8_6) ; wire[3:0] p8_7 = fn_popcount8(w8_7)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

  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) ;
}
```

Creating multi-issue machines

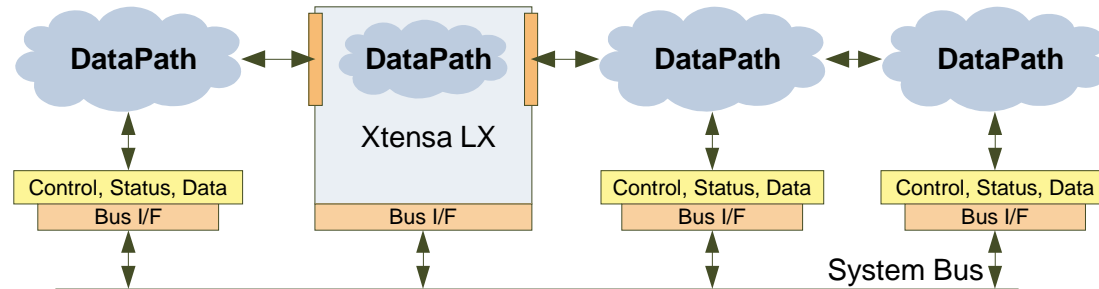
- Example of simple “VLIW” machine creation
- In reality things are much more complex
 - Can have multiple “format” declarations → very flexible machines
- TIE Compiler “wires up” all the hardware for you



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



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!

- 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



SW Aspects of TIE

SW Aspects of TIE extremely important

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

```
#if XCHAL_HAVE_BBEN_EP_DUAL_PEAK_SEARCH
void bbe_dualpeak_mag_32b(int16_t * data_in, uint32_t * __restrict peaks,
    int16_t * __restrict indices, int32_t size, int * ratioNorm)
{
    xb_vecNx16 * __restrict data_ptr = (xb_vecNx16 *)data_in;
    int32_t j, size_16 = (size/SIMD_N) ;
    BBE_SETDUALMAX(0);
    xb_vecNx40 normAccum=0;
    xb_vecNx40 normAccumW=0;
    xb_vecNx16 normAccum16=0;

    // Add loop_count pragmas - this means that the number of peaks to be found
    // should be a multiple of 2 - a reasonable assumption - this is anyway vmax2
    xb_vecNx16 seqInd=BBE_SEQNX16();
    #pragma loop_count min=2, factor=2
    for(j=0; j<size_16; j++)
    {
        xb_vecNx16 vec0,vec1, vec2;
        xb_vecNx40 wvec0;

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

        #if PREC_40 // 40 bit
            wvec0 = BBE_MAGINX16C(vec1,vec0) ; // find norm
            // Accumulate norm and weighted norm
            normAccum = normAccum + wvec0;
            normAccumW += seqInd * BBE_PACKSNX40(wvec0) ;
            seqInd=seqInd + (xb_vecNx16)SIMD_N;

            // Dual max state updates
            BBE_DUALMAXUWNX32(wvec0, 0);
        #else // 16 bit
        }
```

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

        // Peaks and indexes
        uint32_t maxpeak, secondpeak;
        vboolN selector, selector2;
        BBE_RBDUALMAXUR(maxpeak,selector); // get the maximum and lane flag for max
        int32_t maxindex = BBE_SELMAXIDX(selector,1); // get corresponding index

        BBE_MOVDUALMAXT(selector); // replace max with max2 for that lane
        BBE_RBDUALMAXUR(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 ;
        BBE_SETDUALMAX(0);
    }
#endif
```

Source Code - Explained (part 1)

```
#if XCHAL_HAVE_BBEN_EP_DUAL_PEAK_SEARCH
void bbe_dualpeak_mag_32b(int16_t * data_in, uint32_t * __restrict peaks,
    int16_t * __restrict indices, int32_t size, int * ratioNorm)
{
    xb_vecNx16 * __restrict data_ptr = (xb_vecNx16 *)data_in;
    int32_t j, size_16 = (size/SIMD_N);
    BBE_SETDUALMAX(0);
    xb_vecNx40 normAccum=0;
    xb_vecNx40 normAccumW=0;
    xb_vecNx16 normAccum16=0;

    // Add loop_count pragmas - this means that the number of peaks to be found
    // should be a multiple of 2 - a reasonable assumption - this is anyway vmax2
    xb_vecNx16 seqInd=BBE_SEQNX16();
    #pragma loop_count min=2, factor=2
    for(j=0; j<size_16; j++)
    {
        xb_vecNx16 vec0,vec1, vec2;
        xb_vecNx40 wvec0;

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

        #if PREC_40 // 40 bit
        wvec0 = BBE_MAGINX16C(vec1,vec0) ; // find norm
        // Accumulate norm and weighted norm
        normAccum = normAccum + wvec0;
        normAccumW += seqInd * BBE_PACKSNX40(wvec0) ;
        seqInd=seqInd + (xb_vecNx16)SIMD_N;

        // Dual max state updates
        BBE_DUALMAXUWNX32(wvec0, 0);
        #else // 16 bit
        #endif
    }
}
```

- 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 ;"

Source Code - Explained (part 2)

```
#if XCHAL_HAVE_BBE_N_EP_DUAL_PEAK_SEARCH
void bbe_dualpeak_mag_32b(int16_t * data_in, uint32_t * __restrict peaks,
    int16_t * __restrict indices, int32_t size, int * ratioNorm)
{
    xb_vecNx16 * __restrict data_ptr = (xb_vecNx16 *)data_in;
    int32_t j, size_16 = (size/SIMD_N);
    BBE_SETDUALMAX(0);
    xb_vecNx40 normAccum=0;
    xb_vecNx40 normAccumW=0;
    xb_vecNx16 normAccum16=0;

    // Add loop_count pragmas - this means that the number of peaks to be found
    // should be a multiple of 2 - a reasonable assumption - this is anyway vmax2
    xb_vecNx16 seqInd=BBE_SEQNX16();
    #pragma loop_count min=2, factor=2
    for(j=0; j<size_16; j++)
    {
        xb_vecNx16 vec0,vec1, vec2;
        xb_vecNx40 wvec0;

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

    #if PREC_40 // 40 bit
        wvec0 = BBE_MAGINX16C(vec1,vec0) ; // find norm
        // Accumulate norm and weighted norm
        normAccum = normAccum + wvec0;
        normAccumW += seqInd * BBE_PACKSNX40(wvec0) ;
        seqInd=seqInd + (xb_vecNx16)SIMD_N;

        // Dual max state updates
        BBE_DUALMAXUWNX32(wvec0, 0);
    #else // 16 bit
    }
}
```

- Use of an intrinsic
 - BBE_SEQNX16() is an atomic operation that initialises a vector to 0,1,2,3 ... 15 across it's 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 zero 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

Source Code - Explained (part 3)

```
#if XCHAL_HAVE_BBE_N_EP_DUAL_PEAK_SEARCH
void bbe_dualpeak_mag_32b(int16_t * data_in, uint32_t * __restrict peaks,
    int16_t * __restrict indices, int32_t size, int * ratioNorm)
{
    xb_vecNx16 * __restrict data_ptr = (xb_vecNx16 *)data_in;
    int32_t j, size_16 = (size/SIMD_N);
    BBE_SETDUALMAX(0);
    xb_vecNx40 normAccum=0;
    xb_vecNx40 normAccumW=0;
    xb_vecNx16 normAccum16=0;

    // Add loop_count pragmas - this means that the number of peaks to be found
    // should be a multiple of 2 - a reasonable assumption - this is anyway vmax2
    xb_vecNx16 seqInd=BBE_SEQNX16();
    #pragma loop_count min=2, factor=2
    for(j=0; j<size_16; j++)
    {
        xb_vecNx16 vec0,vec1, vec2;
        xb_vecNx40 wvec0;

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

        #if PREC_40 // 40 bit
        wvec0 = BBE_MAGINX16C(vec1,vec0); // find norm
        // Accumulate norm and weighted norm
        normAccum = normAccum + wvec0;
        normAccumW += seqInd * BBE_PACKSNX40(wvec0);
        seqInd=seqInd + (xb_vecNx16)SIMD_N;

        // Dual max state updates
        BBE_DUALMAXUWNX32(wvec0, 0);
        #else // 16 bit
    }
```

- Two atomic operations here

- The multiply '*' is on `xb_vecNx16` types
- `BBE_PACK*` does a shift-round of 40-bit values to produce 16-bit
- The add '+' is on `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` → `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 its own private storage not visible to the compiler
- Reduction processing is done after the loop
- Shown in next slide

Source Code - Explained (part 4)

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

// Peaks and indexes
uint32_t maxpeak, secondpeak;
vboolN selector, selector2;
BBE_RBDUALMAXUR(maxpeak,selector); // get the maximum and lane flag for max
int32_t maxindex = BBE_SELMAXIDX(selector,1); // get corresponding index

BBE_MOVDUALMAXT(selector); // replace max with max2 for that lane
BBE_RBDUALMAXUR(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 ;
BBE_SETDUALMAX(0);
}
#endif
```

- 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.

Body of the Loop after Compilation

```
4000a5eb    addi a3, a5, -2
4000a5ee    or a11, a2, a2
4000a5f1    { bbe_lvnx16_i v5, a2, -32;      bbe_lvnx16_i_n v6, a2, -64;      bbe_maginx16c wv0, v6, v8;      nop }
4000a5fd    loopgtz a3, 4000a63c <bbe_dualpeak_mag_32b+0xcc>
4000a600    { bbe_lvnx16_i v5, a11, 96;      bbe_lvnx16_i_n v6, a11, 64;      bbe_maginx16c wv3, v5, v6;      bbe_packsnx40 v1, wv3 }
4000a60c    { bbe_addnx16 v3, v4, v7;      bbe_lvnx16_i_n v0, a11, 32;      bbe_mulanx16 wv2, v3, v2;      bbe_dualmaxuwnx32 wv0, 0 }
4000a618    { bbe_lvnx16_ip v1, a11, 128;    bbe_packsnx40 v2, wv0;      bbe_mulanx16 wv2, v4, v1;      nop }
4000a624    { nop;                          nop;                          bbe_addnx40 wv1, wv1, wv0 }
4000a62a    { bbe_addnx16 v4, v3, v7;      nop;                          bbe_maginx16c wv0, v0, v1;      bbe_dualmaxuwnx32 wv3, 0 }
4000a636    { nop;                          nop;                          bbe_addnx40 wv1, wv1, wv3 }
4000a63c    { nop;                          bbe_packsnx40 v12, wv3;      bbe_maginx16c wv3, v5, v6; nop }
```

- 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

The screenshot displays an IDE interface for debugging a program. The main window shows the source code of `main.c` with the following content:

```
20 {
21     int i;
22     ae_int64 *vecPtr = (ae_int64 *) data;
23     ae_int64 *vecrPtr = (ae_int64 *) rPtr;
24     for(i=0; i<count/8; i++)
25     {
26         vecrPtr[i] = POP_COUNT64_8(vecPtr[i]);
27     }
28 }
29 void popcount16_TIE(uint16_t * __restrict rPtr, uint16_t *d
30 {
31     int i;
32     ae_int16x4 *vecPtr = (ae_int16x4 *) data;
33     ae_int16x4 *vecrPtr = (ae_int16x4 *) rPtr;
34     for(i=0; i<count/4; i++)
35     {
36         vecrPtr[i] = POP_COUNT64_16(vecPtr[i]);
37     }
38 }
39 void popcount32_TIE(uint32_t * __restrict rPtr, uint32_t *d
40 {
41     int i;
42     ae_int32x2 *vecPtr = (ae_int32x2 *) data;
43     ae_int32x2 *vecrPtr = (ae_int32x2 *) rPtr;
44     for(i=0; i<count/2; i++)
45     {
46         vecrPtr[i] = POP_COUNT64_32(vecPtr[i]);
47 }
```

The `popcount16_TIE` function is currently selected. The `Tie Wires` window shows the following data:

Name	Value
slot 1	
popc_sem	
→ _CPENABLE_in	2'h3
← _res_out	64'h000a000600050008
→ _src_in	64'hcf2d2946b404d878
↯ p8_0	4'h4
↯ p8_1	4'h4
↯ p8_2	4'h1
↯ p8_3	4'h4
↯ p8_4	4'h3
↯ p8_5	4'h3
↯ p8_6	4'h4
↯ p8_7	4'h6
↯ p16_0	5'h08
↯ p16_1	5'h05
↯ p16_2	5'h06
↯ p16_3	5'h0a
↯ p32_0	6'h0d
↯ p32_1	6'h10
↯ TIE_inst_cp1	1'h1
↯ w8_0	8'h78
↯ w8_1	8'hd8
↯ w8_2	8'h04
↯ w8_3	8'hb4
↯ w8_4	8'h46
↯ w8_5	8'h29
↯ w8_6	8'h2d
↯ w8_7	8'hcf
↯ wres	64'h000a000600050008
fn_popcount8(0)	

The `Disassembly` window shows the following assembly code for `popcount16_TIE`:

```
4000056c: entry a1, 32
4000056f: addi.n a5, a4, 3
40000571: movgez a5, a4, a4
40000574: srai a5, a5, 2
40000577: loopgtz a5, 40000586 <popcount16_TIE+0x1a>
4000057a: ae_l16x4.ip aed0, a3, 8
4000057d: { nop; pop_count64_16 aed0, aed0 }
40000583: ae_s16x4.ip aed0, a2, 8
40000586: retw.n
```

Profile View – Pipeline Window

The screenshot displays a software development environment with three main windows:

- Code Editor:** Shows the source code for `popcount16_ISAv2`. The function signature is `void popcount16_ISAv2(uint16_t * restrict rPtr, uint16_t *data, int count)`. The code includes comments and initialization of variables `ae_int16x4`.
- Profile Disassembly:** A table showing the execution count and address for each instruction. The first instruction at address `40000380` has a count of 3.
- Pipeline Window:** A visualization of the instruction pipeline. The x-axis represents pipeline stages (I, R, E, M, W) and the y-axis represents instructions. The diagram shows a diagonal sequence of colored blocks (I, R, E, M, W) representing the flow of instructions through the pipeline stages.

Count	Address
3	40000380
1	40000383
2	40000386
1	40000389
1	4000038c
1	40000392
1	40000398
1	4000039e
1	40000399

Address	Instruction
40000380	entry a1, 32
40000383	l32r a6, 400002c0 <_iram0_text_start>
40000386	ae_16x4.i aed6, a6, 24
40000389	ae_16x4.i aed5, a6, 32
4000038c	{ ae_16x4.i aed4, a6, 16; addi a5, a4, 3 }
40000392	{ ae_16x4.i aed3, a6, 8; movgez a5, a4, a4 }
40000398	{ ae_16x4.i aed2, a6, 0; srai a5, a5, 2 }
4000039e	loopgtz a5, 400003fb <popcount16_ISAv2+0x7b>
400003a1	ae_16x4.ip aed1, a3, 8
400003a4	{ ae_srai16 aed7, aed1, 1; nop }
400003aa	{ nop; ae_and aed7, aed7, aed2 }
400003b0	{ nop; ae_sub16 aed1, aed1, aed7 }
400003b6	{ ae_srai16 aed1, aed1, 2; ae_and aed0, aed1, aed3 }
400003bc	{ nop; ae_and aed1, aed1, aed3 }
400003c2	{ nop; ae_add16 aed0, aed0, aed1 }
400003c8	{ ae_srai16 aed1, aed0, 4; nop }
400003ce	{ nop; ae_add16 aed0, aed0, aed1 }
400003d4	{ nop; ae_and aed0, aed0, aed4 }
400003da	{ nop; ae_mul16x4.l aed1, aed0, aed5 }
400003e0	{ nop; ae_mul16x4.h aed0, aed0, aed5 }
400003e6	ae_srai32 aed1, aed1, 8
400003e9	ae_srai32 aed0, aed0, 8
400003ec	{ nop; ae_sel16i aed0, aed0, aed1, 8 }
400003f2	{ nop; ae_and aed0, aed0, aed6 }
400003f8	ae_s16x4.ip aed0, a2, 8
400003fb	retw.n

The background of the slide features a complex, isometric grid of blue lines. Overlaid on this grid are several semi-transparent, 3D rectangular blocks of varying sizes and orientations, creating a sense of depth and technical precision. The overall color palette is dominated by various shades of blue, from deep navy to light cyan.

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

cā dence[®]