Adding Custom Instructions to Tensilica DSPs
aka “The Modularity of TIE”

Tensilica Day, Hanover
February 2017
Marcus Binining Senior Application Engineering Manager, EMEA

cadence®

Why bother tuning your DSP architecture ?

What do ***your*** customers care about ?

- Performance Efficiency
- Power Efficiency
- Energy Efficiency

Introduction / Agenda

- We explore how to make TIE operations re-usable on different DSPs
  - writing in a modular, parameterisable manner
  - Varying SIMD width, operand sources, programming model (types etc)

- We explore how to make ‘C’ code more portable
  - Exploit TIE intrinsics in different DSP environments
  - Use the HAL layer to make code less “configuration dependent” and more portable

- Finally, we compare performance of simple TIE instructions across different architectures
  - SIMD width
  - Memory width
  - Memory bandwidth

Tensilica - Full Development Automation

Generates RTL + SW Development Tools

Base Processor
Example controller templates

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

Tensilica

Processor Optimization

Application Based
Choose a processor template

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

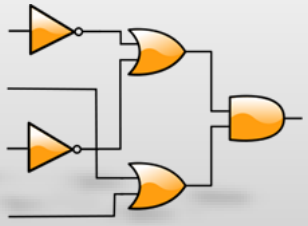
Customer

Processor Generator

Iterate in Minutes!

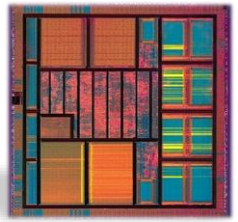
Processor with all customizations

Complete Hardware Design



Pre-verified
RTL
EDA scripts
Test suite

Use standard ASIC/COT design techniques and libraries for *any* IC fabrication process



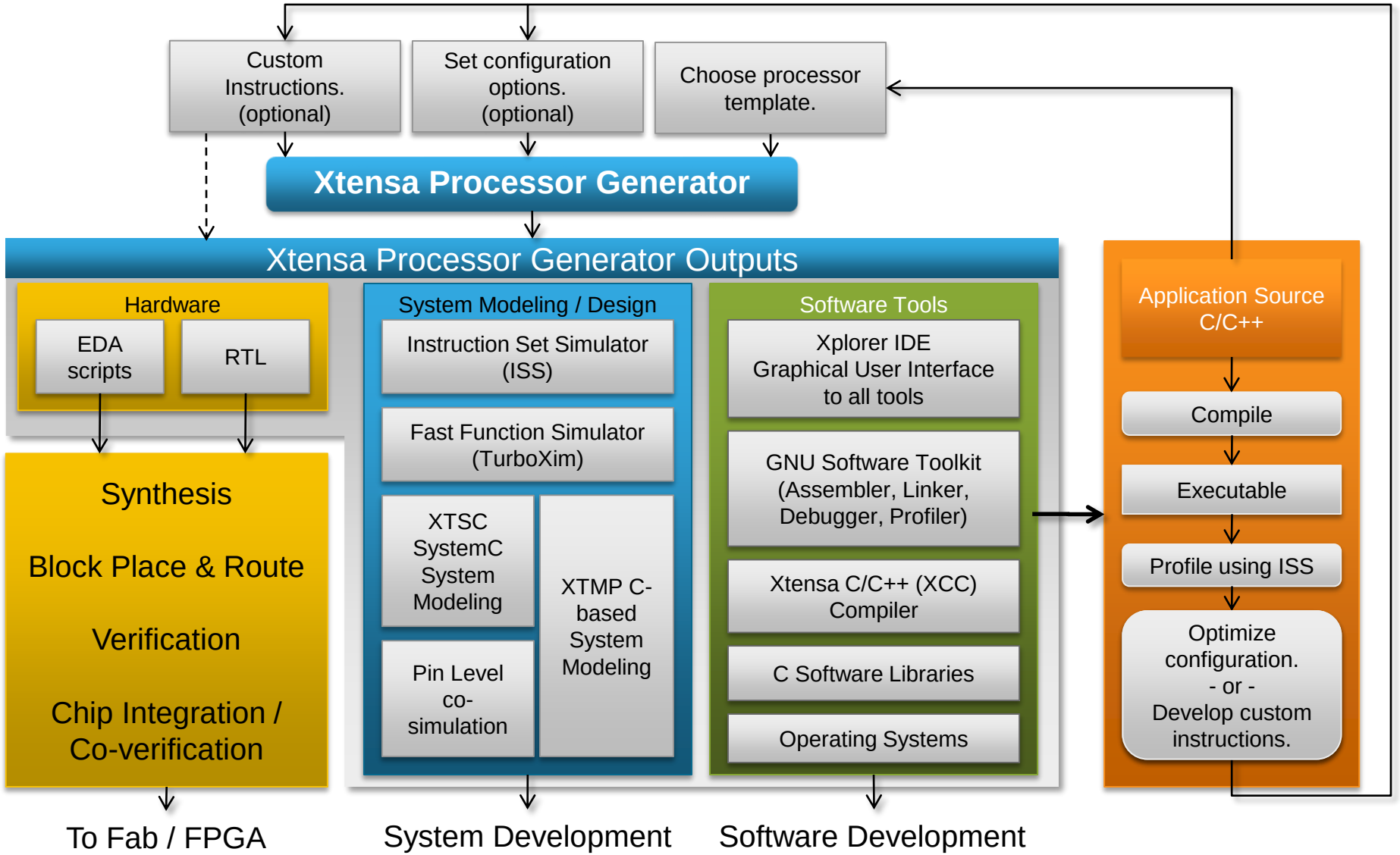
Advanced Software Tools

IDE
C/C++ compiler
Debuggers
Simulators
RTOSes

cadence®

The Tensilica Solution

Fully Automated Hardware and Software Tools Generation



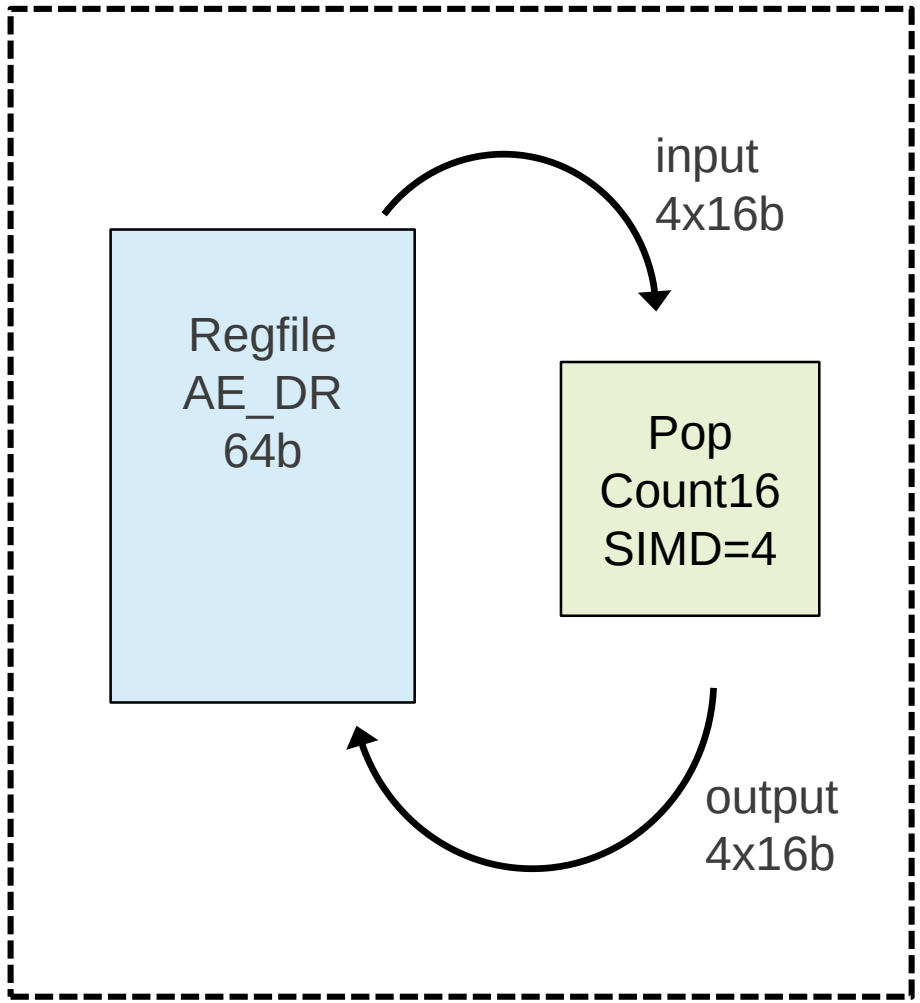
Example – Population Count (aka Hamming weight)
Counting the ‘1’s in a given field

- Let’s take a simple example – “POP_COUNT_16”
  - Count the ‘1’s in 16bit fields and return an answer for each field
  - If the DSP this is attached to has a wide vector register file, we can take advantage and make a SIMD instruction and compute several (many) results per cycle
- Let’s consider two different Fusion DSPs
  - Fusion F1 – a machine with 64b registers
  - Fusion G3 – a machine with 128b registers
  - Vision P6 – a machine with 512b registers (☺ Bonus example ☺)
  - BBE16EP – second bonus example running in FPGA part of the demo ☺ ☺
- How to parameterise the code to make it “work” in all situations with minimal effort?

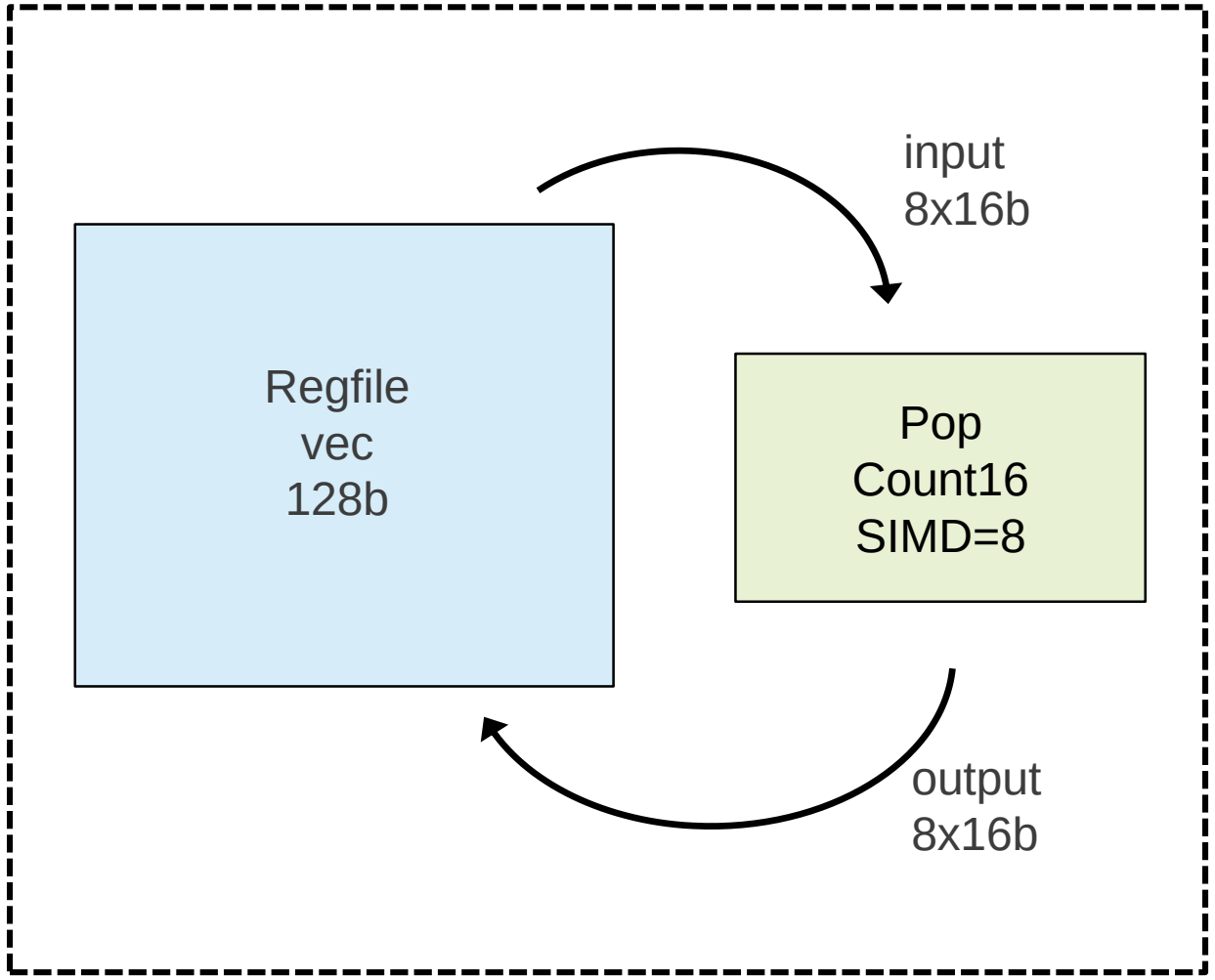
Conceptual dataflow

We consider here the problem of 16b pop counts

Dataflow in Fusion F1



Dataflow in Fusion G3



Considerations

- HW side – the register file operands that we will compute (names etc)
- HW side – the width of the register file operands and therefore the potential SIMD factor of a given implementation
  - MAX SIMD == MAX potential throughput
- SW side – what vector ctypes will we use ?
  - These need to be defined in “protos”
- SW side – in our ‘C’ code we need to provide an indirection mechanism for the vector types
  - use a generic vector type in the ‘C’ and “typedef” it to suit the particular DSP we are using

Writing parameterisable TIE
Using the PERL pre-processor

What is TIE ?

- TIE is a technology that is used to extend a Tensilica processor
  - Used to describe custom execution units, register files, I/O interfaces, load/store instructions, and multi-issue (FLIX) instructions
  - Includes
    - TIE language and TIE compiler
    - Software tools such as the C/C++ compiler, debugger, and instruction set simulator
    - Hardware (RTL) and implementation flows for ASIC/FPGA design
- The TIE Language is used to describe instruction extensions at a high level of abstraction
  - Syntax is a mixture of Verilog and C
  - Describes combinational relation between input and output operands
    - No need to worry about pipelining, control/bypass logic, and interfacing to other processor modules
  - Describes how the instruction extensions are used by software

Parameterising in TIE

Using the PERL pre-processor (simple usage)

```
/*****************************************************************
 * PERL variables - set by the USER - G3 for fusion G3
 *                  P6 for the Vision P6
 *                  and F1 for Fusion F1.  Everything else is automatic.
 *****************************************************************/
;my $DspType = "F1" ;
// E V E R Y T H I N G     E L S E     S E T     A U T O M A T I C A L L Y
;  my $RegType ;                ## name of the register file
;  my $Typedefault ;            ## default ctype of main regfile
;  my $Type8xN ;                ## ctype for vector of 8bit
;  my $Type16xN ;               ## ctype for vector of 16bit
;  my $Type32xN ;               ## ctype for vector of 32bit
;  my $RFW  ;                   ## Register file width
;if("$DspType" eq "F1") {       ## settings for the Fusion F1
;  $RegType = "AE_DR" ;
;  $Type64b = "ae_int64" ;
;  $Type8xN = "ae_int64" ;
;  $Type16xN = "ae_int16x4" ;
;  $Type32xN = "ae_int32x2" ;
;  $RFW = 64 ;
;} elsif("$DspType" eq "G3") {   ## settings for the Fusion G3
;  $RegType = "vec" ;
;  $Type8xN = "xb_vec4Mx8" ;
;  $Type64b = "xb_vec2Mx16" ;
;  $Type16xN = "xb_vec2Mx16" ;
;  $Type32xN = "xb_vecMx32" ;
;  $RFW = 128 ;
;}
```

- Here we define some basic variables that hold some architecture and software parameters
- All lines starting with “;” are treated directly as PERL lines
- In this example we need to set one variable manually – “$DspType”
- Is there a way to automate that ?
  - Hint: *Later* ....

cadence®

Parameterising in TIE

Using the PERL pre-processor (simple usage)

```
/*****************************************************************
 *
 * Function  to calculate pop_count in 8 bits
 *
 *****************************************************************/
function [3:0] fn_popcount8([7:0] a)
{
          wire[3:0] addres    = TIEaddn(a[7], a[6], a[5], a[4],
                                        a[3], a[2], a[1], a[0]) ;
          assign fn_popcount8 = addres ;
}
/*****************************************************************
 *
 * Function  to calculate pop_count in 16 bits
 *
 *****************************************************************/
function [4:0] fn_popcount16([15:0] a)
{
          wire[3:0] addres0   = fn_popcount8( a[ 7: 0] ) ;
          wire[3:0] addres1   = fn_popcount8( a[15: 8] ) ;
          assign fn_popcount16 = TIEadd(addres0, addres1, 1'b0) ;
}
```

- Some simple “building block” functions help make things more readable, and more modular
- TIE functions are similar to Verilog “modules” – they do not create any hardware UNTIL they are instantiated somewhere

Parameterising in TIE

Using the PERL pre-processor (simple usage)

```
 /*************************************************************************
 *
 * This operation performs popcounts across 16bit fields
 *
 *************************************************************************/
 operation POP_COUNT_16       {out `$RegType`     res, in `$RegType` src}
                              {}
 {
;  for($i=0; $i<$RFW/16; $i++) {
     wire[15:0] w16_`$i`, res16_`$i` ;
;}
          assign {
;  for($i=($RFW/16 -1); $i>0 ; $i--) {
     w16_`$i`,
;  }
          w16_0 } = src ;
;  for($i=0; $i<$RFW/16; $i++) {
     assign res16_`$i`         = fn_zpad_5_16(fn_popcount16(w16_`$i`)) ;
;}

    assign          res                                    =         {
;   for($i=($RFW/16 -1); $i>0 ; $i--) {
     res16_`$i`,
;}
          res16_0} ;
 }
```

- Here we can use the variables defined before to create different implementations depending on the DSP we are attaching this TIE to.
- Use of backticks “`” forces evaluation of the variable
- Now we can see that the TIE functions really help to make the code more modular
- We instantiate one “popcount” HW per SIMD lane

Parameterising in TIE – implementation in F1

Using the PERL pre-processor (simple usage)

```
/************************************************************************
*
* This operation performs popcounts across 16bit fields
*
************************************************************************/
operation POP_COUNT_16       {out AE_DR          res, in AE_DR src}
                             {}
{
    wire[15:0] w16_0, res16_0 ;
    wire[15:0] w16_1, res16_1 ;
    wire[15:0] w16_2, res16_2 ;
    wire[15:0] w16_3, res16_3 ;
    assign {
         w16_3,
         w16_2,
         w16_1,
         w16_0 } = src ;
    assign res16_0 = fn_zpad_5_16(fn_popcount16(w16_0)) ;
    assign res16_1 = fn_zpad_5_16(fn_popcount16(w16_1)) ;
    assign res16_2 = fn_zpad_5_16(fn_popcount16(w16_2)) ;
    assign res16_3 = fn_zpad_5_16(fn_popcount16(w16_3)) ;

   assign res =    {
         res16_3,
         res16_2,
         res16_1,
         res16_0} ;
}
```

- Here is the TIE code after the pre-processor has run
- In this case we were compiling the TIE onto “Fusion F1”
- Note the creation of 4 way SIMD engine
- Note the use of AE_DR as the source and destination registers

Parameterising in TIE – implementation in G3

Using the PERL pre-processor (simple usage)

```
/*************************************************************************
*
* This operation performs popcounts across 16bit fields
*
*************************************************************************/
operation POP_COUNT_16       {out vec  res, in vec src}
                                                              {}
{
    wire[15:0] w16_0, res16_0 ;      wire[15:0] w16_1, res16_1 ;
    wire[15:0] w16_2, res16_2 ;      wire[15:0] w16_3, res16_3 ;
    wire[15:0] w16_4, res16_4 ;      wire[15:0] w16_5, res16_5 ;
    wire[15:0] w16_6, res16_6 ;      wire[15:0] w16_7, res16_7 ;
    assign {
    w16_7,      w16_6,      w16_5,      w16_4,
    w16_3,      w16_2,      w16_1,     w16_0 } = src ;
    assign res16_0 = fn_zpad_5_16(fn_popcount16(w16_0)) ;
    assign res16_1 = fn_zpad_5_16(fn_popcount16(w16_1)) ;
    assign res16_2 = fn_zpad_5_16(fn_popcount16(w16_2)) ;
    assign res16_3 = fn_zpad_5_16(fn_popcount16(w16_3)) ;
    assign res16_4 = fn_zpad_5_16(fn_popcount16(w16_4)) ;
    assign res16_5 = fn_zpad_5_16(fn_popcount16(w16_5)) ;
    assign res16_6 = fn_zpad_5_16(fn_popcount16(w16_6)) ;
    assign res16_7 = fn_zpad_5_16(fn_popcount16(w16_7)) ;

   assign res      =         {
    res16_7,      res16_6,      res16_5,      res16_4,
    res16_3,      res16_2,      res16_1,         res16_0} ; }
}
```

- Here is the TIE code after the pre-processor has run
  - *In this case some manual re-formatting has been done to fit the display page*
- In this case we were compiling the TIE onto “Fusion G3”
- Note the creation of 8 way SIMD engine
- Note the use of vec as the source and destination registers

Parameterising in TIE – the software interface

protos – linking the world of software to the processor architecture

```
proto POP_COUNT_16  {out `$Type16xN` z, in `$Type16xN` x} {}
{
          POP_COUNT_16        z, x ;
}
...
in Fusion F1 becomes

proto POP_COUNT_16  {out ae_int16x4 z, in ae_int16x4 x} {}
{
          POP_COUNT_16        z, x ;
}

...
in Fusion G3 becomes

proto POP_COUNT_16  {out xb_vec2Mx16 z, in xb_vec2Mx16 x} {}
{
          POP_COUNT_16        z, x ;
}
```

- protos provide the interface for the compiler to select the correct variable to pass to the intrinsic
- The Intrinsic name is the same but in F1 it's a 4-way SIMD and in G3 it's an 8-way SIMD
- protos are also used for many other software related tasks

Writing portable ‘C’ code –
Using the HAL

The Xtensa HAL – the Hardware Abstraction Layer

- A set of compile-time constants, and runtime used to abstract the details of the HW from the calling application
- In this particular case, we want to control
  - The ‘C’ types that are used to pass information to the intrinsics
  - The SIMD factor and memory accesses
- We can set all this automatically in a header file using the HAL
- Same code will therefore compile on different machines
- This is a simple example, used to show the principles

Using the HAL to define constants and types

```
#ifndef _POPC_DEFS_H_
#define _POPC_DEFS_H_
#if XCHAL_HAVE_FUSION == 1          // really Fusion F1
#include <xtensa/tie/xt_fusion.h>
#define SIMD8  8
#define SIMD16 4
#define SIMD32 2
typedef ae_int64  vec8T ;
typedef ae_int16x4 vec16T ;
typedef ae_int32x2 vec32T ;
#elif XCHAL_HAVE_FUSIONG3 == 1
#include <xtensa/tie/xt_pdx4.h>
#define SIMD8  16
#define SIMD16 8
#define SIMD32 4
typedef xb_vec4Mx8  vec8T ;
typedef xb_vec2Mx16 vec16T ;
typedef xb_vecMx32  vec32T ;
#elif XCHAL_HAVE_VISION == 1
#include <xtensa/tie/xt_ivpn.h>
#define SIMD8  64
#define SIMD16 32
#define SIMD32 16
typedef xb_vec2Nx8  vec8T ;
typedef xb_vecNx16 vec16T ;
typedef xb_vecN_2x32  vec32T ;
#endif
#endif //_POPC_DEFS_H_
```

- XCHAL constants are part of the HAL
- Generated with every core
- There are many of them to cover most features of the core
- Key component of writing portable software (e.g. for RTOS porting)
- Also significant runtime functions

Portable code that will run on multiple platforms

```
/*****************************************************************************
 *
 * Accelerated functions using the new TIE instructions for popcount
 *
 *****************************************************************************/

void popcount16_TIE(uint16_t * __restrict rPtr, uint16_t *data, int count)
{
          int i ;
          vec16T *vecPtr  = (vec16T *) data ;
          vec16T *vecrPtr = (vec16T *) rPtr ;
          for(i=0; i<count/SIMD16; i++)
          {
                    vecrPtr[i] = POP_COUNT_16(vecPtr[i]) ;
          }
}
```

- vec16T has different lengths dependent on the machine
- POP_COUNT_16 intrinsic has different SIMD width dependent on the machine
- SIMD16 #define set according to the underlying DSP width
- Code compiles unmodified on multiple different platforms

The inner loop on different targets ...

Using the exact same ‘C’ source in each case ...

```
ae_l16x4.ip          aed1, a6, 8
{ l32i    a0, a1, 0; pop_count_16       aed1, aed1 }
loopgtz   a3, 60020bf9 <popcount16_TIE+0x2d>
ae_l16x4.ip          aed0, a6, 8
{ ae_s16x4.ip        aed1, a2, 8; pop_count_16     aed1, aed0 }
```

- Fusion F1, single Loadstore, 2 16-bit calcs / cycle
- 2-way FLIX

```
{ nop; nop }
{ loopgtz a4, 400010b0 <popcount16_TIE+0xb0>; pdx_lv_v_i    v2, a2, -16 }
{ pdx_sv_v_ip v1, a7, 32; pdx_lv_v_ip v0, a2, 32; nop; pop_count_16 v1, v0 }
{ pdx_sv_v_i v3, a7, -16; pdx_lv_v_i v2, a2, -16; nop; pop_count_16 v3, v2 }
```

- Fusion G3, dual Loadstore, 8 16-bit calcs / cycle
- Up to 4-way FLIX

```
d000093d: 0020f0                                        nop
{ loopgtz a4, d0000990 <popcount16_TIE+0x11c>; or a8, a2, a2; or      a2, a3, a3;
pop_count_16         v1, v1 }
{ ivp_sv2nx8_i v5, a8, -64; ivp_lv2nx8_ip v0, a2, 192; nop; pop_count_16 v3, v2 }
{ ivp_sv2nx8_ip v7, a8, 64; ivp_lv2nx8_i v2, a2, -128; nop; pop_count_16 v5, v4 }
{ ivp_sv2nx8_ip v1, a8, 192; ivp_lv2nx8_i v4, a2, -64; nop; pop_count_16 v7, v6 }
{ ivp_sv2nx8_i v3, a8, -128; ivp_lv2nx8_ip v6, a2, 64; nop; pop_count_16 v1, v0 }
```

- Vision P6, dual Loadstore, 32 16-bit calcs / cycle
- Up to 4-way FLIX

cadence

Exploiting the HAL in your TIE code

- One way to access HAL features inside TIE is to parse the relevant header file
  - `xtensa-elf/arch/include/xtensa/config/core-isa.h`
- Can parse this in your TIE code and then set variables accordingly:

```
;use lib '/export/bigD/customers/new_cadence/code/tenDay_2017/work/perl' ;

#define XCHAL_HAVE_FUSION               0       /* Fusion*/
#define XCHAL_HAVE_FUSION_FP            0       /* Fusion FP option */
#define XCHAL_HAVE_FUSION_LOW_POWER     0       /* Fusion Low Power option */
#define XCHAL_HAVE_FUSION_AES           0       /* Fusion BLE/Wifi AES-128 CCM option */
#define XCHAL_HAVE_FUSION_CONVENC       0       /* Fusion Conv Encode option */
#define XCHAL_HAVE_FUSION_LFSR_CRC      0       /* Fusion LFSR-CRC option */
#define XCHAL_HAVE_FUSION_BITOPS        0       /* Fusion Bit Operations Support option */
#define XCHAL_HAVE_FUSION_AVS           0       /* Fusion AVS option */
#define XCHAL_HAVE_FUSION_16BIT_BASEBAND 0      /* Fusion 16-bit Baseband option */
#define XCHAL_HAVE_FUSION_VITERBI       0       /* Fusion Viterbi option */
#define XCHAL_HAVE_FUSION_SOFTDEMAP     0       /* Fusion Soft Bit Demap option */
#define XCHAL_HAVE_HIFIPRO              0       /* HiFiPro Audio Engine pkg */
#define XCHAL_HAVE_HIFI4                0       /* HiFi4 Audio Engine pkg */
#define XCHAL_HAVE_HIFI4_VFPU           0       /* HiFi4 Audio Engine VFPU option */
#define XCHAL_HAVE_HIFI3                0       /* HiFi3 Audio Engine pkg */
#define XCHAL_HAVE_HIFI3_VFPU           0       /* HiFi3 Audio Engine VFPU option */
#define XCHAL_HAVE_HIFI2                0       /* HiFi2 Audio Engine pkg */
#define XCHAL_HAVE_HIFI2EP              0       /* HiFi2EP */
#define XCHAL_HAVE_HIFI_MINI            0
```

- simple PERL script parses the core-isa.h file and populates the PERL hash %xchal_db with relevant variables
- Then we can use these variables to set $DspType that we set manually to start with

Calculating Hamming weight using the vector ISA

A more general example

```
/**************************************************************************
 * Faster implementations using some simple vector instructions
 * Note - these were taken from here:
 * https://en.wikipedia.org/wiki/Hamming_weight
 *************************************************************************/
void popcount16_ISAv2(uint16_t * __restrict rPtr, uint16_t *data, int count)
{
```

```
//This uses fewer arithmetic operations than any other known
//implementation on machines with fast multiplication.
//This algorithm uses 12 arithmetic operations, one of which is a multiply.
int popcount64c(uint64_t x)
{
    x -= (x >> 1) & m1;             //put count of each 2 bits into those 2 bits
    x = (x & m2) + ((x >> 2) & m2); //put count of each 4 bits into those 4 bits
    x = (x + (x >> 4)) & m4;        //put count of each 8 bits into those 8 bits
    return (x * h01) >> 56;  //returns left 8 bits of x + (x<<8) + (x<<16) + (x<<24) + ...
}
```

```
  }
}
```

- This is basically scalar code, re-written substituting vector types for scalar types

fferent lengths
the machine

s no intrinsics,
tors working on

s a different
' operators to
structions

cadence

Calculating Hamming weight using the ISA

Loop when compiled onto Fusion F1 – 4 way SIMD

```
6002095e: 41a376              loopgtz   a3, 600209a3 <popcount16_ISAv2+0xc3>
60020961: 98581103001e        { ae_l16x4.ip       aed0, a0, 8; ae_add16         aed3, aed1, aed0 }
60020967: b45a1422001e        { ae_srai16         aed1, aed0, 1; ae_and         aed2, aed2, aed4 }
6002096d: b4583633221e        { ae_s16x4.ip       aed2, a2, 8; ae_and aed3, aed3, aed6 }
60020973: b41f581170fe        { nop; ae_and       aed1, aed1, aed8 }
60020979: a31f501070fe        { nop; ae_sub16     aed0, aed0, aed1 }
6002097f: f21f535170fe        { nop; ae_mul16x4.h aed1, aed3, aed5 }
60020985: f29f535370fe        { nop; ae_mul16x4.l aed3, aed3, aed5 }
6002098b: b45a0702002e        { ae_srai16         aed0, aed0, 2; ae_and         aed2, aed0, aed7 }
60020991: b41e070118ee        { ae_srai32         aed0, aed1, 8; ae_and         aed1, aed0, aed7 }
60020997: 981e321138ee        { ae_srai32         aed3, aed3, 8; ae_add16       aed1, aed2, aed1 }
6002099d: 1c5a0302104e        { ae_srai16         aed0, aed1, 4; ae_sel16i      aed2, aed0, aed3, 8 }
```

- Loop is 11 instructions deep, observing in the ISS tells us “no stalls”
- 11 cycles to compute 4 results.

Calculating Hamming weight using the ISA

Loop when compiled onto Vision P6 – 32way SIMD

```
d0000ffd: 0020f0                                                nop
{ loopgtz a4, d00010a0 <popcount16_ISAv2+0x310>; or a8, a2, a2; or a2, a3, a3; ivp_subnx16 v2, v2, v4; ivp_subnx16 v11, v11,
v19 }
{ ivp_lv2nx8_ip v0, a2, 128; ivp_lv2nx8_i v10, a2, 64; ivp_srainx16 v1, v2, 2; ivp_sranx16 v4, v8, v18 }
{ ivp_and2nx8 v5, v9, v13; ivp_packlnx48 v8, wv0; ivp_srainx16 v9, v11, 2; ivp_and2nx8 v11, v11, v16 }
{ ivp_and2nx8 v2, v2, v16; nop; ivp_and2nx8 v3, v1, v16; ivp_addnx16  v12, v3, v12 }
{ ivp_srainx16 v1, v0, 1; nop; ivp_mulnx16 wv0, v7, v14; ivp_srainx16 v6, v10, 1; ivp_and2nx8 v7, v6, v15 }
{ ivp_addnx16 v4, v2, v3; nop; ivp_and2nx8 v3, v9, v16; ivp_and2nx8 v2, v4, v13 }
{ ivp_and2nx8 v1, v1, v17; ivp_packlnx48 v8, wv0; ivp_and2nx8 v6, v6, v17; ivp_sranx16 v9, v8, v18 }
{ ivp_sv2nx8_ip v5, a8, 128; nop; ivp_srainx16 v5, v4, 4; ivp_addnx16 v3, v11, v3 }
{ ivp_sv2nx8_i v2, a8, -64; nop; ivp_mulnx16 wv0, v7, v14; ivp_subnx16 v2, v0, v1; ivp_subnx16 v11, v10, v6 }
{ ivp_addnx16 v6, v4, v5; nop; ivp_srainx16 v12, v3, 4; ivp_and2nx8 v7, v12, v15 }
```

- No need to analyse in detail ☺ but we can see loop is unrolled by 2 (2 MULNX16)
- Loop is 9 instructions deep, observing in the ISS tells us “no stalls”
- 9 cycles to compute 64 results.

Results of different implementations (1024 calculations)

108mini provides “scalar control machine” performance

Core	Cperf16	Cperf32	Vector ISA 16b	Vector ISA 32b	TIE 16b	TIE 32b
108mini	21,512 (1x)	20,489 (1x)	N/A	N/A	N/A	N/A
Fusion F1	7,702	7,701	2,831	5,135	523	1,035
Fusion G3	7,694	7,694	885	1,717	154	282
Vision P6	5,140	5,138	219	404	58	92

Implementation	Comments
Cperf16	Fast calculation on 16bit fields using standard 'C'
Cperf32	Fast calculation on 32bit fields using standard 'C'
Vector ISA 16b	Fast calculation on 16bit fields using the Vector ISA of the DSP
Vector ISA 32b	Fast calculation on 32bit fields using the Vector ISA of the DSP
TIE 16b	Fast calculation on 16bit fields using TIE instruction and Vector ISA
TIE 32b	Fast calculation on 32bit fields using TIE instruction and Vector ISA

Summary

- TIE provides a highly convenient mechanism for customising the Processor and the processor tool environment
  - Including the programming environment
- ... but sometimes it's useful to make TIE code
  - modular
  - portable
  - automatic
- All the facilities are provided in the Xtensa toolset
  - HAL
  - TIE Preprocessor


cadence®

The Magic of Protos™ (i)

- Protos are used extensively in the Xtensa tools to abstract, guide, “teach” the xcc compiler about the underlying architecture

- Example – Vision P6 does not contain 32x32 multiply instructions
  - We can emulate them with instruction sequences
  - Wouldn’t it be nice if we could hide that from the ‘C’/C++ programmer ?
  - Consider this snippet (vec32T is a vector of 32bit numbers)

```
vec32T h1 = *(vec32T *) &h01L ;

vec32T tmp1, tmp2, tmp3, tmp4 ;
vec32T x ;
for(i=0; i<count/SIMD32; i++)
{
  x = vecPtr[i] ;
  tmp1 = x - ((x >> 1) & m1);
  tmp2 = (tmp1 & m2) + ((tmp1 >> 2) & m2);
  tmp3 = (tmp2 + (tmp2 >> 4)) & m4;
  tmp4 = (tmp3 * h1) >> 24 ;
```

- The ‘*’ operator here must support vector 32x32 multiply – this is not available in a Vision P6 for example
- Protos can “teach” the compiler what to do

The Magic of Protos (ii)

- First we need to define the instruction sequence that creates the function we want
- Then we “link” that sequence to the operator for a given ‘C’ type ...

```
proto MB_MUL32X32   {out xb_vecN_2x32v z, in xb_vecN_2x32v x, in xb_vecN_2x32v y}
                    {xb_vecN_2x64w tmp}
{
  IVP_MULUSN_2X16X32_0        tmp, x, y ;
  IVP_MULAHN_2X16X32_1        tmp, x, y ;
  IVP_PACKLN_2X96             z, tmp ;
}

operator  "*"       MB_MUL32X32
```

- The “proto” statement declares a new intrinsic that takes certain ‘C’ types (already defined) and outputs an operation sequence
- The “operator” statement merely links the proto to a particular ‘C’ operator
- Note the compiler *may* optimise any of the code in the proto in the context of a loop (e.g. register MOVEs the most common candidate)

The Magic of Protos (iii)

- For a bit of fun .... we can create a new ‘C’ type “Hamming Weight Integer” and overload the “-” operator so we can calculate a hamming distance .. using no intrinsics, just ‘C’ operators.
- These two functions are essentially THE SAME (we consider the 16b case here)

```
/*******************************************************
 *
 * Accelerated functions using TIE inst for popcount
 * These functions actually generate the Hamming Distance
 * from a reference  value.
 *
 *******************************************************/
void ham_dist16_TIE(uint16_t * __restrict rPtr,
          uint16_t *data, uint16_t refVal, int count)
{
  int i ;
  vec16T *vecPtr  = (vec16T *) data ;
  vec16T *vecrPtr = (vec16T *) rPtr ;
  vec16T rVal     = (vec16T)(short int) refVal ;
  for(i=0; i<count/SIMD16; i++)
  {
    vecrPtr[i] = POP_COUNT_16(vecPtr[i] ^ rVal ;
  }
}
```

```
/*******************************************************
 *
 * Accelerated functions using TIE inst for popcount
 * These functions actually generate the Hamming Distance
 * from a reference  value.
 *
 *******************************************************/
void ham_dist16_TIE(uint16_t * __restrict rPtr,
          uint16_t *data, uint16_t refVal, int count)
{
  int i ;
  vec16T *vecPtr  = (vec16T *) data ;
  vec16T *vecrPtr = (vec16T *) rPtr ;
  vec16T rVal     = (vec16T)(short int) refVal ;
  for(i=0; i<count/SIMD16; i++)
  {
    vecrPtr[i] = (hw_vec16T)vecPtr[i] - (hw_vec16T) rVal ;
  }
}
```

The Magic of Protos (iv)

- We declare a new ctype “hw_vec16T” (Hamming Weight Vector of 16b)
  - Only SW constructs, no hardware ... just some protos ...
  - This includes some type conversions
- We declare the operation sequence to perform a Hamming Weight Difference
- We associate with the “-” operator ... then we are done ☺
- The type conversions allow the compiler to convert a vec16T ←→ hw_vec16T
  - (These are register moves which get optimised away)
- Then in the hw_vec16T domain we can define “subtract” to mean
  - XOR → POP_COUNT16 ...
  - This is the loop when compiled on Vision P6 ...

```
{ loopgtz a4, d0000cf0 <ham_dist16_TIE+0x17c>; or a8, a2, a2; or a2, a3, a3; ivp_xor2nx8  v1, v1, v12 }
{ ivp_sv2nx8_i v8, a8, -64; ivp_lv2nx8_ip v0, a2, 192; ivp_xor2nx8 v4, v3, v12; pop_count_16 v8, v7 }
{ ivp_sv2nx8_ip v11, a8, 64; ivp_lv2nx8_i v3, a2, -128; nop; ivp_xor2nx8 v7, v6, v12 }
{ ivp_sv2nx8_ip v2, a8, 192; ivp_lv2nx8_i v6, a2, -64; ivp_xor2nx8 v10, v9, v12; pop_count_16 v11, v10 }
{ ivp_sv2nx8_i v5, a8, -128; ivp_lv2nx8_ip v9, a2, 64; pop_count_16 v2, v1; pop_count_16 v5, v4; ivp_xor2nx8
          v1, v0, v12 }
```


cadence®